UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Civil e Arquitetura

Pedro Lima e Silva

MULTI-SCALE MESHING FOR THREE-DIMENSIONAL DISCRETE FRACTURE NETWORKS

GERAÇÃO DE MALHA MULTI-ESCALA PARA REDES DE FRATURAS DISCRETAS TRIDIMENSIONAIS

Campinas
2021

Pedro Lima e Silva

MULTI-SCALE MESHING FOR THREE-DIMENSIONAL DISCRETE FRACTURE NETWORKS

GERAÇÃO DE MALHA MULTI-ESCALA PARA REDES DE FRATURAS DISCRETAS TRIDIMENSIONAIS

Dissertação apresentada à Faculdade de Engenharia Civil, Arquitetura e Urbanismo da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Engenharia Civil, na área de Estruturas e Geotecnia.

Thesis presented to the School of Civil Engineering, Architecture and Urban Planning of the University of Campinas in partial fulfillment of the requirements for the degree of Master of Science in Civil Engineering, in the area of Structures and Geotechnics.

Orientador: Prof. Dr. Philippe Remy Bernard Devloo

Campinas
2021

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

Si38m

Silva, Pedro Lima e, 1992-
Multi-scale meshing for three-dimensional discrete fracture networks / Pedro Lima e Silva. – Campinas, SP : [s.n.], 2021.

Orientador: Philippe Remy Bernard Devloo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Civil, Arquitetura e Urbanismo.

1. Método dos elementos finitos. 2. Geração numérica de malhas - Análise numérica. 3. Meios porosos - Simulação por computador. 4. Reservatórios - Fratura. I. Devloo, Philippe Remy Bernard, 1958-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Civil, Arquitetura e Urbanismo. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Geração de malha multi-escala para redes de fraturas discretas tridimensionais
Palavras-chave em inglês:
Finite element method
Numerical mesh generation - Numerical analysis
Porous media - Computer simulation
Reservoirs - Fractures
Área de concentração: Estruturas e Geotécnica
Titulação: Mestre em Engenharia Civil
Banca examinadora:
Philippe Remy Bernard Devloo [Orientador]
Isaías Vizotto
Paulo Roberto Maciel Lyra
Data de defesa: 21-12-2021
Programa de Pós-Graduação: Engenharia Civil

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-8400-1614
- Currículo Lattes do autor: http://lattes.cnpq.br/70951675339626

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL, ARQUITETURA E URBANISMO

MULTI-SCALE MESHING FOR THREE-DIMENSIONAL DISCRETE FRACTURE NETWORKS

Pedro Lima e Silva

Dissertação de Mestrado aprovada pela Banca Examinadora, constituída por:

Prof. Dr. Philippe Remy Bernard Devloo
Presidente e Orientador / FECFAU Unicamp

Prof. Dr. Isaías Vizotto
FECFAU Unicamp

Prof. Dr. Paulo Roberto Maciel Lyra
DEM UFPE

A Ata da defesa com as respectivas assinaturas dos membros encontra-se no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 21 de dezembro de 2021

Every one I meet is my master in some point,
and in that I learn from them.

(Ralph Waldo Emerson)

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 process number 88882.435161/2019-01.

A special thank you to my great advisor Philippe Devloo, who has been an unending source of knowledge and insight throughout this project.

To all my colleagues and friends from the Laboratory of Computational Mechanics (Labmec) at Unicamp I extend my gratitude. For the incredible and collaborative work environment and for the camaraderie in and out of the lab.

To all other colleagues, professors and staff at the School of Civil Engineering Architecture and Urban Planning (FECFAU|Unicamp), thank you for all the help and welcome extended to me.

To all my family and friends, I am always thankful to you.

Resumo

Simulações de escoamento em de meios porosos é um tópico central de engenharia. Uma técnica (em popularidade crescente) para simular escoamento em meios porosos fraturados é usar os Métodos de Fraturas Discretas (DFMs). Estes idealizam meios fraturados usando elementos de menor dimensão para representar fraturas em uma malha híbrida. DFMs carregam, no entanto, o desafiador requisito de malhas de elementos finitos para domínios crescentemente mais complexos. Tais complexidades rapidamente se intensificam ao passo que o número de intersecções de fraturas se multiplica, e são ainda mais amplificados quando as restrições de duas escalas de discretização de Métodos de Elementos Finitos Multi-escala (MSFEMs) são introduzidas. Nesse contexto, esse trabalho propõe uma nova solução para o estágio pré-processamento de tal classe de problemas, e apresenta uma abordagem para geração de malha de elementos finitos para Redes de Fraturas Discretas (DFNs) em meios porosos, adaptada para MSFEMs. O código, de fonte aberta, que a acompanha foi escrito em C++ moderno e largamente dependente em duas bibliotecas de elementos finitos: NeoPZ e Gmsh. Começando de uma malha-macro definida pelo usuário, lê-se fraturas como polígonos convexos das coordenadas de seus vértices, uma por vez. Os principais passos envolvem: Classificar todos os nós da malha pela sua posição em um dos dois lados da fratura; intersectar arestas checando por nós em lados opostos do plano da fratura; extender interseções de arestas para faces através de conexão de vizinhança; coalescer interseções para o nó existente mais próximo (dada uma tolerância); refinar elementos de interface para se conformar à fratura; identificar subconjuntos complementares da superfície da fratura de acordo com o volume poliedral que os contém; gerar malha de superfície através de triangulações de Delaunay restringidas de cada subconjunto; e seguir repetindo para cada fratura. Com todas as fraturas, localizar contornos e interseções onde elas surgirem. Ao final, o espaço ao redor das fraturas é preenchido com a malha de escala fina não-estruturada, para discretizar o volume da matriz porosa, naturalmente mantida conforme. A implementação, completamente automática, constrói sua confiabilidade e eficiência de uma fundamentação consistente de premissas matematicamente coerentes, como convexidade e conexão de elementos. Resultados mostram que a técnica proposta pode construir malhas tridimensionais adequadas para DFNs, e ainda dar ao usuário a liberdade de ajustar entre fidelidade geométrica e qualidade de malha como for preferível.

Palavras chave: Geração de Malha de Elementos Finitos; Método Multiescala Híbrido-Misto; Redes de Fraturas Discretas; Elementos Finitos; Meios Porosos Fraturados.

Abstract

Simulations of flow through porous media is a core topic in engineering. A growingly popular technique to simulate flow through fractured porous media is to use Discrete Fracture Models (DFMs). These idealize fractured media using lower-dimensional elements to discretely represent fractures within a hybrid mesh. However, DFMs carry the challenging requisite of Finite Element (FE) meshes for increasingly complex domains. Such complexities quickly intensify as the number of fracture intersections multiplies, and are further amplified as the restrictions of two-scale discretizations of Multi-scale Finite Element Methods (MSFEMs) are introduced. In such a context, this work proposes a novel solution for the pre-processing stage of these problems, and presents an approach for FE meshing of Discrete Fracture Networks (DFNs) within porous media, suited to MSFEMs. The accompanying open-source code is written in modern C++ and largely relies on two state-of-the-art FE libraries: NeoPZ and Gmsh. Starting from a user-defined coarse mesh, we read fractures as convex polygons from the coordinates of their corners, one at a time. The main steps involve: Classify every mesh node as being on either side of the fracture; Intersect edges by checking for nodes on opposite sides of fracture plane; Extend intersections from edges to faces through neighbourhood connection; Coalesce intersections to closest existing nodes (given a tolerance); Refine interface elements to conform to fracture; Identify non-overlapping subsets of fracture surface by the polyhedral volume that contains them; Mesh the surface through a restricted Delaunay triangulation of each subset and repeat on to the next fracture. With all fractures, locate boundaries and intersections where they arise. Finally, the space around fractures is filled with the fine-scale unstructured mesh, to discretize the volumetric porous matrix, naturally kept conformal. The implementation pulls its reliability and efficiency from the consistent background of mathematically coherent premises like convexity and element connectivity. Results show that the proposed technique can construct adequate 3D DFN meshes, while still giving users freedom to adjust between geometrical fidelity and mesh quality as they prefer.

Keywords: Finite Element Mesh Generation; Multi-scale Hybrid-Mixed method; Discrete Fracture Networks; Finite Element Method; Fractured Porous Media.

List of Figures

List of Tables

Acronyms and Abreviations

API	Application Programming Interface
CAD	Computer Aided Design
DFM	Discrete Fracture Method
DFN	Discrete Fracture Network
FE	Finite Element
FEM	Finite Element Method
MHM	Multi-scale Hybrid Mixed
MSFEM	Multi-scale Finite Element Method
RHS	Right-Hand Side

Contents

Chapter 1

Introduction

1.1 Outline Of The Thesis

- Chapter 1 introduces the motivations for this work, outlines the goals and global objectives, and sets the literature context with comparable works.
- Chapter 2 treats the core theoretical concepts that support the methodology. There, terms that frequently come up are defined, such as *MHM, DFN, finite element mesh, neighbour element, sides, NeoPZ, Gmsh, refinement patterns*, and others.
- Chapter 3 gives detailed descriptions of all methods used in the code, such that every algorithm is explained and justified.
- Chapter 4 contains examples of use and the type of mesh obtainable with the implementation.
- Chapter 5 finishes with the final discussions and suggestions for extending this research.

1.2 Motivation

Fluid flow through porous media is a core topic of study in engineering. Problems of this type have many applications and are of special interest to Civil and Petroleum Engineering, medical sciences, and others.

Simulations of flow through fractured porous media are routinely applied in the industry of hydrocarbon reservoir exploration, but can also be found applied to groundwater-resource management, radioactive-waste reposition, and carbon capture and storage, among others.

An important observation for a porous medium is that secondary permeability, provided by conductive fractures, can result in a flow pattern that significantly diverges from the flow pattern only considering the primary permeability provided by connected pores. Berre, Doster, and Keilegavlen (2019) explains how, due to tectonic movement, subsurface rocks are always fractured to some degree, and softer soils can often observe occurrence of fractures as well. Further, when fracture flow is the principal mean of transport, it is to be expected that they are inter-connected. Hence, the importance of considering fracture connectivity, in addition to their properties and distribution (LONG et al., 1982).

A frequent choice for modeling the behavior of flow in fractured porous media is the Discrete Fracture Network (DFN) method (Devloo, Teng, and Chen Song Zhang (2019); Jaffré, Mnejja, and Roberts (2011); Jing and Stephansson (2007)). In this type of method, fractures are discretely represented by (d-1)-dimensional elements in a d-dimensional finite element mesh.

The existence of fractures that affect flow and transport are present in different types of porous media for domains ranging from millimeters to hundreds of kilometers (BERRE; DOSTER; KEILEGAVLEN, 2019). When used to faithfully capture the behavior of problems whose data have multiple scales of variation, regular finite element methods can quickly become unfeasible (Durán (2017)). To simulate this kind of problem, recent effort has been put into the creation of Multi-scale Methods, which are capable of coupling different scales (see Efendiev and Hou (2009) and references therein). One of such methods is the Multi-scale Hybrid-Mixed (MHM) method, introduced by Paredes (2013) and Harder, Paredes, and Valentin (2013), which provide a consistent framework for developing multi-scale approximations.

Problem Statement

Multi-scale methods provide a framework for simulating problems with multiple scale heterogeneities and/or large scale fractures, but carry the requirement of special types of meshes that cover two levels of discretization. Therefore, as the number of fractures increases and the network of these discrete fractures grows in complexity, meshing such domains becomes a challenging task. In three dimensions, generating these meshes manually will quickly become unfeasable.

In this context, inspired by the MHM formulation introduced and implemented by Devloo, Teng, and Chen Song Zhang (2019), a demand is clearly identified for a tool that handles the mesh generation for multi-scale meshes of fractured porous media.

1.3 Objectives

The goal of the present work is to build an automatic mesh generator for discrete fracture networks, which must be suited for computing numerical approximations using the Multi-scale Hybrid-Mixed finite element method. The code should be written in C++, require convenient input data, and generate a mesh and corresponding data-structures that are easily manipulated for use in different advanced reservoir simulations.

The following milestones should be achieved:

- Describe arbitrary domains for pertinent numerical formulations using discrete fractures. Specifically:

    - Tridimensional porous medium;

    - Fracture surface;

    - Fracture-fracture intersections, and;

  - Fracture boundaries;

- Establish the theory and general methodology behind the algorithms;
- Verify the code through benchmarks and examples.

1.4 Literature review

Multiple publications have provided solutions for mesh generation of Discrete Fracture Networks. The closest example might be found in Erhel, De Dreuzy, and Poirriez (2009), where a conforming mesh generator was implemented with some optimizations to avoid sharp angles, and the mesh is applied in mixed hybrid finite element methods. Nevertheless, these authors consider only the flow through the fractures with an impervious rock matrix, and find no need for volumetrical meshing. For their implementation, fractures are represented by planar ellipses in $\mathbb{R}^3$ that can be truncated by the boundaries of the domain.

Also with application for mixed hybrid approximations with meshes for DFNs, the work of Maryška, Severýn, and Vohralík (2005) is similar to that of Erhel, De Dreuzy, and Poirriez (2009), but uses circular geometries for fractures.

A popular package is the software developed by Hyman, Karra, et al. (2015). in the package dfnWorks, which uses the LaGriT[1] mesh generator to mesh, and run classical Finite Element simulations on, polygonal three-dimensional fracture networks.

Other noteworthy examples of works in meshing DFNs are: Fourno et al. (2019), Mustapha (2011), Hyman, Gable, et al. (2014), Yongbin Zhang et al. (2015) and Wang et al. (2017)

In the multi-scale context, Akkutlu, Efendiev, and Vasilyeva (2016) give a formulation for the Generalized Multi-scale Finite Element Method (GMsFEM) applied to DFNs in porous media, but with little discussion on how they use neighbourhood information to associate the fine-level discretization to the coarse elements.

However, likely due to their recent origin, the literature seems to lack efforts on mesh generation specifically tailored to DFNs in porous media simulations using multi-scale methods.

Finally, if not clear from the sections on the problem statement and objectives, it should be noted that this work is closely tied with geometric mesh manipulations and MHM solvers implemented in the NeoPZ library, especially the works of Lima, Devloo, and Villegas (2020), Devloo, Teng, and Chen Song Zhang (2019), Durán (2017) and Lucci (2015).

[1]Los Alamos Grid Toolbox, (LaGriT) Los Alamos National Laboratory `http://lagrit.lanl.gov`

Chapter 2

Theoretical background

In order to establish the concepts that are central to our methodology, this chapter is dedicated to discuss the concepts and terminologies which form the basis that guides the choices of implementation in the following chapters.

Formal definitions are given for certain concepts to avoid ambiguity, and graphics are used whenever they are more effective to convey a concept.

2.1 Discrete Fracture Networks in Porous Media

Fractured porous-media systems appear in many fields of applications in earth science, such as reservoir engineering, groundwater-resource management, carbon capture and storage, radioactive waste reposition, and coal bed methane migration in mines or landslides (SCHWENCK et al., 2015).

The concept of fractured porous media can also be applied to biomedical engineering applications, e.g. Erbertseder et al. (2012), where multidimensional flow in capillary tissues is coupled with the one-dimensional flow in vases.

In all, consideration of the presence of fractures in porous media flow is motivated by the fact that interconnected fractures can serve as conduits or barriers for fluid and chemical transport. According to Jing and Stephansson (2007), under such a state, a minor change of fracture connectivity (say, from the addition of one small-scale fracture that links two previously unlinked fractures) might lead to significantly different flow patterns. The connection of fractures in a discrete model (see example in Figure 2.1.1) introduces the influence of such a system in the global flow and conceives the concept of Discrete Fracture Networks (DFN).

A widely studied class of techniques for reservoir simulations is the so-called Discrete Fracture Model (DFM). For those, fractures are modeled through the insertion of lower-dimensional elements in the porous matrix (DEVLOO; TENG; ZHANG, C. S., 2019), and, by doing so, fractures are represented explicitly with unstructured meshes. The primary motivation for DFMs is to idealize fractured media by emphasizing realistic fracture geometries (LA POINTE et al., 1997; KARIMI-FARD; DURLOFSKY; AZIZ, 2004). Such a feature is in direct contrast with the other popular alternative of dual-porosity models.

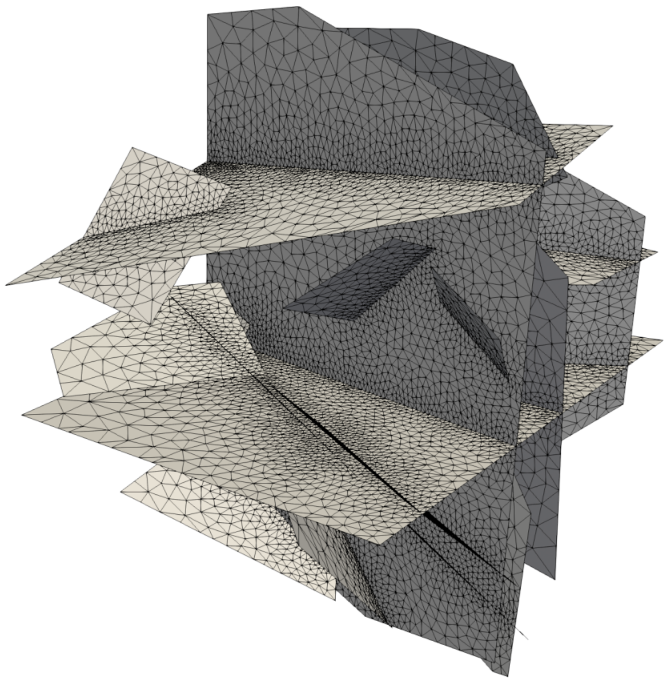


Figure 2.1.1: An example of a DFN with complex arrangements of fractures. Source: DfnWorks (2019) (User Manual)

We note, moreover, that discretely representing each fracture of the domain can lead to expensive simulations. Fractures commonly present in a large variety of geometries and demand unstructured meshes with a rapidly growing number of degrees of freedom. As such, complex meshes are naturally associated with such models, especially as the number of connections between fractures increases.

The class of mathematical models we are interested in stems primarily from considering the balance of mass along (and in between) rock and fissures, and can be seen as an extension (to multiple dimensions) of the classical presentation of Darcy's flow. The main hypotheses are:

- **Fracture aperture is a magnitude smaller than the dimensions of the domain.** Hence the fractures are modeled as lower-dimensional elements;
- **The flow through fractures respects the constitutive characterization of Darcy's law.** Which is a sensible assumption attributable to the presence of debris in their interior;
- **Fluid exchange is present, not only in between intersecting fractures but also between rock and fractures.** The latter is mostly observable in the transverse direction of their surface;
- **Fractures can be regarded as interfaces between porous rock volumes**, which implies the discontinuity to the pressure field on either side of their plane.

To exemplify and give support to the hybrid mixed variational formulation presented in the subsequent section, consider a suitable mathematical model (for Darcy flow) based on those treated by Martin, Jaffré, and Roberts (2005), Ahmed, Jaffré, and Roberts (2017) and Berre, Boon, et al. (2021).

The model starts with a problem domain Λ, of which we can characterize the subdomain corresponding to the volumetric porous matrix (Ω_3). Over this subdomain the flow follows the classic Darcy model, with which we seek the paired pressure and fluid velocity fields $(p_3, \mathbf{u}_3)$, measured in m and m/s respectively, such that:

$$\mathbf{K}_3^{-1}\mathbf{u}_3 + \nabla p_3 = \mathbf{0} \qquad \text{in } \Omega_3; \tag{2.1.1a}$$

$$\nabla \cdot \mathbf{u}_3 = f_3 \qquad \text{in } \Omega_3; \tag{2.1.1b}$$

and the following boundary conditions hold

$$p_3 = \overline{p} \qquad \text{on } \partial\Omega_3 \cap \partial\Lambda_p, \tag{2.1.2a}$$

$$\mathbf{u}_3 \cdot \mathbf{n} = \overline{u} \qquad \text{on } \partial\Omega_3 \cap \partial\Lambda_u, \tag{2.1.2b}$$

where:

- The boundary is split into 2 complementary subsets:
  - $\partial\Lambda_p$: over which pressure is known;
  - $\partial\Lambda_u$: over which the normal component of the fluid velocity is known;
- $\partial\Omega_3$ is the boundary of a volumetric subdomain (implying that its dimension is 2);
- $\mathbf{n}$ is an outward pointing normal vector associated to that boundary;
- $\mathbf{K}_3$ (m/s) is a symmetric, positive definite tensor representative of the volumetric fluid conductivity;
- f_3 $(1/s)$ is a known source/sink term.

The next subdomain considered is the fracture surface (Ω_2). The quantities of interest here are the integrated pressure and tangential fluid velocity $(p_2, \mathbf{u}_2)$, measured in m and m^2/s respectively, and given for every $x \in \Omega_2$:

$$\mathbf{u}_2(x) = \int_{\varepsilon_2(x)} \mathbf{u}_t$$

$$p_2(x) = \frac{1}{\varepsilon_2(x)} \int_{\varepsilon_2(x)} p$$

The reduced Darcy law and mass balance over these domains dictate that:

$$\mathbf{K}_2^{-1}\mathbf{u}_2 + \nabla p_2 = \mathbf{0} \qquad \text{in } \Omega_2; \tag{2.1.3a}$$

$$\nabla \cdot \mathbf{u}_2 - \sum \mathbf{u}_3 \cdot \mathbf{n} = f_2 \qquad \text{in } \Omega_2; \tag{2.1.3b}$$

$$\mathbf{u}_3 \cdot \mathbf{n} + \frac{2\kappa_2}{a_2}(p_2 - p_3) = 0 \qquad \text{in } \Gamma_2; \tag{2.1.3c}$$

and the following boundary conditions hold

$$p_2 = \overline{p} \qquad \text{on } \partial\Omega_2 \cap \partial\Lambda_p, \tag{2.1.4a}$$

$$\mathbf{u}_2 \cdot \mathbf{n} = \varepsilon_2 \overline{u} \qquad \text{on } \partial\Omega_2 \cap \partial\Lambda_u, \tag{2.1.4b}$$

$$\mathbf{u}_2 \cdot \mathbf{n} = 0 \qquad \text{on } \partial\Omega_2 \backslash (\Gamma_1 \cup \partial\Lambda) \tag{2.1.4c}$$

where:

- $\partial\Omega_2$ is the boundary of a surface (implying that its dimension is 1);
- $\Gamma_2 = \Omega_2 \cap \partial\Omega_3$ is the interface between the fracture surface and the boundary of a volumetric subdomain that intersect over it;
- $\partial\Omega_2 \backslash (\Gamma_1 \cup \partial\Lambda)$ is the subset of fracture boundaries completely embedded within the porous matrix (excluding the parts that coincide with the intersections with other fractures);
- $\mathbf{n}$ is an outward pointing normal vector associated to the limits of a domain;
- $\mathbf{K}_2$ (m/s) is a symmetric, positive definite tensor representative of the fluid conductivity tangential to the fracture surface;
- κ_2 (m/s) represents a positive scalar for the normal hydraulic conductivity for fluid coming from the porous matrix into the fracture surface;
- ε_2 (m) is the fracture aperture;
- $a_2 = \varepsilon_2$ (m) is the typical length for that fracture;
- $f_2 = \displaystyle\int_{\varepsilon_2(x)} f$ (m/s) is a known integrated source/sink term;

These equations extend analogously to the fracture intersections (Ω_1). There, we seek the paired integrated pressure and fluid velocity fields $(p_1, \mathbf{u}_1)$, measured in m and m^3/s respectively, such that:

$$\mathbf{K}_1^{-1}\mathbf{u}_1 + \nabla p_1 = \mathbf{0} \qquad \text{in } \Omega_1; \tag{2.1.5a}$$

$$\nabla \cdot \mathbf{u}_1 - \sum \mathbf{u}_2 \cdot \mathbf{n} = f_1 \qquad \text{in } \Omega_1; \tag{2.1.5b}$$

$$\mathbf{u}_2 \cdot \mathbf{n} + \frac{2\kappa_1}{a_1}(p_1 - p_2) = 0 \qquad \text{in } \Gamma_1; \tag{2.1.5c}$$

and the following boundary conditions hold

$$p_1 = \overline{p} \qquad \text{on } \partial\Omega_1 \cap \partial\Lambda_p, \tag{2.1.6a}$$

$$\mathbf{u}_1 \cdot \mathbf{n} = \varepsilon_1 \overline{u} \qquad \text{on } \partial\Omega_1 \cap \partial\Lambda_u, \tag{2.1.6b}$$

$$\mathbf{u}_1 \cdot \mathbf{n} = 0 \qquad \text{on } \partial\Omega_1 \backslash (\Gamma_0 \cup \partial\Lambda) \tag{2.1.6c}$$

where:

- $\partial\Omega_1$ is the boundary of a curve (implying that its dimension is 0);
- $\Gamma_1 = \Omega_1 \cap \partial\Omega_2$ is the interface between the intersection length and one of the surfaces that creates it;
- $\partial\Omega_1 \backslash (\Gamma_1 \cup \partial\Lambda)$ is the set of boundary points of these curves that are completely embedded within the porous matrix (excluding the parts that coincide with interior nodes of other curves);
- $\mathbf{n}$ is an outward pointing normal vector associated to the limits of a domain;
- $\mathbf{K}_1$ (m/s) is a symmetric, positive definite tensor representative of the fluid conductivity tangential to the fracture intersection;
- κ_1 (m^2/s) represents a positive scalar for the normal hydraulic conductivity for fluid coming from a fracture surface into the intersection;
- ε_1 (m^2) denotes the cross-sectional area of that intersection;
- $a_1 = \varepsilon_1^{1/2}$ (m) is the typical length for that intersection;
- $f_1 = \displaystyle\int_{\varepsilon_1(x)} f$ (m^2/s) is a known integrated source/sink term;

The last domain considers intersections of fracture intersections, which is defined by points (Ω_0). There, the conservation of mass is completed and the flux through it still follows a Darcy-type law:

$$\sum \mathbf{u}_2 \cdot \mathbf{n} = f_0 \qquad \text{in } \Omega_0; \tag{2.1.7a}$$

$$\mathbf{u}_1 \cdot \mathbf{n} + \frac{2\kappa_0}{a_0}(p_0 - p_1) = 0 \qquad \text{in } \Gamma_0; \tag{2.1.7b}$$

where:

- p_0, mesured in (m), is the pressure at that point;
- $\Gamma_0 = \Omega_0 \cap \partial\Omega_1$ is the interface between the (0D)-intersection length and one of the surfaces that created it;
- $\mathbf{n}$ is an outward pointing normal vector associated to the limits of a domain;
- κ_0 (m^3/s) represents a positive scalar for the normal hydraulic conductivity for fluid coming from a curve into the point intersection;
- ε_0 (m^3) denotes the volume of that point intersection;
- $a_0 = \varepsilon_0^{1/3}$ (m) is the typical length for that intersection;
- f_0 (m^3/s) is a known point source/sink term;

2.2 Multi-scale Mixed Finite Elements for Flow in DFNs

The Finite Element Method (FEM) is a widely popular numerical method for approximating engineering problems (enunciated as boundary value problems) through space discretization. Here, we refer to the Finite Element Method in its most common meaning: Weak form Galerkin approximations over element-wise-defined basis functions. A touch of the essentials is made but, for in-depth treatments, the reader is referred to specialized works (e. g. Oden, Becker, and Carey (1981), Zienkiewicz, Taylor, and Zhu (2013), Reddy (2015) and Surana and Reddy (2017)).

The core ideas of finite element approximations are to

- Divide the problem domain into closed subdomains (K) of simple topologies, which can be associated, through an invertible affine map, to a reference geometry called *master element*;
- Derive a variational formulation of the original boundary value problem, stated as an equation of linear forms (B), which implies a set of admissible solutions;
- Construct an approximation space ($\{\phi_i\}$) by choosing linearly independent basis functions (in the aforementioned set of admissible solutions) defined over the parametric space of the master elements. The triple $(K, \{\phi_i\}, B)$ defines the so-called *finite element*;
- Project the solution onto the approximation space and, by doing so, transform the problem into an algebraic system of equations;
- Assemble the element-wise algebraic systems into a global system that is solved for the whole domain.

2.2.1 Multi-scale Hybrid Mixed method

Many engineering problems involve data that observe variations in multiple scales. A frequently cited example comes from problems of groundwater transport, in which heterogeneity of subsurface formations can vary permeability accross layers of rock and soil, and faults that can typically reach from tens to hundreds of meters. In fact, multiple scales parameters are bound to dominate simulation efforts wherever large disparities in spatial scales are encountered (EFENDIEV; HOU, 2009). The motivation behind multi-scale methods comes from the need to capture small-scale effects on the large-scales without demanding computation of all small-scale features within the global algebraic problem.

The Multi-scale Hybrid Mixed method (MHM) was introduced by (Paredes (2013); Harder, Paredes, and Valentin (2013)). An extension of the method, implemented in the NeoPZ environment, is given by Durán et al. (2019). More recently, the MHM method has been applied to discretely fractured porous media by Devloo, Teng, and Chen Song Zhang (2019).

The main idea of multi-scale finite element approximations is to associate two levels of discretization to efficiently simulate problems dominated by multi-scale features. For this, Efendiev and Hou (2009) describe how they mainly rely on two traits:

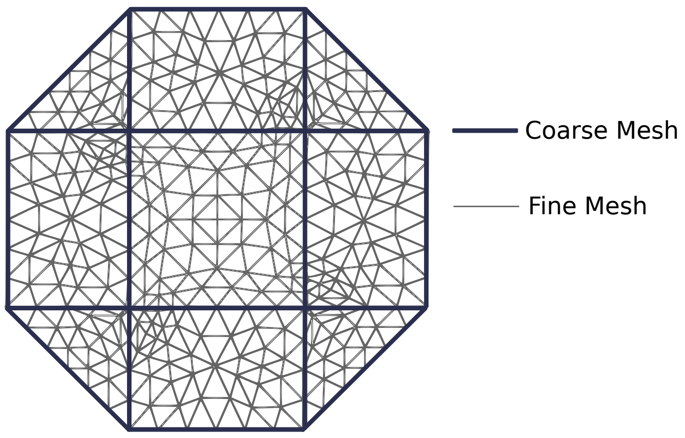


Figure 2.2.1: Example two-level discretization for MHM methods.

- The construction of multi-scale basis functions by solving local flow problems to incorporate fine-scale effects into coarse-scale equations and
- A global numerical formulation that couples multi-scale basis functions.

Let $\mathcal{T}_H$ be a first-level polyhedral discretization of the problem domain (written with a capital H to emphasize the larger-scale size of elements at this level). We call this partition a **coarse mesh**, and it is comprised of the union of **coarse elements**. For each coarse element we introduce a second level partition $\mathcal{T}_h$ (with a lower-case h) and call it a **fine mesh**, composed of **fine elements**. For illustrating, the reader is referred to the example mesh in Figure 2.2.1.

For more details on the MHM implementation for Darcy flow problems, see Durán et al. (2019) or Harder, Paredes, and Valentin (2013).

A Mixed FEM formulation for discrete fractures in porous media

With the goal of reducing the size of the global algebraic system, the MHM-H(div) method takes a 2-scale discretization of the computational space. It groups the fine elements, and its interfaces normal fluxes are the global degrees of freedom. The fine mesh is a partition of each coarse element, and its degrees of freedom are viewed as internal to the coarse elements and, as such, condensed into the global algebraic system without affecting its size. Discrete fractures are two-dimensional elements that exist as the interface between fine elements, and thus are part of the fine mesh.

Following the methodology introduced by Durán et al. (2019), approximation of the interior mesh problem is constructed over an H(div) conforming space, which allows us to carry the multi-scale computations by restricting the fine-scale shape functions to the space of macro fluxes at the interface of macro-regions. Thus, the variational formulation for the MHM-H(div) method matches that of a mixed formulation over an H(div) space. The added feature is that, in the multi-scale setup, the flux approximation space is partitioned between macro fluxes associated with the boundary of the macro domains, and internal fluxes and pressures are associated with the interior (DEVLOO; TENG; ZHANG, C. S., 2019).

As such, the starting point for this framework is the variational mixed problem for the tridimensional porous matrix. For that, the following approximation spaces are taken:

$$\begin{aligned} \mathscr{V}(\mathcal{T}_d) &\subset H(\text{div}, \Omega_d); \\ \mathscr{P}(\mathcal{T}_d) &\subset L^2(\Omega_d) \end{aligned} \tag{2.2.1}$$

piece-wise defined over a hybrid, conformal at each discretization level, finite element mesh $\mathcal{T}_d$ (d=2 for fractures and d=3 for porous medium); Here, the the inner-product spaces $H(\text{div}, \Omega)$ and $L^2(\Omega)$ hold their usual meaning:

The $L_2(\Omega)$ is a Hilbert space for square integrable functions, defined as

$$L_2(\Omega) = \left\{ f \ : \int_\Omega f^2 \, d\Omega < \infty \right\} \tag{2.2.2}$$

The $H(\text{div}, \Omega)$ is the Hilbert space for functions whose divergence are square integrable

$$H(\text{div}, \Omega) = \{ f \in L_2(\Omega) \ : \ \nabla \cdot f \in L_2(\Omega) \} \tag{2.2.3}$$

and the divergence-compatibility property is verified $\nabla \cdot \mathscr{V} \subset \mathscr{P}$. Detailed constructions of such spaces can be found in De Siqueira, Devloo, and Gomes (2013).

The tridimensional mixed FE problem that follows is constructed from the strong statement given in Equations 2.1.1–2.1.7 and directly derived from those given by Devloo, Teng, and Chen Song Zhang (2019) and Villegas et al. (2021).

For the porous matrix, we seek the pair $(\mathbf{u}_3, p_3) \in \mathscr{V}(\mathcal{T}_3) \times \mathscr{P}(\mathcal{T}_3)$ such that:

$$\int_{\mathcal{T}_3} \mathbf{K}_3^{-1} \mathbf{u}_3 \cdot \boldsymbol{\psi}_3 \, d\mathcal{T} - \int_{\mathcal{T}_3} p_3 \, \nabla \cdot \boldsymbol{\psi}_3 \, d\mathcal{T} + \int_{\Gamma_2} \boldsymbol{\psi}_3 \cdot \mathbf{n} \, p_2 \, d\Gamma = -\int_{\partial\Lambda_p} \boldsymbol{\psi}_3 \cdot \mathbf{n} \, \overline{p} \, da \quad \forall \boldsymbol{\psi}_3 \in \mathscr{V}(\mathcal{T}_3), \tag{2.2.4}$$

$$-\int_{\mathcal{T}_3} \varphi_3 \, \nabla \cdot \mathbf{u}_3 \, d\mathcal{T} = \int_{\mathcal{T}_3} \varphi_3 f_3 \, d\mathcal{T} \qquad \forall \varphi_3 \in \mathscr{P}(\mathcal{T}_3) \tag{2.2.5}$$

Likewise, we add to the system the variational constitutive and conservation laws within the fracture surface. There we seek the pair $(\mathbf{u}_2, p_2) \in \mathscr{V}(\mathcal{T}_2) \times \mathscr{P}(\mathcal{T}_2)$, and the Lagrange multiplier with the interpretation of pressure across fracture intersections $p_1 \in \mathscr{P}(\mathcal{T}_1)$, such that:

$$\int_{\mathcal{T}_2} \mathbf{K}_2^{-1} \mathbf{u}_2 \cdot \boldsymbol{\psi}_2 \, d\mathcal{T} - \int_{\mathcal{T}_2} p_2 \, \nabla \cdot \boldsymbol{\psi}_2 \, d\mathcal{T} + \int_{\Gamma_1} \boldsymbol{\psi}_2 \cdot \mathbf{n} \, p_1 \, d\Gamma = -\int_{\partial\Omega_2} \boldsymbol{\psi}_2 \cdot \mathbf{n} \, \overline{p} \, da \quad \forall \boldsymbol{\psi}_2 \in \mathscr{V}(\mathcal{T}_2) \tag{2.2.6}$$

$$-\int_{\mathcal{T}_2} \varphi_2 \, \nabla \cdot \mathbf{u}_2 \, d\mathcal{T} + \int_{\Gamma_2} \varphi_2 \sum \mathbf{u}_3 \cdot \mathbf{n} \, d\Gamma = 0 \qquad \forall \varphi_2 \in \mathscr{P}(\mathcal{T}_2) \tag{2.2.7}$$

Finally, the system is completed by the mass conservation for fracture intersections:

$$\int_{\Gamma_1} \varphi_1 \sum \mathbf{u}_2 \cdot \mathbf{n} \, d\mathcal{T} = 0 \qquad \forall \varphi_1 \in \mathscr{P}(\mathcal{T}_1) \tag{2.2.8}$$

In essence, these equations can be seen as direct extensions to the classical mixed formulation of a Darcy flow (approximated on an H(div)-conforming space). To couple the mixed-dimensional flow in-and-out of fractures from-and-to porous rocks, this formulation takes the terms which observe the reciprocal conservation of mass: Where there is fluid exchange from the rock to the fracture, flux that leaves the tridimensional domain enters as a source term for the bidimensional conservation law; Reciprocally, the pressure of the fluid in the fracture surface acts as a boundary pressure for the porous matrix fluid flow. The same logic applies to fracture intersections.

Other numerical formulations of Darcy flow in fractured porous media using discrete fractures can be found in Juanes, Samper, and Molinero (2002) and Jaffré, Mnejja, and Roberts (2011). For the Brinkman version of Darcy flow, coupled with Biot's poroelasticity, see Bukač, Yotov, and Zunino (2017). For models that simplify rocks to be impervious, such that flow is limited to the fractures, the reader can refer to Maryška, Severýn, and Vohralík (2005) and Erhel, De Dreuzy, and Poirriez (2009).

2.2.2 Mesh Requirements

We close this section by underscoring what exactly are the constraints the framework of MHM solvers applied to DFNs puts on the pre-processing step of mesh generation.

To clearly connect this mixed finite element formulation into our core discussion of mesh generation, we are exclusively interested on over what integration domains these linear forms should be computed. To label these domains, take the saddle-point presentation of Equations 2.2.4–2.2.8, organized in a tabular manner in Equation 2.2.9, where emphasis is given to how trial and test functions are paired at each inner-product.

$$\begin{array}{c|ccccc|c}
 & \mathbf{u}_3 & p_3 & \mathbf{u}_2 & p_2 & p_1 & \text{RHS} \\
\hline
\boldsymbol{\psi}_3 & t_{11} & t_{12} & 0 & t_{14} & 0 & r_1 \\
\varphi_3 & t_{21} & 0 & 0 & 0 & 0 & r_2 \\
\boldsymbol{\psi}_2 & 0 & 0 & t_{33} & t_{34} & t_{35} & r_3 \\
\varphi_2 & t_{41} & 0 & t_{43} & 0 & 0 & 0 \\
\varphi_1 & 0 & 0 & t_{53} & 0 & 0 & 0
\end{array} \tag{2.2.9}$$

The integration domains are thus evidently associated to each term:

- t_{11}, t_{12}, t_{21} and r_2 are computed over the tridimensional porous matrix;
- r_1 is computed at the boundary of the domain;
- $t_{14}, t_{41}, t_{33}, t_{34}$ and t_{43} are computed at the fracture surfaces;
- r_3 is computed at the boundary of each fracture (excluding portions where fracture boundaries coincide with fracture intersections);
- t_{35} and t_{53} are computed at fracture intersections;

Furthermore, other than establishing the required integration sub-domains, our discussion has identified the following constraints on what defines a suitable mesh for these methods:

- The domain has two levels of discretization, a fine and a coarse;
- Fine elements are a discretization of the internal domain of coarse elements;
- Fracture surface elements are lower-dimensional elements and part of the fine mesh;

2.3 Finite Element Mesh Generation

A basic premise of the finite element method is a process called *discretization*, through which the problem domain is partitioned into an acceptable finite number of subdomains called finite elements. These elements, as argued by Lo (2015), should neither overlap nor have gaps between each other in order to form a coherent representation of the real domain of the problem. The strength of finite element methods lies in their ability to break complex multidimensional domains into simpler subdomains whose behaviour is readily understood (Oden, Becker, and Carey (1981)). As such, the quality of the mesh is central to the quality of the approximation.

Let Ω be a closed bounded domain in $\mathbb{R}^d$ with $d \in \{1, 2, 3\}$. Following a usual definition, similar to that given by George (1996), a partition $\mathcal{T}_h$ is a mesh of Ω, split into elements K of simple geometry, if the following conditions hold:

1. $\Omega = \bigcup_{K \in \mathcal{T}_h} K$
2. The interior of every element K in $\mathcal{T}_h$ is non-empty
3. The intersection of the interior of two elements is empty

The subscript h is representative of a metric for the size of the elements of the mesh. A frequent value taken for h is the maximum circumscribed diameter among all elements. Condition 1 asserts that a proper polygonal representation of the domain gets more faithful as the size of elements reduce (which implies that the number of elements is increased) and, in collaboration with condition 2, states that it should completely cover the domain. Condition 3 is the enforcement that no elements should overlap.

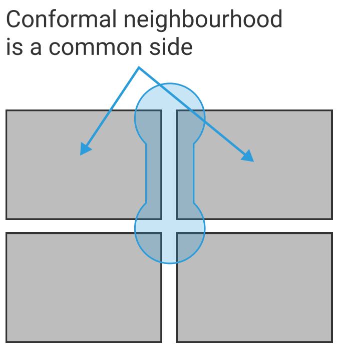
Introducing another condition, we are able to define a **conformal mesh** (or conforming mesh):

4. In a conformal mesh, the intersection of two elements is either the empty set, a vertex, an edge, or a face (for d = 3).

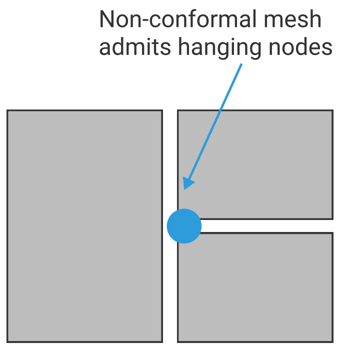
Conformal meshes are the conventional type of meshes used in finite element computations (Frey and George (2008)), and this last condition formalizes the rejection of features like *hanging nodes* (Figure 2.3.1) and edges intersecting the interior of a 2D area.

The customary geometries employed to build meshes are quadrilaterals and triangles for 2D, and tetrahedra, pyramids, prisms or hexahedra for 3D. All of which are contemplated by our proposed methods in Chapter 3.

Lo (2002) reviews the most popular types of methods applied to the construction of finite element meshes:



(a) Conformal mesh



(b) Non-conformal mesh with a hanging node

Figure 2.3.1: Comparison of conformal and non-conformal meshes.

- Octree techniques;
- Delaunay-based;
- Advancing-front method;
- Selective refinement;

Broadly speaking, works on unstructured numerical grid generation typically divide procedures for partitioning arbitrary domains into 2 groups. They either achieve it by filling an unmeshed interior with elements directly (Advancing-front); or they modify an existing base mesh that already covers the domain in order to give it the desired properties (Octrees, Delaunays, Selective refinement).

For this work, focus is given to the Delaunay-based algorithms – since they are the ones used in key parts of our methodology. The reader is referred to (Lo (2002) or Lo (2015)) and references therein for treatments of the others.

2.3.1 Mesh Quality

The answer for the question *"is this a good quality mesh"* is usually one that can be answered from multiple criteria. Since the role of the mesh is to provide a discrete domain into which to run a numerical simulation, the best judge of a mesh quality is the error measure for the simulation itself (FREY; GEORGE, 2008). Nonetheless, it is often the case when numerical approximations will take a lot of computational power and time, and hence, a good quality mesh is desirable beforehand. The reduction of error due to poor discretization is what motivates the concept of quality metrics based on parameters independent of the approximate solution of the problem.

The common attempt on setting a measure for quality is based on the idea, which is defended by Frey and George (2008) and widely held, that a nice-looking mesh is a good mesh.

By attributing reasonably computable numbers to the notion of a *"nice-looking mesh"*, authors tend to adopt measures based on the shape of elements. Thus, a factor of the aspect ratio of triangles and tetrahedra can be set to an acceptable interval (e.g. George (1996); Lo (2002));

and minimization of the number of sharp internal angles is targeted, which gives birth to the notion that a Delaunay mesh is the optimal mesh (BERG et al., 2001), since it is defined from that very same premise.

Lo (2015) gives some consistent quality metrics. For a triangle ABC, its α-quality factor is taken as the ratio of its area by the sum of the squared lengths of its edges, multiplied by a normalizing factor that ensures an equilateral triangle has α set to one.

$$\alpha(ABC) = 2\sqrt{3}\frac{\|AB \times AC\|}{\|AB\|^2 + \|AC\|^2 + \|BC\|^2} \tag{2.3.1}$$

and, for a tetrahedron ABCD, the γ-quality comes from its volume and the sum of its edges squared.

$$\gamma = 72\sqrt{3}\frac{\mathrm{v}}{s^{3/2}} \tag{2.3.2}$$

where v is the volume of the tetrahedron and s is the sum of squared edge lengths:

$$\mathrm{v} = \frac{1}{6}\left|AD \cdot (AB \times AC)\right| \tag{2.3.3}$$

$$s = \|AB\|^2 + \|BC\|^2 + \|CA\|^2 + \|DA\|^2 + \|DB\|^2 + \|DC\|^2 \tag{2.3.4}$$

For the Gmsh library (Geuzaine and Remacle (2009)), quality of triangles are imposed as

$$\alpha(ABC) = 4\frac{\sin\hat{A}\sin\hat{B}\sin\hat{C}}{\sin\hat{A} + \sin\hat{B} + \sin\hat{C}} \tag{2.3.5}$$

where $\hat{A}$ denotes the angle at vertex A. Also in Gmsh, tetrahedra quality is measured through

$$\gamma = \frac{6\sqrt{6}\ \mathrm{vol}}{\left(\sum_i^4 a(f_i)\right)\max(l)} \tag{2.3.6}$$

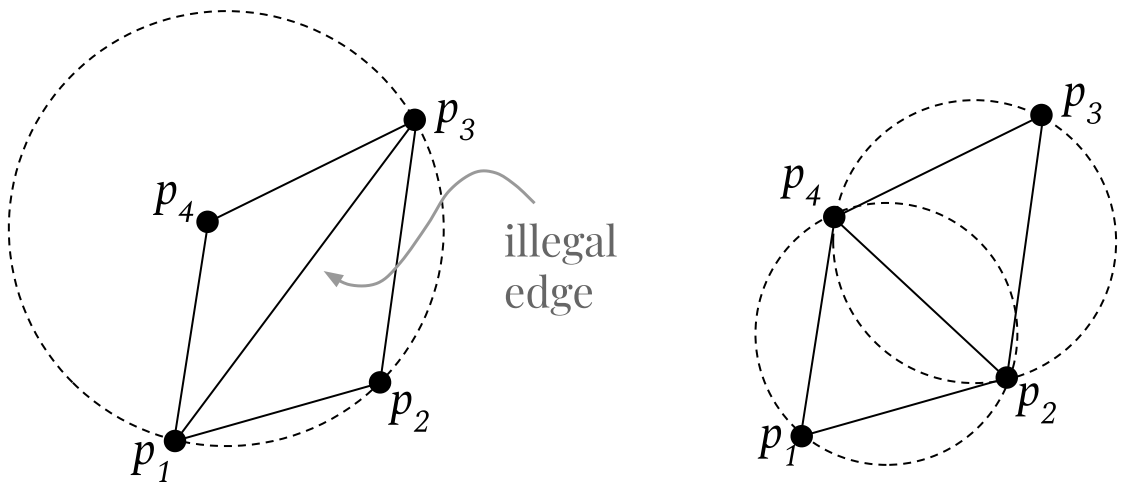
where $a(f_i)$ is the area of face i, and $\max(l)$ is the length of the biggest edge of the tetrahedron.

A common feature to all these metrics is the interval [0,1], wherein equilateral triangles and regular tetrahedra (whose faces are all equilateral) have quality value 1, and degenerate elements will yield zero quality values.

2.3.2 Delaunay Triangulations

As discussed in Section 2.3.1, a widely accepted criterion for a good mesh, is that it contains the minimal possible amount of sharp internal angles. This very notion is what motivates the description of a Delaunay triangulation. However, instead of defining it in terms of the internal angles, a more general definition, which is valid for higher dimensional versions of triangles (i.e. tetrahedra) is most opportune, and, by doing so, *angle-optimality* remains but is seen as a property.

It should be noted that, although *mesh* and *triangulations* are often used interchangeably in the context of finite elements, they are not necessarily the same thing. The concept of meshes



(a) Non-Delaunay triangulation (b) Delaunay triangulation

Figure 2.3.2: Illegal edges and edge-flipping operation in Delaunay triangulations.

does not demand that elements be triangles (or tetrahedra). Additionally, triangulations are usually prescribed for a set of points and are bounded by their convex hull.

Following a definition similar to that of Berg et al. (2001), let $\mathcal{P}$ be a set of points in $\mathbb{R}^d$, and let $\mathcal{T}$ be a triangulation of $\mathcal{P}$. Then, $\mathcal{T}$ is a **Delaunay triangulation** if, and only if, the open discs[1] that circumscribe any simplex[2] in $\mathcal{T}$ do not contain a point of $\mathcal{P}$ in its interior.

The dual of the Delaunay triangulation is the Voronoi diagram defined on the same point set. For any point $\mathbf{p}_j$ in the point set $\mathcal{P}$, Si (2013) defines the Voronoi cell of $\mathbf{p}_j$ as the set of coordinates $\mathbf{x}$ with distance to $\mathbf{p}_j$ not greater than to any other point of $\mathcal{P}$, i.e. it is the set $\mathcal{V}(\mathbf{p}_j) = \{\mathbf{x} \in \mathbb{R}^d : \|\mathbf{x} - \mathbf{p}_j\| \leq \|\mathbf{x} - \mathbf{p}_k\|, \ \forall \mathbf{p}_j, \mathbf{p}_k \in \mathcal{P}\}$

Two important concepts that are frequently present in discussions about Delaunay triangulations are *illegal edges* and the *edge flipping* operation.

Take a set of four points, non co-linearly disposed, such as that illustrated in Figure 2.3.2. A naive, although perfectly valid, triangulation of those points can be obtained from triangles $p_1p_2p_3$ and $p_1p_3p_4$ (Figure 2.3.2a). To test if this triangulation obeys the Delaunay criterion, we trace the circumscircle of $p_1p_2p_3$. Clearly, since p_4 is contained in the circle, we do not have a Delaunay triangulation. Now, because edge $\overline{p_1p_3}$ is incident to two triangles, we can obtain a new triangulation by switching that edge into $\overline{p_2p_4}$. We refer to that operation as an **edge flip**, and, because the internal angles were locally increased through this flip, we say that edge $\overline{p_1p_3}$ is an **illegal edge**. In 3D triangulations (i.e. tetrahedralizations), the concept of illegal edges and edge flipping are extended to oriented illegal faces and face flipping.

The Delaunay criterion itself is not an algorithm for mesh generation. It merely provides a rule to connect a set of existing points in space. As a result, it is necessary to design a method

[1]Here a disc represents a multidimensional concept for a sphere or a circle. Both being the 3D and 2D versions of a disc, respectively.

[2]Here a simplex represents a multidimensional concept for a tetrahedron or a triangle. Both being the 3D and 2D versions of a simplex, respectively.

to determine the number and the locations of node points to be inserted within the domain of interest (LO, 2002).

The most popular strategies in mesh generations of this type are oriented by incremental insertion algorithms (CHENG; DEY; SHEWCHUK, 2016). Their approach is to create a triangular mesh large enough to contain the entire domain, then insert boundary nodes progressively, while connecting nodes as they enter the domain. Through edge flipping operations, the algorithms maintain the triangulation to be Delaunay as points are inserted. After the boundary points enter, optimization of the mesh begins and new points are inserted sequentially at the circumcenter of the element that has the largest adimensional circumradius, while the quality of the mesh is improved through a combination of edge flipping and vertex re-positioning. Methods similar to this are described in Geuzaine and Remacle (2009) for the implementation in the Gmsh library.

Constrained Delaunay Triangulations

A strong point of Delaunay triangulations is that, through them, mesh generation is based on a sound theoretical background from which efficient and reliable algorithms can be formulated (LO, 2002). It is the case, however, as noted by Frey and George (2008), that these methods carry no guarantee that boundaries will be preserved, and this is indispensable for the mesh generation scope.

Constructing a mesh that conforms to a user defined geometry is what prompts the discussion of constraints. For any geometry to be meshed, unless it is simply a cloud of points, there will be prescribed edges, faces and/or volumes. This is consistently true for the boundary of the domain, which is expected to bound the corresponding triangulation. Delaunay-based algorithms understand these predefined geometrical entities as constraints to the triangulation (Cheng, Dey, and Shewchuk (2016)).

Constrained Delaunay triangulations can, thus, be seen as triangulations that try to be *"as much Delaunay as possible"*. Accordingly, constrained triangulations tend to obey the Delaunay criteria throughout the domain, with the exception of the neighbourhoods of the constraints.

In order to respect the constraints, not much modification of the original algorithms is required. Generally, the approach is to either skip any edge flip and point insertion/relocation that would, otherwise, interfere with a constraint, or to run a boundary recover algorithm after the triangulation is set, in order to reconstruct the constraints. According to Gmsh documentation, their implementation follows the latter logic, and surface mesh is recovered using Si (2013) algorithm (Tetgen/BR).

Much more can be said on the topic of Delaunay triangulations and Delaunay-based meshing techniques, and, for that, the reader is referred to Berg et al. (2001), Lo (2015), Cheng, Dey, and Shewchuk (2016) and references therein.

2.4 NeoPZ Paradigms

NeoPZ (or simply: PZ) is a scientific computing environment idealized by Professor Philippe R. B. Devloo (DEVLOO, 1997) at Unicamp during the 1990s. It is an evolving project that has been the main framework on which research is conducted at the Laboratory of Computational Mechanics - LabMeC/Unicamp.

The goal of NeoPZ is to provide a class structure that is general enough to embrace a large number of finite element algorithms. The majority of the code is written in C++, and its approach is aimed toward a coherent framework guided by an object-oriented philosophy.

For either one, two or three-dimensional problems, the main blocks of work in NeoPZ library are:

1. Geometry modeling;
2. Approximation space generation;
3. Variational formulation definition;
4. Assembly and resolution of algebraic systems;
5. Post-processing.

For purposes of this work, focus is given to the paradigms that portray major roles in the assembly of our methodology. These are mainly geometrical tools, definitions and data-structures.

At the time of this work, all the code for NeoPZ and other parallel projects are hosted open source at GitHub (`https://github.com/labmec`). For project documentation, the reader is referred to `https://labmec.github.io/neopz/`.

2.4.1 Topology

Under the context of geometry, Gemignani (1990) defines topology as the study of properties preserved under homeomorphisms. Meaning continuous deformations such as stretching and bending but not tearing or gluing.

Defining some data structures based on topological consistencies is a demand that follows the adoption of master element spaces in finite element codes. But it is also a useful way to handle geometries through manipulation of properties that will remain true even after successive transformations (as long as they are homeomorphic).

Topologies available in the PZ library (Figure 2.4.1) are classical in element-wise discretization methods: Points, Lines, Triangles, Quadrilaterals, Tetrahedra, Pyramids, Triangular prisms and Hexahedra.

To further discuss topologies in the PZ context, the concepts of sides and elements should be clearly established. A **side** (Ω_j) is an open set of points associated to one of the available topologies. Zero-dimensional sides are corners and one-dimensional sides are edges. An **element** ($\bar{\Omega}^e$) is a closed set of points and the union of its N sides (Equation 2.4.1). The intersection of sides is empty, and thus, each element can be partitioned into its sides. Sides are numbered from 0 to N-1, from lowest to highest dimensional sides, starting from the corners, then edges followed by faces and volumes. There is an unique side whose closure[3] is the proper element, this side is usually referred to as the *interior* of the element and its index is conventionally set to N-1.

$$\bar{\Omega}^e = \bigcup_{j=0}^{N-1} \Omega_j \tag{2.4.1}$$

Table 2.1 contains the number of sides for each topology available in PZ. As an example, a quadrilateral element (Figure 2.4.1) has 9 sides: 4 corners ($N^{(0)} = 4$), 4 edges ($N^{(1)} = 4$) and 1 surface ($N^{(2)} = 1$).

Table 2.1: Number of sides associated to each topology.

Topology	$N^{(0)}$	$N^{(1)}$	$N^{(2)}$	$N^{(3)}$	$N = \sum N^{(d)}$
Point	1	0	0	0	**1**
Edge	2	1	0	0	**3**
Triangle	3	3	1	0	**7**
Quadrilateral	4	4	1	0	**9**
Tetrahedron	4	6	4	1	**15**
Pyramid	5	8	5	1	**19**
Prism	6	9	5	1	**21**
Hexahedron	8	12	6	1	**27**

2.4.2 Neighbourhood

Finite element methods require, by definition, the partitioning of the problem domain into a mesh of subdomains. This discretization is posed in the literature, such as in Surana and Reddy (2017) and Frey and George (2008), as not arbitrary and intended to cover the whole domain in a continuous disposition. Such continuity implies the existence of inter-element interfaces, which motivates the concept of elemental neighbourhood.

Neighbourhood is one of the fundamental concepts in the study of topology. Bredon (1993) gives a formal definition of neighbourhood for topological spaces without assuming a metric: Let x be a subset of a topological space X, a neighbourhood $\mathscr{N}^x$ of x is a set that includes an open set U that contains x.

[3]The closure of a topological subset Ω_j is the intersection of all closed sets ($\bar{\Omega}$) which contain Ω_j.

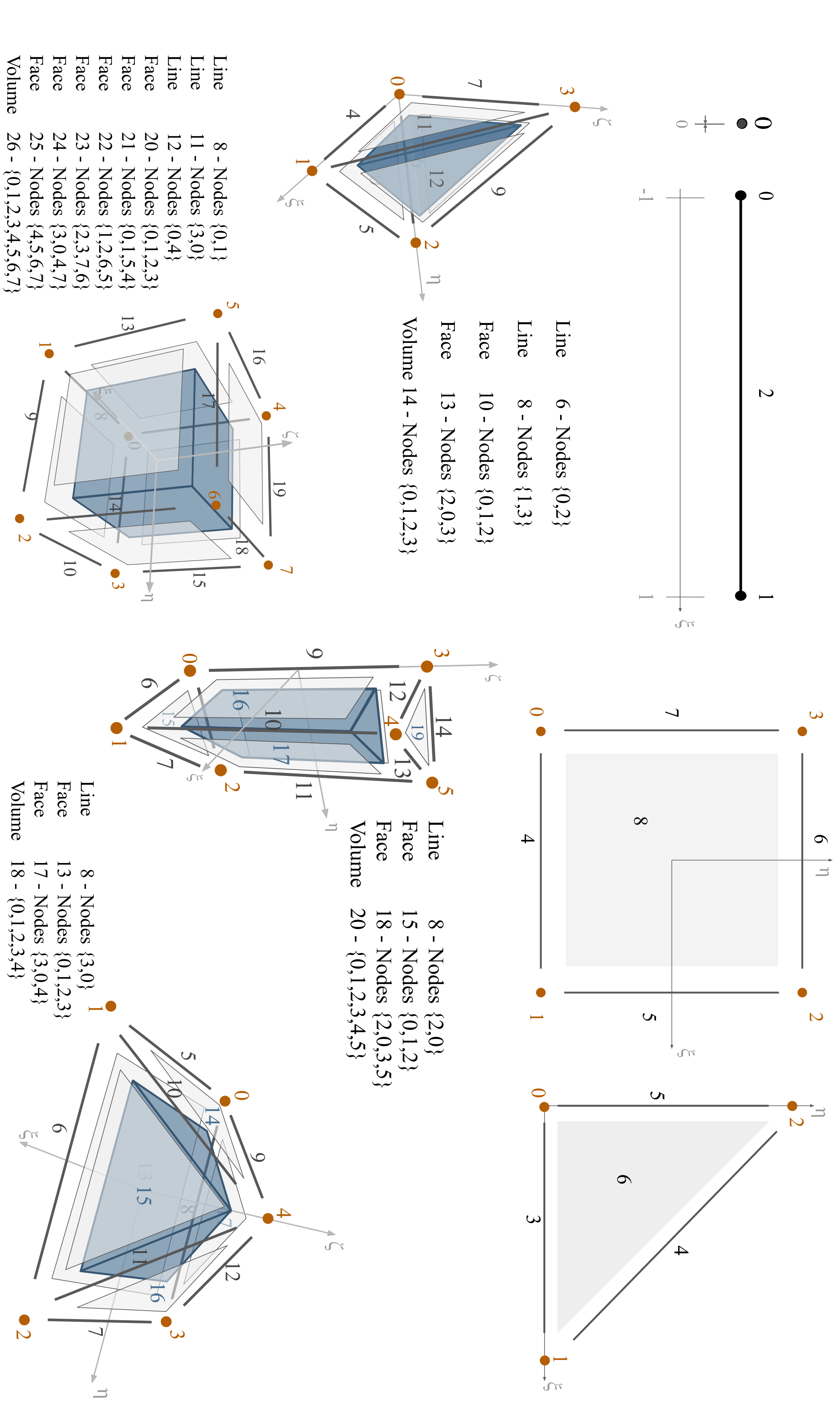

0
Line 6 - Nodes {0,2}
Line 8 - Nodes {1,3}
Face 10 - Nodes {0,1,2}
Face 13 - Nodes {2,0,3}
Volume 14 - Nodes {0,1,2,3}
Line 8 - Nodes {0,1}
Line 11 - Nodes {3,0}
Line 12 - Nodes {0,4}
Face 20 - Nodes {0,1,2,3}
Face 21 - Nodes {0,1,5,4}
Face 22 - Nodes {1,2,6,5}
Face 23 - Nodes {2,3,7,6}
Face 24 - Nodes {3,0,4,7}
Face 25 - Nodes {4,5,6,7}
Volume 26 - {0,1,2,3,4,5,6,7}
Line 8 - Nodes {2,0}
Face 15 - Nodes {0,1,2}
Face 18 - Nodes {2,0,3,5}
Volume 20 - {0,1,2,3,4,5}
Line 8 - Nodes {3,0}
Face 13 - Nodes {0,1,2,3}
Face 17 - Nodes {3,0,4}
Volume 18 - {0,1,2,3,4}
ξ
η
ζ

Figure 2.4.1: Topologies for geometrical elements available in PZ.

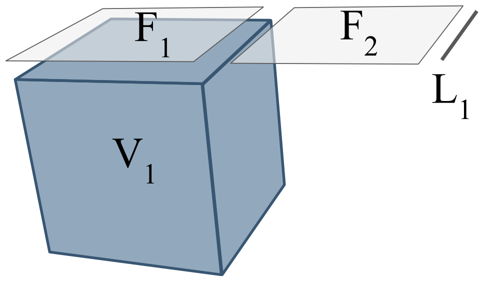


Figure 2.4.2: Neighbouring elements.

In the context of NeoPZ, neighbourhood is consistently defined through shared sides, so neighbours are usually referred-to in the meaning of a *neighbour through a specific side.* Two elements are considered neighbours through a side, if both of them contain a side that exists in the exact same space. In fact, for a conformal mesh, the following construct is often useful and frequently what we mean by the term *neighbourhood*: take an element $\bar{\Omega}^e$ that belongs to a geometrical finite element mesh $\mathcal{T}$, the set of all its neighbours ($\mathscr{N}_j^e$) through a side Ω_j is defined as the set of all elements $\bar{\Omega}^k$ of that mesh which contain that side, excluding the element itself (Equation 2.4.2).

$$\mathscr{N}_j^e = \{\bar{\Omega}^k \in \mathcal{T} \ : \ \Omega_j \subset \bar{\Omega}^k\} \backslash \{\Omega^e\} \tag{2.4.2}$$

For example, Figure 2.4.2 illustrates an arrangement of 4 elements (exploded to make a clear distinction of each element): Two faces F_1 and F_2, a volume V_1 and a line L_1. Element F_1 is neighbour of V_1 through all of its sides, neighbour of F_2 through an edge and 2 corners, and is not neighbour of L_1. The set of neighbours of F_2 through the lower-left corner are $\{F_1, V_1\}$. The set of neighbours of V_1 through its bottom face is an empty set.

Elementary connectivity is a common trait in finite element codes, but NeoPZ stands out with a group of libraries that find use in elements tracking neighbourhood, as a topological construction, through each one of their sides. Neighbour queries are handled through objects of the class `TPZGeoElSide`, which is a class of objects that pair a geometrical element (`TPZGeoEl`) with an integer for the local index of the side of interest. Requesting a neighbour of an object of the type `TPZGeoElSide` means asking for a neighbour of the pointed element through that side, and the return type is another `TPZGeoElSide` that represents that neighbour. Most interestingly, neighbours, within NeoPZ's data structure, are cleverly organized as an (implied) *circularly linked list*; Hence, repeatedly calling the next neighbour of a `TPZGeoElSide` is guaranteed to loop over all neighbours (without skipping or repeating) and finally return to the initial `TPZGeoElSide`. Such mechanism is quite advantageous in allowing for effortless navigation through neighbour lists and is exhaustively used through our methodology.

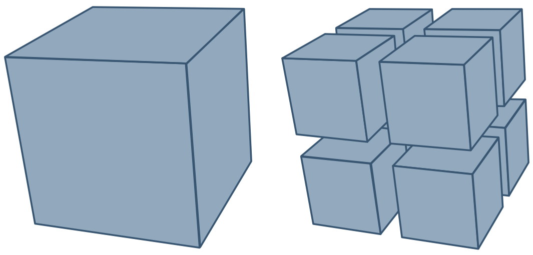


Figure 2.4.3: Example of uniform refinement

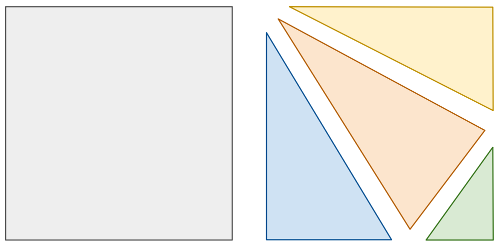


Figure 2.4.4: Example of non-uniform refinement

2.4.3 Refinement Patterns and Refinement Trees

A common topic in treatments of finite elements is h-adaptivity, on which elements of the mesh are subdivided into smaller elements in order to improve the representation of the domain and increase degrees of freedom as necessary. Typically, this is done with the specification of an error norm, and refinement is guided towards its minimization (SURANA; REDDY, 2017). In its adaptive FEM origins, NeoPZ has innately contemplated h-refinement in its attributions.

Geometric elements may be divided uniformly, as put by Lucci (2015): edges are split in 2 and higher-dimensional sides follow correspondingly by generating smaller versions of itself to conform (Figure 2.4.3). Yet, there may rise the need for partitions that are not uniform (Figure 2.4.4).

In PZ, arbitrary refinement is handled through refinement patterns. The definition of a **refinement pattern** is rather straight forward:

1. **A set of nodes**. This includes the original father corner nodes, and any new node that children elements may require. All defined in master element space;
2. **A father element**, that is identified through its topological sequence (i.e. ordered allocation of corner nodes).
3. **A set of children** also defined from the indices of their corner nodes and completely cover their fathers volumes.

Arbitrary refinement patterns, in NeoPZ, are of the class `TPZRefPattern`. It is important to note that, by covering the whole volume of the father, each side of the sub-elements is contained in one of the father sides. As such, for most refinement patterns, there exists an affine transformation which maps all points in the parametric space of the child side to the corresponding father side parametric space. Computing such transformations is an attribution of this class.

We can use these to compare father-element and sub-element orientations, since the sign of the determinant tells us if an affine transformation is orientation preserving.

`TPZRefPattern`s can be constructed from a string file containing information on a predefined format, which is convenient to statically determine uniform refinements. Conversely, for dynamically setting refinement patterns, a more convenient way is to use a geometrical mesh that contains all required nodes, the father as element zero, and the children elements. Is clear from Chapter 3, that the approach of our methodology is not to replace an old mesh with a new fractured one, but rather split a coarse mesh to conform to inserted fractures using refinement patterns defined at execution time. Accordingly, all refinement patterns discussed here on are non-uniform and dynamically defined.

The class responsible for implementing h-refinement in the NeoPZ library is the specialized geometric element `TPZGeoElRefPattern`. Objects of this class are geometric elements with an arbitrary refinement pattern and a vector of sub-element indices. It is through these elements that the complete **refinement tree** is kept in the mesh if children elements are further refined. Useful queries come from navigating this structure, such as:

- **Eldest Ancestor**: Gives a pointer to the geometric element that originated that tree;
- **Youngest Children**: Fills a container with pointers to all sub-elements, up the branches from the queried element, that have not yet been refined (elements that do not have children of their own).

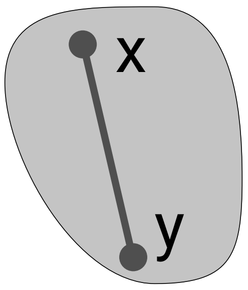
Finally, as we navigate the refinement tree, the concept of **refinement levels** inevitably arises. As each element is connected to its sub-elements and father element (if existent), we count the level of refinement from the *eldest ancestor* which has level zero or, equivalently, exists in the lowest refinement level. Moving up the tree means progressively accessing sub-elements, such that all unrefined elements in the mesh exist at the highest refinement levels.

2.5 Notes On Convex Polyhedra

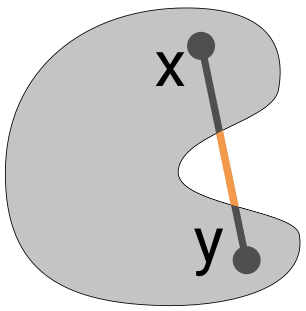
Throughout our methodology presentation, frequent mentions are made to convex geometric figures. As they certainly play a key role to our discussions, what follows in this section are some definitions and simple theorems that we can use to support algorithmic choices presented in the next chapter.

Since these concepts are of rather introductory nature, most of them are frequently found in the initial chapters of textbooks on geometry. All definitions here are either directly taken or slightly adapted from one of the following references: Aleksandrov (2005), Bredon (1993) and Boissonnat and Yvinec (1998)

We start, of course, with a formal definition of a convex region (Figure 2.5.1a).



(a) Convex region



(b) Non-Convex (or concave) region

Figure 2.5.1: Illustration of convexity.

Definition 2.5.1. (Convex region)
A region (or body) $\mathfrak{R} \subset \mathbb{R}^n$ is said convex if, for each pair of points $x, y \in \mathfrak{R}$, it contains the entire segment $\overline{xy}$ that connects them.

To avoid any confusion on terminology, we refrain from refering to *concave* bodies, and instead use *"non-convex"* to convey lack of convexity (Figure 2.5.1b).

A polygon is a 2D region bounded by finitely many edges that connect in pairs at vertices. Likewise, a polyhedron is a 3D region bounded by finitely many polygons that connect in pairs at edges. To impose convexity on those, we simply attach to them the description of a convex region. Particularly to these discrete regions (and directly due to their discrete and affine nature), we get the following equivalent definitions:

Definition 2.5.2. (Convex polyhedron)
A polyhedron is said convex if either of the two following equivalent properties is verified:

- The entire figure lies on one side of the plane that contains each of the facets that bound them;
- All internal dihedral-angles, between each pair of facets, are up to 180°.

The use of the general term *facet* makes it so that this definition trivially extends to convex polygons.

In more rigorous treatments (e.g. in works of Convex Analysis) a distinction between *convexity* and *strict-convexity* is usually present. For our consideration, this would imply limiting the upper bound of internal angles of convex polyhedra to 180°, excluding 180°. Through our methods, however, we relax this definition to include 180°, since it carries no risk to robustness.

Other than the definition of convexity, the next Theorem and the Lemma that follows it are directly invoked.

Theorem 2.5.3. (Intersection of convex sets)
The intersection $\mathfrak{I}$ of any collection of convex regions $\mathfrak{R}_i$ is convex itself.

Proof: By definition, any pair of points $x, y \in \mathfrak{I}$ are also points in the regions $\mathfrak{R}_i$. By invoking the convexity of those regions, we know that the segment that connects x and y is also contained within all the regions $\mathfrak{R}_i$, and therefore belongs to their intersection $\mathfrak{I}$, making it convex as well. ■

Lemma 2.5.4. (Cross-section of convex polyhedron is convex)
Any planar cross-section of a convex polyhedron is a convex polygon.

Proof: By definition, a convex polyhedron is bounded by planar polygons. The cross-section of a polyhedron is, therefore, bounded by the continuously connected straight segments which are the cross-sections of these polygons.

Without loss of generality, we can pick any convex subset of the unbounded crossing plane which is big enough that it completely contains the cross-section of the plane with the polyhedron. From Theorem 2.5.3 we know that the intersection of 2 convex regions yields a convex region, which completes the proof. ■

Chapter 3

Methodology

The context of Multi-scale Hybrid-Mixed methods imposes clear constraints on the geometric description of a Discrete Fracture Network. The chief requirements are for sub-meshes built around the fracture surface elements to fill the interior of each coarse region, and a data structure that consistently associates fine/coarse elements. The steps of pre-processing we now set out to implement should then abide to these constraints and ultimately deliver the integration domains described in Section 2.2.2.

Positioning nodes, lines and surfaces; manually setting which subsets of surfaces lie within which volume; and having conventional meshing software generate an unstructured mesh for such geometric configuration, is not beyond a reasonable realm of complexity for engineering and scientific endeavors; but it can be a rather dull and time consuming job to perform manually. This is specially true for the very common case of fractures that intersect multiple coarse elements and other fractures. From this we take the motivation to find a systematic and robust approach, which is delegated to a computer.

This chapter contains the proposed method to automatically generate suitable meshes from simple input data, under an object oriented philosophy. We start with a superficial mention to the main steps of the program, in order to illustrate the global idea and motivations of the method. Right after, we dive into the details and principles of the developed algorithms.

The entirety of the code is written in C++ and, at the time of writing, hosted open source at `https://github.com/labmec/dfnMesh`. We largely rely on classes and functions available from two open source finite element libraries, namely NeoPZ[1] (Devloo (1997)) and Gmsh[2] (Geuzaine and Remacle (2009)).

In addition to geometrical meshes and refinement patterns from NeoPZ, and mesh modules from Gmsh, very simple blocks make up our implementation. Along this chapter we motivate and define each of them. Namely, the main components are:

- Coarse mesh (Section 3.1);

[1]Source code currently at: `https://github.com/labmec/neopz`
documentation currently at: `https://labmec.github.io/neopz/`

[2]Source code currently at: `https://gitlab.onelab.info/gmsh/gmsh`
documentation currently at: `http://gmsh.info/doc/texinfo/gmsh.html`

- Polyhedral volumes (Section 3.1.2);
- Faces sorted around edges (Section 3.1.2);
- Fractures (a construction made with);
    - Intersected edges (Section 3.2.3);
    - Intersected faces (Section 3.2.5);
    - A convex polygon (Section 3.2.1);
    - A surface mesh (Section 3.3);

A remark in notation should be pointed out for the rest of this chapter: We shall refer to **fracture plane** or **fracture polygon** as the object generated from a (user-informed) set of coplanar points that define a convex polygon, which is, in fact, planar. **Fracture surfaces** are a reference to the set of all *2D* elements that amount to the actual description of the fracture, and are likely **not** planar.

From the user, we take only 3 inputs: The coarse mesh, a list of convex polygons to define a desired position for the fractures, and geometrical tolerances which determine how the code should balance between mesh quality and fidelity to the user input. Appendix A contains an example input file in `json` format.

Starting from a coarse mesh and treating the fracture insertion as a refinement problem, gives this methodology the ability to conserve the geometry of the coarse elements and the relationship coarse/fine mesh, since fine elements stem from a refinement. To support the conservation of the coarse mesh, we also add a policy of *not changing coordinates of any node once it has been set.*

By taking fractures one-by-one, this method enjoys the advantage of being able to follow a quasi-linear logic: The problem of meshing a set of fractures reduces to repeating the one of inserting a single fracture into whatever state the previous fracture left the mesh.

Globally, the logic behind the construction of a fractured mesh arises somewhat naturally, once we impose the constraints that make it suitable to MHM methods. The main steps that add up to our proposed method are:

- **Initialize a coarse mesh and a list of fractures**: Read an input file and translate it into objects the code is expected to work with;
- **Initialize the skeleton mesh**: In between coarse volumes, lower-dimensional elements are created. This allows us to define more abstract volumes in terms of boundary faces;
- **Divide the skeleton mesh by the fracture plane**: Exploring mesh conformity and the convexity of most of our geometric figures to perform our construction by refining elements from 1D to 3D;
- **Mesh fracture surface**: Identifying subsets of the fracture surface, we mesh simple regions that cover all the surface;

- **Recover the fracture surface boundaries**: Orthogonal virtual planes are used to refine extensions of the fracture surface to recover the boundaries set by the input data;
- **Identify fracture-fracture intersections**: After all fractures were created, a graph is setup with edges common to each pair of surfaces, over which we can solve a shortest path problem that recovers the intersection of these 2 fractures;
- **Mesh volumes**: Abstract polyhedral volumes, kept throughout the running of the program, are tetrahedralized to define a regular finite element mesh[3].

The methods proposed are, in many ways, a composition of neighbourhood information, convexity of key geometrical figures, some level of mesh conformity and very simple geometrical constructions.

As a final remark for this introduction, we note that, for most of the procedures proposed in this chapter, we shall assume the fracture is unbounded, or at least that its limits do not intersect any element in the mesh. In Section 3.4, the program conception gets completed by our description of the operations necessary to handle the fracture limits in such way that all algorithms initially described remain valid.

3.1 Mesh Setup

The coarse mesh taken from the user input is our starting point. It is assumed to contain a typical PZ geometrical mesh of 3D connected and conformal elements. On this data we set up some preliminary constructions for further use.

3.1.1 Initializing Coarse and Skeleton Mesh

After the coarse mesh is read, the next step is to create a skeleton mesh in between coarse elements. The *skeleton mesh* is the set of all elements of 1 and 2 dimensions (assuming a tridimensional coarse mesh). We shall also refer to its members as *skeleton elements*, *lower-dimensional elements*, or even *interface elements*, depending on which of their properties is most pertinent to emphasize.

The existence of the skeleton mesh allows to define the intersections from lower to higher dimensional elements. The skeleton mesh allows to uniquely identify the lower-dimensional domains (e.g. elements) occupied by the lower-dimensional sides of the coarse volumes. Dividing these domains one dimension at a time allows to set up simple constructions that aggregate into more complex intersections.

For the creation of the skeleton mesh, following the example distribution on Figure 3.1.1, we superpose a 2D element, called *face*, on all the bidimensional sides of volumes. Similarly, over one-dimensional sides of faces, we superpose a 1D element called *edge* or *rib*. Since our primary goal is to uniquely identify these lower-dimensional subdomains, wherever sides are shared by

[3]If the numerical simulation method accepts polyhedral meshes, the final meshing could even be avoided

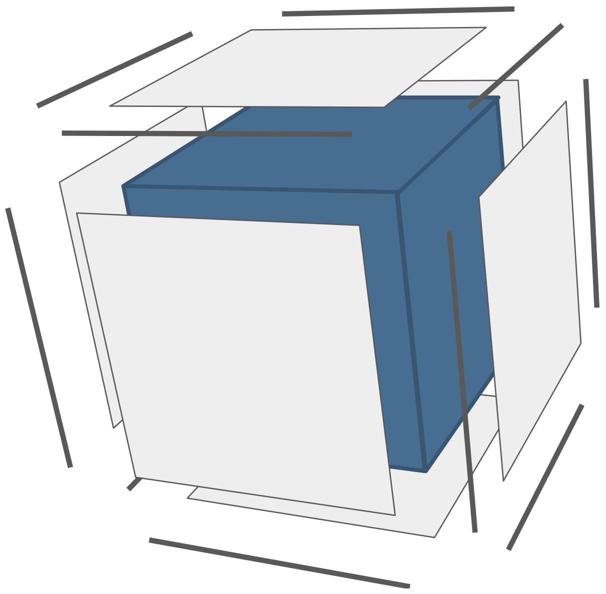


Figure 3.1.1: Skeleton elements around a coarse volume.

more than one neighbour, this superposition is performed only once and no duplicated elements are created at this step. The data-structure of NeoPZ that keeps track of element neighbours facilitates the creation of this mesh.

We include into our framework the notion of more arbitrary polyhedral volumes (Section 3.1.2). Similarly, the mesh representing the surface of the fractures are defined as a set of 2D elements which are part of the skeleton mesh.

Mesh Conformity Assumptions

Aiming to simplify the algorithms, mesh conformity[4] is conserved at all steps of the fracture insertion process. Conformal meshes have the assurance that the common space between neighbouring elements has the known topology of, and completely covers, one of their sides. This is easily seen through the example in Figure 2.3.1, where the shared space between the 2 faces is a single edge.

The main contribution of this setup is that, whenever a side of an element is intersected by a fracture, the same intersection is extended to all elements that share that side. This is the foundation of our choice of building intersections from lower to higher dimensional elements, and can be superficially summarized with the statement: *"All faces neighbours of intersected edges are intersected faces".*

Nonetheless, we choose to maintain conformity only for the skeleton mesh[5], and justify this choice with the claim that it enables us to build a surface mesh for the fracture only with its intersections with skeleton elements.

[4]Mesh conformity is defined in Section 2.3

[5]In the following section (3.1.2), we use the skeleton mesh to define arbitrary polyhedral volumes. As a consequence, this also implies conformity for those volumes.

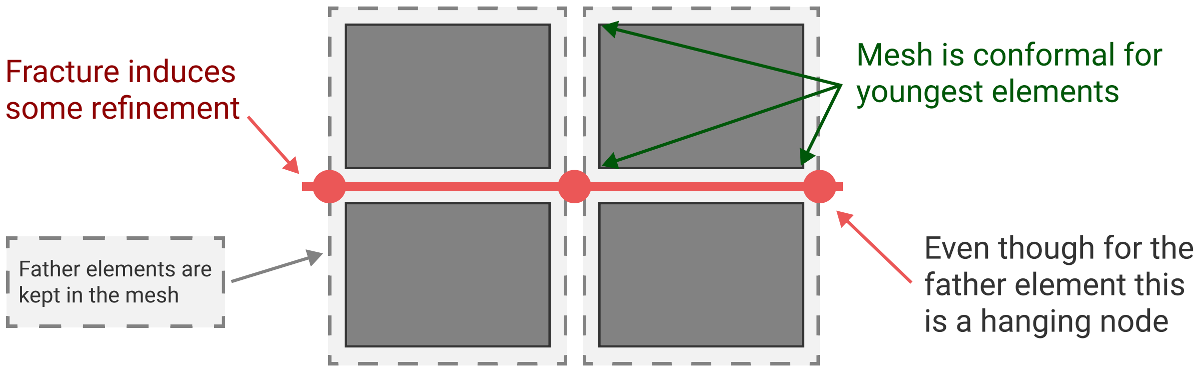

Fracture induces some refinement
Mesh is conformal for youngest elements
Father elements are kept in the mesh
Even though for the father element this is a hanging node

Figure 3.1.2: Illustration of conformity at the highest level of refinement.

It is important to note that the element refinement trees are extensively employed in our algorithms. Refined elements are not deleted from the mesh. We shall often use *father elements* to refer to those that were refined and still exist in the mesh. As existing nodes are never moved, plane elements will remain plane.

Yet, elements are only refined with the direct goal of creating conformity with (or indirectly due to) a new fracture. This promptly implies that there are elements to which father elements are not conforming (Figure 3.1.2). As a result, to take conformity as a premise, another level of detail has to be addressed.

Since father elements are refined to conserve conformity, we define conformity in a sense that applies only to their youngest sub-elements: We adopt the notion of conformity *at the highest level of refinement* (sometimes referred to as the *youngest children level*). In practice, this translates to: Whenever an algorithm requires conformity, we sample only those elements in the mesh which were not refined.

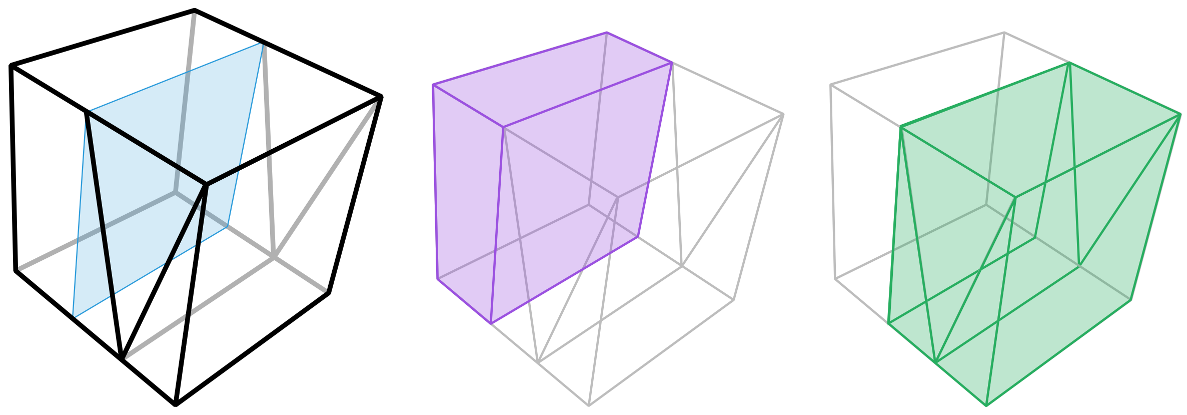
3.1.2 Recovering Polyhedral Volumes

Within our meshes, by polyhedral volume (or region) we mean a simply-connected volumetric region, bounded by oriented 2D elements at the highest level of refinement, with no skeleton element in its interior.

Mesh conformity for the skeleton mesh (conserved at the highest level of refinement) guarantees that every skeleton face element will interface two, and always two, polyhedral volumes. This is indeed what motivates taking into account the orientation of such faces. The unique association of a volume to an orientation of a face provides us with a reference from the pair (face+orientation) to a polyhedron and back. Hence, mentions to *oriented-faces* should be interpreted as a remark on (exclusively) one of the two possible pairings of face and orientation: One that matches the normal direction of the element, and one opposite to that.

The majority of the operations we perform on polyhedral volumes assume that they are convex. Therefore, a volume created after a division that is found to violate convexity will always be meshed[6].

[6]see Section 3.6 for further discussion



(a) An arbitrarily refined cube whose interior was split in 2 by a blue plane

(b) Resulting polyhedral volumes from a refined cube

Figure 3.1.3: Example of the occurrence of arbitrary polyhedral volumes

Given its definition, it should be clear that constructing one of these volumes reduces to gathering the set of oriented 2D elements that bound their interior. We refer to this set as the *shell of the polyhedron*.

This approach observes one justifiable exception: The boundary of the mesh is taken to be a polyhedral region, unique in the way that its interior contains all elements which are not in its shell, and its convexity is of no importance. The exception that defines the boundary polyhedron is, in fact, what motivates its construction; Most of the operations we perform on polyhedral volumes do not apply to the boundary, so we build it with the deliberate intention of skipping it.

The purposes behind the gathering of ployhedral volumes are:

1. We use them to separate fracture surfaces into complementary subsets (Section 3.3);
2. They provide control volumes to isolate and handle fracture limits (Section 3.4);
3. Each of them are simple regions inside which we can generate the sub-meshes;
4. They track the association of sub-mesh to coarse element.

In Figure 3.1.3 we observe the occurrence of arbitrary polyhedral volumes through the refinement of a cube (caused by the introduction of at least two fractures). Figure 3.1.3a shows how the original cubic volume has its interior split into 2 regions by a blue plane, and part of its shell is also refined (although without actually splitting its interior). Figure 3.1.3b highlights the resulting volumes created: The parallelepiped in purple has certainly simple topology; But the one in green, albeit occupying the geometrical space of a parallelepiped, has an arbitrary shell of triangles and quadrilaterals.

The only ways polyhedral volumes are created are through their input in the coarse mesh or by refining the existing volumes. This is, of course, what motivates our third and fourth uses of these structures: they start with a one-to-one correspondence with coarse elements and, as refinement progresses, they carry these references to smaller polyhedra, up until we finally generate the sub-mesh inside each of them.

Before any fracture has been inserted to the mesh, the first construction of polyhedral volumes is trivial. The boundary polyhedron is simply the set of all 2D elements with only one 3D neighbour through its 2D side. All other polyhedral volumes start as volumetric mesh elements, and so their shell is simply collected from the skeleton elements that are neighbour of their 2D sides. The orientation is easily computed through the sign of the determinant of a 2x2 matrix that defines the parametric transformation between elements that share a bidimensional side in NeoPZ.

Once fractures are inserted into the mesh, polyhedral volumes are progressively split by these fractures into subsets of themselves that can be bound by any number of faces.

When a polyhedron is divided, the shell elements that compose the boundary of the newly created polyhedrons will need to substitute their reference from the original polyhedron to the son polyhedrons. The next subsection documents further constructions required to robustly define arbitrary polyhedral regions.

Sorting Faces Around an Edge

The more general assembly of polyhedral volumes is primarily based on the imposition that it should not contain faces in its interior. For this, the first operation we perform is to sort faces around edges. To manage this setup, we introduce these key concepts:

- **Oriented dihedral angle** - (Figure 3.1.4) The right-handed angle between 2 faces that share an edge, according to the positive or negative orientation of the one-dimensional side of the first face. Which is computed using vectors in the plane of each face, and one at the axis shared by both planes according to Equation 3.1.1. Note the use of the 2-argument arc tangent function ($\arctan2(y,x)$), to set the range of $\theta \in [0, 2\pi)$.

$$\cos\theta = \frac{(\mathbf{b_3} \times \mathbf{b_2}) \cdot (\mathbf{b_1} \times \mathbf{b_2})}{|\mathbf{b_3} \times \mathbf{b_2}||\mathbf{b_1} \times \mathbf{b_2}|} \tag{3.1.1a}$$

$$\sin\theta = \frac{\mathbf{b_2} \cdot ((\mathbf{b_3} \times \mathbf{b_2}) \times (\mathbf{b_1} \times \mathbf{b_2}))}{|\mathbf{b_2}||\mathbf{b_3} \times \mathbf{b_2}||\mathbf{b_1} \times \mathbf{b_2}|} \tag{3.1.1b}$$

$$t = \arctan2(\sin\theta, \cos\theta) \tag{3.1.1c}$$

$$\theta = \begin{cases} t, & t \geq 0 \\ t + 2\pi, & t < 0 \end{cases} \tag{3.1.1d}$$

- **Rolodex Card** - (Figure 3.1.5a) An unrefined face (i.e. highest level face) with a one-dimensional side shared by other cards, an angle to a reference-card, and an orientation.

- **Rolodex** - (Figure 3.1.5a) A set of cards all sorted by their angle to a reference, and a one-dimensional element which is neighbour of all cards.

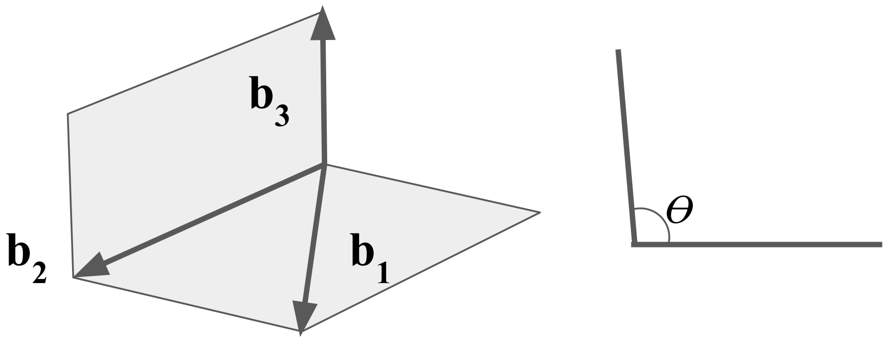
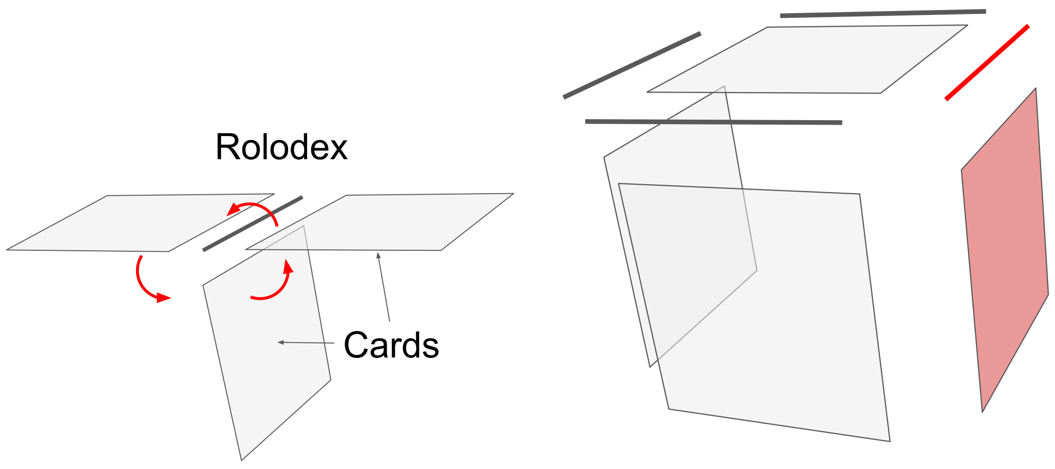


Figure 3.1.4: Dihedral angle θ and vectors used to compute it.



(a) Rolodex and card (b) Recursive assembly of a polyhedral volume

Figure 3.1.5: Using sorted faces to assemble polyhedral volumes

To establish this sorting we first choose a reference, and then take the oriented dihedral angle of each of the faces neighbours to the current edge. The reference face can be chosen by any arbitrary criteria. We seek only the sorted listing of these faces, and the oriented loop they form around the edge is useful regardless of where it starts. The angle of the reference-card to itself is, of course, zero.

These sorting operations need only be performed when a new edge is introduced in the mesh and when the a neighbouring face of an edge is divided. Such events are observed at the refinement of faces (when sub-elements and their 1D skeletons are created in the mesh), and when the surface mesh is created for a fracture. Although the construction or update of a rolodex entails a small sequence of floating point computations, they are restricted to certain triggers and should be infrequently carried out during the creation of fractures. We check for any of these triggers right before we start defining polyhedral volumes.

Updating Polyhedral Volumes

In possession of the sorted distribution of faces around edges, the key principle that guides the assembly of generalized polyhedra follows: Every consecutive pair of Rolodex Cards will share a polyhedron. An assertion directly owed to the definition of a polyhedron not containing faces in its interior.

From this guiding principle, the algorithm moves through an inherently recursive logic. We start in any oriented face which has not been assigned a polyhedron; Access the Rolodex of each of its edges; Check that the orientation match of the Rolodex edge against the face side so we know what direction to move up the card list; Query each Rolodex for the next card (given the proper orientation); and repeat the process on each added card. To ensure the algorithm terminates, we simply skip the next card if it has already been added to the sought for polyhedral shell. The pseudo-code to assemble a single volume is provided in Algorithm 3.1.1.

As hinted before, the use of the rolodex and card structures is also used to check for convexity of the volume. A convex polyhedron will have internal dihedral angles up to, and including, 180^o, so we always verify this as we move from a card to the next in the rolodex by taking the difference of their angles to the reference card.

Algorithm 3.1.1: AssemblePolyhedron

```
Input: An initial oriented face, and a mesh with sorted faces around each edge;
Add initial face to the polyhedron;
edgelist ← list of edges for the face;
newfaces ← new array for oriented faces added to polyhedron;
foreach edge in edgelist do
    rolodex ← sorted faces around edge;
    initialcard ← card corresponding to initial face in this rolodex;
    nextcard ← next card down the rolodex according to orientation;
    if nextcard already in polyhedron then
        continue to next edge;
    else
        newfaces.Append(nextcard);
        internalAngle ← difference of angles to reference between the cards;
        if internalAngle > $180^o$ then
            Tag polyhedron as non-convex;
        end
    end
end
foreach card in newfaces do
    card.AssemblePolyhedron();
end
```

3.2 Conforming the Skeleton Mesh to a Fracture

Observing the directive of conserving the user-informed coarse mesh, we exploit the skeleton mesh defined in Section 3.1 to tackle the creation of a fracture as, primarily, a refinement problem.

This approach is still backed by our claim that the fracture surface can be created from intersections with the skeleton elements. Thus, in this section we describe algorithms to: Find fracture-to-skeletons intersections; Define refinement patterns that include the geometrical domain of these intersections; Impose geometrical tolerances by coalescing intersections that violate a threshold and; Split elements according to the coalesced refinement patterns, to create the geometrical definition of intersections as elements or nodes in the mesh.

The refinements performed have the goal of conforming the skeleton elements to the surface mesh that is subsequently generated for the fracture. In a sense, this step can be seen as the specification of a path through which a (convex) fracture polygon passes.

The intersection is defined from the lowest dimension (point) to the highest dimension. First we classify the nodes as being above or below the plane. Then we take the intersection of an edge with a plane. From this very simple construction we are able to extend intersections to higher dimensions by exploring mesh conformity, neighbourhood information, and convexity of geometrical figures.

As a first step the representation of a fracture polygonal plane is discussed.

3.2.1 Fracture Polygon Setup

A fracture is defined as a convex planar polygon in $\mathbb{R}^3$. Aiming for simplicity of input data, the polygon is read by the program as an N×3 matrix in which every line is a corner point of the polygon, numbered counter-clockwise from zero to N-1.

In order to compute intersections of a fracture plane with the mesh, it is useful to determine its orientation first.

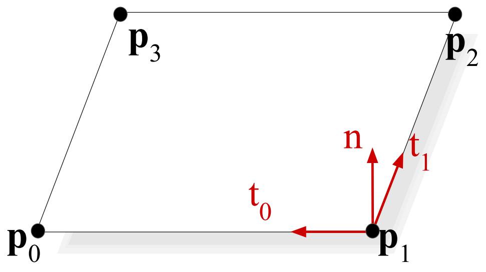


Figure 3.2.1: Plane points with its respective axis.

For example, Figure 3.2.1 shows a plane formed by four points, and those points are the base to compute the vectors that define its orientation.

$$\mathbf{t_0} = \frac{\mathbf{p_0} - \mathbf{p_1}}{\|\mathbf{p_0} - \mathbf{p_1}\|} \tag{3.2.1}$$

$$\mathbf{t_1} = \frac{\mathbf{p_2} - \mathbf{p_1}}{\|\mathbf{p_2} - \mathbf{p_1}\|} \tag{3.2.2}$$

$$\mathbf{n} = \mathbf{t_1} \times \mathbf{t_0} \tag{3.2.3}$$

where $\|\bullet\|$ denotes the Euclidean vector norm.

It should be noted that, by choosing point $\mathbf{p_1}$ as a reference to the plane, we give this definition a general aspect, since any polygon will necessarily have points $\mathbf{p_0}, \mathbf{p_1}$ and $\mathbf{p_2}$ in an Euclidean space, if their corner nodes are numbered starting from zero.

Checking Data Consistency

Given the coordinates of the corner points, a consistency is put in place: All points must be coplanar.

The coplanarity of corner points is a basic hypothesis for most of the algorithms in this work. This condition is inevitably met for triangular planes. However, other polygons require further evaluation. As an example, for a quadrilateral assumed to be perfectly planar, constructing a vector from point $\mathbf{p}_1$ to $\mathbf{p}_3$ should result in finding a third vector that is tangent to the plane and, therefore, orthogonal to its normal vector $\mathbf{n}$. Ensuring that:

$$(\mathbf{p_3} - \mathbf{p_1}) \cdot \mathbf{n} = 0 \tag{3.2.4}$$

or using a small number ϵ as tolerance

$$|(\mathbf{p_3} - \mathbf{p_1}) \cdot \mathbf{n}| < \epsilon \tag{3.2.5}$$

is condition enough to state that a set of four points numbered counter-clockwise from zero to three are coplanar (where $|\bullet|$ denotes the absolute value).

If considering usage of polygons of a bigger number of vertices to define a fracture plane, this operation should be performed for each vertex, other than those used to define the orientation of the plane.

$$|(\mathbf{p_j} - \mathbf{p_1}) \cdot \mathbf{n}| < \epsilon \qquad \forall j \in \{2, ..., k\} \tag{3.2.6}$$

ensures a set of k coplanar points to a tolerance ϵ.

3.2.2 Step 0: Node Binary Classification

At the core of our intersection search is a classification of nodes with respect to each fracture plane. Once the vectors that describe the plane orientation have been determined, we can check where, to which of its sides, lie all nodes $\mathbf{p}$ of the mesh. Just construct a vector from the plane to the node and the scalar product of this vector with the normal vector of the plane should fall in one of two cases:

$$(\mathbf{p} - \mathbf{p_1}) \cdot \mathbf{n} > 0 \qquad (\textit{point above plane}) \tag{3.2.7a}$$

$$(\mathbf{p} - \mathbf{p_1}) \cdot \mathbf{n} \leqslant 0 \qquad (\textit{point below plane}) \tag{3.2.7b}$$

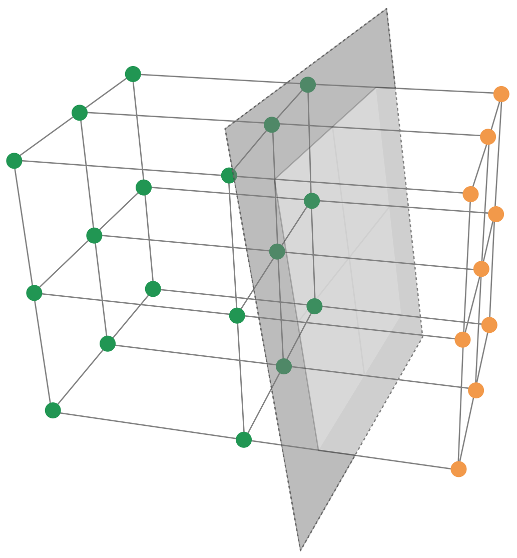


Figure 3.2.2: Example of binary classification of nodes according to a plane.

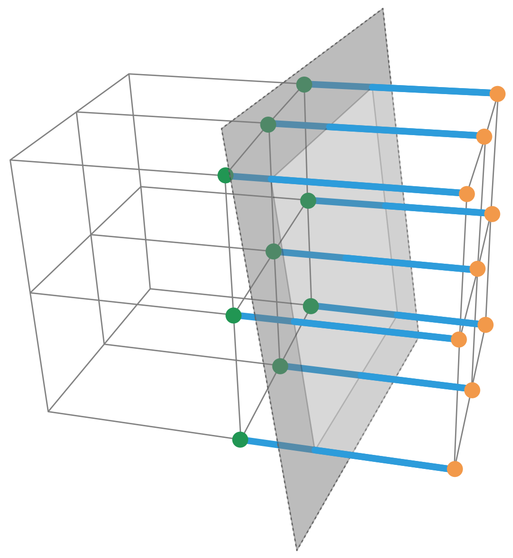


Figure 3.2.3: Intersected edges have nodes on opposite sides of the fracture plane.

Building an array (with length equal to the current number of nodes in the mesh) with the result of this operation, gives us a binary classification of all nodes with respect to the current fracture plane (Figure 3.2.2). Such classification is obviously O(n) on the number of mesh nodes, and significantly narrows the number of edges we verify for intersection with the fracture polygon.

3.2.3 Step 1 : Edge-to-Fracture Intersection

Once all nodes have been partitioned as either being on the positive, or the negative, side of the plane, finding intersected edges is trivial: we simply get all one-dimensional elements with nodes on opposite sides of the plane.

For an intersected edge, we can compute the coordinates of the intersection point ($\mathbf{p_{int}}$) on the fracture plane. Consider a segment connecting between the two nodes, $\mathbf{p_A}$ and $\mathbf{p_B}$ (as shown in Figure 3.2.4). The position vector for $\mathbf{p_{int}}$ can be written as (Equation 3.2.8):

$$\mathbf{p_{int}} = \mathbf{p_B} + t(\mathbf{p_A} - \mathbf{p_B}) \tag{3.2.8}$$

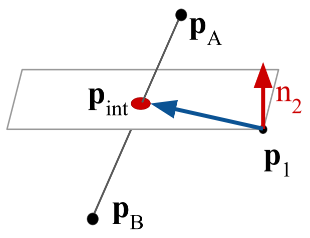


Figure 3.2.4: Two points on opposite sides of the plane, and the intersection point $\mathbf{p}_{int}$ of the segment that connects them.

A vector from the intersection to any point in plane is coplanar to the fracture, and, consequently, orthogonal to the normal vector. Taking the reference corner point $\mathbf{p_1}$, we write from the definition of orthogonality:

$$(\mathbf{p_{int}} - \mathbf{p_1}) \cdot \mathbf{n} = 0 \tag{3.2.9}$$

which provides an equation that can be solved for t

$$(\mathbf{p_B} + t(\mathbf{p_A} - \mathbf{p_B}) - \mathbf{p_1}) \cdot \mathbf{n} = 0 \tag{3.2.10}$$

$$t = \frac{(\mathbf{p_1} - \mathbf{p_B}) \cdot \mathbf{n}}{(\mathbf{p_A} - \mathbf{p_B}) \cdot \mathbf{n}} \tag{3.2.11}$$

allowing for computing the intersection coordinates from equation (3.2.8).

3.2.4 Check Whether the Intersection Point is in the Interior of the Fracture Polygon

The intersection coordinates obtained in Section 3.2.3, paired with the separation of nodes of a rib element being on opposite sides of the plane, are only indicative of an intersection with the unbounded plane that contains the fracture polygon. The ultimate decision on wether a rib is cut through by our current fracture is conditioned to that point being in the interior of the polygon that initializes that fracture.

The **Point-in-Polygon** test is a common problem, specially within the *Computer Graphics* community, and has multiple solutions going as far back as Sutherland, Sproull, and Schumacker (1974).

The more numerically stable solutions for this problem are limited to two-dimensional space where no truncation error can invalidate the assumption that all points considered are in the same plane. In consideration, we can project our 3D points into 2D space with no significant com-

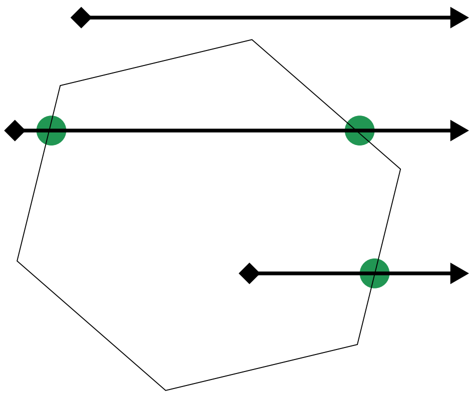


Figure 3.2.5: Checking for point inside a projected polygon through ray intersection.

putational cost: Following a 2D projection suggested by Snyder and Barr (1987) *apud* Badouel (1995), since we already know the normal vector of the fracture polygon, we simply ignore the maximum absolute component of the normal direction for a costless 3D-to-2D projection of all pertinent points. The choice of the maximum absolute component guarantees us that the polygon will not be projected into null area.

With a 2D polygon our test for an interior point is based on the Jordan Curve Theorem[7]: From the testing point ($\mathbf{p_1}$) construct a ray (a line with a starting point and no end) going on the positive $\mathbf{x}$ direction; Test all edges ($\overline{\mathbf{p_A p_B}}$) of the polygon for an intersection point ($\mathbf{p_{int}}$) with the ray; If the number of intersections with the ray is *odd*, the point is in the interior of the polygon; If *even* the point lies in the exterior. This strategy is intuitively seen on Figure 3.2.5.

The intersection of a ray and an polygon-edge is the 2D version of the problem solved in Subsection 3.2.3. It is made even more efficient by having rays parallel to the $\mathbf{x}$ direction, which implies the $\mathbf{y}$ direction as the normal vector.

As a final remark, we note that the usual adoption of this algorithm comes with a warning of failure for points lying perfectly on a vertex of the polygon, and (since we have limited ourselves to only one ray going horizontally) that it will also fail for points perfectly on top of horizontal edges. Such instabilities, however, do not affect our methods due to the *snap operations* further discussed on Section 3.2.6. If points on these unstable domains are wrongly considered on the exterior of the polygon, the tested rib is still *"snapped into being intersected"*. Hence, as far as our methodology is concerned, the final product of this algorithm is unconditionally stable.

3.2.5 Step 2 : Face-to-Fracture Intersection

Having the neighbourhood information and a set of intersected edges, finding intersected faces is straightforward: We simply take those which are neighbours of intersected edges. Figure 3.2.6 shows the intersected faces in continuation to the example initiated in Figures 3.2.2 and 3.2.3.

In accordance to our mesh conformity directives (see Section 3.1.1), it is clear that, if an edge is refined, all neighbour faces should observe a matching refinement as well. Nonetheless,

[7] Jordan Curve Theorem states: A closed, non-self-intersecting, curve separates the plane ($\mathbb{R}^2$) into 2 connected components: The bounded interior and the unbounded exterior.

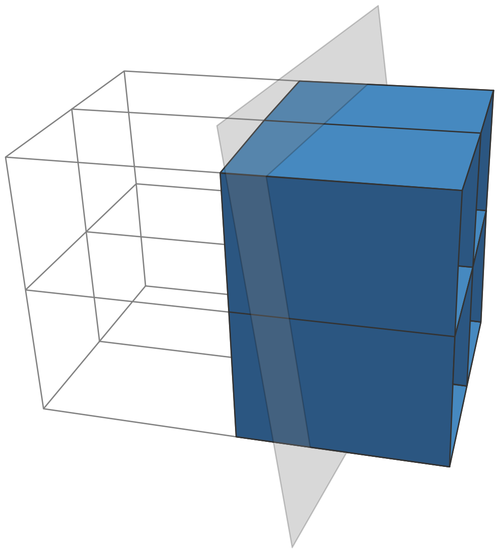


Figure 3.2.6: Intersected faces.

we can push some considerations further to enjoy a stronger claim: *The intersection of a face is completely determined by its intersected edges.*

First, the binary classification of nodes (Section 3.2.2) allows us to unambiguously define intersection points to have a one-to-one correspondence with an intersected rib. Even when a fracture plane passes arbitrarily close to a node (which can be shared with multiple other edges), that node is exclusively on either one of its sides.

Second, we can safely assume every 2D skeleton to be a convex element. They are all triangles and quadrilaterals which we read from the user and/or create with convexity as a given property. Pairing this assumption with the binary node placement, we can state that **faces intersected by an unbounded plane will have two (and only two) intersected edges**[8].

Finally, by direct application of the definition of convexity (2.5.1) and Theorem (2.5.3), we know the intersection of a face with the (also convex) fracture polygon to simply be the continuous line segment connecting the intersection points of its edges.

This construction outlines how we define an *intersected-face* and is perhaps the best example of how we use convexity to build our meshes from lower to higher dimensions. We expand on it by setting up its data representation.

Identifying Split Patterns

To conform the *intersected faces* to a fracture, we should be able to discretely represent the intersection line decide how to consistently refine the face.

Assume intersection points at edges were far apart enough not to trigger a coalescing of those points into each other. Then, over the segment that connects these points, we create a one-

[8] We later weaken this notion to include faces with only one intersected edge by the extension of a fracture polygon. That is, however, an utility fabricated to recover fracture boundaries (as discussed in Section 3.4), and does not affect the validity of the constructions of this subsection.

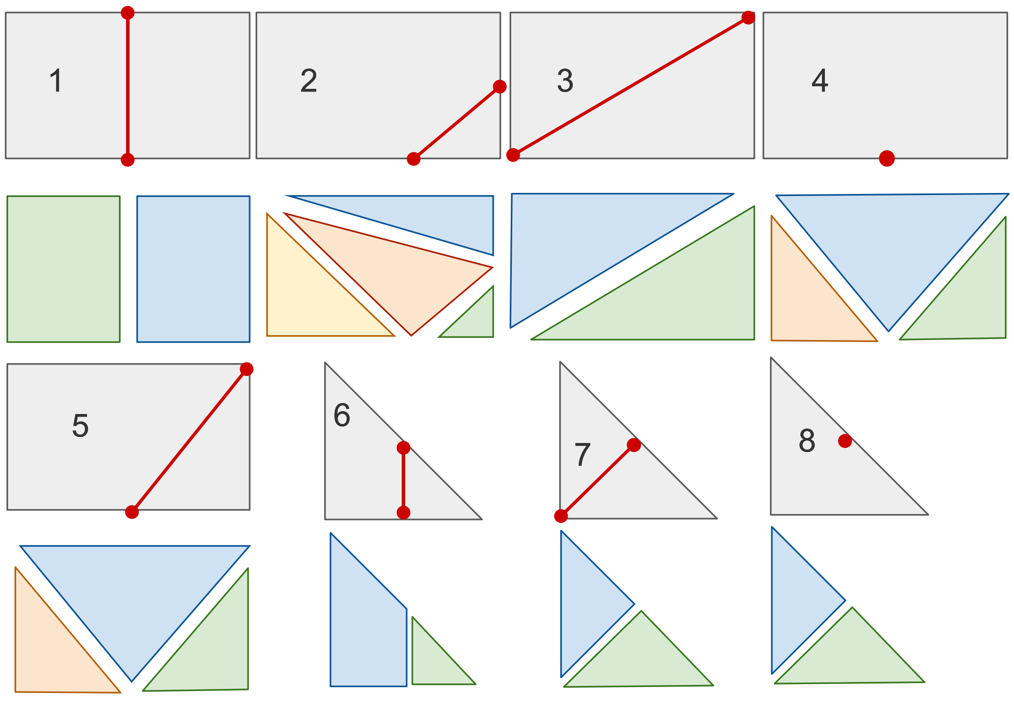


Figure 3.2.7: Identity permutations of all Split Patterns considered.

dimensional element, and refer to it as *interior-edge*. Naturally, if these points were coalesced into the same node, then, this edge does not exist.

At the topology level, two-dimensional mesh elements may be split in a finite number of manners, and many of those are permutations of one another, which further narrows the number of split cases to be considered.

This introduces the notion of a *Split Pattern* (or *Split Case*). These are similar to the *Refinement Patterns*, defined in the NeoPZ environment (see Section 2.4.3), in the sense that their core information is how children elements are connected to a father element. They differ, however, in that *Split Patterns* are more general (within our set of possible intersections) in their goal to contemplate a topological description of a refinement and its possible permutations. *Split patterns*, thus, carry much less information, that we can efficiently manipulate, up until the event of an element actually being refined, when we ultimately use them to create a *Refinement Pattern*.

Figure 3.2.7 illustrates all *Split Patterns* that can arise from the intersection of a convex fracture polygon and 2D mesh elements. Any possible intersection within our framework is a permutation of one of these.

Cases 1, 2, and 6 are the most simple and clearly match our description of face refinement. Cases 3, 5, and 7 correspond to cases where one or more intersection point coalescing to existing nodes. These cases are discussed in Section 3.2.6. Cases 4 and 8 have only a single intersected rib. These latter cases are included as a deliberate exception we later use as a tool to recover fracture boundaries and keep conformity. Detailed description of how to use them are postponed to Section 3.4.

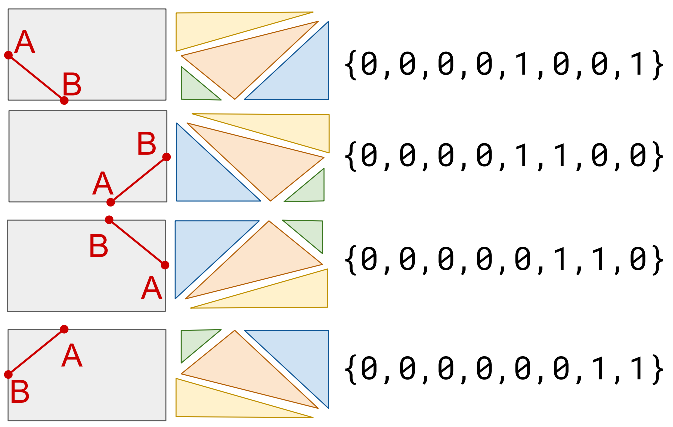


Figure 3.2.8: Examples of permutations and corresponding status vector.

The Split Patterns of the faces determined by the intersected ribs naturally lead to a globally conforming mesh.

We propose identification of each split case through a boolean vector that describes where the cut happens (true for side cut, false for side uncut) which is here on defined as the *status vector*. The information of how (and if) a face is cut comes from its ribs, so when iterating over those to check for a cut, a status vector is gradually constructed according to how each rib was divided. In this way, the *status vector* is the condensed data representation made possible by our initial observation that *The intersection of a face is completely determined by its intersected edges*.

For a quadrilateral element, this vector should have 8 elements: 4 for each corner node, and 4 for each rib. Likewise, triangles will have status vectors of size 6. Such distributions are chosen, and ordered, so that they directly match the side enumeration for these topologies within the NeoPZ environment (which we have thoroughly detailed in Section 2.4.1). As an example, Figure 3.2.8 has all permutations of split pattern 2.

Once such vector has been given, an algorithm was written identify which refinement pattern should be used to divide the element.

Refining Intersected Faces

The first step when refining a face is identifying which split case is appropriate. After finding the case, we simply inform the face which refinement pattern it should follow. The face is divided and its children are stored in the geometric mesh data structure.

An algorithm organizes a list of children based on the status vector. Considering that NeoPZ's geometric elements are constructed from node indices, a child is defined from the vector of the local node indices of its corners.

For example, Algorithm 3.2.1 contains an excerpt (useful for split case 2) of the code that generates such list of children. Figure 3.2.8 gives the reader a graphical reference of the logic of the algorithm. The goal is to find a pair of consecutive cut ribs (respectively `ribA` and `ribB`) and then use those as a reference to define all children.

Algorithm 3.2.1: Refine face (excerpt for split case 2)

```
Input: A status vector (status) and a face object (face);
SplitCase ← DetermineSplitCase(status);
i ← 0;
switch SplitCase do
    ⋮ ;                                        // ('%' is the modulo operator)
    case 2 do
        while (status[i+4] = 0 or status[(i+3)%4+4] = 0) do
            i ← i+1;
        end
        ribA  ← face.Rib[(i+3)%4];
        nodeA ← intersection node at RibA;
        ribB  ← face.Rib[i];
        nodeB ← intersection node at RibB;

        children[1] ← {nodeA, nodeB, i};
        children[2] ← {nodeB, (i+1)%4,(i+2)%4};
        children[3] ← {nodeA, nodeB, (i+2)%4};
        children[4] ← {nodeA, (i+2)%4,(i+3)%4};
    end
    ⋮
end
return children
```

It is importante to note that, as demonstrated in Algorithm 3.2.1, children connectivity are completely determined from logical operations, without requiring any floating point operations. This is made possible by the construction of *Split Patterns* paired with the explicit topologies of the NeoPZ environment. The refinement of the skeleton mesh is a reasonably efficient step in our methodology.

In continuity to the example of intersected faces identified in Figure 3.2.6, Figure 3.2.9 shows their resulting sub-elements whose boundary conforms to the fracture surface.

3.2.6 Geometrical Tolerance Imposition

Writing geometrical algorithms using floating-point operations is a major challenges of problems involving computational geometry: The handling of close/very close cases often leads to complicated decision algorithms which hurt the robustness of the code. It is necessary to associate with these problems projection of points and/or lines with the purpose of eliminating ambiguity to the domain definition of all geometrical figures involved.

In Section 3.2.2 we took a first step into handling geometric ambiguity by classifying mesh nodes with respect to a plane as being either above or below the plane. A second projection is define which guarantees that two points are never created within a tolerable distance ϵ, related to the smallest mesh size chosen by the user.

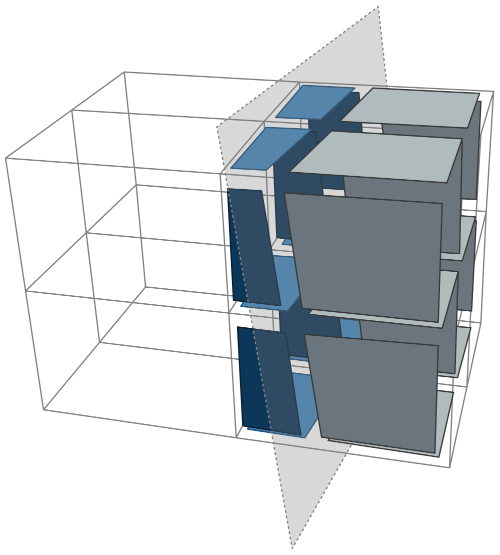


Figure 3.2.9: Refined intersected faces conforming to fracture plane.

Another important aspect in generating a good quality geometric representation of a DFN is to avoid generating lines which are too close to existing planes/lines and/or who form too small an angle. We can take these directives as the more practical interpretation of imposing the quality metrics given in Section 2.3.1.

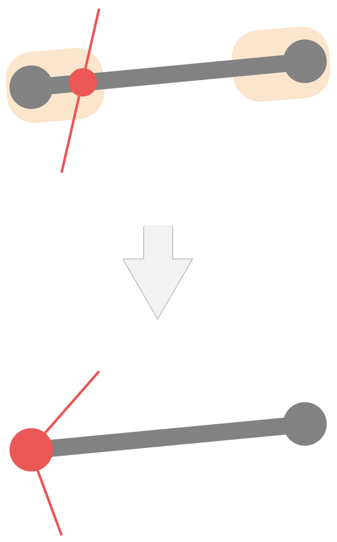
There are many algorithms geared towards localizing coordinates within a point cloud (with reasonable computational cost), which are largely employed in the literature of computational geometry to verify features like these. The most known methods are based on the partitioning of the points in *octrees*. Optimized algorithms, such as those developed by Eppstein, Goodrich, and Sun (2005), can build octrees in $\mathcal{O}(n \log n)$ and locate a close point in approximately $\mathcal{O}(\log n)$ time, depending on the depth of the octree.

In this work, such problems is tackled within the domains of intersected elements. Implicitly, this choice carries the assumption that the user-defined coarse-mesh satisfies the quality measures imposed. Our point projection algorithm does not optimize elements for mesh quality, they just try to conserve the features initially set by the user with a user-defined tolerance. This can be interpreted as a deliberate trade-off: we forgo actual mesh optimization algorithms that would deliver optimal meshes (maximizing quality metrics), but gain conservation of the user-defined coarse-mesh, and robustness.

Point Snapping Operations

Following the guideline of generating meshes from lower-to-higher dimensions, at this level, we impose geometrical tolerances on the *Split Patterns* created for the skeleton mesh. In this approach small distances and sharp angles are rejected:

1. **Points closer than a tolerance to existing points are rejected/snapped**: We have already established that the only points that can be moved are the newly created intersection points, all other points are fixed (they were created either by a previous fracture or


(a) Tolerable distance is checked within the domain of intersected ribs

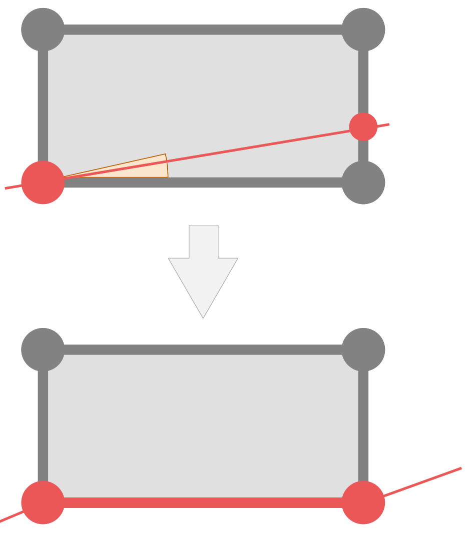

(b) Tolerable angle is checked for internal angles of sub-elements of the intersected face

Figure 3.2.10: Illustration of geometrical tolerances.

as part the original coarse mesh). Intersection points are always created in the alignment with intersected edges. Hence, this tolerance check is simply a matter of measuring the distance of the intersection point to both nodes of a rib (Figure 3.2.10a).

If any distance is smaller than the tolerance, we force it to coalescing (or *snap*) the point to the closest of the two nodes.

2. Secondly **angle below a tolerance** are verified (e.g. Figure 3.2.10b). Within the domain of the skeleton mesh, the angles an intersecting fracture creates (as it refines necessary elements) are the internal angles of the sub-elements. Therefore, all these angles should be checked. To impose this tolerance, we note that all sub-elements have at least one corner defined by one of the intersection points the fracture plane has created at a rib of its father. This is easily verified in Figure 3.2.7.

   If any internal angle violates the tolerable angle, we again impose it by coalescing the intersection point to a corner of the sub-element.

It should be emphasized that both snap operations are actually the same. What differs between them is the criterion which triggers each one.

A snapped intersection is still associated with a ribs. The intersection point still *belongs* to that rib, even though its coordinates have been modified to match one of the two nodes of the rib.

With regard to data representation, the snap is represented within a rib as an integer that marks the side of the rib that currently contains the point. To represent a snap from the perspective of a face, we modify its *Status Vector*, which has exclusively been defined to store the sides containing the two intersection points.

The snap operations have a tendency to propagate. Recall that the goal of these refinements, which we are creating and improving upon, are to create a conformal path for the fracture surface. Naturally, since our snap operations happen at the domain of ribs, and since ribs are the interface between faces, this approach implies that neighbouring faces that share that rib need to update their split pattern accordingly.

A tolerance violated within a face induces a modification of refinement patterns on all neighbour faces through the snapped edge-side, even if all their features had been verified *legal* beforehand.

This is an important second order effect, and attaches to these algorithms a level of recursion. To guarantee that they terminate, however, is easy enough. There is only one projection operation performed: it moves a point from a 1D domain (the intersected rib) to a 0D domain (the closest rib node). When an intersection point has been projected to a lower-dimensional domain, it cannot be moved again. The algorithm has to terminate as we have a finite number of elements.

Finally, to close this discussion, let us remark on the usefulness of this approach of defining, taking and imposing geometrical tolerances:

The way snap operations are here presented are primarily motivated by removing geometrical ambiguities through simple decision algorithms. We are still able to centralize some decision power to the user of the program through the definition of smallest distances and angles. The method conserves the *user-defined* coarse-mesh while still rejecting undesirable geometrical features to *user-defined* tolerances. In this sense, other than being an instrument to robustness, these operations are the main way through which we can retain configurability to better adjust a larger group of finite element discretization problems. They work as a trade off between geometrical fidelity and user defined tolerances, in exchange for more aggressive feature rejection.

3.3 Step 3: Meshing a Fracture Surface

The refinement of lower-dimensional elements conforming to the desired position of the fracture defines a fracture line used to create the surface mesh. The core idea is to partition the surface intersection into the simple subdomain intersections, and resort to Gmsh meshing only for those subdomains whose *tesselation* is not trivial.

A former version of our surface meshing process, in the early stages of this research, pursued the construction of a frame of 1D elements, defined by the *interior edges* of intersected faces, and bounded by the limits of the fracture polygonal definition (also 1D elements). Such frame outlines the complete intersection of a fracture with the skeleton mesh and forms the constraints for a constrained Delaunay triangulation. This approach proved itself revealed itself as limited for two reasons: First, it depended on costly geometrical searches to match the fracture original limit definition to their resulting snapped position. An operation with an excessive amount of special cases to our every attempt at generalizing it. Second, it required Gmsh to optimize a progressively more complex setup, and reached extreme cases where it pushed the meshing

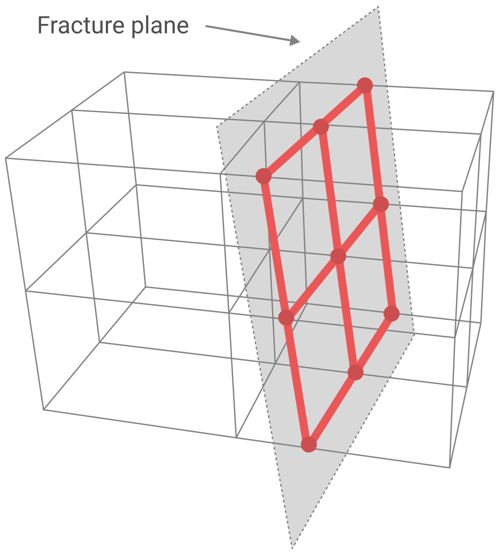


Figure 3.3.1: Subsets of the fracture surface isolated within polyhedral volumes.

algorithms beyond their numerical stability limits, specially with the growing number of fracture-fracture intersections.

We therefor opt to break down the process of meshing a fracture surface into a sequence of simpler operations. The first observation that motivates our method is the fact that, as a fracture enters the mesh, it naturally gets partitioned into smaller subsets of itself (Figure 3.3.1).

We refer to these regions as **sub-polygons**, and define them to be the non-overlapping subsets of the fracture surface which are contained and isolated inside each unrefined polyhedral volume intersected by the fracture.

These sub-polygons are an appealing approach, as it allows to pursue the problem through a divide-and-conquer strategy using much simpler subdomains that can be tackled one at a time to solve our problem. Other convenient properties are revealed to justify their use:

1. These subsets are non-overlapping and their union completely covers the surface. This follows immediately from the fact that they are the cross-section of non-overlapping and continuously adjacent polyhedral volumes. So meshing each of them separately will produce a mesh that completely covers our fracture surface;

2. Each of these subsets are convex polygons[9]. This property is taken to follow directly from Lemma 2.5.4. Convex polygons are much simpler domains to generate a mesh to cover. Indeed, it is one of the basic properties of Delaunay triangulations that they will be bounded by the convex hull of the triangulated point set;

3. Frequently, these subsets are simple enough that their “mesh” will contain only one simple element (a quadrilateral or a triangle). A state so simple that we can solve without any aid from advanced meshing tools like Gmsh;

4. Every one of these sub-polygons, by definition, have a one-to-one correspondence to well defined polyhedral volumes, and we can use this correspondence to gather each of them separately.

By tackling the surface meshing problem one subdomain at a time, we then need to check that neighbouring subsets will not interfere and introduce inconsistencies on each other. Simply put, we need to take special care that these surface meshes do not produce non-conforming meshes. This inconsistency is avoided through the requirement that, whatever is the mesh we produce for a sub-polygon, it will not introduce any new nodes on its interfaces with other sub-polygons (i.e. no edge gets refined during a sub-polygon tesselation).

A pseudo-code summarizing the surface meshing process (leveraging one sub-polygon at a time) is documented in Algorithm 3.3.1. From this procedure, we single out the functions `BuildSubPolygon()` and `TryFaceIncorporation()`, for more discussions in the immediately following subsections.

Algorithm 3.3.1: MeshFractureSurface

```
foreach intersected-volume do
    initialFace ← Pick any intersected face in the volume shell;
    entry-side ← Pick any of the 2 edge sides intersected at the initialFace;
    sub-polygon ← BuildSubPolygon(initialFace, entry-side, volume-index);
    if TryFaceIncorporation(sub-polygon) then
        // TryFaceIncorporation returns true if no new mesh is needed for
           sub-polygon
        continue to next intersected-volume;
    end
    MeshSubPolygon(sub-polygon);
end
```

3.3.1 Finding Sub-polygons

The procedure to build these sub-polygons is based on their one-to-one correspondence with intersected polyhedral volumes. Other properties revealed to be necessary to consistently define a meshable domain. We desire to know how the edges that bound each sub-polygon are connected and to instill each of them with an orientation. The approach is thus, not only to simply list the edges of the sub-polygon, but also to set them up as an oriented closed loop of one-dimensional elements (Figure 3.3.2). Such a set up is a requirement for defining surfaces with Gmsh's API. With this guideline, we exploit the fact that each intersected face has 2 intersected edges, and tag each of them as an entry- and an exit-side. Then on, it becomes straighforward to build the sequence of the sub-polygon in a circular manner that completes the desired edge loop.

Figure 3.3.2 serves as a graphical reference of the logical routine that follows.

[9]As is discussed in subsection 3.3.2, snap operations arbitrarily deform the fracture surface, and 2D convexity only truly makes sense in a manifold. However, here we can talk about convexity in the sense of an "orthogonal projection of the sub-polygon to the plane that best fits its nodes" without hurting the relevance of this argument.

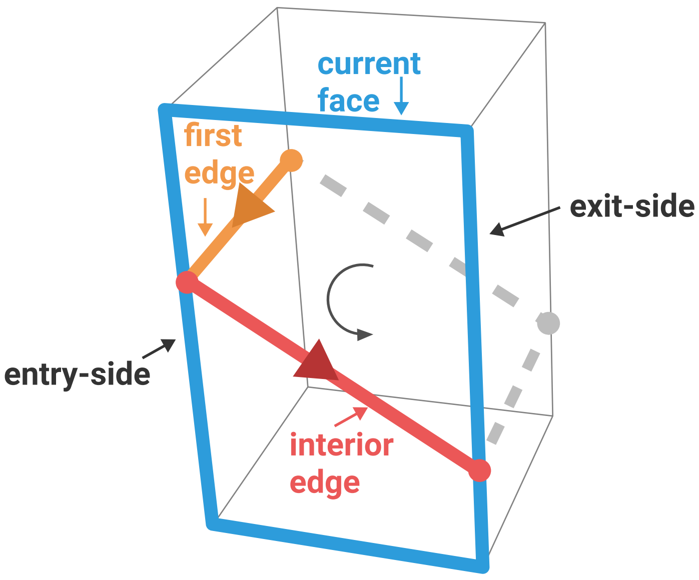


Figure 3.3.2: Features for the assembly of a sub-polygon.

Starting from any intersected face within the shell of an intersected polyhedral volume; Get the one-dimensional element created in the interior of this face (the one which delimits the domain of the fracture-face intersection); Define one of its 2 intersected one-dimensional sides to be the entry-side, which consequently classifies the other to be the exit-side; The next intersected face is uniquely defined by being neighbour of the current face through the aforementioned exit-side and because they both bound the same polyhedral volume; Move to the next face and repeat the process, only now the previous exit-side becomes the new entry-side; The algorithm terminates when we complete the loop and reach the initial face; For a schematic summary of this procedure, see Algorithm 3.3.2.

Algorithm 3.3.2: BuildSubPolygon

```
Input  : An intersected-face element (face); An intersected 1D-side (entry-side); Index
         of the current volume within which to search (volume-index)
Output: A closed loop of oriented edges (sub-polygon)
interior-edge ← face.GetInteriorEdge();
exit-side ← face.OtherSide(entry-side);
neighbour ← First face neighbour through exit-side;
while neighbour $\notin$ volume-shell do
    neighbour ← Next neighbour;
end
if neighbour = initialFace of the sub-polygon then
    return // Loop is complete
else
    BuildSubPolygon(neighbour, exit-side, volume-index);
end
```

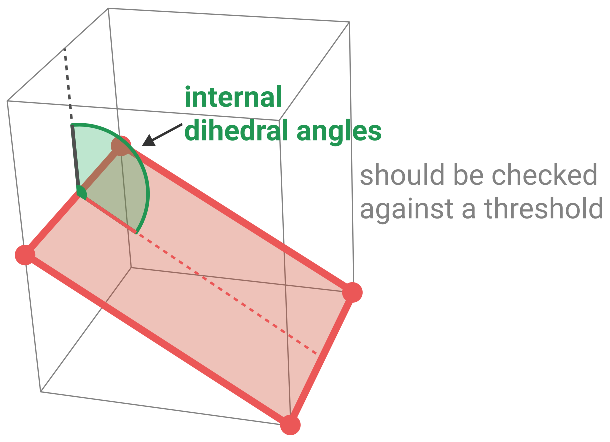


Figure 3.3.3: Internal dihedral angle created by a subset of the fracture surface

3.3.2 Effects of Intersection Coalescing on the Fracture Surface

Arguably, the name sub-polygon can be somewhat misleading. More often than not, intersection nodes are coalesced to existing nodes by the snapping algorithms discussed in Section 3.2.6. These operations, of course, cause the surface of the fracture to deform away from its initial definition, such that some of its subsets similarly tend not to conform to a single plane. The generalized denomination for such geometrical figures are *skew polygons*[10], which somewhat deviates from the implied perception of planarity that accompanies the word *polygon*. Regardless, the algorithm to find the edge loop that defines them has no dependence on planarity. So we keep the name sub-polygon for the intuition it adds to the algorithm that assembles them.

The lack of planarity is not taken into account completely free of computational cost. Although loosely supported, the latest Gmsh release (at the time of this writing) does not robustly delivers surface meshes for non-planar *"polygonal"* surfaces. We can get around this minor limitation by projecting the corners of a non-planar sub-polygon onto its *Best Fit Plane*. The projected sub-polygon is then meshed through a suitable Delaunay based algorithm. Ultimately, as we import elements to our mesh from Gmsh, the projection is reversed by simply using the original node coordinates.

Whenever snap-operations coalesce the intersection of a face down to a node, the intersected-face data structure will, nonetheless, allow us to know which pair of its edges were intersected. Therefore, even completely collapsed sub-polygons are contemplated by this algorithm. However we do need to remove these coalesced edges from the loop of edges that define the sub-polygon. Actually, it is occasionally the case when enough of the edges are coalesced to cause a sub-polygon to collapse into null area; Such a configuration is easily identified, so we skip them and move to find the next sub-polygon.

Two-dimensional meshing algorithms by Gmsh to triangulate a sub-polygon are well suited to reasonably optimize the elements according to 2D quality metrics imposable over the surface.

[10] Actually, an argument can be made that our sub-polygons are related to the concept of *Petrie Polygons*. They are constructed at the surface of a polyhedron and even have a one-to-one correspondence to the facets of volume. They differ in that, within our methodology, triangles are perfectly valid sub-polygons, we have no imposition of regularity and do not admit self-intersecting polyhedral volumes.

However, after these elements are incorporated to our mesh, we must also take care to verify and impose some quality metrics related to their tridimensional disposition, which are beyond the scope of this particular use of Gmsh API. The main metric of our particular interest, as is discussed in sections 2.3.1 and 2.3.2, is the sharpness of element internal angles, so that is the one we choose to verify. Even though sub-polygon triangulations do not generate any 3D elements, they do split polyhedral volumes and, therefore, define their internal angles. For every meshed subset of the fracture surface, we check the internal dihedral angles they define around each edge that bound the sub-polygon (Figure 3.3.3). If any of these internal angles violates a threshold then:

- revert the sub-polygon together with the mesh created for it;
- break the corresponding polyhedral volume down to tetrahedra;
- retry the surface meshing for the now simpler sub-polygons.

Incorporating Existing Mesh Elements to the Fracture Surface

Snapping intersections often cause the fracture surface to overlap with existent 2D elements in the mesh. As a consequence, to avoid duplicating these overlapped elements, a procedure has to be put in place to incorporate them to the surface (as opposed to the operation of creating new ones).

Elements whose nodes are all on the surface are incorporated in the fracture intersection mesh. Yet, checking every node of every 2D element against every node on the fracture surface is sure to lead to an undesirable amount of operations. We resort to the already assembled data for a more reasonable solution: In possession of the edges that define the sub-polygon, one can easily employ element connectivity, and some cleaver navigation of refinement trees, to determine which 2D elements should be incorporated into the surface mesh.

Assuming tridimensional mesh-conformity, the condition for a face to be incorporated is:

- The sub-polygon completely loops around a 2D element. In which case, all edges are neighbours of a common 2D element. This common neighbour should be incorporated.

However, since we are only holding conformity over the skeleton mesh, we need to further extend this construction. The more robust condition, which is general enough to include most mesh configurations of our interest, requires an additional level of nuance:

- Moving down the refinement tree of each edge of the sub-polygon, keep swapping edges for their father element if both nodes of the father are in the original set of nodes of the sub-polygon. The resulting list of edges is guaranteed to still form a loop, which we call the *reduced-sub-polygon*. If these father edges share a common neighbour, the subelements of the father element should be incorporated in the fracture surface.

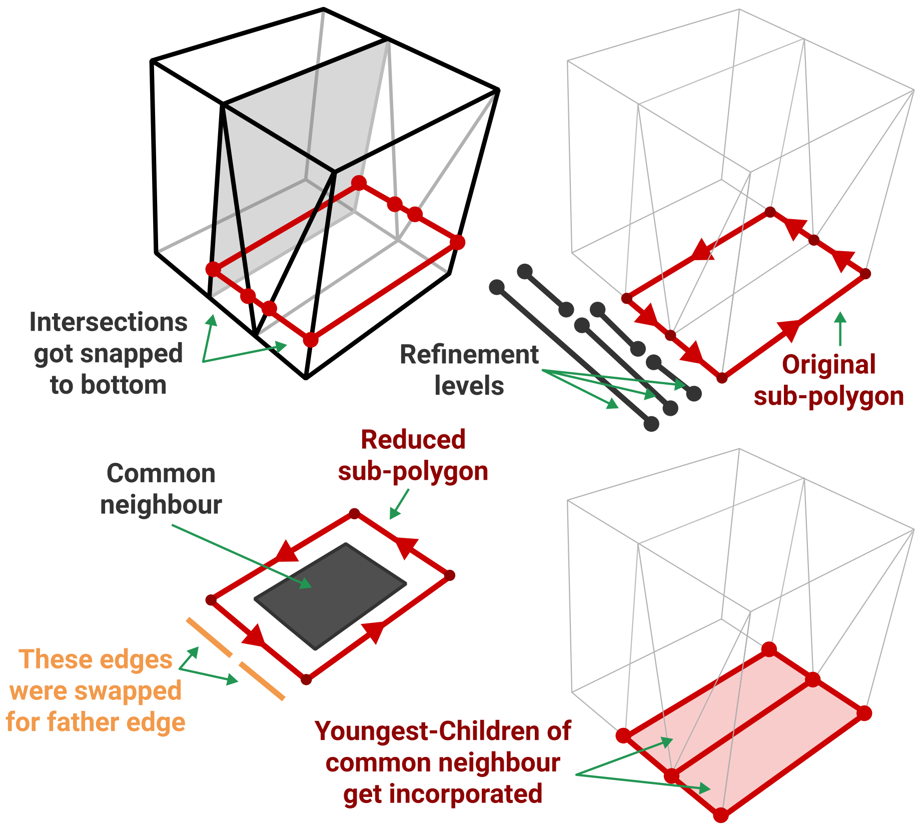


Figure 3.3.4: Steps to incorporate existing elements to fracture surface.

Figure 3.3.4 provides a configuration not contemplated by the first condition but solved by the second. We take the polyhedral volume tagged 'green' in Figure 3.1.3b and analyse the sub-polygon created by a hypothetical fracture passing close enough to the bottom faces to trigger a snap.

One must add a last condition: if a father is incorporated to a fracture surface, its youngest children will in fact be incorporated. The devised logic is described in Algorithm 3.3.3.

Algorithm 3.3.3: TryFaceIncorporation

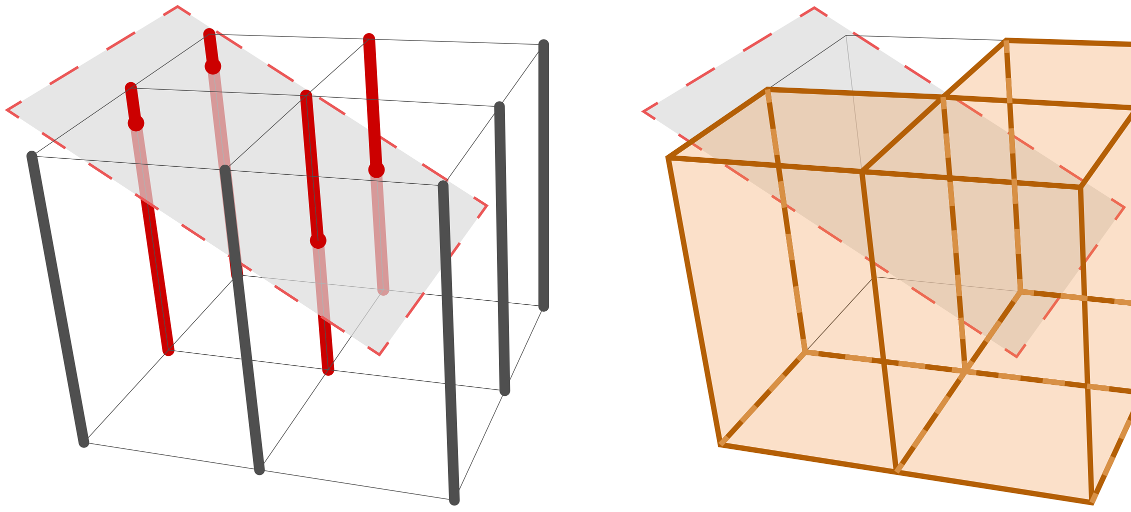
```
Input: A loop of edges (sub-polygon);
polygonNodes ← nodes of sub-polygon;
reduced-sub-polygon ← new empty list of edges;
foreach edge in sub-polygon do
    currentEdge ← edge;
    while currentEdge.HasFather() do
        father ← currentEdge.Father();
        if polygonNodes contains father.Nodes() then
            currentEdge ← father;
        else
            break while loop;
        end
    end
    reduced-sub-polygon.Append(currentEdge);
end
if reduced-sub-polygon.Length() > 4 then
    return false ;              // there is no 2D element with more than 4 edges
end
CommonNeighbour ← FindCommon2DNeighbour(reduced-sub-polygon);
if CommonNeighbour.Exists() then
    FractureSurface.Append(CommonNeighbour.YoungestChildren());
    return true;
end
return false;
```

Finally, we point out that these operations are only dependent on the collection of edges that define the sub-polygon and on element connectivity. The algorithm does not depend on the triangulation for the sub-polygon (nor any other kind of floating point operation) to detect an overlap. Therefore, such a check can be executed with great efficiency. The need for triangulation is dismissed when an element incorporation occurs.

For the rare cases not contemplated by Algorithm 3.3.3, our check for internal angles close to zero is the ultimate step in completing the robustness (with reduced performance).

3.4 Step 4: Fracture Boundary Recovery

Up until this point we have treated each fracture as unbounded. Little reference was made on how the fracture boundaries factor into our methods. Even though it is not hard to recognize that naiveness in their handling can invalidate multiple of our adopted premises.



(a) In-bounds cut-ribs (red) *vs* ribs with nodes on opposing sides of plane, but off-bounds (black).

(b) Volumes that contain some segment of the fracture boundaries.

Figure 3.4.1: Isolating the boundaries of a fracture using polyhedral volumes.

The fracture boundary is represented by dividing the fracture surface representation built in Steps 1 to 3. The strategy is focused on conserving all assumptions previously made. Through the logic that follows, we can state all of the following:

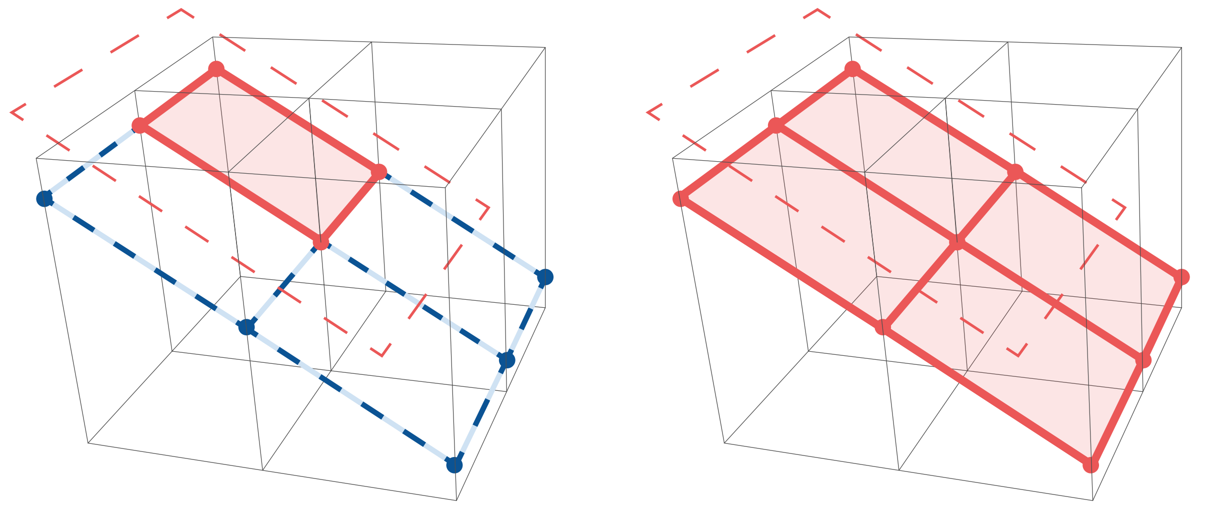
- Coordinates that have not yet been made into mesh nodes are still the *intersection points* which have a one-to-one correspondence with *intersected-ribs*;
- Volumetric regions introduced by the delimiting presence of a fracture surface are still *polyhedral volumes*;
- Mesh conformity is kept for the *skeleton mesh* and naturally extends to conformity of *polyhedral volumes*;
- All possible refinements of 2D *skeleton elements* are contemplated by a permutation of one of the *split patterns* layed out in Figure 3.2.7;
- Subsets of the fracture surface are defined by *interior edges* of *intersected-faces* of *polyhedral volumes*, and always form a closed loop.

Isolate Fracture Boundaries

The first step is to isolate the boundaries within some control region, to which we can restrict any operations we may have to perform; We use polyhedral volumes for this.

Convex faces cut by the limits of convex fracture polygons are trivially found: They are the ones which only have one intersected-rib (Figure 3.4.1a).

This set of faces directly lead to the polyhedral volumes which contain parts of the fracture boundary. Any volume whose shell contains at least one face cut by a boundary of a convex fracture polygon must contain a segment of the fracture boundary. We refer to these polyhedra as *Boundary Control Volumes* (Figure 3.4.1b).



(a) Truncated fracture boundary (red) with possible projection onto bounds of volumes that contain them (blue dashed).

(b) Extended fracture boundary projected onto bounds of volumes that contain them.

Figure 3.4.2: Admissible projections of fracture boundaries within control volumes.

Extend or Restrict Fracture Boundaries

Polyhedral volumes that are completely crossed by the fracture polygon decisively match our introductory description of fracture-to-volume intersections (reported in Section 3.3 to generate meshable *sub-polygons*). Triangulating only those, however, would truncate the surface into a smaller area than what is possibly intended by the input.

We can over-correct this construction, as a second stream of operations, and project the fracture plane to the shell of the *Boundary Control Volumes*. Thus extending the area covered by the fracture surface. Such an increase in area is kept coherent by the limiting presence of our established control volumes. As a graphical reference, we can look to Figure 3.4.2a for an example of an in-bounds sub-polygon region (in red), and the new sub-polygons fabricated (only within the control volumes) through the extension of the fracture (dashed blue lines).

Within the set of edges at the shell of the *Boundary Control Volumes*, search for ribs cut by the *'unbounded'* fracture plane; For those, we skip the Point-in-Polygon verification described in Section 3.2.4 and take all ribs with nodes on opposite sides of the fracture plane. We then need to be able to differentiate between ribs that were in fact intersected by the fracture, and those that were only considered intersected by this projected extension of the fracture polygon. Figure 3.4.1a differentiates ribs that would be found intersected through this construction by the dark gray color (against the conventionally intersected ribs in red).

Once this new set of intersected skeleton elements were found, normal snap algorithms and tolerance impositions apply. Likewise, the added sub-polygons of our, now extended, fracture are meshed to cover its surface (Figure 3.4.2b).

The usefulness of *Split Patterns* 4 and 8 (from Figure 3.2.7) is now apparent: At the limit of control volumes, onto which we project the fracture, there might be nodes created within edges that are shared with neighbours out of these volumes. The same reasoning would apply, even

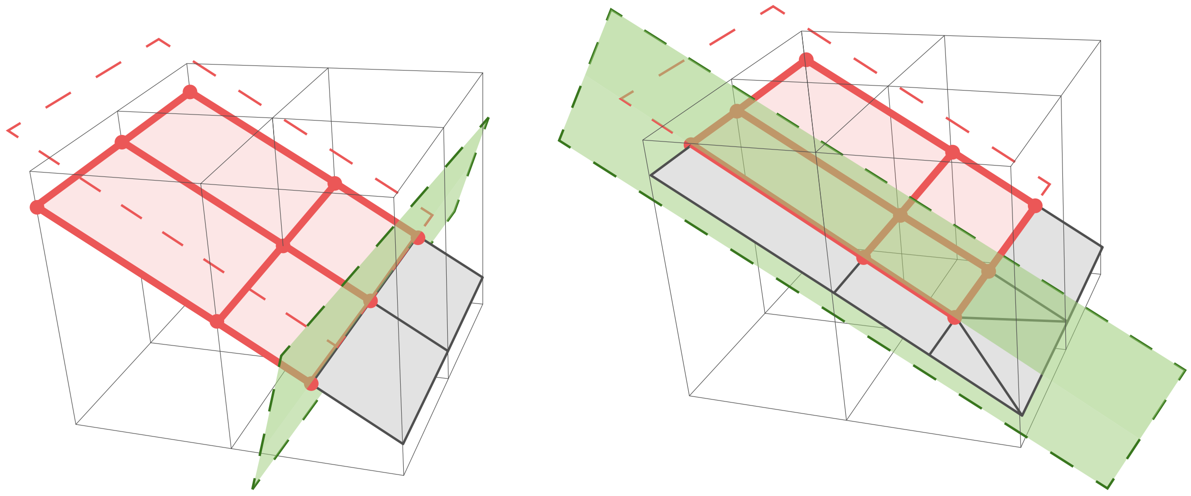

(a) First orthogonal plane refining surface (b) Second orthogonal plane refining surface

Figure 3.4.3: Boundary recovery algorithm through the refinement of extended fracture surfaces using orthogonal planes.

if we were not to perform this projection. To keep our premise of mesh conformity, then, these faces must also be refined. And the pattern they should follow is either one of those two.

Although not discussed in depth in Section 3.2.6, the projection onto the shell of intersected polyhedra is actually our last feature geared to eliminating geometrical ambiguity. Through it we can ensure that all intersection nodes exist only in a domain of an intersected rib, and can enjoy a one-to-one correspondence which is instrumental to our methods.

Recover Fracture Boundaries

In order to create the geometrical representation of the boundary within the mesh, we refine this extended fracture surface and partially retreat from our over-correction. This is the third level of operations performed to handle the fracture limits.

For each line segment that connects two consecutive corners of the fracture polygon, there is an edge we would wish to recreate as a set of geometrical elements. Over each of these domains, we can construct an unbounded plane that is orthogonal to the plane of the fracture. Two examples of these can be consulted on Figures 3.4.3a and 3.4.3b.

Using this orthogonal plane, we can look for intersected skeleton elements within the set of 1-and-2D elements that exist on the fracture surface. This search is pretty much the same we have performed in Section 3.2, differing in the sense that it is restricted to only the elements already grouped as being on the surface. The restriction to the collection of surface elements is indeed the only one we are interested, and in that we do not have to perform the check for *Point-in-Polygon* described in Section 3.2.4.

Moreover, we can employ this orthogonal plane to choose what elements should remain in the fracture surface, and remove those that should not. We can orient the orthogonal plane such that its normal vector points to the interior of the surface. Henceforth, elements that are kept or removed are separated by being, respectively, above or below the orthogonal plane.

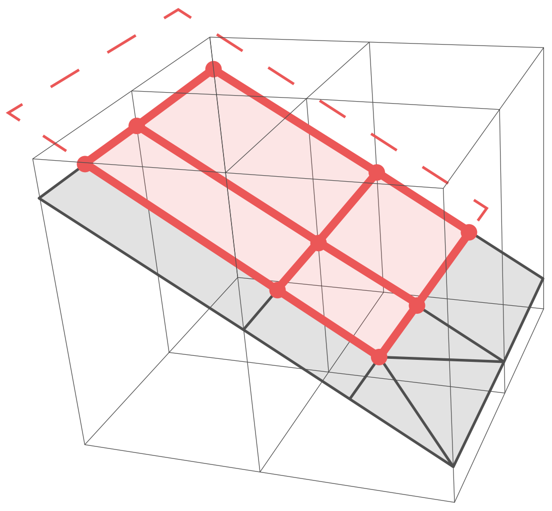


Figure 3.4.4: Recovered fracture boundary.

This logic is sequentially repeated for each of these segments we can define on the boundaries of the fracture polygon. And its progressive execution is exemplified in Figures 3.4.3a and 3.4.3b.

The recovered fracture surface of this example is given in red in Figure 3.4.4 Note that the elements that were discarded from the surface (light gray) are kept in the mesh to conserve conformity.

The one-dimensional geometrical representation of the fracture boundary is a *simple* use of neighbourhood information and mesh conformity. Since we know what 2D elements are part of the fracture surface, the boundary is finally defined as the set of 1D elements with only one 2D neighbour belonging to the surface.

We observe that this algorithm is a direct application of the first of the two equivalent definitions of a convex polygon (stated in Definition 2.5.2).

Fracture Boundary Options

From a review perspective of the 3 levels of operations performed to geometrically represent the fracture boundaries, a convenient observation is in order: This construction in fact gives us 3 valid solutions on how to handle the limits of a fracture.

1. We can follow through the complete logic and attempt to recover a boundary that best approximates the straight lines connecting the user-informed fracture-corners. This will result in what we can denominate as a *"recovered"* fracture boundary (Figure 3.4.2a);

2. We can stop this logic after the projection of the fracture onto the limits of the volumes that contain them, and get an *"extended"* fracture boundary (Figure 3.4.2b);

3. And we can skip the extension and ignore the search for Sub-Polygons within volumes intersected by the fracture limits to get a *"truncated"* fracture boundary (Figure 3.4.4);

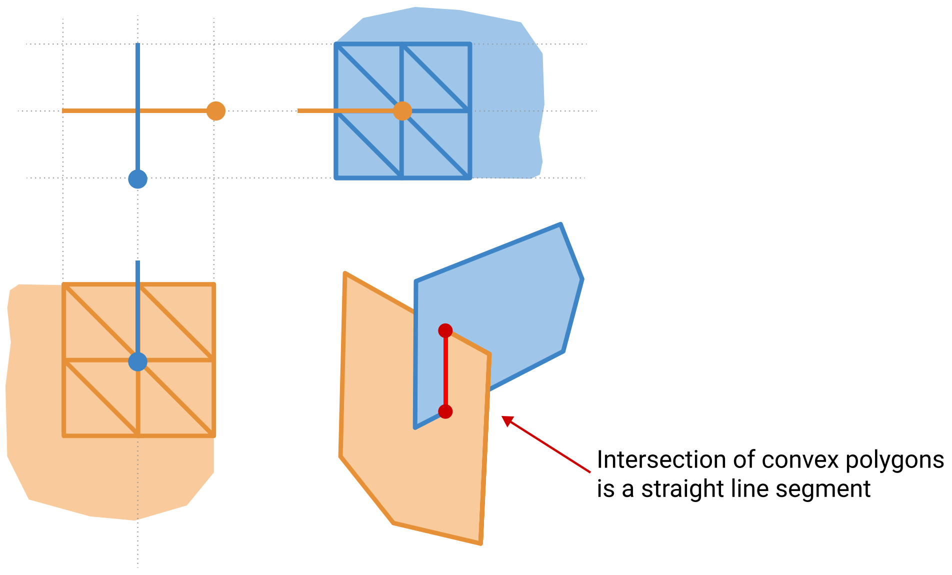


Figure 3.5.1: Multiple views of two intersecting fractures.

The choice of what strategy to use can be different for each fracture. Of course, we leave it as a user option, and make it configurable within the input file. An example of how we can take this option from the user is present in Appendix A.

3.5 Fracture-Fracture Intersections

The target in the construction of which elements correspond to the interface between two fractures is, perhaps, clear at this point in our discussion. We are working in a 3D conformal mesh, for which we have guaranteed no duplicated elements, and therefore: A search is in place for unrefined, one-dimensional elements that are shared by the surfaces of both fractures (Figure 3.5.1).

This description, however, does not fully address the possible intricacies resulting from our methodology.

In our attempt to remove geometrical ambiguity and remove undesired features, we have admitted distortions to the surface mesh of fractures away from their initial plane. Directly, this suggests that, without moving existing nodes, the best we can hope for is some approximation of the straight line segment that defines the intersection of 2 planar polygons. But there is another level of complexity we should account for.

Assume that geometrical tolerances are set high enough that some intersection coordinates will get coalesced (as explained in Section 3.2.6). If we gradually rotate two coincident fractures into the same plane, at some point, they will, unavoidably, form a relative angle that crosses the threshold of the tolerable angle. Beyond this point, we can reasonably assume some subset of both fractures will get snapped onto the same surface (Figure 3.5.2).

Naive assembly of shared edges between the fractures is in this case not truly representative of the intersection between 2 planes. Such a setup would imply edges that fork, and loop around

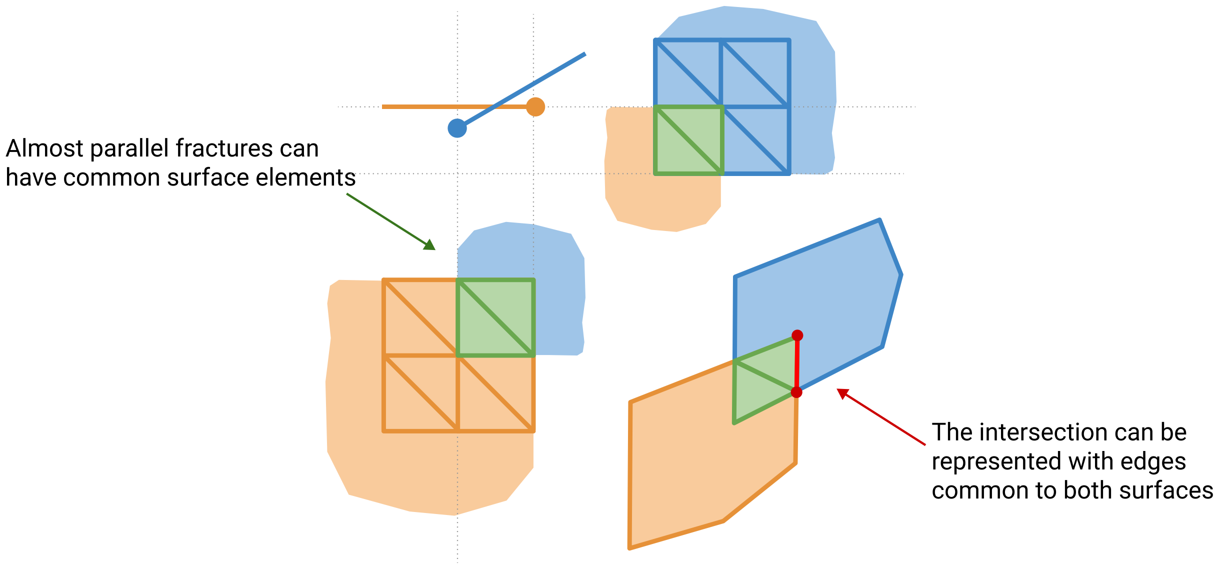

Almost parallel fractures can have common surface elements
The intersection can be represented with edges common to both surfaces

Figure 3.5.2: Multiple views of almost-parallel intersecting fractures.

faces that are common to both surfaces. Surely, we need to search for a less ambiguous definition of such an intersection.

The more consistent representation should form a continuous open curve. Or, in algorithmic terms: these edges should connect in pairs at the interior nodes, and the boundary nodes should be only 2.

Hence, to solve the problem of defining fracture-to-fracture intersections we propose:

- Compute a segment defining the intersection between the 2 *fracture-polygons*;
- Assemble all edges common to both surfaces; Through element connectivity we can define a planar graph for these edges;
- Through geometrical search, choose a start and an end node. Those should be uniquely defined by being those closest to the points of the intersection segment;
- Solve for the shortest path connecting these nodes within the graph.

The shortest path solution is, within the graph, the best possible approximation of the intended geometric intersection.

We assume the user only wants a geometrical representation of the intersection for fractures whose defining positions indeed make contact with each other. Thus, if an intersection segment cannot be found between the fracture-polygons, no attempt is made to list common elements to a pair of surfaces. This is true even if the surfaces do, nonetheless, share elements. Such a configuration can, for example, occur due to the extension of their boundaries (Section 3.4), or the coalescing of intersection points (Section 3.2.6).

3.6 Importance And Imposition Of Convexity

If not clear by now, some light should be shed on the fact that many of the operations described up until this section are conditional to the convexity of many of the geometrical figures involved.

The complete domain itself may not be convex, and even the resulting fractures may slightly distort away from convexity. Yet, through out our discussion, we have invoked this property for: Each coarse and skeleton elements individually; For the more abstract polyhedral volumes; And for the polygons which initially define the fractures.

To illustrate, take the following examples of how the lack of convexity would hurt our methodology:

- The early premise, that an intersection of a fracture and a 2D element is a continuous straight line segment connecting 2 nodes at 2 different edges of that element, is a direct application of our binary classification of nodes being exclusively on either of the sides of a plane (Subsection 3.2.2) and the definition of convex bodies (Definition 2.5.1). It is only valid for convex fracture planes intersecting convex 2D elements;

- All of the split patterns described in Section 3.2.5 can only be guaranteed to remain consistent (no *self-intersection*, positive *jacobian of geometric map*, and accumulated area of subelements matching that of their parent element) if they are the product of the refinement of convex elements;

- Indirectly, since lack of convex 2D skeleton elements (at the shell of each polyhedral volume) induce lack of simple continuous straight line segments as intersections, our *"circular search"* for the surface sub-polygons (Section 3.3.1) would not make sense.

- Without convex polyhedral volumes, there is no guarantee that the sub-polygons will be simply-connected, which is contrary of our aim of favoring simplicity.

- The boundary recovery algorithm (Section 3.4) is also a direct application of the definition of a convex polygon. Assuming a lack of convexity, as the orthogonal planes classify which elements are kept on the surface and which are removed, we could construct geometries that would completely remove all elements from the surface. Failing to recover a boundary since there would be no surface to bound.

- The shortest path algorithm, proposed for fracture-fracture intersections, assumes unique (and connected) *initial* and *end* points, so a path can be defined. Non-convex fractures would frequently observe discontinuous intersections, which would result in disconnected graphs and no shortest path solution[11].

Fortunately the convexity of the lower-dimensional skeleton elements is a given state; We can reasonably expect the user to input a coarse mesh that only contains convex elements, and we

[11]Depending on the aggressiveness of user-defined geometrical tolerances this can happen with the implementation as it currently stands. But it should be much less common, and removable through an adjustment of the tolerances.

have chosen our split patterns in such a way that refinements only generate convex quadrilaterals and triangles. This state is, however, not so trivially carried over into three dimensions.

It is important to point out that the methods described so far gave no means through which to impose convexity of the polyhedral volumes, even though we have established them to depend on it. In fact, it should be clear to the reader that we did start from a position of guaranteed convexity for 2 reasons: First, we have extended the limits of every fracture to the bounds of the polyhedral volume that contains each subset of it. Through this, we are assured that every intersection fracture/polyhedron is a complete cut-through of that volume, and that it will be divided into two complementary polyhedral sub-volumes. Second, Theorem 2.5.4 assures that the cross section is convex, which in turn implies that the (at most) 2 complementary sub-volumes created by this cut-through are convex themselves.

We did give up security of 3D convexity due to snap operations on intersection nodes (Subsection 3.2.6). This loss of volumetric convexity materializes through the moving of points away from the initial fracture plane definition, which frequently results in the distortion of the previously referred *"sub-polygons"* into arbitrarily deformed surfaces.

The solution for this comes in 2 steps: First we need to actually identify that we have created a non-convex polyhedron, second we need to refine it into tetrahedra (which are guaranteed to be convex).

Assemble the newly created polyhedra from the newly created surface elements (which should have no polyhedra associated to them, since they were just created). As described before, during the assembly of a polyhedral shell, we move through every rolodex and collect every "next face" in that rolodex. During this step, access is made to the angle of each face relative to the reference of the rolodex. The oriented difference between these angles gives the angle between 2 faces. This angle gives the measure through which to check if a polyhedron is convex or not. We simply verify that all internal dihedral angles are less than or equal to π radians.

Once a polyhedral shell has been assembled and verified not to be convex, we tetrahedralize it using the Gmsh API.

This operation is done for each fracture after a surface mesh is created for it. Through this strategy, every fracture is guaranteed to only find convex volumes and will only leave convex volumes. Thus completing the validity and demonstration of our taken set of premises.

3.7 Volumetric Porous Matrix And The Global Algorithm

To close this chapter we now address the final domain required by the numerical formulation: the volumetric fine-mesh for the porous matrix.

Recall that our method has started with a setup for the coarse mesh with the introduction of skeleton elements and the volumes that they bound. Those were then followed by the discretization for the domains of a fracture: Points, ribs, surface, boundary, and intersections. In the process, all the volumetric regions, introduced by the surfaces of fractures, were systematically represented in the form of polyhedral volumes.

Any new volume introduced was, naturally, a refinement of the ones that came before. Starting from the coarse mesh.

The methods take it a step further and represent those volumes mainly through the list of faces that bound their interior. Such a boundary representation nicely ties with how Gmsh represents their volumes. Therefore, we are conveniently positioned, in the end of the methodology, to easily mesh the porous matrix: To complete the typical problem domain representation, we export the unrefined polyhedral volumes to Gmsh as the CAD domains in which to create the fine-mesh. This strategy ultimately delivers fine-meshes that are conformal, since polyhedral volumes are conformal amongst themselves.

We describe a global algorithm that ties all described components in a structured disposition. The proposed global steps can be summarized through the pseudo-code in Algorithm 3.7.1.

Algorithm 3.7.1: Create Multi-scale DFN Mesh

```
Input  : A coarse mesh for the porous medium, a set of fractures, and geometrical
         tolerances
Output: A Gmsh CAD file
$dfn \leftarrow$ Create DFN object pointing to coarse mesh;
$dfn$.InitializeSkeletonMesh (1D = ribs, 2D = faces);
$dfn$.InitializePolyhedralVolumes;
foreach $fracture$ $polygon$ do
    $fracture \leftarrow dfn$.CreateNewFracture($polygon$);
    Sort nodes above and below fracture plane;
    Search for ribs cut by fracture;
    Add 2D neighbours of cut ribs as intersected faces;
    $fracture$.ExtendBoundaries();
    Check geometrical tolerances and snap intersections coordinates;
    Refine skeleton elements cut by $fracture$;
    $fracture$.MeshFractureSurface();
    $fracture$.RecoverBoundaries();
    $dfn$.UpdatePolyhedralVolumes;
end
for $j \leftarrow 0$ to $NumFractures$ do
    $fracJ \leftarrow DFN$.Fracture[$j$];
    for $k \leftarrow j+1$ to $NumFractures$ do
        $fracK \leftarrow DFN$.Fracture[$k$];
        $fracJ$.FindFractureIntersection($fracK$);
    end
end
Export Gmsh CAD file;
```

Chapter 4

Examples and Result Analysis

To explore the extent of the detailed methods, we now move to some examples.

4.1 Effects of geometrical tolerances

To study the effects of varying geometric tolerances (see Section 3.2.6), this example is setup such that the gradual increase in the minimum tolerable angle will distort the fracture surface. Figure 4.1.1 contains the coarse mesh and fracture illustration for this test. The fracture is embedded in a 3x3 cubic coarse mesh, and has a rotation on all 3 directions. Its corner nodes are close to the boundaries of the coarse mesh, but not directly on them.

The first run takes a minimum tolerable angle of 0.1 radians, and delivers a mesh with no distortion from the initial plane definition. Figure 4.1.2a contains the resulting surface mesh, accompanied by the histogram of element quality for the complete fine-mesh in Figure 4.1.2b.

The second run takes a minimum tolerable angle of 0.3 radians, and delivers a mesh with noticeable distortion. In particular, the corner nodes of the fracture are seen snapped onto the faces of the coarse mesh. The surface elements still reside in the initial plane, but the resulting surface mesh occupies a non-convex region. Figure 4.1.3a contains the resulting surface mesh,

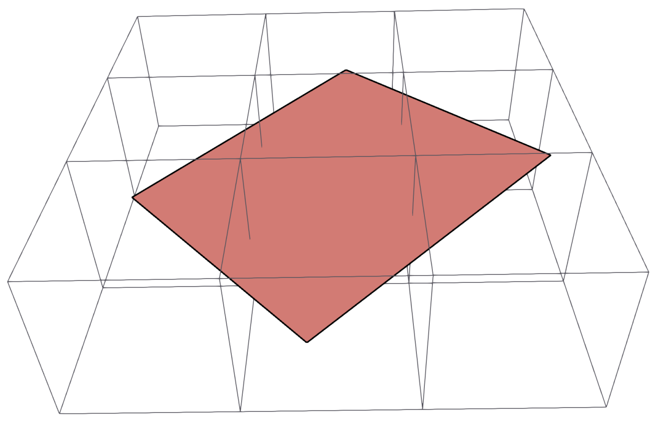
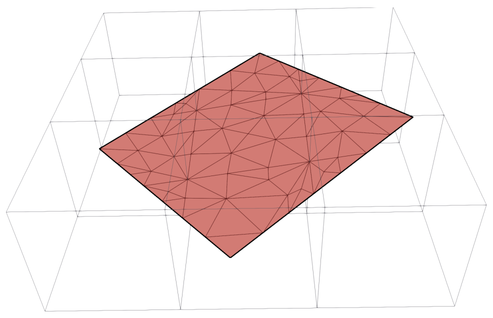
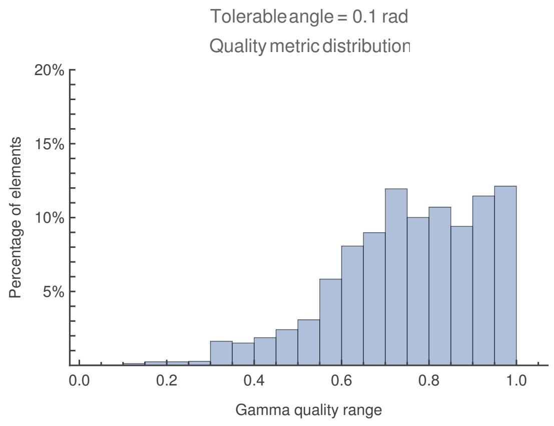


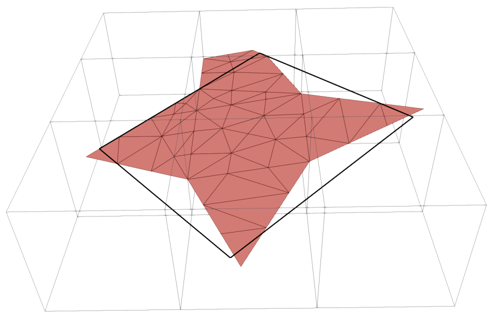
Figure 4.1.1: Coarse mesh and fracture polygon configuration to test varying geometric tolerance.

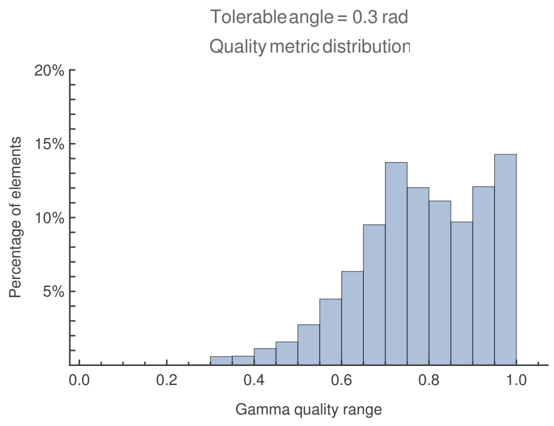

(a) Surface mesh



(b) Element quality histogram

Figure 4.1.2: Resulting mesh with tolerance set to 0.1 radians.


(a) Surface mesh



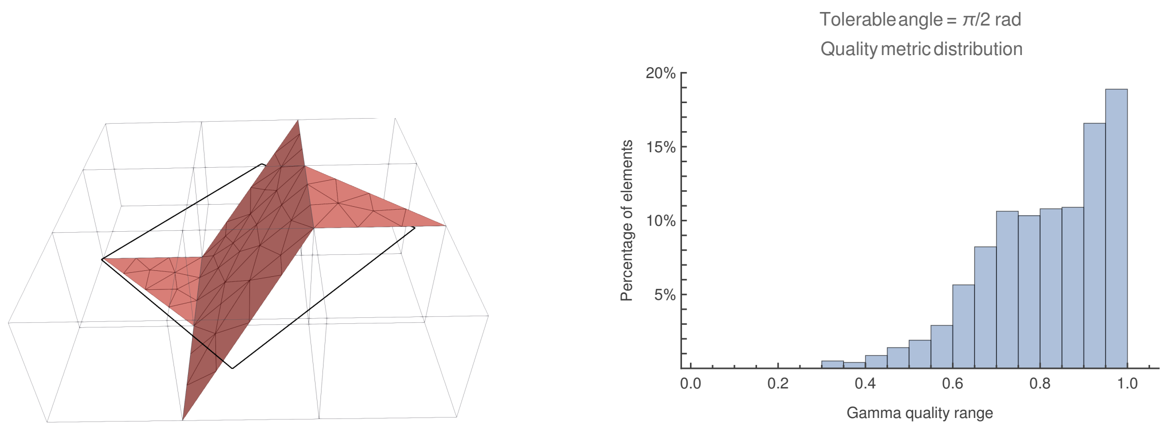
(b) Element quality histogram

Figure 4.1.3: Resulting mesh with tolerance set to 0.3 radians.

accompanied by the histogram of element quality for the complete fine-mesh in Figure 4.1.3b. A slight increase in mesh quality is also noticeable, specially to the minimum values.

The final run takes a minimum tolerable angle of $\pi/2$ radians, and delivers a highly distorted surface mesh. At this limit, all intersection points are coalesced and no rib of the original skeleton mesh is refined. Figure 4.1.4a contains the resulting surface mesh, accompanied by the histogram of element quality for the complete fine-mesh in Figure 4.1.4b. The gamma quality measure is visibly being pushed onto the 1.0 upper bound, but, depending on the use case, such a gain in mesh quality may be insignificant in face of the high amounts of distortion to the fracture.

This example was designed to maximize distortions to the fracture surface, so it should be taken as an exaggeration to the feature it demonstrates. In practice, we would expect the coarse mesh and tolerances to better match the geometry of fractures such that a balance is achieved to keep surface distortions within a more reasonable range.



(a) Surface mesh (b) Element quality histogram

Figure 4.1.4: Resulting mesh with tolerance set to $\pi/2$ radians.

4.2 Performance

To evaluate the scalability of the implementation, we are mostly interested in how computational time grows with some variability of input data. All results here presented were measured in a typical personal computer (current generation at the time of this writing): CPU Intel i7-8550U, and 16MB of DDR4 RAM memory. Each test was run and measured 6 times, with any strong outliers discarded.

4.2.1 Test 1: Single Fracture Crossing a Progressively More Refined Mesh

The first example corresponds to a progressive refinement of the same box domain. Figure 4.2.1 details the constant geometry of a triangular fracture that diagonally crosses through the increasingly more populated coarse mesh. This example is designed to be independent in Gmsh: Every volume is guaranteed convex, and all subsets of the fracture surfaces are triangles.

Table 4.1 shows the average measurements for each of the 11 coarse meshes. Figure 4.2.2 contains a scatter plot of all measurements over a linear regression of the results.

It is clear that, at least within this range of up to a million coarse elements, the computation time scales linearly to the number of coarse elements.

4.2.2 Test 2: Multiple Fractures On Constant Coarse Mesh

We can also look at performance from the perspective of number of fractures as a parameter. On a limiting case, where all fractures coincide to the same coarse volume, we can try to get a sense for the upper bound on scalability cost of the current implementation.

This example follows the fracture distribution illustrated in Figure 4.2.3. Fractures are introduced parallel to the 3 directions crossing a cube. Even though this example starts with a single coarse element, the progressive refinement induced by each new fracture causes the number of skeleton elements to grow in higher order.

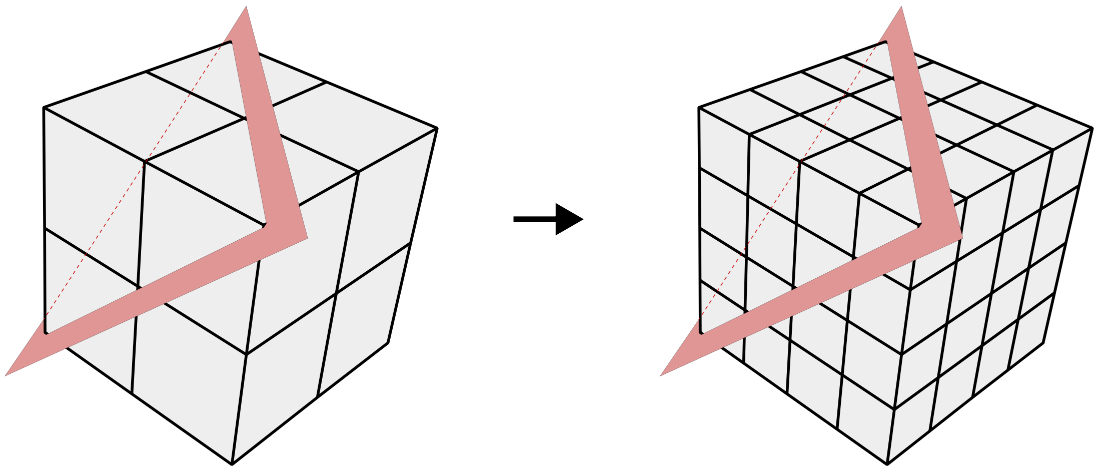


Figure 4.2.1: Illustration of performance test 1.

Table 4.1: Average performance measurements for multiple fractures

Number of Coarse Elements	time (s)
8	0.0111148
64	0.0276025
125	0.0425578
512	0.118278
1 000	0.20837
4913	0.857347
10648	1.75272
50653	7.6214
103823	13.6862
512000	63.2472
10^6	124.997

Results are organized by average time in Table 4.2. Figure 4.2.4 contains a scatter plot of all measurements up to 84 fractures.

For this limiting case, the raising number of skeleton elements with every fracture pushes the computation time to grow in higher order. A good polynomial curve fit for this range can be achieved with cubic order on the number of fractures.

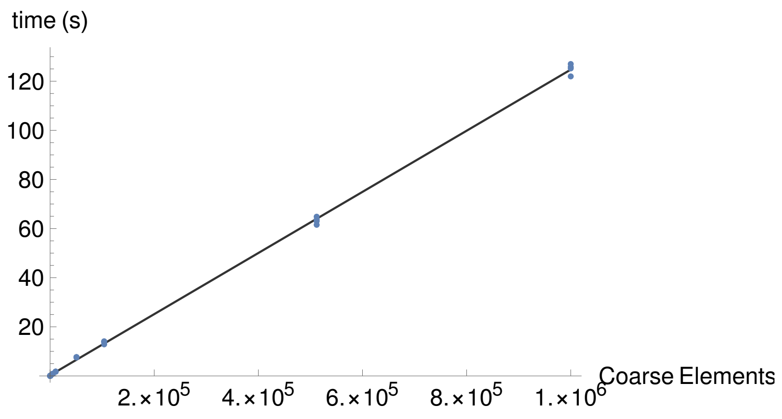


Figure 4.2.2: Scalability test 1: Computational time growth with respect to number of coarse elements.

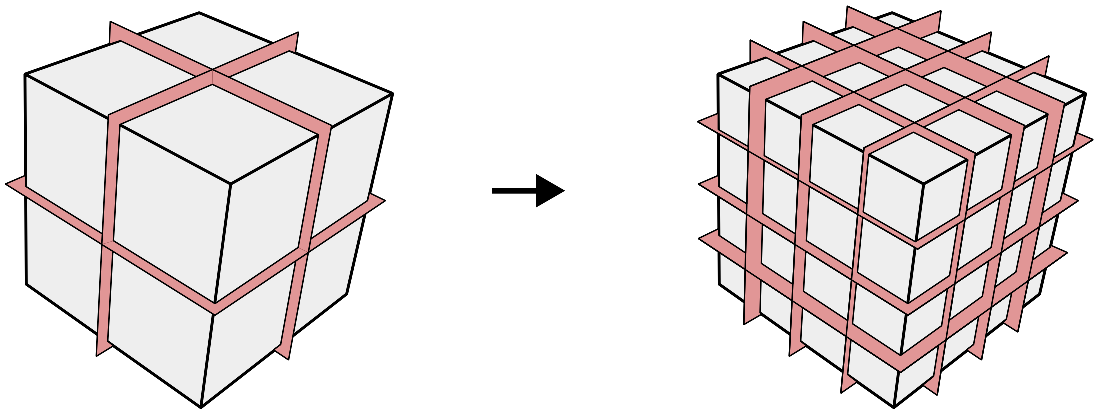


Figure 4.2.3: Illustration of performance test 2.

4.3 Fracture 3D Benchmarks

To explore mesh quality and robustness, we look to the literature for examples that have been recently taken as benchmarks for discrete fracture problems.

All 4 benchmarks cases here presented were taken from Berre, Boon, et al. (2021), and should be of interest to those seeking some reference to evaluate the quality of their implementation. The original paper is accompanied with single-scale meshes hosted in an open repository, but imposes no coarse regions restrictions that can be used as coarse mesh. Hence, the coarse meshes presented here are author-defined and constrained only by three goals:

1. To have enough resolution so that all fractures are represented;
2. The boundary conditions can be depicted with coarse level discretization;
3. Fractures accept some minor level of distortion off their original plane.

Table 4.2: Average performance measurements for multiple fractures

Number of Fractures	time (s)
3	0.0359803
9	0.122523
15	0.367046
21	0.840523
27	1.71195
33	2.97722
39	4.93305
60	21.3819
84	60.6731

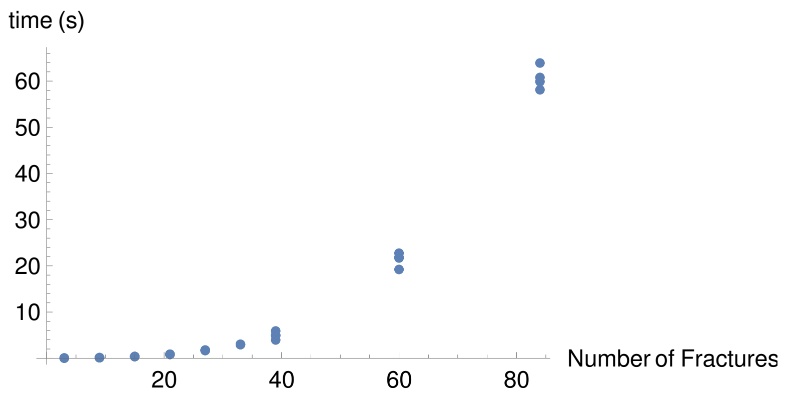


Figure 4.2.4: Scalability test 2: Computational time growth with respect to number of coincident fractures.

4.3.1 Case 1

Case 1 starts simple with a single fracture crossing the complete domain. To solve it, there is even no need for the boundary recovery and fracture-fracture intersection definition described in Sections 3.4 and 3.5.

Figure 4.3.2 shows the input coarse mesh and fracture polygon. Figure 4.3.3 shows a clipped view of the fine mesh to display some detail for the fracture surface mesh and how it occupies the interface between volumetric fine-elements.

In Table 4.3 and Figure 4.3.1 the element quality metrics distribution are summarized.

Table 4.3: Case 1: Element quality metric range count

Quality range	Element count	**Quality range**	Element count
0. – 0.05	0	**0.6 – 0.65**	726
0.05 – 0.1	0	**0.65 – 0.7**	1039
0.1 – 0.15	6	**0.7 – 0.75**	1154
0.15 – 0.2	7	**0.75 – 0.8**	649
0.2 – 0.25	12	**0.8 – 0.85**	142
0.25 – 0.3	10	**0.85 – 0.9**	66
0.3 – 0.35	8	**0.9 – 0.95**	418
0.35 – 0.4	80	**0.95 – 1.**	54
0.4 – 0.45	116	**1.**	0
0.45 – 0.5	229	**Minimum**	0.13108
0.5 – 0.55	243	**Average**	0.682999
0.55 – 0.6	449		

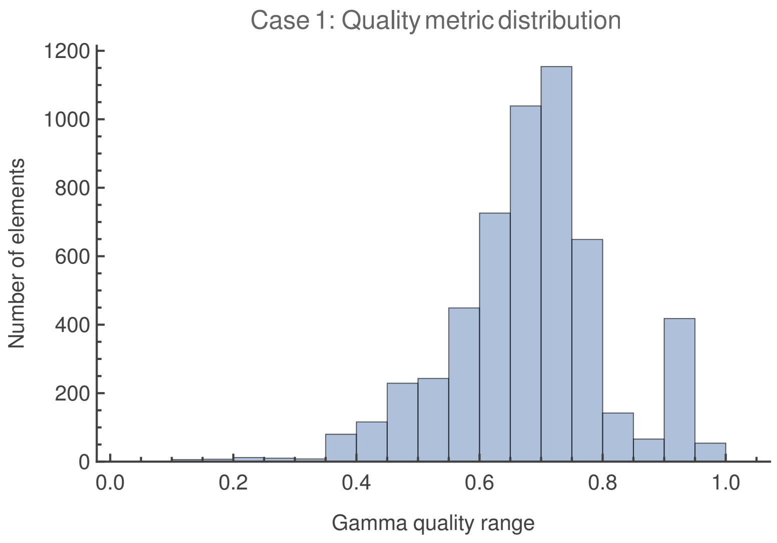


Figure 4.3.1: Case 1: Histogram of element quality metric for the fine mesh.

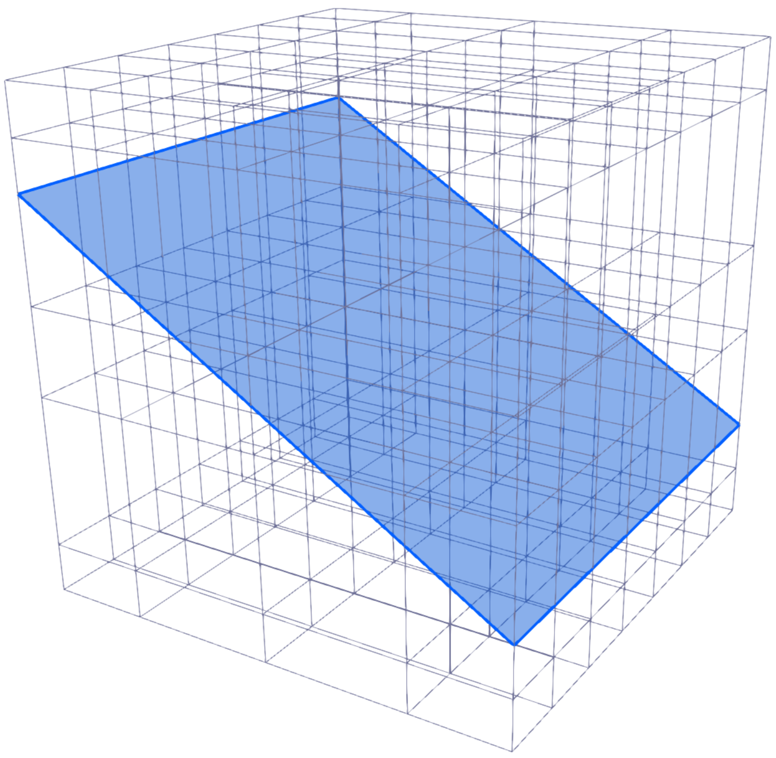


Figure 4.3.2: Case 1: Input coarse mesh and polygons.

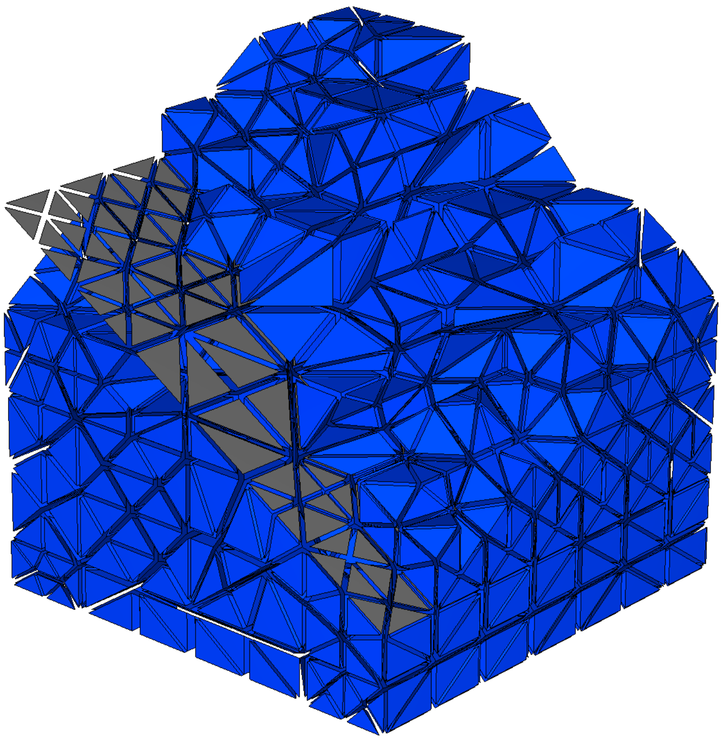


Figure 4.3.3: Case 1: Clipped view for the fine mesh and fractures surface meshes.

4.3.2 Case 2

Case 2 introduces an example with multiple fracture-fracture intersections, but with an orthogonal and easily predictable configuration. To solve it with convenience, all fractures are slightly scaled up and their boundary directives are set to *truncated* (see last Subsection in 3.4 for details). All fractures in this example coincide with existing faces of the coarse mesh, and thus all elements that compose their surfaces are incorporated from the existing skeleton mesh.

Figure 4.3.5 shows the input coarse mesh and fracture polygons, translucid and colored by the index of each of the 9 polygons. Figure 4.3.6a shows a clipped view of the structured fine mesh.

Due to all elements in this mesh being perfect squares and cubes, Table 4.4 and Figure 4.3.4 do not add much information other than demonstrating the capacity of our implementation to also deliver structured grids (limited to the support of Gmsh to those cases).

We can, however, take advantage of the structured and orthogonal configuration of this example, and look at Figure 4.3.6b to display the details of all fracture intersections and boundaries.

Table 4.4: Case 2: Element quality metric range count

Quality range	Element count
0.0 – 1.0	0
1.	32 768
Minimum	1.0
Average	1.0

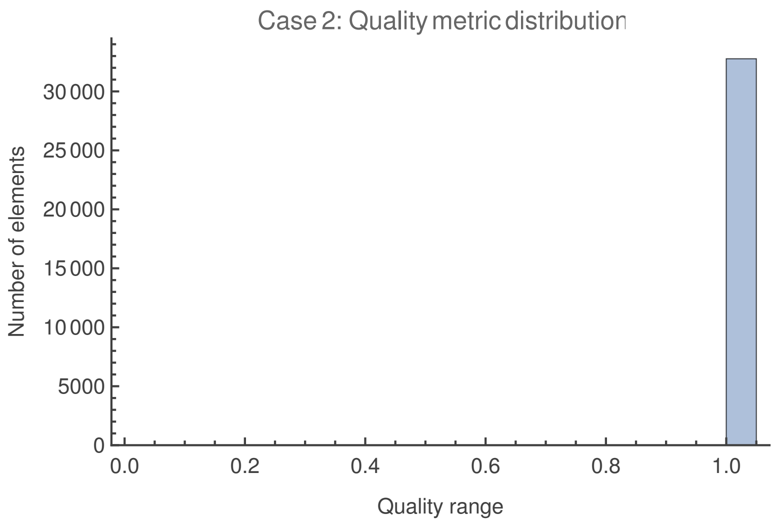


Figure 4.3.4: Case 2: Histogram of element quality metric for the fine mesh.

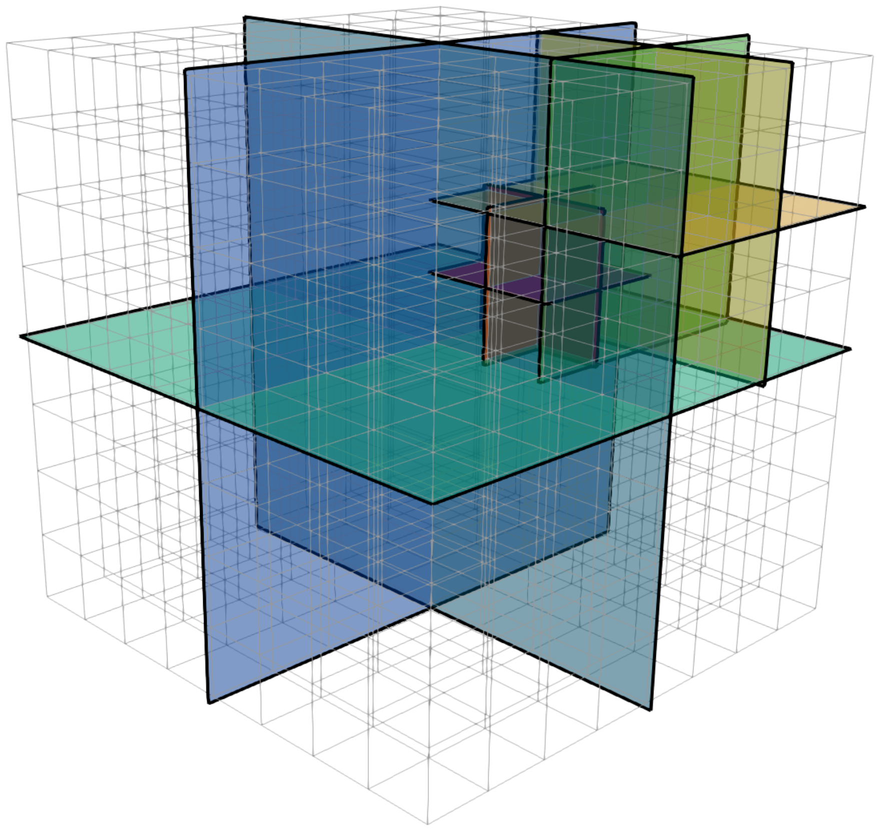
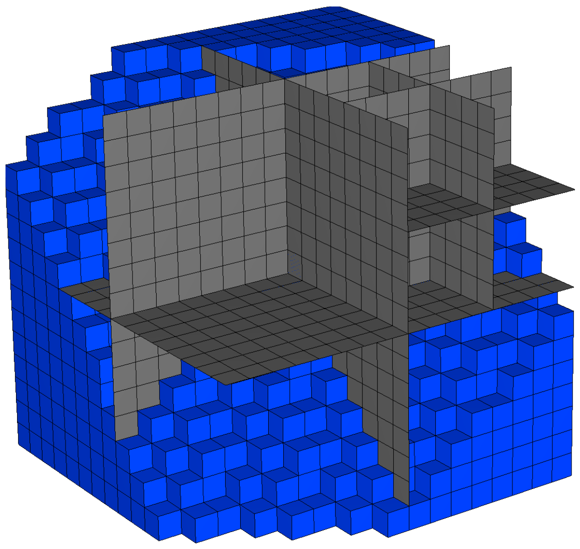
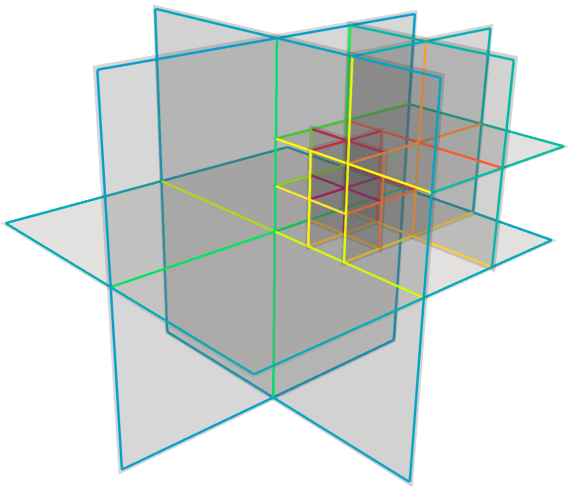


Figure 4.3.5: Case 2: Input coarse mesh and polygons.



(a) Case 2: Clipped view for the fine mesh and fractures surface meshes.



(b) Case 2: Fracture-fracture intersections and fracture boundaries.

Figure 4.3.6: Case 2: Fine-mesh features.

4.3.3 Case 3

Case 3 takes a step towards complexity, with fractures crossing each other at different angles and limits that require the complete run of boundary recovery operations.

Figure 4.3.8 shows the input coarse mesh and fracture polygons. Figure 4.3.9 shows a clipped view of the resulting fine mesh. Note how the imposition of geometrical tolerances causes some of the fracture surface to slightly distort.

In Table 4.5 and Figure 4.3.7 the element quality metrics distribution are summarized. The choice for a structured coarse mesh of cuboids here causes enough incidence of fractures at sharp angles to induce the creation of a minority of (fine-mesh) tetrahedra below gamma quality of 0.1, but well over 80% of fine-elements are over 0.6.

Table 4.5: Case 3: Element quality metric range count

Quality range	Element count	**Quality range**	Element count
0. – 0.05	13	**0.6 – 0.65**	5280
0.05 – 0.1	116	**0.65 – 0.7**	631
0.1 – 0.15	90	**0.7 – 0.75**	12667
0.15 – 0.2	111	**0.75 – 0.8**	324
0.2 – 0.25	75	**0.8 – 0.85**	95
0.25 – 0.3	191	**0.85 – 0.9**	202
0.3 – 0.35	241	**0.9 – 0.95**	54
0.35 – 0.4	339	**0.95 – 1.**	1
0.4 – 0.45	349	**1.**	0
0.45 – 0.5	557	**Minimum**	0.0321483
0.5 – 0.55	704	**Average**	0.656189
0.55 – 0.6	785		

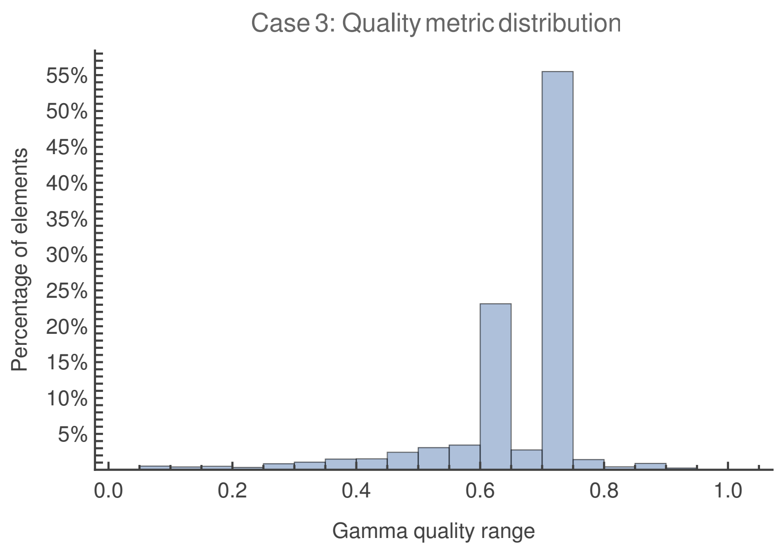


Figure 4.3.7: Case 3: Histogram of element quality metric for the fine mesh.

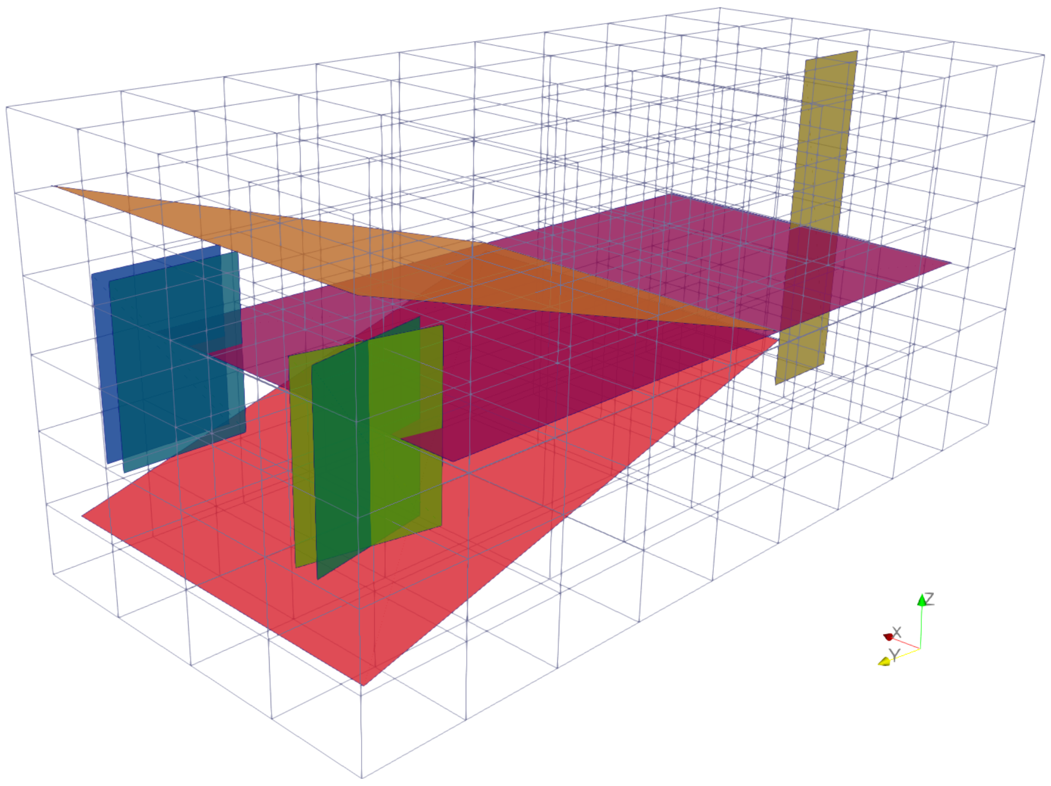


Figure 4.3.8: Case 3: Input coarse mesh and fracture polygons.

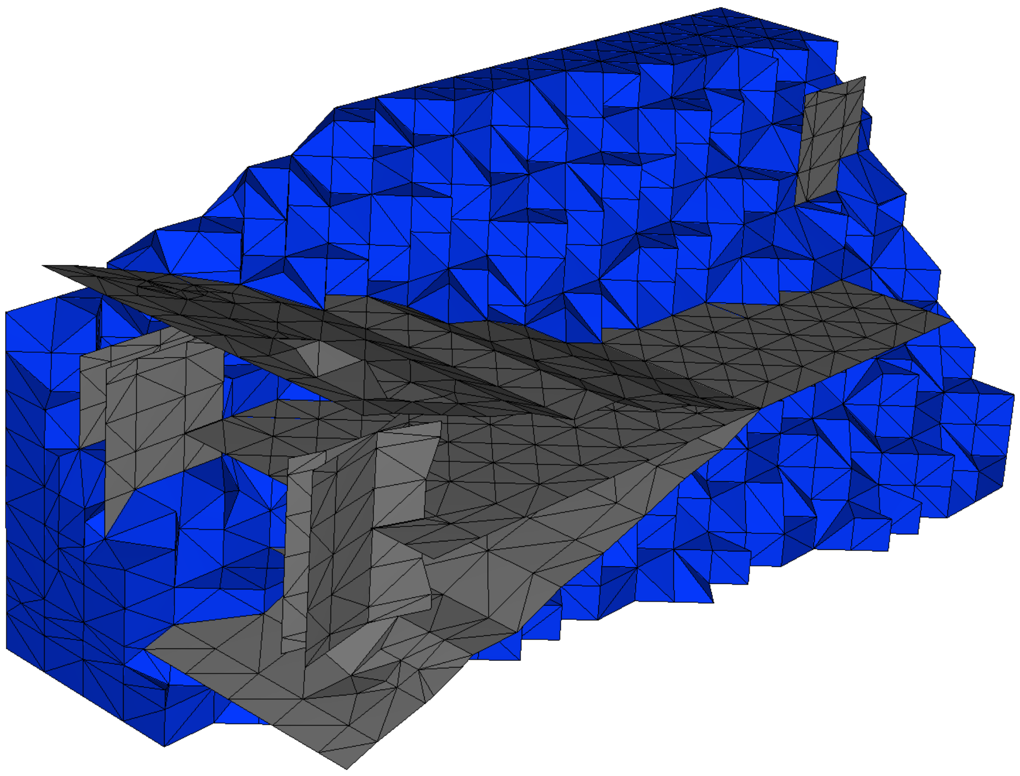


Figure 4.3.9: Case 3: Clipped view for the fine mesh and fractures surface meshes.

4.3.4 Case 4

The most challenging configuration is defined in Case 4. With 52 fractures of arbitrary distribution and intersections, it is the first example with polygons of more than 4 corners. Running it with a coarse mesh of 5712 elements with one level of uniform pre-refinement (to create the resolution necessary to capture 2 very small fractures), takes the initial setup to 51408 cuboid elements (8 sub-elements added to each coarse volume). Using the same machine whose performance was measured in Section 4.2, amounts to around 28 minutes of computation time.

Figure 4.3.11 shows the input coarse mesh, and fracture polygons colored by fracture index. Figure 4.3.12 shows a clipped view of the resulting conformal fine mesh. In Table 4.6 and Figure 4.3.10 the element quality metrics distribution are summarized.

Note how, similarly to case 3, the imposition of geometrical tolerances partially distorts most fracture surfaces to meet the nodes previously defined in the mesh.

Table 4.6: Case 4: Element quality metric range count

Quality range	Element count	**Quality range**	Element count
0. – 0.05	11	**0.6 – 0.65**	13529
0.05 – 0.1	182	**0.65 – 0.7**	27580
0.1 – 0.15	187	**0.7 – 0.75**	9037
0.15 – 0.2	2549	**0.75 – 0.8**	20630
0.2 – 0.25	3012	**0.8 – 0.85**	637649
0.25 – 0.3	3741	**0.85 – 0.9**	374
0.3 – 0.35	1043	**0.9 – 0.95**	657
0.35 – 0.4	5937	**0.95 – 1.0**	427
0.4 – 0.45	1420	**1.**	312545
0.45 – 0.5	2242	**Minimum**	0.0353279
0.5 – 0.55	12016	**Average**	0.841418
0.55 – 0.6	6575		

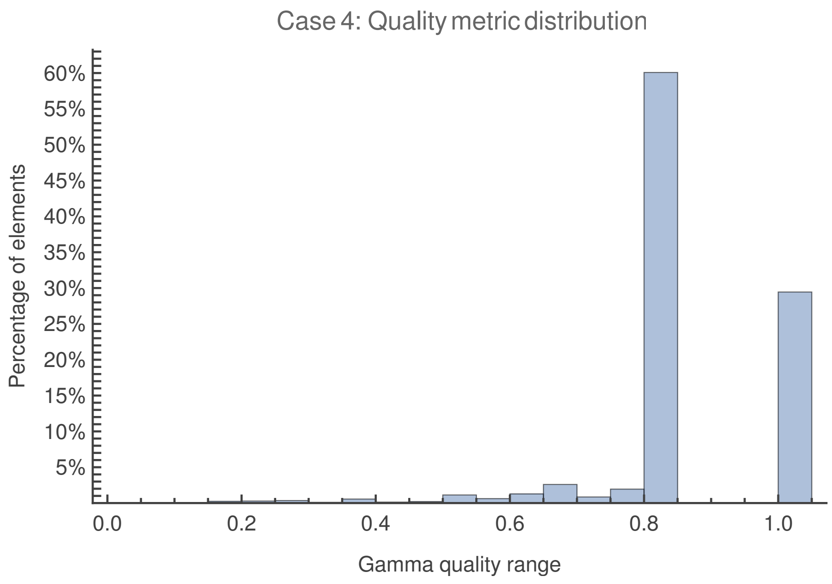


Figure 4.3.10: Case 4: Histogram of element quality metric for the fine mesh.

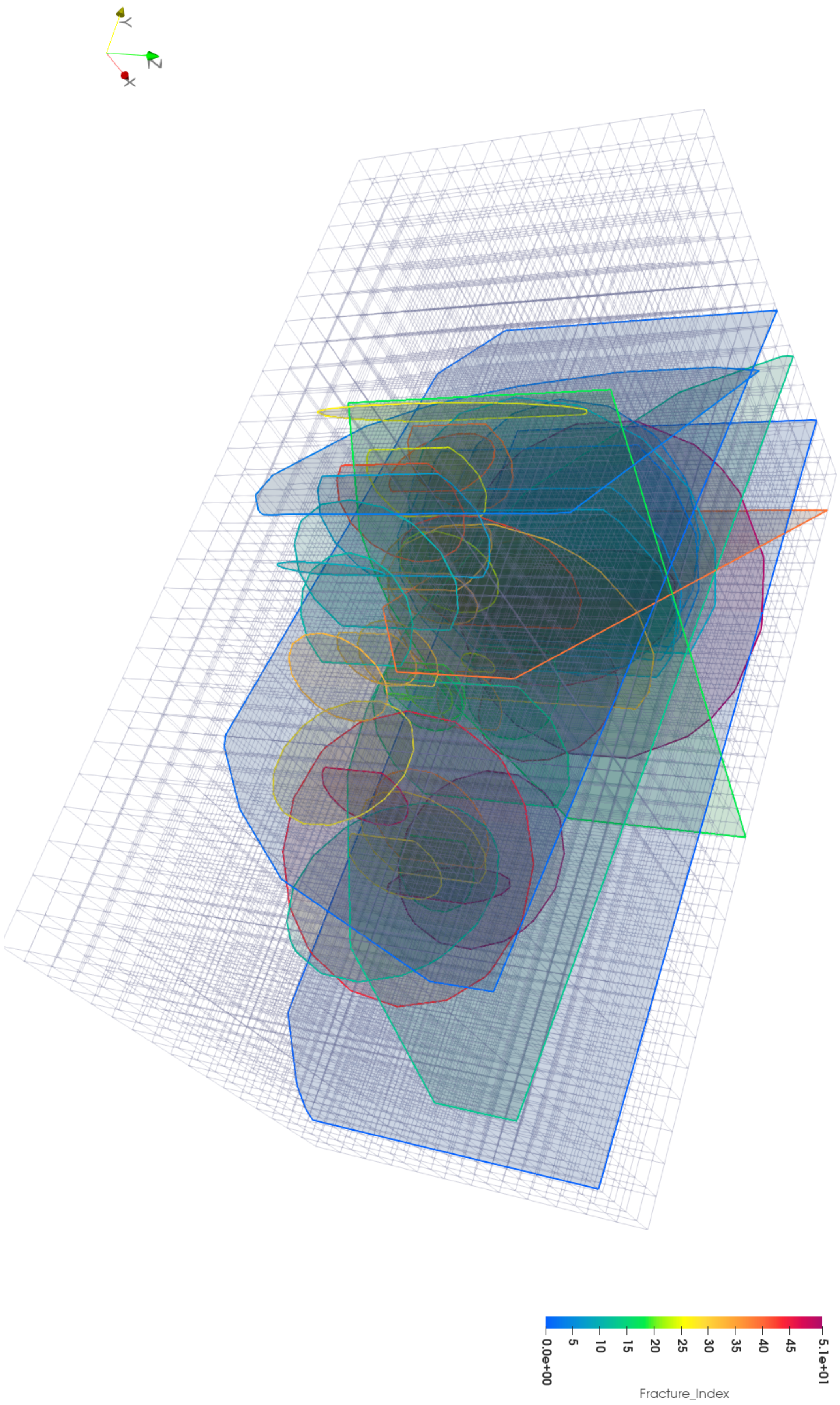


Figure 4.3.11: Case 4: Input coarse mesh and polygons.

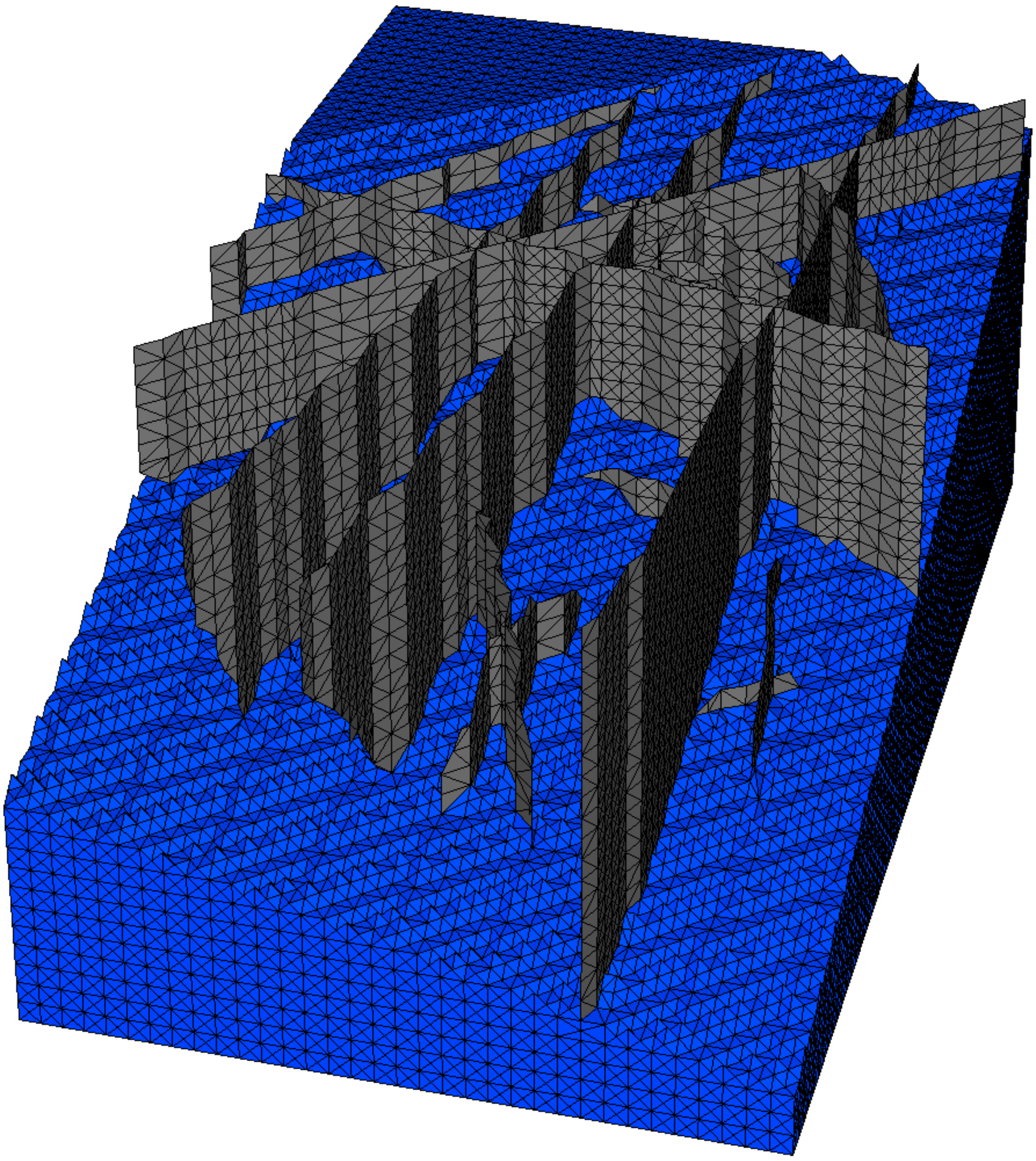


Figure 4.3.12: Case 4: Clipped view for the fine mesh and fractures surface meshes.

Chapter 5

Conclusions

In view of the solved examples on top of the detailed methodology, the intended goal was achieved: We have built a fractured mesh generating tool restricted to a pre-defined first-level mesh. Results show that the proposed technique can robustly construct adequate 3D meshes.

Novelty introduced by the methodology is present in the freedom given to the user to set their coarse mesh as best appropriate for their problem: Be it a reservoir geology and well distribution, or complex layers for ground-water management. We can expect this feature to allow analysts to make the best use of multi-scale finite element simulators.

The accompanying code pulls its reliability from the consistent background of mathematically coherent premises, like: convexity, element connectivity and systematic explicitly-defined topologies. The robustness of the algorithms has been demonstrated against popular benchmarks available in the literature of over 50 fractures that mimic real geological occurrences.

Another noteworthy feature lies in the relatively sparse presence of floating-point operations, considering an algorithm of geometrical nature.

There are:

- Angle computations and sorting. Computed once for each pair of a 2D and a 1D element sharing the 1D side, and updated only when a new face is added to the neighbourhood of that edge (Subsection 3.1.2);
- Computations to characterize a polygonal plane for fracture definition, i. e. orientation computation and checking for co-planarity (Subsection 3.2.1);
- There are the node binary classification: O(n) dot products for each fracture, n being the number of nodes in the mesh (Subsection 3.2.2);
- Intersection node interpolated from two edge nodes (O(n) dot products and divisions, n = number of edges) and checking for points within bounds of a polygon (O(n) divisions + 2 subtractions, n being the number of edges of the polygon); Only done for edges with nodes on opposite sides of the fracture plane; (Subsection 3.2.3)

- Floating-point numbers subtractions and comparisons to check for polyhedra convexity (of O(n), n being the number of edges of a polyhedron, done for each assembled polyhedra); (Subsection 3.1.2)

- Edge length and shortest path computations to determine the intersection of 2 fractures (only done for fractures that share elements in their surfaces, and limited to the number of edges that are shared by those two fractures); (Section 3.5)

  And these are accompanied by the meshing algorithms. Geometric and strongly dominated by floating-point operations. Done by Gmsh:

- Delaunay 2D meshing of a few elements (one simple fracture subset at a time, performed by Gmsh API when fractures subsets have more than four edges);

- Delaunay 3D meshing of a few elements (one polyhedron at a time, only when those are found to be non-convex, or we cannot solve for fracture trying to partially incorporate mesh elements to its surface);

All other operations are logical by nature and/or of list lookups. This results in a targeted implementation, where most of the computational effort is taken by dynamic memory allocation and the meshing operations done by the Gmsh API. Room for performance improvement should be then mainly present in details of memory allocation, but that is out of scope for this work.

The most significant contribution of this work (to the state-of-the-art) exists in our approach to the volumetric meshing of the porous matrix. We employ the (always manifold) polyhedral volumes, created along the refinement of the coarse mesh and fracture insertions, as simple CAD subdomains for easy 3D mesh generation by Gmsh. This divide and conquer approach can be expected to significantly improve numerical stability of meshing algorithms, since other methods unavoidably require fractures to act as non-manifold mesh constraints.

This proposed technique is described in Section 3.7 as a natural final step in the methodology. Although naturally evolving, the key features that enable this strategy are very deliberate design decisions we have made leading up to it:

- Polyhedral volumes are one of our core building blocks;

- We start with a pre-defined coarse mesh and introduce fractures as a refinement problem. Thus easily tracking fine-to-coarse geometric figures relations, and;

- The proposal we have given to handle the fracture boundaries, by always completing a cut-through of polyhedral volumes (by extension or restriction), ensures these volumes are kept manifold.

This feature is truly a product of the restriction of 2-levels of discretization and, in a sense, a reframing of this constraint into a useful construction. From our reasonable efforts of literature review, no similar strategy appears to have been published to date.

We note, moreover, that there is no intrinsic characteristic of the described methods that make them unusable to single-scale finite element meshes. Indeed, all choices made, which tailor these algorithms to multi-scale meshes, were of restrictive nature. They generally aim to conserve the coarse mesh, and to ensure the constructed sub-mesh to be coherently paired to their coarse counterpart.

Although usable for single-scale discretizations, the methodology discussed and implemented in this work is likely to be outperformed by other meshing tools, especially in the important criteria of mesh quality measures for fracture surfaces. This is attributed to the fact that the traditional approach (adding fractures and fracture-fracture intersections to the geometry definitions which constraint the meshing algorithms) will put as little as possible constraints into the optimizations performed by meshing algorithms; That freedom is used by these algorithms with the primary intent of maximizing quality measures at a fine-tuned balance for conserving the fidelity to the geometry definition. We have consciously given up that freedom as a trade-off for the feature of giving the user the ability of constructing, however they best see it fit, the first level of discretization (i.e. the coarse mesh). This, of course, being the intended use for the tool.

Limitations

- Our proposal to tackle the meshing process as a refinement problem carries the implicit assumption that there exist intersections for all fractures. An implementation of the given methods, as here written, should, therefore, fail to mesh fractures for which no intersection with the coarse mesh exists. Such a situation can be expected, for example, for fractures that are very small and simply do not cross a single edge of the coarse mesh. We have left out these cases from this text, due to current lack of interest.

  Assume that, relative to the coarse mesh, such a fracture is small enough that its corner nodes are all contained within adjacent polyhedral volumes. To contemplate such cases, is not a feature that would come with much challenge: One can easily verify that, for an unusual fracture, all ribs with nodes on opposite sides of the fracture-plane have failed the Point-in-Polygon test; Naively search for the polyhedral volumes that contains each of this fracture's corner nodes; And get a set of intersected-ribs as a subset to the edges of these volumes (and all volumes in-between them); This would entail skipping the Point-in-Polygon test this time, and checking only for nodes on opposite sides of this unusual fracture plane. The rest of the process should then follow the normal procedure layed-out in the methodology.

  For a fracture of greater size, for which the adjacent volumes hypothesis fails, a more considerable challenge is present. Nevertheless, these geometries can be discarded as unsupported since the user can alternatively tweak their coarse mesh slightly to create the necessary intersections for the conventional flow of our methods.

- If the complete domain is not convex and a fracture leaves then re-enters it, this takes the implementation to undefined behavior. It can be tweaked to work by manually separating the resulting fracture export to the cad file: The leaving and then re-entering of the fracture in the domain is likely to divide the fracture surface into two or more separate fractures, with separate boundaries, which the code will interpret as a single fracture, but the list may be simple enough that a willing user could split it manually.

- The methods discussed were strongly motivated by topological concepts of the geometrical entities used. This not only is a frequent source of optimizations but also is the foundation upon which we build robustness. We cannot state, however, that our approach is free of numerical instabilities introduced by geometrical floating-point operations:

  - At some stress tests which we have not covered in this text, extremely irregular polygons (e.g. naive-randomly generated polygons which have edges of very small lengths compared to its other edges) have introduced instability to the code. For now, we present this as limitation to our implementation (as it is currently described), and leave quality impositions to the polygons of fracture definition as a suggestion for future expansion of this work; A desirable and convenient pre-processing operation.

  - We have explicitly required in our methodology that the restricted triangulation performed for each subset of a fracture surface must not refine the edges that bound that subset (Subsection 3.3.1). This approach is the path we chose in order to efficiently impose mesh conformity for the skeleton mesh. A trait shared by the tessellation given to polyhedral volumes that need to be broken into tetrahedra. Such an imposition, in the presence of small tolerable distances (Section 3.2.6), was observed to cause Gmsh to generate collapsed (or nearly collapsed) elements; The so-called *slivers*. In practice, this means that geometrical tolerances cannot be arbitrarily prescribed. They must reasonably surround the scale of the size of coarse elements. We can, however, get around this limitation through the pre-refinement of the coarse mesh. This workaround reduces the lower bound on the robustly prescribable geometrical tolerance, and also improves the quality of outputted fine mesh. This strategy was, for example, used in Benchmark Case 4 (Example 4.3.4).

Suggestions For Future Research

- The present set of algorithms still have wide room for improvement when it comes to imposition of quality measures. The setup usually results in fair quality fractures when the initially defined coarse mesh is itself of good quality, but this infrequently can be stated for subpar coarse elements. Post processing algorithms for minimization of surface area for fractures surfaces meshes, either through the moving of points or swapping of elements can be implemented with reasonable effort and much gain in mesh quality.

- The meshing of a fracture surface, done one *subset* at a time (Section 3.3), has one restriction which was not explored to its full extent. We have divided the surface meshing to be done on each of the *sub-polygons* separately and, to keep conformity, have restricted the acceptable meshes to no refinement of the edges that bound them.

Such a constraint isolates these subsets enough that it could, for example, enable the parallelization of this meshing step to gain considerable performance.

Otherwise, one could relax this constraint by assembling all *sub-polygons*, and then using Gmsh API to mesh them all together. As separate 2D regions, but connected by shared edges. On a positive side, following this logic would add to the fracture surface the mesh optimizations possible within the scope of Gmsh; Also, the possibility to refine edges that may be larger than the rest of the elements, could frequently improve mesh quality metrics. On a negative side, however, this approach would be much more affected by the distortion of the fracture surface, since Gmsh requires polygons with more than 4 edges to be projected onto the same plane, and the optimizations are really computed for that plane of projection. The eventual reversion of this projection could completely negate any improvement to quality measures.

- In the process of robustly constructing tridimensional algorithms we have left little treatment on how to build 2D meshes with the presented methodology. It should be possible, however, to simplify the current implementation to support 2D DFNs. Most of the effort will be put on discarding some unnecessary data structures and algorithms of tridimensional nature: There is no need for defining polyhedral volumes or rolodexes; Surface meshing is trivial as the fracture surface is simply defined by one-dimensional elements inside each intersected face. Indeed most of the operations necessary are contemplated by how we have here proposed to intersect and refine the skeleton mesh up to Section 3.2.

- For convenience of input data, and since this project is built by NeoPZ library's developers, this implementation is currently limited to coarse meshes comprised exclusively of tridimensional topologies from the NeoPZ environment (cuboids, tetrahedra, pyramids and triangular prisms). This setup enjoys the comfort of easy creation of multiple input data for testing and debugging. We note, however, that all those topologies are immediately attributed the status of arbitrary polyhedral volumes, in the very first steps of our methodology. Thus, extending the range of valid input coarse meshes to arbitrary polyhedra can be done with relatively small effort.

- The exploration of different mesh tools implementations should also be worth of pursuit. We have taken Gmsh for its popularity, demonstrated reliability as a general finite element meshing tool, and stable and easy-to-use C++ API. However, in our current implementation (and hence also in this text) no attempt was directed into testing similar meshing algorithms from other projects. Some example of codes that have been applied in the current literature are the Los Alamos Grid Toolbox (LaGriT) and the Computational Geometry Algorithms Library (CGAL).

References

AHMED, E.; JAFFRÉ, J.; ROBERTS, J. E. A reduced fracture model for two-phase flow with different rock types. Elsevier BV, v. 137, p. 49–70, July 2017. DOI: 10.1016/j.matcom.2016.10.005. Available from: <https://doi.org/10.1016/j.matcom.2016.10.005>.

AKKUTLU, I. Y.; EFENDIEV, Y.; VASILYEVA, M. Multiscale model reduction for shale gas transport in fractured media. **Computational Geosciences**, v. 20, n. 5, p. 953–973, 2016. ISSN 15731499. DOI: 10.1007/s10596-016-9571-6. arXiv: 1507.00113.

ALEKSANDROV, A. D. **Convex Polyhedra**. Berlin/Heidelberg: Springer-Verlag, 2005. (Springer Monographs in Mathematics). ISBN 3-540-23158-7. DOI: 10.1007/b137434. Available from: <http://link.springer.com/10.1007/b137434>.

BADOUEL, D. An Efficient Ray-Polygon Intersection. In: GLASSNER, A. S. (Ed.). **Graphics Gems**. [S.l.]: Academic Press, 1995.

BERG, M. de; CHEONG, O.; KREVELD, M. van; OVERMARS, M. **Computational Geometry: Algorithms and Applications**. [S.l.: s.n.], 2001. v. 85, p. 175. ISBN 9783540779735. DOI: 10.2307/3620533.

BERRE, I.; BOON, W. M., et al. Verification benchmarks for single-phase flow in three-dimensional fractured porous media. **Advances in Water Resources**, Elsevier, v. 147, p. 103759, 2021.

BERRE, I.; DOSTER, F.; KEILEGAVLEN, E. Flow in Fractured Porous Media: A Review of Conceptual Models and Discretization Approaches. **Transport in Porous Media**, Springer Netherlands, v. 130, n. 1, p. 215–236, 2019. ISSN 15731634. DOI: 10.1007/s11242-018-1171-6. arXiv: 1805.05701. Available from: <https://doi.org/10.1007/s11242-018-1171-6>.

BOISSONNAT, J.-D.; YVINEC, M. **Algorithmic geometry**. [S.l.]: Cambridge university press, 1998.

BREDON, G. E. **Topology and Geometry**. [S.l.]: Springer-Verlag, 1993. v. 139, p. 557. ISBN 0387979263.

BUKAČ, M.; YOTOV, I.; ZUNINO, P. Dimensional model reduction for flow through fractures in poroelastic media. **ESAIM: Mathematical Modelling and Numerical Analysis**, v. 51, n. 4, p. 1429–1471, 2017. ISSN 12903841. DOI: 10.1051/m2an/2016069.

CHENG, S.-W.; DEY, T. K.; SHEWCHUK, J. R. **Delaunay mesh generation**. 1. ed. [S.l.]: Chapman and Hall/CRC, 2016. p. 410. ISBN 978-1-58488-731-7.

DE SIQUEIRA, D.; DEVLOO, P. R. B.; GOMES, S. M. A new procedure for the construction of hierarchical high order Hdiv and Hcurl finite element spaces. **Journal of Computational and Applied Mathematics**, v. 240, p. 204–214, Mar. 2013. ISSN 03770427. DOI: 10.1016/

j.cam.2012.09.026. Available from: <https://linkinghub.elsevier.com/retrieve/pii/S0377042712003998>.

DEVLOO, P. R. B. PZ: An object oriented environment for scientific programming. **Computer Methods in Applied Mechanics and Engineering**, v. 150, n. 1-4, p. 133–153, 1997. ISSN 00457825. DOI: 10.1016/S0045-7825(97)00097-2. Available from: <https://linkinghub.elsevier.com/retrieve/pii/S0045782597000972>.

DEVLOO, P. R. B.; TENG, W.; ZHANG, C. S. Multiscale hybrid-mixed finite element method for flow simulation in fractured porous media. **CMES - Computer Modeling in Engineering and Sciences**, v. 119, n. 1, p. 145–163, 2019. ISSN 15261492. DOI: 10.32604/cmes.2019.04812.

DFNWORKS. **dfnWorks Documentation - Release 2.2**. Los Alamos, USA: Subsurface Flow and Transport Team LANL, 2019.

DURÁN, O. Y. **Development of a Surrogate Multiscale Reservoir Simulator Coupled With Geomechanics**. 2017. PhD Thesis – Univerdade Estadual de Campinas - Unicamp. Available from: <http://repositorio.unicamp.br/handle/REPOSIP/331516>.

DURÁN, O. Y.; DEVLOO, P. R.; GOMES, S. M.; VALENTIN, F. A multiscale hybrid method for Darcy's problems using mixed finite element local solvers. **Computer Methods in Applied Mechanics and Engineering**, Elsevier B.V., v. 354, p. 213–244, 2019. ISSN 00457825. DOI: 10.1016/j.cma.2019.05.013. Available from: <https://doi.org/10.1016/j.cma.2019.05.013>.

EFENDIEV, Y.; HOU, T. Y. **Multiscale Finite Element Methods**. New York, NY: Springer New York, 2009. v. 4, p. 202. ISBN 978-0-387-09495-3. DOI: 10.1007/978-0-387-09496-0. arXiv: arXiv:1011.1669v3. Available from: <http://books.google.com/books?id=HIGOCRf6msoC%20http://link.springer.com/10.1007/978-0-387-09496-0>.

EPPSTEIN, D.; GOODRICH, M. T.; SUN, J. Z. The skip quadtree: A simple dynamic data structure for multidimensional data. In: PROCEEDINGS of the Annual Symposium on Computational Geometry. Pisa: [s.n.], 2005. p. 296–305. DOI: 10.1145/1064092.1064138. arXiv: 0507049 [cs].

ERBERTSEDER, K. et al. A coupled discrete/continuum model for describing cancer-therapeutic transport in the lung. **PLoS ONE**, v. 7, n. 3, 2012. ISSN 19326203. DOI: 10.1371/journal.pone.0031966.

ERHEL, J.; DE DREUZY, J. R.; POIRRIEZ, B. Flow simulation in three-dimensional discrete fracture networks. **SIAM Journal on Scientific Computing**, v. 31, n. 4, p. 2688–2705, 2009. ISSN 10648275. DOI: 10.1137/080729244.

FOURNO, A.; NGO, T. D.; NOETINGER, B.; LA BORDERIE, C. FraC: A new conforming mesh method for discrete fracture networks. **Journal of Computational Physics**, Elsevier Inc., v. 376, p. 713–732, 2019. ISSN 10902716. DOI: 10.1016/j.jcp.2018.10.005. Available from: <https://doi.org/10.1016/j.jcp.2018.10.005>.

FREY, P. J.; GEORGE, P.-L. **Mesh Generation. Application to finite elements**. 2. ed. [S.l.]: ISTE, 2008. ISBN 978-1-84821-029-5. Available from: <https://dl.acm.org/citation.cfm?id=1205626>.

GEMIGNANI, M. C. **Elementary Topology**. 2. ed. [S.l.]: Addison-Wesley, 1990. p. 270. ISBN 978-0201023404.

GEORGE, P.-L. Automatic mesh generation and finite element computation. **Handbook of Numerical Analysis**, v. 4, Part 2, p. 69–190, 1996. ISSN 15708659. DOI: 10.1016/S1570-8659(96)80003-2.

GEUZAINE, C.; REMACLE, J. F. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. **International Journal for Numerical Methods in Engineering**, v. 79, n. 11, p. 1309–1331, 2009. ISSN 00295981. DOI: 10.1002/nme.2579.

HARDER, C.; PAREDES, D.; VALENTIN, F. A family of Multiscale Hybrid-Mixed finite element methods for the Darcy equation with rough coefficients. **Journal of Computational Physics**, Elsevier Inc., v. 245, p. 107–130, 2013. ISSN 10902716. DOI: 10.1016/j.jcp.2013.03.019. Available from: <http://dx.doi.org/10.1016/j.jcp.2013.03.019>.

HYMAN, J. D.; GABLE, C. W.; PAINTER, S. L.; MAKEDONSKA, N. Conforming Delaunay Triangulation of Stochastically Generated Three Dimensional Discrete Fracture Networks: A Feature Rejection Algorithm for Meshing Strategy. **SIAM Journal on Scientific Computing**, v. 36, n. 4, a1871–a1894, Jan. 2014. ISSN 1064-8275. DOI: 10.1137/130942541. Available from: <http://epubs.siam.org/doi/10.1137/130942541>.

HYMAN, J. D.; KARRA, S., et al. DfnWorks: A discrete fracture network framework for modeling subsurface flow and transport. **Computers and Geosciences**, Elsevier, v. 84, p. 10–19, 2015. ISSN 00983004. DOI: 10.1016/j.cageo.2015.08.001. Available from: <http://dx.doi.org/10.1016/j.cageo.2015.08.001>.

JAFFRÉ, J.; MNEJJA, M.; ROBERTS, J. E. A discrete fracture model for two-phase flow with matrix-fracture interaction. **Procedia Computer Science**, v. 4, p. 967–973, 2011. ISSN 18770509. DOI: 10.1016/j.procs.2011.04.102.

JING, L.; STEPHANSSON, O. Discrete Fracture Network (DFN) Method. In: DEVELOPMENTS in Geotechnical Engineering. [S.l.: s.n.], 2007. v. 85. chap. 10, p. 365–398. DOI: 10.1016/S0165-1250(07)85010-3.

JUANES, R.; SAMPER, J.; MOLINERO, J. A general and efficient formulation of fractures and boundary conditions in the finite element method. **International Journal for Numerical Methods in Engineering**, Wiley Online Library, v. 54, n. 12, p. 1751–1774, 2002.

KARIMI-FARD, M.; DURLOFSKY, L. J.; AZIZ, K. An efficient discrete-fracture model applicable for general-purpose reservoir simulators. **SPE journal**, OnePetro, v. 9, n. 02, p. 227–236, 2004.

LA POINTE, P.; EIBEN, T.; DERSHOWITZ, W.; WADLEIGH, E. **Compartmentalization analysis using discrete fracture network models**. [S.l.], 1997.

LIMA, P.; DEVLOO, P. R. B.; VILLEGAS, J. B. Multi-scale meshing for 3D discrete fracture networks. In: IBERO-LATIN-AMERICAN Conference on Computational Methods for Engineering. Foz-do-Iguaçu: [s.n.], 2020. Available from: <https://github.com/PedroLima92/papers/blob/master/2020/cilamce/dfnmesh/2020_CILAMCE_Lima.et.al-2020_Multi-scale_Meshing_for_3D_DFNs.pdf>.

LO, S. H. **Finite element mesh generation**. [S.l.]: CRC Press, 2015. ISBN 978-1-4822-6687-0.

LO, S. H. Finite element mesh generation and adaptive meshing. **Progress in Structural Engineering and Materials**, v. 4, n. 4, p. 381–399, 2002. ISSN 1365-0556. DOI: 10.1002/pse.135.

LONG, J. C.; REMER, J.; WILSON, C.; WITHERSPOON, P. Porous media equivalents for networks of discontinuous fractures. **Water resources research**, Wiley Online Library, v. 18, n. 3, p. 645–658, 1982.

LUCCI, P. C. A. **Implementação de Simulador Numérico de Propagação Hidráu-lica de Fratura Plana em Meio Tridimensional Multicamadas**. 2015. s. 151. PhD Thesis – Universidade Estadual de Campinas - Unicamp.

MARTIN, V.; JAFFRÉ, J.; ROBERTS, J. E. Modeling fractures and barriers as interfaces for flow in porous media. **SIAM Journal on Scientific Computing**, SIAM, v. 26, n. 5, p. 1667–1691, 2005.

MARYŠKA, J.; SEVERÝN, O.; VOHRALÍK, M. Numerical simulation of fracture flow with a mixed-hybrid FEM stochastic discrete fracture network model. **Computational Geosciences**, v. 8, n. 3, p. 217–234, 2005. ISSN 14200597. DOI: 10.1007/s10596-005-0152-3.

MUSTAPHA, H. G23FM: A tool for meshing complex geological media. **Computational Geosciences**, v. 15, n. 3, p. 385–397, 2011. ISSN 14200597. DOI: 10.1007/s10596-010-9210-6.

ODEN, J. T.; BECKER, E. B.; CAREY, G. F. **Finite elements: An introduction**. 1. ed. [S.l.]: Prentice-Hall, 1981. v. 1. ISBN 0-13-317057-8.

PAREDES, D. **Novos Métodos de Elementos Finitos Multi-Escalas: Teoria e Aplicações**. 2013. PhD Thesis – Laboratório Nacional de Computação Científica - LNCC.

REDDY, J. N. **An Introduction to Nonlinear Finite Element Analysis**. 2. ed. [S.l.]: Oxford University Press, 2015. ISBN 978–0–19–964175–8.

SCHWENCK, N.; FLEMISCH, B.; HELMIG, R.; WOHLMUTH, B. I. Dimensionally reduced flow models in fractured porous media: crossings and boundaries. **Computational Geosciences**, v. 19, n. 6, p. 1219–1230, 2015. ISSN 15731499. DOI: 10.1007/s10596-015-9536-1.

SI, H. **TetGen: A quality tetrahedral mesh generator and a3D Delaunay triangulator**. Berlin: Wias Berlin, 2013. Available from: <http://wias-berlin.de/software/tetgen/1.5/doc/manual/manual.pdf>.

SNYDER, J. M.; BARR, A. H. Ray tracing complex models containing surface tessellations. **ACM Computer Graphics**, v. 21, n. 4, 1987.

SURANA, K. S.; REDDY, J. N. **The Finite Element Method for Boundary Value Problems**. 1. ed. [S.l.]: CRC Press, 2017. ISBN 9781498780506.

SUTHERLAND, I. E.; SPROULL, R. F.; SCHUMACKER, R. A. A characterization of ten hidden-surface algorithms. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 6, n. 1, p. 1–55, 1974.

VILLEGAS, J. B.; LIMA, P.; DEVLOO, P. R. B.; DURAN, O. A MULTI-SCALE MIXED METHOD FOR A TWO-PHASE FLOW IN FRACTURED RESERVOIRS CONSIDERING PASSIVE TRACER. In: CILAMCE - Proceedings ofthe XLII Ibero-Latin-American Congress on Computational Methods in Engineering. Rio de Janeiro, RJ: ABMEC, 2021.

WANG, Y.; MA, G.; REN, F.; LI, T. A constrained Delaunay discretization method for adaptively meshing highly discontinuous geological media. **Computers and Geosciences**, Elsevier Ltd, v. 109, January, p. 134–148, 2017. ISSN 00983004. DOI: 10.1016/j.cageo.2017.07.010. Available from: <https://doi.org/10.1016/j.cageo.2017.07.010>.

ZHANG, Y.; GONG, B.; LI, J.; LI, H. Discrete fracture modeling of 3D heterogeneous enhanced coalbed methane recovery with prismatic meshing. **Energies**, v. 8, n. 6, p. 6153–6176, 2015. ISSN 19961073. DOI: 10.3390/en8066153.

ZIENKIEWICZ, O.; TAYLOR, R.; ZHU, J. Z. **The Finite Element method: its basis and foundamentals**. 7. ed. Oxford, UK: Butterworth-Heinemann, 2013. ISBN 978-1-85617-633-0. Available from: <https://www.elsevier.com/books/the-finite-element-method-its-basis-and-fundamentals/zienkiewicz/978-1-85617-633-0>.

Appendix A

Example input file

The following code snippet is an example of the current expected input file for the program formatted as `json`.

```json
{
    "$schema": "./dfn_schema.json",
    "PZGenGrid":{
        "minX": [0.0, 0.0, 0.0],
        "Dimensions": [4.0, 4.0, 2.0],
        "MMeshType": "EHexahedral",
        "Nels": [4,4,2]
    },
    "TolDist": 0.09,
    "TolAngle": 0.001,
    "Fractures":[
        {
            "Index": 0,
            "Limit": "Erecovered",
            "MaterialID": 10,
            "Nodes":[
                [1.50, 1.70, 0.30],
                [3.55, 1.70, 0.30],
                [3.55, 1.70, 1.50],
                [1.50, 1.70, 1.50]
            ]
        },
        {
            "Index": 1,
            "Limit": "Erecovered",
            "MaterialID": 11,
            "Nodes":[
                [1.5,       0.25,       0.7],
                [2.38388,   0.616117,   0.7],
                [2.75,      1.5,        0.7],
                [2.38388,   2.38388,    0.7],
                [1.5,       2.75,       0.7],
                [0.616117,  2.38388,    0.7],
                [0.25,      1.5,        0.7],
                [0.616117,  0.616117,   0.7]
            ]
        }
    ]
}
```

Listing 1: Example input file.