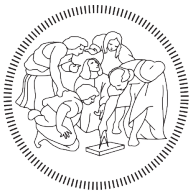
A C++ code to solve a Contact Mechanics problem with Coulomb Friction for slipping faults

Stefano Galati



POLITECNICO
MILANO 1863

Report for *Advanced Programming for Scientific Computing*
MSc in Mathematical Engineering
Computational Science and Computational Learning
Politecnico di Milano

July 2025

Supervisors:
Prof. Luca Formaggia
Prof.ssa Anna Scotti

Introduction

The main objective of this project is to address a contact mechanics problem involving a slipping fault between two bulk media, governed by a Coulomb friction law at the interface. This type of contact condition presents significant challenges, both theoretically and computationally, due to its non-smooth nature. The system's behavior changes at the fault and depends on the solution itself, introducing strong nonlinearities.

Several approaches have been proposed to tackle this problem, most commonly based on standard methods such as Lagrange multipliers or penalty techniques. A more recent alternative, developed by F. Chouly and Y. Renard, is a Nitsche-based method, which can be interpreted as a consistent form of penalization.

Chapter One covers the theoretical and numerical formulation of the problem. We introduce the contact algorithms used in each method and explain how the fault is modeled within the domain. Chapter Two focuses on the implementation: we provide guidance on how to use the code, outline its structure and design choices, and describe the implementation in detail. In Chapter Three, we present the results. Through a series of simple tests, we compare the performance, robustness, and effectiveness of the different methods.

Contents

Chapter 1

Problem Setting

1.1 Strong formulation

We consider two elastic bodies $\Omega^{(i)} \subset \mathbb{R}^d$ ($i \in \{m, s\}$, $d \in \{2, 3\}$), with Lipschitz boundaries $\partial\Omega^{(i)}$ defined as the disjoint union of three boundary portions:

$$\begin{aligned}
\Gamma_D^{(i)} &\coloneqq \{\mathbf{x} \in \partial\Omega^{(i)} : \mathbf{u}|_{\Gamma_D^{(i)}} = \mathbf{u}_D^{(i)}\} \\
\Gamma_N^{(i)} &\coloneqq \{\mathbf{x} \in \partial\Omega^{(i)} : \boldsymbol{\sigma}(\mathbf{u})\mathbf{n}|_{\Gamma_N^{(i)}} = \mathbf{g}^{(i)}\} \\
\Gamma_C^{(i)} &\coloneqq \text{portion that possibly comes into contact}
\end{aligned}$$

We make the following assumptions:

$$|\Gamma_D^{(i)}| > 0, \qquad \Gamma_D^s \Subset \partial\Omega \setminus \overline{\Gamma}_C^s, \qquad \Gamma_C^{eff} \Subset \Gamma_C^s \tag{1.1}$$

The first assumption guarantees that Korn's inequality holds on each boundary, the second assumption implies that the trace space on Γ_C^s does not "see" any boundary condition from Γ_D^s, while the third assumptions ensures that the support of the surface tractions on Γ_C^s is compactly embedded in Γ_C^s, begin Γ_C^{eff} the actual contacct zone. Moreover, we assume to be in the small-deformation hypothesis and that the two bodies have an initial gap g_0 along the interface $\Gamma_C^{(i)}$. The goal is to determine the displacement fields $\mathbf{u}^{(i)} : \Omega^{(i)} \to \mathbb{R}^d$ satisfying the equilibrium equations and boundary conditions:

$$\begin{cases}
-\nabla \cdot \sigma^{(i)}(\mathbf{u}^{(i)}) = \mathbf{b}^{(i)} & \text{in } \Omega^{(i)}, \quad (1.2a) \\
\mathbf{u}^{(i)} = \mathbf{u}_D^{(i)} & \text{on } \Gamma_D^{(i)}, \quad (1.2b) \\
\sigma^{(i)}(\mathbf{u}^{(i)})\mathbf{n}^{(i)} = \mathbf{f}^{(i)} & \text{on } \Gamma_N^{(i)}, \quad (1.2c)
\end{cases}$$

for $i \in \{m, s\}$, where $\sigma^{(i)}$ denotes the Cauchy stress tensor, given under linear elasticity as $\sigma^{(i)} = \mathbb{C}^{(i)} : \varepsilon(\mathbf{u}^{(i)})$, with $\mathbb{C}^{(i)}$ the fourth-order elasticity tensor and $\varepsilon(\mathbf{u}) = \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^\top)$ the linearized strain tensor.

In order to formulate the contact conditions, we first need to define the normal. Let $\mathbf{n}^{(i)}$ be the normal associated to each side of the interface. Within a biased formulation, for any $\mathbf{x} \in \Gamma_C^s$ let $\mathbf{y} \in \Gamma_C^m$ be the point that potentially comes into contact with $\mathbf{x}$. Since we impose the equilibrium conditions 1.2 in the actual configuration, we should define a map

$$\begin{aligned} \Pi : \Gamma_C^s &\to \Gamma_C^m \\ \mathbf{x} &\mapsto \mathbf{y} = \Pi(\mathbf{x}) \end{aligned}$$

that projects the points belonging to the master on the slave side. For other possible choices of this map, see Mlika 2018. In the context of small deformations such map is defined in the reference configuration and remains fixed, since reference and actual configurations can be identified. To express the contact conditions between two elastic bodies, we define the normal and tangential components of the stress tensor at the interface. Let $\boldsymbol{\sigma}(\mathbf{u}^s)$ denote the Cauchy stress tensor associated to the displacement $\mathbf{u}^s$. On the slave contact boundary, we define the normal and tangential components of the stress as:

$$\sigma_{\mathbf{n}} := (\boldsymbol{\sigma}(\mathbf{u}^s)\mathbf{n}^s) \cdot \mathbf{n}, \qquad \boldsymbol{\sigma}_{\mathbf{t}} := (\mathbf{I} - \mathbf{n} \otimes \mathbf{n})\, \boldsymbol{\sigma}(\mathbf{u}^s)\mathbf{n}^s,$$

where $\mathbf{n}^s$ is the outward unit normal to Ω^s, and $\mathbf{n}$ is the contact normal determined by the geometry and contact pairing mapping Π. Note that $\mathbf{n}$ and $\mathbf{n}^s$ are not necessarily equal in the hypothesis of finite deformations.

We define the initial gap between the contact surfaces as

$$g_0 := (\mathbf{y} - \mathbf{x}) \cdot \mathbf{n},$$

where $\mathbf{x} \in \Gamma_C^s$ and $\mathbf{y} = \Pi(\mathbf{x}) \in \Gamma_C^m$.

The jump in displacement across the interface is defined by

$$[\![\mathbf{u}]\!] := \mathbf{u}^s - \mathbf{u}^m \circ \Pi, \qquad [\![u_n]\!] := [\![\mathbf{u}]\!] \cdot \mathbf{n}.$$

With these definitions, the contact (Signorini-type) conditions can be written as:

$$\begin{cases} [\![u_n]\!] \leq g_0, & \text{(1.3a)} \\ \sigma_n \leq 0, & \text{(1.3b)} \\ ([\![u_n]\!] - g_0)\,\sigma_n = 0. & \text{(1.3c)} \end{cases}$$

These conditions enforce non-penetration (the relative normal displacement must not exceed the initial gap), compressive contact force (the normal stress is non-positive), and the complementarity condition that ensures contact force is active only when the bodies are touching.

To describe friction along the contact interface, we introduce the tangential displacement increment or *slip*, denoted by

$$[\![\mathbf{u}_t]\!] = (\mathbb{I} - \mathbf{n} \otimes \mathbf{n})[\![\mathbf{u}]\!],$$

where $(\mathbb{I} - \mathbf{n} \otimes \mathbf{n})$ is the tangential projection operator. The static Coulomb law can be expressed as:

$$\begin{cases} ||\sigma_t|| \leq -\mathcal{F}\sigma_n, & \text{(1.4a)} \\ [\![\mathbf{u}_t]\!] - \mathcal{F}\sigma_n ||[\![\mathbf{u}_t]\!]|| = 0 & \text{(1.4b)} \end{cases}$$

while the quasi-static Coulomb law can be obtained in 1.4 by replacing the tangential displacements with the tangential velocities, leading to an evolutionary problem. In such cases, an initial displacement $\mathbf{u}|_{t=0}$ has to be provided.

1.2 Weak formulation

To solve our problem we need to write it in a weak form. Different formulations will lead to different numerical methods, depending on how the contact conditions are rewritten and on how the inequalities are treated from a numerical point of view. Before diving into the weak formulation of the problem, we shall make precise the notation on the functional spaces that we will be using. We use the standard notation for the Sobolev space $H^s(\omega), s \geq 1$, where ω is a subset of $\Omega^{(i)}, i \in \{m, s\}$, with associated norm $||\cdot||_{s;\omega}$. We define the Sobolev broken space:

$$\mathbf{V}^{(i)} := (H^1(\Omega^{(i)}))^d$$
$$\mathbf{V} := \mathbf{V}^{\mathbf{m}} \times \mathbf{V}^{s}$$

and denote by $\mathbf{v} = (\mathbf{v}^m, \mathbf{v}^s) \in \mathbf{V}$ its elements. Moreover, let

$$\mathbf{W}^{(i)} \coloneqq (H^{\frac{1}{2}}(\Gamma_D^{(i)}))^d$$
$$\mathbf{W} \coloneqq \mathbf{W}^m \times \mathbf{W}^s$$

be the trace spaces for the Dirichlet boundaries, with dual $\mathbf{W}' = (\mathbf{W}^m)' \times (\mathbf{W}^s)'$ made of elements $\mathbf{w} = (\mathbf{w}^m, \mathbf{w}^s) \in \mathbf{W}'$. We denote as

$$\langle \mathbf{v}, \mathbf{w} \rangle_{\Gamma_D} \coloneqq \langle \mathbf{v}^m, \mathbf{w}^m \rangle_{\Gamma_D^m} + \langle \mathbf{v}^s, \mathbf{w}^s \rangle_{\Gamma_D^s}$$

the duality product for the trace operator on the Dirichlet boundary portions. The trace spaces for the contact boundary Γ_C^s are given by:

$$\mathbf{W}_C \coloneqq (H^{\frac{1}{2}}(\Gamma_C^s))^d, \qquad W_{C,n} \coloneqq \{\mathbf{w} \cdot \mathbf{n}^s : \mathbf{w} \in \mathbf{W}_C\}, \qquad \mathbf{W}_{C,t} \coloneqq \{(\mathbb{I} - \mathbf{n} \otimes \mathbf{n})\mathbf{w} : \mathbf{w} \in \mathbf{W}_C\}$$

Then we can define:

$$\mathbf{V}_0 \coloneqq \{\mathbf{v} \in \mathbf{V} : \langle \mathbf{v}, \mathbf{w} \rangle_{\Gamma_D} = 0 \quad \forall \mathbf{w} \in \mathbf{W}'\}$$
$$\mathbf{V}_D \coloneqq \{\mathbf{v} \in \mathbf{V} : \langle \mathbf{v}, \mathbf{w} \rangle_{\Gamma_D} = \langle \mathbf{u}_D, \mathbf{w} \rangle_{\Gamma_D} \quad \forall \mathbf{w} \in \mathbf{W}'\}$$
$$\mathbf{K} \coloneqq \{\mathbf{v} \in \mathbf{V}_D : \langle [\![v_n]\!] - g_o, w \rangle_{\Gamma_C^s} \leq 0 \quad \forall w \in W'_{C,n}\}$$

where $\mathbf{V}_0, \mathbf{V}_D$ are broken Sobolev spaces enforcing the Dirichlet boundary conditions, $\mathbf{K}$ is a closed, convex, non-empty set of admissible displacements strongly prescribing the contact conditions, and $\langle \cdot, \cdot \rangle_{\Gamma_C^s}$ represents the duality pairings for the trace operators on the contact boundary Γ_C^s. The broken norm associated to $\mathbf{V}$ is given by $||.||_{s;\Omega} = ||.||_{s;\Omega^m} + ||.||_{s;\Omega^s}$. For the definition of $\mathbf{K}$ to make sense we should assume (Kikuchi and Oden 1988) that $\Omega^{(i)} \in C^{1,1}$ so that, thanks to the second assumption in 1.1, we have:

$$\overline{\Gamma}_C^{(i)} \Subset \partial\Omega^{(i)} \Rightarrow H^{\frac{1}{2}}(\Gamma_C^{(i)}) \subset H^{\frac{1}{2}}_{0,0}(\partial\Omega^{(i)})$$

and we are thus allowed to work with $H^{\frac{1}{2}}(\Gamma_C^s)$ instead of the more complicated space $H^{\frac{1}{2}}_{00}(\Gamma_C^s)$. (Wohlmuth 2011)

A classical weak formulation of problem (1.2,1.3,1.4) is given by the following variational inequality (see Kikuchi and Oden 1988 for a formal derivation):

$$\begin{aligned} &\text{Find } \mathbf{u} \in \mathbf{K} \text{ s.t.} \\ &a(\mathbf{u}, \mathbf{v} - \mathbf{u}) + j(S(\mathbf{u}), \mathbf{v}) - j(S(\mathbf{u}), \mathbf{u}) \geq L(\mathbf{v} - \mathbf{u}) \qquad \forall \mathbf{v} \in \mathbf{K} \end{aligned} \tag{1.5}$$

where:

$$
\begin{aligned}
&a(\mathbf{v},\mathbf{w}) \coloneqq a_m(\mathbf{v}^m,\mathbf{w}^m) + a_s(\mathbf{v}^s,\mathbf{w}^s) && \forall \mathbf{v},\mathbf{w} \ \text{ in } \mathbf{V}\\
&a_i(\mathbf{v}^{(i)},\mathbf{w}^{(i)}) \coloneqq \int_{\Omega^{(i)}} \sigma(\mathbf{v}^{(i)}) : \varepsilon(\mathbf{w}^{(i)}) && i \in \{m,s\}\\
&L(\mathbf{v}) \coloneqq L_m(\mathbf{v}^m) + L_s(\mathbf{v}^s) && \forall \mathbf{v} \text{ in } \mathbf{V}\\
&L_i(\mathbf{v}^{(i)}) \coloneqq \int_{\Omega^{(i)}} \mathbf{b}^{(i)} \cdot \mathbf{v}^{(i)} + \int_{\Gamma_N^{(i)}} \mathbf{f}^{(i)} \cdot \mathbf{v}^{(i)} && i \in \{m,s\}\\
&j(S,\mathbf{v}) \coloneqq \langle S, ||[\![\mathbf{v}_t]\!]|| \rangle_{\Gamma_C^s} && \forall S \in W'_{C,n}, \forall \mathbf{v} \in \mathbf{V}
\end{aligned}
$$

The bilinear form $a(\mathbf{v},\mathbf{v})$ corresponds to the standard stiffness elasticity term, while the pairing $j(S,\mathbf{v})$ is a nonconvex, nondifferential, nonquadratic functional of the admissible displacement $\mathbf{v} \in \mathbf{K}$ that incorporates the virtual work done by the frictional forces. The frictional threshold depends on the chosen friction law, and is given by

$$S = S^T \tag{1.6}$$

in the case of Tresca (fixed, constant threshold) and

$$S(\mathbf{u}) = -\mathcal{F}\sigma_n(\mathbf{u}) \tag{1.7}$$

for the Coulomb friction law. If the contact threshold is regular enough, $j(\cdot,\cdot)$ can be written as:

$$j(S,\mathbf{v}) = \int_{\Gamma_C^s} S||[\![\mathbf{v}_t]\!]||$$

In the case with no friction ($\mathcal{F} = 0$), the analysis simplifies a lot. Since $j(0,\cdot) = 0$, thanks to Stampacchia's theorem, one can prove existence and uniqueness of a solution and that problem 1.5 is equivalent to the constrained minimization problem:

$$\text{Find } \mathbf{u} = \arg\min_{\mathbf{v}\in\mathbf{K}} J(\mathbf{v}) \tag{1.8}$$

where the following functional on $\mathbf{K}$ has been defined:

$$J(\mathbf{v}) \coloneqq \frac{1}{2}a(\mathbf{v},\mathbf{v}) - L(\mathbf{v}) \tag{1.9}$$

Since the Coulomb law does not directly come from a potential, one cannot write the frictional case as a minimization problem. However, in the case of Tresca threshold - known contact threshold S - problem 1.5 becomes a variational inequality of the second kind, and can thus be reformulated as a constrained minimization problem of

the following functional on $\mathbf{K}$:

$$\tilde{J}(\mathbf{v}) = \frac{1}{2}a(\mathbf{v},\mathbf{v}) + j(S,\mathbf{v}) - L(\mathbf{v}) \tag{1.10}$$

In the Tresca case, one can still prove the existence and uniqueness of a solution, being $j(\cdot,\cdot)$ convex and lower semi-continuous with respect to its second argument.(Kikuchi and Oden 1988; Chouly, Hild, and Renard 2023b)

The well-posedness of the frictional case with Coulomb law is still an open problem. Recently (Ballard and Iurlano 2023), an existence result has been proven for the bi-dimensional case, in the hypothesis of isotropic elasticity, for any value of friction coefficient; an example of non-existence has been provided for anisotropic elasticity. For the three-dimensional case, existence was proven only for small values of the friction coefficient (Chouly, Hild, and Renard 2023a), namely

$$\mathcal{F} \leq \frac{\sqrt{3-4\nu}}{2-2\nu}$$

Some results of uniqueness were proved for small values of the friction coefficient (Eck, Jarusek, and Krbec 2005), while examples of multiple solutions were found for large values of $\mathcal{F}$ e.g. in (Hild 2003).

1.2.1 Mixed formulation

The introduction of multipliers allows us to relax the constraints on the admissible displacement from the functional space $\mathbf{K}$ through the addition of the pressure unknown λ. The key idea is to characterize the convex set $\mathbf{K}$ in terms of a dual cone. Given the contact threshold S, we define the following trace spaces for the traction:

$$\begin{aligned}
&\boldsymbol{\Lambda}_C(S) \coloneqq \Lambda_{C,n} \times \boldsymbol{\Lambda}_{C,t}(S)\\
&\Lambda_{C,n} \coloneqq \{\lambda \in W'_{C,n} : \langle \lambda, w\rangle \geq 0 \quad \forall w \in W_{C,n},\ w \leq 0\ a.e.\}\\
&\boldsymbol{\Lambda}_{C,t}(S) \coloneqq \{\boldsymbol{\lambda} \in \mathbf{W}'_{C,t} : -\langle \boldsymbol{\lambda}_t, \mathbf{w}\rangle_{\Gamma_C^s} \leq \langle S, ||\mathbf{w}||\rangle_{\Gamma_C^s} \quad \forall \mathbf{w} \in \mathbf{W}_{C,t}\}
\end{aligned}$$

where $\Lambda_{C,n}$ is the dual cone of weakly nonpositive admissible normal tractions on Γ_C^s and $\boldsymbol{\Lambda}_{C,t}$ is the space for admissible tangential tractions on Γ_C^s. Notice that, since the latter depends on the contact threshold S, in the Coulomb friction case the traction space is unknown - i.e., it depends on the solution itself. Moreover, with abuse of notation, we have denoted as $\langle\cdot,\cdot\rangle_{\Gamma_C^s}$ the duality product for both the pairings $(W'_{C,n}, W_{C,n})$ and

$(\mathbf{W}'_{C,t}, \mathbf{W}_{C,t})$. Moreover, we will denote by

$$I_{\mathbf{X}}(\mathbf{x}) \coloneqq \begin{cases} 0 & \text{if } \mathbf{x} \in \mathbf{X}, \\ +\infty & \text{otherwise.} \end{cases}$$

the indicator function on the space $\mathbf{X}$.

To write the saddle-point problem involving the multipliers, one can proceed by using the principle of duality. The idea is to introduce an auxiliary variable $\boldsymbol{\eta} = (\eta_n, \boldsymbol{\eta}_t) \in \mathbf{W}_C$ and to define the functional

$$\Theta(\mathbf{v}, \boldsymbol{\eta}) = J(\mathbf{v}) + \langle S, ||[\![\mathbf{v}_t]\!] - \boldsymbol{\eta}_t|| \rangle_{\Gamma_C^s} + I_{K_0}([\![u_n]\!] - g_0 - \eta_n)$$

such that $\Theta(\mathbf{u}, \mathbf{0}) = \tilde{J}(\mathbf{u})$ and $\Theta(\mathbf{u}, \cdot)$ is convex. Therefore, the solution of the Tresca problem is the minimizer of $\Theta(\mathbf{u}, \mathbf{0})$ and, by applying the Fenchel-Legendre conjugate to $\Theta(\mathbf{u}, \cdot)$, one gets the following Lagrangian:

$$\mathcal{L}(\mathbf{u}, \boldsymbol{\lambda}) = J(\mathbf{u}) - b(\boldsymbol{\lambda}, \mathbf{u}) - I_{\Lambda_{C,n}}(\lambda_n) - I_{\boldsymbol{\Lambda}_{C,t}(S)}(\boldsymbol{\lambda}_t)$$

where

$$b(\lambda, \mathbf{u}) \coloneqq b_n(\lambda_n, u_n) + b_t(\boldsymbol{\lambda}_t, \mathbf{u}_t)$$
$$b_n(\lambda_n, u_n) \coloneqq \langle \lambda_n, [\![u_n]\!] - g_0 \rangle, \qquad b_t(\boldsymbol{\lambda}_t, \mathbf{u}_t) \coloneqq \langle \boldsymbol{\lambda}_t, [\![\mathbf{u}_t]\!] \rangle_{\Gamma_C^s}$$

in which the presence of the indicator functions for the spaces $\Lambda_{C,n}, \boldsymbol{\Lambda}_{C,t}$ enforces the constraints on the multipliers. By (sub)differentiation of the Lagrangian, we finally obtain the optimality system for the mixed formulation:

$$\begin{aligned} &\text{Find } (\mathbf{u}, \boldsymbol{\lambda}) \in \mathbf{V}_0 \times \boldsymbol{\Lambda}_C(S) \text{ s.t.} && && (1.11) \\ &a(\mathbf{u}, \mathbf{v}) - b(\boldsymbol{\lambda}, \mathbf{v}) = L(\mathbf{v}) && \forall \mathbf{v} \in \mathbf{V}_0 \\ &b(\boldsymbol{\mu} - \boldsymbol{\lambda}, \mathbf{u}) \geq 0 && \forall \boldsymbol{\mu} \in \boldsymbol{\Lambda_C} \end{aligned}$$

where we have sticked to the homogeneous Dirichlet case for the case of simplicity. Clearly, one could account for non-homogeneous boundary conditions with the usual techniques. For a more detailed derivation, see Wohlmuth 2011; Chouly, Hild, and Renard 2023a; Chouly, Hild, and Renard 2023b. In particular, by taking $S(\lambda_n) = -\mathcal{F}\lambda_n$, the solution space for the multipliers depends on the solution itself.

1.3 Numerical approximation

Moving towards the discretization of the problem with the Finite Element Method, we consider a bounded polytopal domain $\Omega \subset \mathbb{R}^d$ ($d = 2$ or 3) with a Lipschitz boundary, possibly containing internal interfaces representing faults. To discretize the problem, we introduce a conforming, shape-regular triangulation $\mathcal{T}_h$ of the domain into non-overlapping elements K such that $\overline{\Omega} = \bigcup_{K \in \mathcal{T}_h} \overline{K}$. For each subdomain $\Omega^{(i)}$, we define $\Omega_h^{(i)} := \{K \in \mathcal{T}_h : K \subset \Omega^{(i)}\}$ as the collection of elements composing a disjoint partition of $\Omega^{(i)}$.

To approximate the displacement field $\mathbf{u}$, we construct suitable finite-dimensional subspaces $\mathbf{V}_h \subset \mathbf{V}$, where $\mathbf{V}$ is the Sobolev space corresponding to the weak formulation of the continuous problem. In the case where the mesh $\mathcal{T}_h$ consists of simplicial elements - triangles in 2D or tetrahedra in 3D - we adopt piecewise linear Lagrangian finite elements ($\mathbb{P}_1$). Alternatively, if the domain is discretized using quadrilateral or hexahedral elements, we employ multilinear ($\mathbb{Q}_1$) elements.

The discrete displacement field $\mathbf{u}^h \in \mathbf{V}_h$ is then represented as a linear combination of basis functions associated with the finite element space. That is,

$$\mathbf{u}^h(\mathbf{x}) = \sum_{i=1}^{N} \mathbf{u}_i \, \boldsymbol{\varphi}_i(\mathbf{x}),$$

where $\{\boldsymbol{\varphi}_i\}_{i=1}^N$ denotes the vector-valued basis functions of the discrete space $\mathbf{V}_h$, and $\mathbf{u}_i \in \mathbb{R}^d$ are the nodal degrees of freedom representing the displacement at the mesh nodes.

Denoting with g_0^h an approximation of the gap function, and as $\mathbf{K}_h = \mathbf{K} \cap \mathbf{V}_h$ the discrete convex cone of admissible displacements, the discrete counterpart of problem 1.5 can be written as:

$$\begin{aligned} &\text{Find } \mathbf{u}^h \in \mathbf{K}_h \text{ s.t.} \\ &a(\mathbf{u}^h, \mathbf{v}^h - \mathbf{u}^h) + j(S(\sigma_n(\mathbf{u}^h)), \mathbf{v}^h) - j(S(\sigma_n(\mathbf{u}^h)), \mathbf{u}^h) \geq L(\mathbf{v}^h - \mathbf{u}^h) \qquad \forall \mathbf{v}^h \in \mathbf{K}_h \end{aligned} \tag{1.12}$$

In the case of non-matching meshes, one has to be precise on the definition of $\mathbf{K}_h$. Some possibilities are given in Chouly, Hild, and Renard 2023a.

In the following we present some different formulations, given in the aforementioned book, that impose the contact and friction conditions in a weak sense, relaxing the solution space to be the whole $\mathbf{V}_h$. The presented methods are those implemented in

the code and that we aim to compare.

1.3.1 Penalty formulation

The simplest model to approximate the contact problem in the context of Finite Elements is a standard penalty method. The Signorini conditions and the friction law are imposed through appropriate penalization terms:

$$\begin{aligned}&\text{Find } \mathbf{u}^h \in \mathbf{V}_h \text{ s.t., } \forall \mathbf{v^h} \in \mathbf{V}_h \\ &a(\mathbf{u}^h, \mathbf{v}^h) + \int_{\Gamma_C^s} \gamma_P([\![u_n^h]\!] - g_0^h)_+ [\![v_n^h]\!] + \int_{\Gamma_C^s} \gamma_P([\![\mathbf{u}_t]\!]^h)_{S^h(\mathbf{u}^h)} \cdot [\![\mathbf{v}_t]\!] = L(\mathbf{v}^h) \end{aligned} \tag{1.13}$$

where γ_P is the penalty parameter to be tuned appropriately and $S^h(\mathbf{u}^h)$ is an approximation of the contact threshold, e.g.,

$$S(\mathbf{u}) = -\mathcal{F}\sigma_n(\mathbf{u})$$

for the Coulomb law. The principal limitation of this formulation lies in its lack of consistency in both the weak and strong senses.

1.3.2 Nitsche method

A relatively recent approach for the weak enforcement of contact and friction conditions relies on a Nitsche-based formulation. Originally introduced by (Nitsche 1971) for the weak imposition of Dirichlet boundary conditions, this method has only more recently been extended to contact problems in finite elasticity. Owing to its structure, it is often described as a "consistent penalization" method: it introduces penalty-like terms that weakly enforce constraints that would otherwise require restricting the solution space. Like traditional penalty methods, it is simple to implement and computationally efficient, but it avoids the additional complexity associated with Lagrange multipliers.

The basic idea of the method is to rewrite conditions 1.3 and 1.4 in an equivalent form. We report here two crucial results from Chouly, Hild, and Renard 2023a.

Proposition 1 *Let $\gamma_N : \Gamma_C^s \to \mathbb{R}^+$ be a positive integrable function, $\Pi : \Gamma_C^m \to \Gamma_C^2 \in L^\infty(\Gamma_C^m)$ the projector for the contact pairing and $\mathbf{u}^{(i)}$ be regular enough on $\Omega^{(i)}$ such that 1.3 hold in $L^2(\Gamma_C^s)$. Then the contact conditions 1.3 are equivalent to:*

$$-\sigma_n(\mathbf{u}) = [[\![u_n]\!] - g_0 - \sigma_n(\mathbf{u})]_+ \tag{1.14}$$

where $\sigma_n(\mathbf{u})$ is defined above and $[\cdot]_+$ is the projection operator on $\mathbb{R}^+$.

Proposition 2 *Let $\gamma_N : \Gamma_C^s \to \mathbb{R}^+$ be a positive integrable function, $\Pi : \Gamma_C^m \to \Gamma_C^2 \in L^\infty(\Gamma_C^m)$ the projector for the contact pairing and $\mathbf{u}^{(i)}$ be regular enough on $\Omega^{(i)}$ such that 1.4 hold in $L^2(\Gamma_C^s)$. Then the friction conditions 1.4 are equivalent to:*

$$\boldsymbol{\sigma}_t(\mathbf{u}) = [[\mathbf{u}_t]] - \boldsymbol{\sigma}_t(\mathbf{u})]_{\mathcal{F}[\gamma_N u_n - \sigma_n(\mathbf{u})]_+} \tag{1.15}$$

where $\sigma_n(\mathbf{u})$ is defined above and $[\cdot]_a$, with $a > 0$, is the projection operator on the ball of radius a, i.e.

$$[\mathbf{x}]_a := \begin{cases} \mathbf{x} & \text{if } ||\mathbf{x}|| \leq a \\ a\frac{\mathbf{x}}{||\mathbf{x}||} & \text{otherwise} \end{cases}$$

Integrating by parts, rewriting the integrals on Γ_C^s in terms of the expressions 1.14 and 1.15 and adding some symmetrizing terms we obtain the Nitsche-based formulation:

$$\text{Find } \mathbf{u}^h \in \mathbf{V}_h \text{ s.t.} \tag{1.16}$$

$$a(\mathbf{u}^h, \mathbf{v}^h) - \int_{\Gamma_C^s} \frac{\theta}{\gamma_N} \sigma(\mathbf{u})_n \sigma(\mathbf{v})_n \tag{1.17}$$

$$+ \int_{\Gamma_C^s} \frac{1}{\gamma_N} [P^n(\mathbf{u}^h)]_+ P_\theta^n(\mathbf{v}^h) \tag{1.18}$$

$$+ \int_{\Gamma_C^s} \frac{1}{\gamma_N} [\mathbf{P}^t(\mathbf{u}^h)]_{S_h(\mathbf{u}^h)} \cdot \mathbf{P}_\theta^t(\mathbf{v}^h) = L(\mathbf{v^h}), \qquad \forall \mathbf{v}^h \in \mathbf{V}_h \tag{1.19}$$

where $\theta \in \{-1, 0, 1\}$ is the symmetrization parameter and the following quantities have been defined:

$$P^n(\mathbf{u}^h) := \gamma_N([[u_n]] - g_0^h) - \sigma_n(\mathbf{u}^h)$$
$$\mathbf{P}^t(\mathbf{u}^h) := \gamma_N[[\mathbf{u}_t]] - \boldsymbol{\sigma}_t(\mathbf{u}^h)$$
$$S_h(\mathbf{u}^h) = \mathcal{F}(\gamma_N([[u_n^h]] - g_0^h) - \sigma_n(\mathbf{u}^h))_+$$

In particular, $\theta = 0$ yields the equation arising from the suggested derivation, while $\theta = 1$ and $\theta = -1$ correspond to the symmetric and skew-symmetric modifications, respectively. The effect of parameter θ has been studied in (Chouly, Fabre, et al. 2016).

Notice that, since the previous propositions state an equivalence relation, the Nitsche formulation could be written also at the continuous level. However, such formulation was derived formally, i.e. without caring about the functional setting, and thus has no precise meaning at the continuous level. Moreover, we are able to substitute the original contact and friction conditions, which involve inequality constraints, with some non-smooth requests on the displacement. From a numerical viewpoint, we're shifting the implementation complexity from the need to handle the inequality constraints -

e.g. using an active set strategy - to the solution of a non-smooth system, which will require some care.

1.3.3 Augmented Lagrangian formulation

A main drawback of the use of multipliers to relax the constraints on the admissible displacement, as shown in section 1.2, is that this introduces constraints on the multipliers' solution space. In Chouly, Hild, and Renard 2023b the authors derive an optimality system by using an augmented Lagrangian technique presented in Rockafellar and Wets 1998 obtaining the following formulation:

$$
\begin{aligned}
&\text{Find } \mathbf{u}^h \in \mathbf{V}_h,\ \boldsymbol{\lambda}^h \in \mathbf{W}_h, \text{ such that, for all } (\mathbf{v}^h, \boldsymbol{\mu}^h) \in \mathbf{V}_h \times \mathbf{W}_h \\
&a(\mathbf{u}^h, \mathbf{v}^h) + \int_{\Gamma_C^s} \left(\gamma_L \left([\![u_n^h]\!] - g_0^h \right)_+ - \lambda_n^h \right) [\![v_n^h]\!] \\
&\qquad + \int_{\Gamma_C} \left(\gamma_L [\![\mathbf{u}_t^h]\!] - \boldsymbol{\lambda}_t^h \right)_{Sh(u_n^h, \lambda_n^h)} \cdot [\![\mathbf{v}_t^h]\!] = L(\mathbf{v}^h) \\
&\qquad - \int_{\Gamma_C^s} \frac{1}{\gamma_L} \left(\lambda_n^h + \left(\gamma_L \left([\![u_n^h]\!] - g_0^h \right)_+ - \lambda_n^h \right) \right) \mu_n^h \\
&\qquad - \int_{\Gamma_C^s} \frac{1}{\gamma_L} \left(\boldsymbol{\lambda}_t^h + \left(\gamma_L [\![\mathbf{u}_t^h]\!] - \boldsymbol{\lambda}_t^h \right)_{S_h(u_n^h, \lambda_n^h)} \right) \cdot \boldsymbol{\mu}_t^h = 0.
\end{aligned} \tag{1.20}
$$

where the discrete frictional threshold is now defined as:

$$
S_h(u_n^h, \lambda_n^h) = \mathcal{F}(\gamma_L([\![u_n^h]\!] - g_0^h) - \lambda_n^h)_+
$$

1.3.4 Faults approximation

The numerical treatment of faults - modeled as lower-dimensional interfaces with non-penetration or friction conditions - can be approached in different ways. One can either align the fault with the mesh (conforming discretization), embed the fault independently of the mesh (non-matching or unfitted methods), or enrich the approximation space locally (e.g., via XFEM or mortar methods). The choice depends on the desired accuracy, complexity of the geometry, and computational efficiency. We choose to use a conforming mesh with continuous elements defined in the whole domain, with doubled DOFs on the nodes lying on the interface to enable jumps (i.e. discontinuities in the displacement field) across the fault. Technical details on how those DOFs were doubled will be given in the next chapter.

Chapter 2

Implementation

This chapter examines the implementation of the problem presented in the first chapter. The implemented code allows to solve 3-dimensional problems where the geometry is a parallelepiped bulk which is entirely cut by a fault intersecting the top and the bottom boundary, possibly with a small inclination. The codebase is compiled with `CMake` and is used to generate a small static library `mycontactlib.a` that is used by the `main.cpp` source code. It supports Gmsh integration for the generation of inclined and/or unstructured meshes. The main library used is GetFEM++ with an internal binding to some common linear algebra libraries. Before compiling the code, the user needs to have those libraries installed in their system directories.

The project root directory is structured according to the following layout:

```
project_root/
    build/      -> CMake build options
    examples/   -> Input data and output folders for tests and user
    external/   -> The muparserx source code
    include/    -> Header files
    lib/        -> Compiled libraries
    src/        -> Source code of the library
```

2.1 Compile and Install

To install GetFEM++, first download the last stable version and unpack it:

```
wget http://download-mirror.savannah.gnu.org/releases/getfem/stable/getfem-5.4.4
tar xzf getfem-5.4.4.tar.gz
cd getfem-5.4.4/
```

Before starting the configuration process, make sure that the following libraries are installed on your system:

- `Qhull`: used for Delaunay triangulations;
- `SuperLU`: A direct solver for large sparse linear systems;
- `Mumps`: A direct solver for large sparse linear systems. Be sure to have the sequential version installed;
- `Lapack`, `BLAS`.

Then configure GetFEM++ with:

```
./configure --enable-shared --enable-mumps --enable-superlu
```

This supposes that the used libraries are installed in system directories. If you want to use different libraries installed in non-standard paths, you can specify it with the options:

```
--with-blas=<lib>             use BLAS library <lib>
--with-superlu=<lib>          use SuperLU library <lib>
--with-superlu-include-dir    directory in which the
                              superlu/sl*.h or just sl*.h   headers can be found
--with-mumps=<lib>            use MUMPS library <lib>
--with-mumps-include-dir
```

For more information on the configuration, run

```
./configure --help
```

To build `getfem`:

```
gmake -j<nprocs>
```

You can optionally check that the installation was successful:

```
gmake check
```

Finally, to install the library run:

```
sudo gmake install
```

Since the code supports mesh generation via the open-source software Gmsh, you may need to install that if you plan to use it. Other optional packages are `doxygen` and `graphviz` for generating documentation.

Once you have installed the GetFEM++ library, go to the PROJECT_ROOT and run:

```
cd build
cmake ..
```

with possible options (all defaulted to OFF):

```
-DEXAMPLES_VERBOSE=ON       to be verbose when testing the examples
-DEXAMPLES_USE_GMSH=ON      to test the examples using a mesh generated via gmsh
```

and then:

```
make -j<nprocs>
```

Optionally, you can execute the example both to check that the installation of the library was successful and to check that the output is correct, by doing:

```
ctest <--debug>
```

To see all the possible options run `ctest --help`.

To clean the result of compilation, you can do

```
make clean-all
```

Finally, to generate the documentation, run

```
make doc
```

2.2 How to use the code

The user may build its own problem modifying the datafile contained in `examples/user/`. The datafile is given in a .pot file containing the following parameters:

```
[domain]
angle = 2                   % Inclination degree (ignored if the mesh is built-in)
Lx = 1                      % Lenght of domain side along x
Ly = 2                      % Lenght of domain side along y
Lz = 2                      % Lenght of domain side along z
h = 0.25                    % Grid spacing
meshType = GT_PK(3,1)       % Element type: GT_QK(3,1) for hexahedron,
                            % GT_PK(3,1) for tetrahedron
order_u = 1                 % Order of Lagrangian Finite Elements for displacement
order_lm = 0                % Order of Lagrangian Finite Elements for LM
                            % (used iff method==augLM)
[../]
[it]
```

```
tol = 1.0e-14                % Tolerance for contact
maxit = 50                   % Maximum number of iteration
[../]
[contact]
method = nitsche             % nitsche penalty augLM
theta = 1                    % +1 0 -1 Nitsche
gammaN = 10                  % Nitsche parameter
gammaP = 1.0e5               % Penalty parameter
gammaL = 10                  % augmented Lagrange parameter
[../]
[time]
t0 = 0.0                    % Starting time
tend = 10.5                 % Ending time
dt = 0.5                    % Timestep
[../]
[physics]
E = 25000                    % Young modulus
nu = 0.25                    % Poisson ratio
bulkLoad = [0.,0.,-0.]       % Gravity
mu_friction = 0.4            % Friction coefficient
...
boundary conditions parameters
...
[../]
```

Listing 2.1: data.pot

For the imposition of boundary conditions, `muparserx` is used. Each boundary region is tagged with an ID ranging from 1 to 10 as shown in Figure 2.1. The user should give the list of tags where to apply a specific boundary region and the relative function (of vectorial field) written using the muparserx syntax, for instance:

```
...
regionLoad = [7,10]          % Neumann bd regions
bdLoad1 = {0,x[1]^2,-t}
bdLoad2 = {-3,0,sqrt(x[0])+0.1}
...
```

Listing 2.2: bd_data

will apply `bdLoad1` on region with tag 7 and `bdLoad2` on region with tag 10. The user may add comma-separated region tags to the list and add `bdLoad<i>` corresponding to the $i-th$ region in the list, **avoiding any blank space**.

The supported boundary conditions are:

- Dirichlet: use the fields `regionDisp` and `bdDisp<i>`;

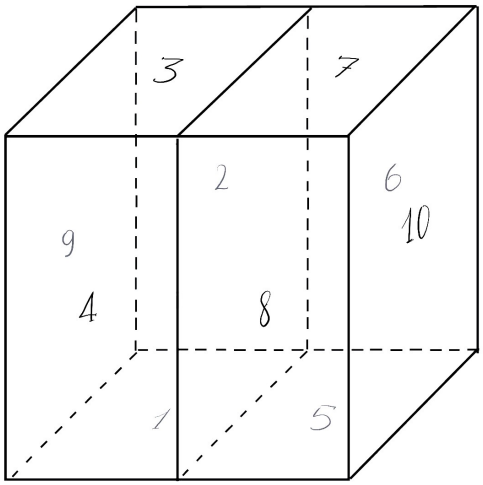


Figure 2.1: Tag IDs for boundary regions

- normal Dirichlet: use the fields `regionDispNormal` and `bdDispN<i>`;
- Neumann: use the fields `regionLoad` and `bdLoad<i>`;

Then, to run the executable, type:

```
./main <-v> <-m>
```

with the optional flags `-v` for verbosity and `-m` to generate the mesh via Gmsh. This will allow for unstructured meshes with possibly inclined faults. To see the other options available, run

```
./main -h
```

2.3 Core Components

Basically the `mycontactlib` library is made of classes built on top of classes and data structures of the GetFEM++ library and provides a modular and extensible code to solve problems of the type mentioned in the introduction to the current chapter. From a developer point of view, thanks to the design choices, it's easy to extend the code to implement other methods for solving the same problem. Here we present the main data structures and algorithms that constitute the code. The reference directories are `src/` and `include/`.

2.3.1 Core.hpp

This header file contains the `#include` directives for including the required `getfem` headers, type aliases and enum type definitions that are used throughout the code. It is included by all the other header files.

2.3.2 Parameters: `Params`

Basically a `struct` holding all the parameters of the problem, divided in internal `struct`s. It overloads the constructor operator to read the options passed to the command line and reads the datafile using a GetPot object stored as a member of the `struct` itself. Moreover, it overloads the output streaming operator for printing information on the parameters. Notice that it does not handle the data fields related to the boundary conditions, since those are treated by the `BCHandler` class. A `Params` object is first instantiated in the `main` function and then passed as argument (entirely or not) to the `Mesh` and `ContactProblem` constructors.

2.3.3 Mesh Handler: `Mesh` and `MeshBuilderStrategy`

`Mesh`

The `Mesh` class encapsulates the creation and management of the computational mesh used in the simulation. It serves as a high-level interface to a `getfem::mesh` object and abstracts away the details of mesh construction via a strategy pattern.

Responsibilities.

- Construct the mesh using a specified strategy (either built-in or Gmsh-based).
- Provide access to the underlying `getfem::mesh` object.
- Expose boundary and interior regions through region accessors.

Design Highlights. The mesh is constructed using a runtime-selected mesh builder strategy, implemented via the `MeshBuilderStrategy` abstract base class. Depending on the user-specified input (GMSH or domain identifier), the constructor chooses either a `GmshBuilder` or a `BuiltInBuilder`:

```
if (M_params.gmsh)
    M_meshBuilder = std::make_unique<GmshBuilder>(M_params.domain);
else
    M_meshBuilder = std::make_unique<BuiltInBuilder>(M_params.domain);
```

Listing 2.3: Mesh constructor code snippet

The selected strategy then constructs the mesh via the `construct()` method, which fills the internal `getfem::mesh` instance.

Key Methods.

- `const getfem::mesh& get() const`
  Returns a const reference to the internal GetFEM mesh.
- `const mesh_region region(RegionType r) const`
  Provides access to a named mesh region, used for applying conditions.
- `size_type dim() const`
  Returns the spatial dimension of the mesh.

Debugging Support. If the verbosity option is passed at runtime, the constructed mesh is written to a log file (`mesh_getfem_info.log`) to facilitate debugging and verification of the mesh generation phase.

`MeshBuilderStrategy`

The `MeshBuilderStrategy` class defines an abstract interface for constructing and initializing a finite element mesh. It follows the strategy design pattern, allowing for interchangeable mesh-building methods, encapsulated in concrete subclasses. This component is used directly by the `Mesh` class, which delegates mesh construction logic through the `construct()` method.

Purpose. The `MeshBuilderStrategy` provides a uniform interface for generating the computational mesh either internally (via programmatic logic) or externally (by importing a mesh from Gmsh). This is controlled by the command-line option `-m` passed at runtime.

Interface. The core public method is:

- `void construct(getfem::mesh& mesh) const;`
  Calls the protected virtual methods `buildMesh()` and `initRegions()` in sequence to generate the mesh and initialize associated regions.

Inheritance. Two concrete implementations are provided:

- `BuiltInBuilder`
  Constructs the mesh using domain parameters such as lengths and subdivisions. It also partitions the domain into logical regions such as `BulkLeft`, `BulkRight`, and `Fault`, and labels external boundaries. Everything is done using methods internal to `getfem`.

- `GmshBuilder`

  Automatically generates a Gmsh-compatible `.geo` file. Below is a simplified snippet showing how the `.geo` file is generated and written to disk:

```
std::ofstream geo("fractured_mesh.geo");
geo << "L = " << L << ";\n";
geo << "H = " << H << ";\n";
geo << "Point(1) = {0, 0, 0, lc};\n";
geo << "Point(2) = {L, 0, 0, lc};\n";
geo << "Point(3) = {L, H, 0, lc};\n";
geo << "Point(4) = {0, H, 0, lc};\n";
geo << "Line(1) = {1, 2};\n";
geo << "Line(2) = {2, 3};\n";
// Additional lines and physical groups...
geo.close();
```

Listing 2.4: Example for generating a .geo file

After creating the `.geo` file, the builder runs Gmsh as a subprocess to produce the mesh:

```
std::string cmd = "gmsh -3 mesh.geo -format msh2 -o mesh.msh";
int status = std::system(cmd.c_str());
if (status != 0) {
    throw std::runtime_error("Gmsh failed to generate mesh.");
}
```

Listing 2.5: Invoking Gmsh to generate .msh

This is done by using the private method `GmshBuilder::generate_mesh()`. The resulting `fractured_mesh.msh` file is then imported using GetFEM's Gmsh loader:

```
getfem::import_mesh_gmsh("fractured_mesh.msh", mesh, regmap);
```

Listing 2.6: Importing the mesh into GetFEM

Region labeling is then adjusted to resolve potential overlap between `Fault` and `Bulk` regions.

Extensibility. Additional mesh-building strategies can be implemented by subclassing `MeshBuilderStrategy` and overriding its two protected pure virtual methods:

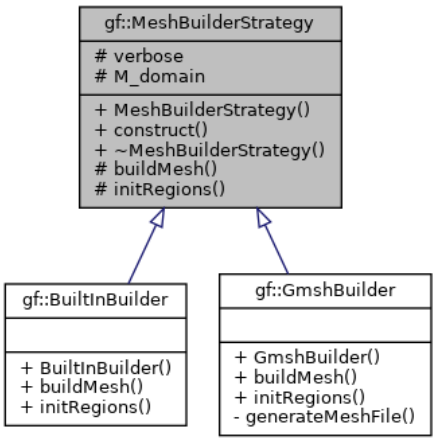


Figure 2.2: Inheritance diagram for `MeshBuilderStrategy`

- `virtual void buildMesh(getfem::mesh&) const = 0;`
- `virtual void initRegions(getfem::mesh&) const = 0;`

Usage in Mesh. The `Mesh` constructor instantiates the appropriate strategy object and invokes its `construct()` method. This design decouples mesh generation logic from the rest of the simulation framework and enables flexible integration with different mesh sources.

2.3.4 Boundary Condition Handler: `BCHandler` and `BC`

`BC`

Purpose. The `BCHandler` class is responsible for reading, storing, and managing boundary condition (BC) data associated with a given `getfem::mesh` object. It parses expressions from a `GetPot` input file and constructs BC objects of different types (`Dirichlet`, `Neumann`, or `Mixed`).

Core Data Members.

- `const getfem::mesh& M_mesh`: reference to the mesh over which BCs are applied.
- `BCListType M_BCList`: a map from `BCType` to lists of BC objects.
- `BCstringsType M_BCStrings`: stores the symbolic expressions as strings for output/logging.
- `muParserXInterface M_parser`: interface to the expression parser for evaluating symbolic BC data.

Interface. The key public methods include:

```
void readBC(const GetPot& gp);

const std::vector<std::unique_ptr<BC>>& Dirichlet() const;
const std::vector<std::unique_ptr<BC>>& Neumann() const;
const std::vector<std::unique_ptr<BC>>& Mixed() const;
```

Listing 2.7: Key methods in `BCHandler`

The method `readBC()` serves as the central entry point, delegating to the templated `read<>()` function for each BC type.

Reading BCs from Parameters. BC regions and expressions are defined in the input file (via `GetPot`) using naming conventions such as:

- `physics/regionDisp, physics/bdDisp1, ...` for Dirichlet BCs.
- `physics/regionLoad, physics/bdLoad1, ...` for Neumann BCs.
- `physics/regionDispNormal, physics/bdDispN1, ...` for Mixed BCs.

These are read and parsed with code like:

```
if constexpr (T == BCType::Dirichlet)
    regionsStr = datafile("physics/regionDisp", "");
...
std::ostringstream varname;
varname << "physics/bdDisp" << (i + 1);
std::string stringValue = datafile(varname.str().c_str(), "");
M_parser.set_expression(stringValue);
```

Listing 2.8: Reading Dirichlet BCs

Constructing BC Objects. For each region, the appropriate BC object is constructed using the parsed expression and stored in the corresponding container. An example for Dirichlet BCs:

```
auto bc = std::make_unique<BCDir>(
    M_mesh.region(regionsID[i]), regionsID[i], M_parser, T, n
);
M_BCList[T].emplace_back(std::move(bc));
```

Listing 2.9: Building Dirichlet BCs

Here, `n` denotes the mean normal of a boundary face, computed via:

```
base_small_vector n = M_mesh.mean_normal_of_face_of_convex(it.cv(), it.f());
```

Design Notes. The use of `std::unique_ptr<BC>` ensures safe memory management, while the `unordered_map<BCType, ...>` structure provides efficient lookup and separation of BC types. The templated design of `read<>()` minimizes code duplication and leverages compile-time dispatch for type-specific logic.

`BC`

The abstract class `BC` provides a common interface for all boundary conditions in the framework. It stores the target mesh region, the boundary function (as a `VectorFunctionType`), the region ID, and the normal vector of the associated boundary face.

Purpose. The main role of this class is to enable polymorphic access to boundary condition data via the pure virtual methods `type()` and `name()`.

Inheritance. Three concrete subclasses implement specific boundary types:

- `BCDir` for Dirichlet conditions
- `BCNeu` for Neumann conditions
- `BCMix` for Mixed (normal/tangential) conditions

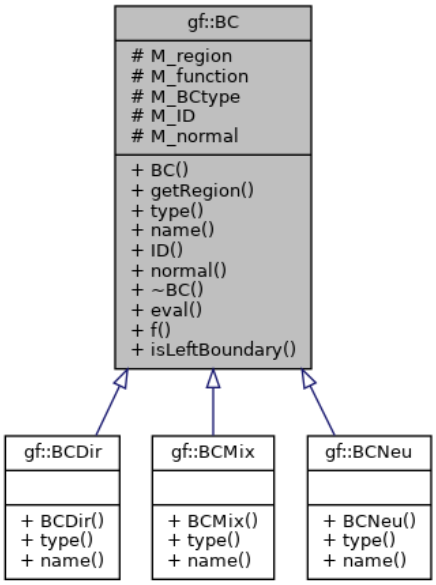


Figure 2.3: Inheritance diagram for the `BC` class

Each subclass overrides the `type()` method to return the corresponding `BCType` and assigns a unique name based on the tag of the region where it is imposed. All classes share the same evaluation interface:

```
base_small_vector eval(const base_node& x, scalar_type t) const;
```

which may be used for debugging purposes.

This method allows the user to evaluate the BC function at any spatial point and time.

The class also includes a utility method `isLeftBoundary()` based on the predefined list of region IDs. This will be used during the assembly phase.

Design Notes. The hierarchy ensures extensibility and clean separation of concerns, allowing new boundary types to be added with minimal disruption.

2.3.5 Finite Element Space Manager: `FEMManager`

The `FEMManager` class encapsulates the creation and configuration of finite element method (FEM) spaces required by the simulation.

Purpose. This class provides a central place to initialize and access the mesh-based finite element spaces for various physical quantities, including displacement, stress, right-hand side terms, and Lagrange multipliers. It simplifies the management of FEM spaces by grouping related functionality into a single component.

Responsibilities.

- Initialize the `getfem::mesh_fem` objects over the provided mesh.
- Set the finite element types for each field using the descriptors provided in the `Numerics` configuration object.
- Provide access to the configured FEM spaces via dedicated getter methods.

Code Design. The class internally stores four `mesh_fem` instances:

- `M_mfU`: FEM space for the displacement field.
- `M_mfStress`: FEM space for the stress tensor, with a fixed Q-dimension of 3×3.
- `M_mfRhs`: FEM space for body loads or other right-hand side contributions.
- `M_mfLM`: FEM space for Lagrange multipliers, used for enforcing constraints when the Augmented Lagrange method is adopted.

All FEM spaces are initialized from the same mesh, and the actual finite element types are assigned using:

```
M_mfU.set_finite_element(pfU);
M_mfStress.set_finite_element(pfStress);
M_mfStress.set_qdim(3,3);
M_mfRhs.set_finite_element(pfRhs);
M_mfLM.set_finite_element(pfLM);
```

These descriptors are determined at runtime by querying the FEM type strings in the `Numerics` object via `getfem::fem_descriptor`.

The class adheres to single responsibility and low coupling principles: it only concerns itself with FEM setup, leaving usage and assembly to other components (e.g., the assembler or solver modules).

2.3.6 The `ContactProblem` Class

Purpose and Responsibilities. The `ContactProblem` class is the central component of the codebase. It orchestrates the overall setup and solution of the contact mechanics problem. Its responsibilities include:

- Managing references to essential components such as the mesh, boundary conditions, finite element spaces, and contact enforcement strategy.
- Hosting and configuring a `getfem::model` object, which acts as the main container for all variables, parameters, and equations. This model is then used by GetFEM to perform the assembly and solution processes.
- Providing an interface for initializing, assembling, solving, and exporting the simulation.

Key Members. The most significant member is the `getfem::model` object `M_model`, which collects all variables, data, and equations required by GetFEM to define and solve the problem in its high-level *Generic Weak Form Language* (GWFL). Additional members include:

- `M_mesh`: a reference to the mesh object.
- `M_params`: global simulation parameters.
- `M_BC`: a `BCHandler` that manages boundary conditions.
- `M_FEM`: a `FEMManager` that defines and stores all finite element spaces.
- `M_integrationMethod`: the numerical integration method used for assembly.

- `M_contactEnforcement`: a polymorphic pointer to the chosen contact enforcement strategy.

Initialization. The constructor initializes all internal references and objects based on the input `Mesh` and `Params`. The method `init()` performs key setup tasks:

- Loads boundary condition data from the parameter file.
- Defines finite element spaces for all variables (displacement, stress, etc.).
- Configures the integration method using GetFEM descriptors.
- Selects and instantiates the contact enforcement strategy (e.g., Nitsche, Penalty, Augmented Lagrangian) based on user input.

```
void ContactProblem::init() {
  M_BC.readBC(M_params.datafile);
  M_FEM.setMeshFem(M_params.numerics, M_mesh.get());

  auto ppi = getfem::int_method_descriptor(M_params.numerics.integration);
  M_integrationMethod.set_integration_method(M_mesh.get().convex_index(), ppi);

  if (M_params.contact.method == "nitsche")
    M_contactEnforcement = std::make_unique<NitscheContactEnforcement>(...);
  else if (...) { /* other strategies */ }
}
```

Listing 2.10: The `init()` method

Assembly Procedure. The `assemble()` method is responsible for preparing the weak formulation of the contact problem by populating the internal `getfem::model` object with all necessary variables, data, macros, and weak form terms. While this method does not perform the actual numerical assembly (which is triggered later during the solver phase), it defines the full variational problem through symbolic representations (using the GWFL) that GetFEM interprets and assembles internally.

The method begins by defining the primary unknowns of the problem - the displacement fields on the left and right sides of the contact interface - using:

```
M_model.add_fem_variable("uL", M_FEM.mf_u());
M_model.add_fem_variable("uR", M_FEM.mf_u());
```

More details about this choice are provided in the next section. Here, the finite element space used for displacement is retrieved from the `FEMManager` attribute `M_FEM`, which was initialized in `init()`.

Next, scalar physical parameters such as the Lamé coefficients (`lambda`, `mu`) and friction coefficient (`mu_fric`) are added to the model using `getfem::add_initialized_scalar_data`. These parameters are read from the global `Params` object, which centralizes all simulation data.

The method then defines a set of symbolic expressions known as *macros*. These macros are string-based formulas recognized by GetFEM's internal weak form interpreter. They define complex tensor operations, such as stress tensors, tractions, and various projections of displacement jumps across the contact interface. For example, the normal component of the displacement jump is defined as:

```
M_model.add_macro("un_jump", "uL - Interpolate(uR,neighbor_element) . n");
```

Macros enable expressive, compact, and flexible representation of terms in the variational formulation, making the weak form both human-readable and computationally interpretable.

The core of the assembly process involves the addition of *bricks*, which are the modular components of GetFEM's weak form engine. Bricks represent contributions to the global system matrix or right-hand side vector. Two types of bricks are commonly used, even though one could also define his own brick (see GetFEM++ User's Documentation):

- **Predefined bricks**, such as the elasticity brick, which models isotropic linear elasticity:

```
getfem::add_isotropic_linearized_elasticity_brick(
    M_model, M_integrationMethod, "uL", "lambda","mu", RegionType::BulkLeft);
```

- **Generic bricks**, such as nonlinear or linear term bricks, where arbitrary expressions can be inserted, including stabilization or contact enforcement terms:

```
getfem::add_nonlinear_term(
    M_model, M_integrationMethod,
    "eps * uL . Test_uL + eps * uR . Test_uR", BulkLeft);
```

The enforcement of contact conditions is delegated to the polymorphic `M_contactEnforcement` attribute, which points to an instance of a derived class of `ContactEnforcementStrategy`. This design enables flexible switching between penalty, Nitsche, or augmented Lagrangian approaches. Each strategy is responsible for injecting its own weak form contributions (i.e., contact constraints) into the model using the same macro/bricks infrastructure.

Boundary conditions are imposed using information stored in the `BCHandler` attribute `M_BC`. Dirichlet, Neumann, and mixed (normal Dirichlet) conditions are imposed via specialized bricks:

- `getfem::add_source_term_brick` for Neumann conditions;
- `getfem::add_Dirichlet_condition_with_multipliers` for standard Dirichlet constraints;
- `getfem::add_normal_Dirichlet_condition_with_multipliers` for enforcing only normal components.

For each boundary region, the associated prescribed function is interpolated over the finite element mesh using `getfem::interpolation_function` and then added as `getfem::initialized_fem_data`.

In summary, the `assemble()` method symbolically encodes the full variational formulation into the `getfem::model` through a structured and extensible pipeline. This symbolic model is then passed to a solver, which performs the actual numerical assembly and solution of the resulting linear or nonlinear system.

Time Integration and Solution. The `solve()` method executes the actual numerical solution of the contact problem over time. It performs a simple time-stepping loop using the parameters defined in `M_params.time`, where at each time step, updated Neumann boundary conditions are interpolated from user-defined time-dependent functions and injected into the model using `getfem::set_real_variable`. The method then invokes `getfem::standard_solve(...)` which triggers GetFEM's internal nonlinear or linear solver to assemble and solve the system previously defined in `assemble()`. Notice that, based on the information given during the addition of the GetFEM bricks, `getfem::standard_solve(...)` internally chooses the best option for both the nonlinear algorithm and the linear solver to be used among the different possibilities provided by its interface with Mumps (which is used by default) and superLU. Iterative convergence is monitored using a `gmm::iteration` object configured with user-defined tolerances. Upon convergence (or failure), results are exported via `exportVtk()` for visualization. Notice that what we are solving in time is an independent problem for each time step. We chose a static Coulomb friction law and we are neglecting inertial terms, thus no time derivatives are involved in the equations. The dependence on time is only given in the change of boundary condition and/or forcing term with time, and the solution from a time instant is used as an initial guess for the solution of the problem at the next time instant. Hence we decided not to stop the simulation if a converged is not reached for a a certain time step, since some initial time steps, though physically

incorrect, may be useful for the following timesteps to reach convergence. In summary, this method thus encapsulates both the transient logic and the final execution of the symbolic weak formulation that was set up earlier.

Custom VTK Export. The `exportVtk()` method is responsible for exporting simulation data into the VTK format, enabling visualization of displacement and stress fields. While GetFEM++ includes native functions to export data in VTK, those functions do not directly support the combination of multiple displacement variables into a single cohesive output. In this problem, the displacement fields on the two sides of the contact interface, `uL` and `uR`, need to be merged into one unified displacement vector for meaningful post-processing and visualization. To achieve this, the method first filters and offsets the degrees of freedom on each side to avoid overlapping contributions at the fault interface, ensuring a clear spatial separation in the exported geometry. It then interpolates the stress tensors on both sides, converts them into the Voigt notation format, and matches the stress values to the corresponding nodal points, carefully handling shared nodes along the fault. The mesh connectivity is reconstructed with proper node ordering, and a slight geometric offset is applied to visually distinguish the two domains. Finally, the method writes all this information - including node coordinates, cell connectivity, displacement vectors, and stress data - to a custom VTK file using ASCII format, making it compatible with common visualization tools like ParaView. After the export process completes, the original displacement variables are restored to their previous state to preserve the integrity of the model for subsequent computations.

2.3.7 Import and Export of the Solution

To support various features such as initialization from a previous run or computation of errors, the code provides two utility methods: `exportCSV()` and `importCSV()`. These functions allow for exporting and importing the solution vector `U` to and from a plain text file in CSV format.

The method `exportCSV()` writes the displacement field at a given time step to a file. Each line corresponds to the displacement components (u_x, u_y, u_z) of a single node, preceded by a header line indicating the time step. If the time step is zero, the file is opened in write mode; otherwise, it is appended to preserve data across multiple time steps.

Conversely, `importCSV()` reads the displacement field corresponding to a specific time step from a CSV file. It searches for the block of values marked by a line of the form `t = i`, where `i` is the time index, and reads until an empty line or the end of the file.

These functions are employed in the `solve()` method under different runtime configurations:

- **Initialization:** When the `-i` (resp. `-ir`) flag is set, the solution is imported (resp. exported) to (resp. from) files named `initSolutionLeft.txt` and `initSolutionRight.txt`.
- **Reference Export:** When the `-r` flag is passed, the computed solution is exported to `refSolutionLeft.txt` and `refSolutionRight.txt` for later comparison.
- **Testing and Convergence:** When the `-t` flag is used, the code imports the reference solutions and compares them to the current simulation result to compute the error, enabling convergence and accuracy assessments.

This mechanism enables the use of continuation strategies and robust testing pipelines without requiring manual intervention, while maintaining clear separation between simulation phases.

2.3.8 Treatment of the Discontinuity

The numerical treatment of the fault interface is built upon a conforming mesh, meaning that the discretization does not contain hanging nodes or non-matching elements across the interface; rather, the mesh is topologically consistent and continuous across the entire domain. In GetFEM++, the `getfem::mesh_fem` object associated with the displacement field is defined globally over all elements of the mesh. Although I aim to use continuous finite elements (e.g., P_1 or Q_1) for the displacement fields to maintain physical realism, capturing the mechanical discontinuity introduced by the fault requires a relaxation of continuity across that interface.

To accommodate this, I explicitly define two distinct mesh regions - `BulkLeft` and `BulkRight` - in the `Mesh` class. These regions cover the subdomains on either side of the fault, and crucially, they overlap along the interface, meaning that nodes on the fault belong to both regions (you can check the `mesh_getfem_info.log` file running your executable with the `-v` option). However, by assigning the variables `uL` and `uR` exclusively to `BulkLeft` and `BulkRight`, respectively, each side is allowed to have independent degrees of freedom on the shared nodes, thereby modeling a displacement discontinuity in an otherwise conforming mesh. This approach necessitates a clear distinction between "left" and "right" parts of the domain, both in geometry and in assembly logic.

When constructing the model, I associate the standard elasticity bricks (such as linear

elasticity terms or internal forces) and boundary condition bricks with their respective regions: `uL` terms are assembled over `BulkLeft`, and `uR` terms over `BulkRight`. Since I'm using a biased formulation, i.e., the constraints are imposed only on one side of the fault, the integrals are computed only on the faces belonging to such side. To define a region of codimension one in `getfem`, one needs to add the faces of a convex (element) to such region. Thus, the idea behind the code was to create the `Fault` region by adding the faces lying on the interface that belong to convexes of the `BulkLeft` region only. This way, when defining a brick passing `Fault` as region, the assembly will be performed only on the left side of the interface. Finally, since the test functions of the displacement on the other side are defined on the same physical points but have different DOFs, I need to use the GetFEM's transformation `Interpolate(u, neighbor_element)`.

A subtle but critical detail arises from this formulation: although the finite element method nominally assigns DOFs continuously across elements, splitting them between two regions may cause the global stiffness matrix to exhibit near-singular behavior due to the artificial duplication of DOFs at the interface. To circumvent this issue and stabilize the system matrix, I include negligible "stabilizer terms" during assembly through weak artificial stiffness contributions (scaled by 10^{-20}) that serve to anchor the degrees of freedom without affecting the physical solution. These stabilizers are purely technical and act as regularization artifacts, ensuring numerical robustness while faithfully preserving the intended discontinuity across the fault interface.

2.3.9 Implemented methods: the `ContactEnforcementStrategy` class

Purpose. The `ContactEnforcementStrategy` class hierarchy provides an abstraction for enforcing the contact constraints, specifically the non-penetrability and the friction conditions described in the first chapter.

Class Structure and Inheritance.

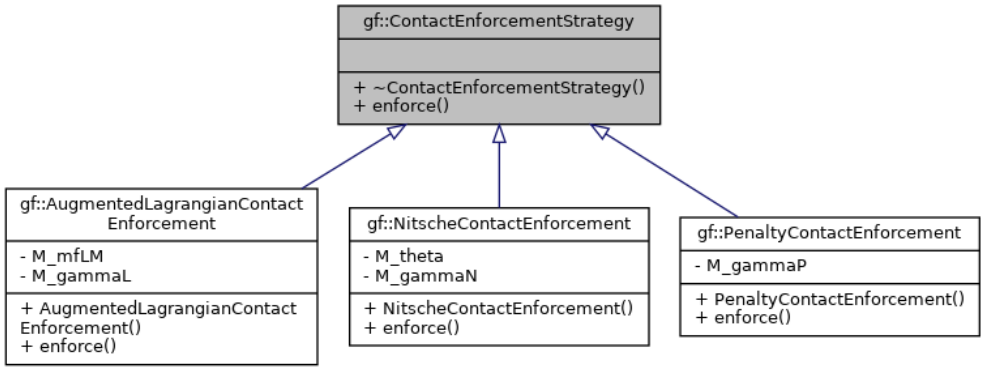


Figure 2.4: Inheritance diagram for the `ContactEnforcementStrategy` class

The class hierarchy consists of the following elements:

- `ContactEnforcementStrategy`: Abstract base class defining the interface for contact enforcement.
- `NitscheContactEnforcement`: Implements the Nitsche method.
- `PenaltyContactEnforcement`: Implements the penalty method.
- `AugmentedLagrangianContactEnforcement`: Implements the Augmented Lagrangian method.

Interface Description

```
class ContactEnforcementStrategy {
public:
    virtual ~ContactEnforcementStrategy() = default;

    virtual void enforce(getfem::model& md,
                         const getfem::mesh_im& im) const = 0;
};
```

Listing 2.11: Interface of `ContactEnforcementStrategy`

All concrete strategies override the `enforce` method, which injects bricks and macros into the GetFEM model to impose contact conditions and is called by the `assemble()` method of the `ContactProblem` class.

Design Considerations. The strategy pattern is used here to decouple the enforcement logic from the rest of the simulation. This promotes modularity and extensibility, enabling users to switch between different enforcement methods with minimal code modification. Future implementations might include the addition of different methods for imposing the contact conditions. Each method encapsulates specific parameters (e.g., penalty constants or Lagrange multipliers) and their associated terms for the model.

Nitsche Contact Enforcement

The Nitsche method allows for weak imposition of contact conditions through additional terms in the variational formulation, maintaining symmetry and consistency.

Key Parameters.

- `M_theta`: Governs symmetry of the Nitsche formulation (1 0 or -1).
- `M_gammaN`: Penalty scaling parameter for contact stiffness.

Method Outline. The following terms are injected into the GetFEM model:

- Linear stress term: ensure stress transfer, arising from integration by parts:

```
std::cout << " Adding linear stress brick...";
getfem::add_linear_term(
    md,
    im,
    "- theta/gammaN * sig_u_nL * sig_v_nL", // expression
    Fault, // region
    false, // symmetric
    false, // coercive
    "linear_stress",
    false // check
);
std::cout << "done.\n";
```

Listing 2.12: Linear stress brick for Nitsche method

- Nonlinear KKT term: impose the non-penetration condition:

```
std::cout << " Adding KKT condition brick...";
getfem::add_nonlinear_term(
    md,
    im,
    "1/gammaN * pos_part(Pn_u) * Pn_v_theta",
    Fault,
    false, // symmetric
    false, // coercive
    "KKTbrick"
);
std::cout << "done.\n";
```

Listing 2.13: KKT condition brick for Nitsche method

- Nonlinear friction terms: impose Coulomb-law friction condition:

```
std::cout << " Adding Coulomb friction brick...";
getfem::add_nonlinear_term(
    md,
    im,
    "(1/gammaN) * (proj_Pt1_u * Pt1_v_theta + proj_Pt2_u * Pt2_v_theta)",
    Fault,
    false, // symmetric
    false, // coercive
    "CoulombBrick"
);
std::cout << "done.\n";
```

Listing 2.14: Coulomb friction brick for Nitsche method

Penalty Contact Enforcement

The penalty method introduces a soft constraint by penalizing the violation of contact conditions via artificial stiffness.

Key Parameter.

- `M_gammaP`: Penalty parameter, must be sufficiently large to approximate the constraint.

Method Outline. Contact enforcement proceeds by injecting nonlinear terms that penalize normal and tangential gaps:

```
std::cout << " Adding KKT condition brick...";
getfem::add_nonlinear_term(
    md,
    im,
    "gammaP * pos_part(un_jump) * vn_jump",
    Fault,
    false,
    false,
    "KKTbrick"
);
std::cout << "done.\n";
```

Listing 2.15: KKT condition brick for Penalty method

```
std::cout << " Adding Coulomb friction brick...";
getfem::add_nonlinear_term(
    md,
    im,
    "gammaP*(proj_ut1_jump * vt1_jump + proj_ut2_jump * vt2_jump)",
    Fault,
    false,
    false,
    "CoulombBrick"
);
std::cout << "done.\n";
```

Listing 2.16: Coulomb friction brick for Penalty

This method is straightforward to implement and efficient, but is too permissive with respect to the parameter `gammaP`, which has to be tuned carefully.

Augmented Lagrangian Enforcement

This method combines Lagrange multipliers with penalty terms to provide accurate enforcement with improved numerical conditioning compared to pure penalty methods.

Key Elements

- `M_mfLM`: Mesh finite element for the Lagrange multipliers.
- `M_gammaL`: Augmentation parameter (penalty-like).

Method Outline. The method adds:

- Nonlinear KKT and Coulomb terms.
- Multiplier variable (`mult`).
- Bricks for contact conditions and constraints on the multiplier:

```
std::cout << " Adding KKT condition brick...";
getfem::add_nonlinear_term(
    md,
    im,
    "pos_part(Pn_u) * vn_jump",
    Fault,
    false,
    false,
    "KKTbrick"
);
std::cout << "done.\n";
```

Listing 2.17: Coulomb friction brick for augmented Lagrangian

```
std::cout << " Adding Coulomb friction brick...";
getfem::add_nonlinear_term(
    md,
    im,
    "proj_Pt1_u * vt1_jump + proj_Pt2_u * vt2_jump",
    Fault,
    false,
    false,
    "CoulombBrick"
);
std::cout << "done.\n";
```

Listing 2.18: Coulomb friction brick for augmented Lagrangian

```
std::cout << " Adding Lagrange multiplier term for normal gap...";
getfem::add_nonlinear_term(
    md,
    im,
    " - 1/gammaL * (lambdan + pos_part(Pn_u)) * mun",
    Fault,
    false,
```

```
    false,
    "LM_NormalGapBrick"
);
std::cout << "done.\n";

std::cout << " Adding Lagrange multiplier term for tangential gap...";
getfem::add_nonlinear_term(
    md,
    im,
    " -1/gammaL * (lambdat1 + proj_Pt1_u) * mut1 + (lambdat2 + proj_Pt2_u)
        * mut2",
    Fault,
    false,
    false,
    "LM_TangentialGapBrick"
);
std::cout << "done.\n";
```

Listing 2.19: Multiplier constraints bricks

This approach provides more accurate satisfaction of constraints without the high stiffness penalties of the basic penalty method, at the cost of the overhead due to the use of a multiplier.

Chapter 3

Examples and Results

In this chapter, we present a series of numerical experiments aimed at comparing the performance and behavior of the three proposed methods. The focus is not on assessing accuracy against a known exact solution - since such a solution is not available - but rather on evaluating the methods in terms of their robustness, consistency, and enforcement of problem-specific properties.

The comparison is carried out along the following axes:

- **Conservation checks and enforcement of the contact constraints**: We verify whether each method preserves physically meaningful quantities (such as mass or momentum) and how accurately the contact conditions are satisfied in the computed solution.
- **Sensitivity to the parameters**: We investigate how the solutions change in response to variations in the internal parameters specific to each method (e.g., penalty parameters), and identify ranges where the methods remain reliable.
- **Convergence of the Newton method**: We study the nonlinear solver performance by monitoring the convergence of the Newton iterations, both in terms of iteration counts and residual decay.
- **Sensitivity to the initial condition**: We explore how the choice of initial guess for the nonlinear solver affects the convergence and the final solution.

Through these tests, we aim to highlight not only the strengths and limitations of each method, but also their practical suitability for solving the underlying contact problem.

For this tests we will use first-order hexahedral elements with linear Lagrangian elements ($\mathbb{Q}^1$) for the displacement, with the following fixed parameters:

```
[domain]
Lx = 2                          % Lenght of domain side along x
Ly = 4                          % Lenght of domain side along y
Lz = 4                          % Lenght of domain side along z
h = 0.25                        % Grid spacing
meshType = GT_QK(3,1)           % Element type
order_u = 1                     % Order of Lagrangian Finite Elements for displacement
order_lm = 1                    % Order of Lagrangian Finite Elements for LM
[../]
[it]
tol = 1.0e-14                   % Tolerance for contact
maxit = 50                      % Maximum number of iteration
[../]
[physics]
E = 25000                       % Young modulus
nu = 0.25                       % Poisson ratio
bulkLoad = [0.,0.,-0.]          % Gravity
mu_friction = 0.4               % Friction coefficient
...
boundary conditions parameters
...
[../]
```

Listing 3.1: Simulation parameters

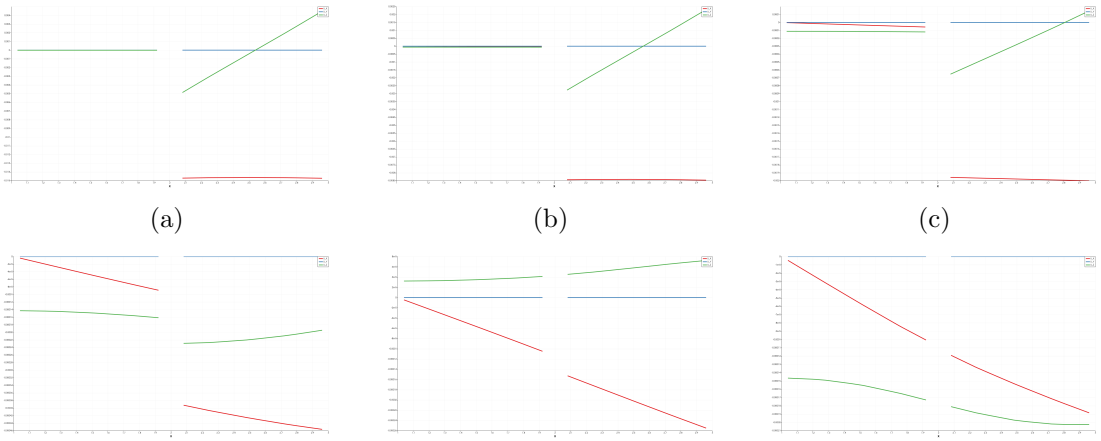
We impose a constant compressive load $\mathbf{t}_x = [-3, 0, 0]$ along the x-axis, pushing the right side of the bulk along negative x, with a liensarly increasing tangential load imposed only on the top right face that pushed the right portion of the bulk toward negative z, that is, $\mathbf{t}_z = [0, 0, -t]$. The bottom face remains fixed (homogeneous Dirichlet) while the left face of the left bulk side is constrained only in the x direction.

The key parameter that controls the weak enforcement of the contact conditions, namely the non-penetrability and the Coulomb friction conditions, is the penalty parameter.

3.1 Penalty method

We consider $\gamma_P = 1.\text{e}\{1, 2, 3, 4, 5, 6\}$. For the first three choices of γ_P, the Newton solver converges quadratically in one or two iterations, but stress and displacement continuity along x is not enforced. In particular, the left bulk is practically not affected at all by the loads and by friction, being displacement and stress identically null. For $\gamma_P = 1.\text{e}4$ constraints start to be visible in the solution. The case $\gamma_P = 1.\text{e}6$ shows the best enforcement of the contact conditions, though it is highly sensitive to the initial

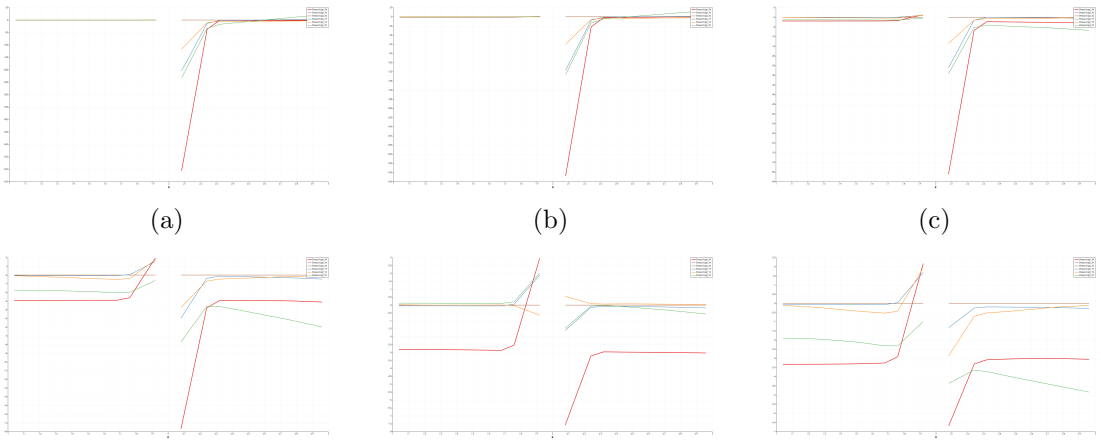
condition and starts showing some issues in the non-linear solver. The case $\gamma_P = 1.e5$ seems to be a good compromise between robustness and respect of the conservation of crucial quantities.



(a) (b) (c)

(d) (e) (f)

Figure 3.1: Continuity of the displacement (u_x in red, u_y in blue, u_z in green) along the x-axis at t=6: (a) $\gamma_P = 1.0e1$, (b) $\gamma_P = 1.0e2$, (c) $\gamma_P = 1.0e3$, (d) $\gamma_P = 1.0e4$, (e) $\gamma_P = 1.0e5$, (f) $\gamma_P = 1.0e6$.



(a) (b) (c)

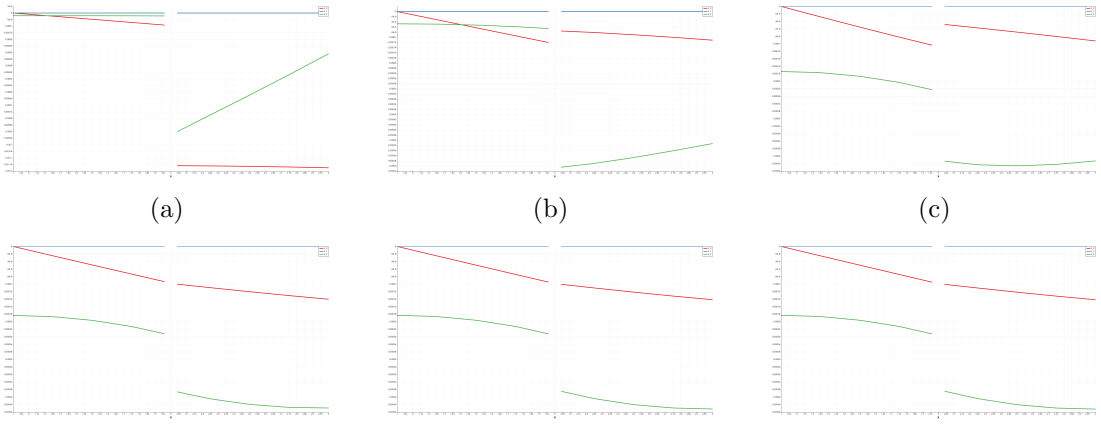
(d) (e) (f)

Figure 3.2: Continuity of the stress (S_{xx} in red, S_{yy} in blue, S_{zz} in green, S_{yz} in orange, S_{xz} in brown, S_{xy} in purple), along the x-axis at t=6: (a) $\gamma_P = 1.0e1$, (b) $\gamma_P = 1.0e2$, (c) $\gamma_P = 1.0e3$, (d) $\gamma_P = 1.0e4$, (e) $\gamma_P = 1.0e5$, (f) $\gamma_P = 1.0e6$.

3.2 Nitsche Method

We still consider $\gamma_N = 1.e\{1, 2, 3, 4, 5, 6\}$. Already for $\gamma_N = 1.e2$ we have a good fulfillment of displacement and stress continuity, which gets more realistic for increasing γ_N.

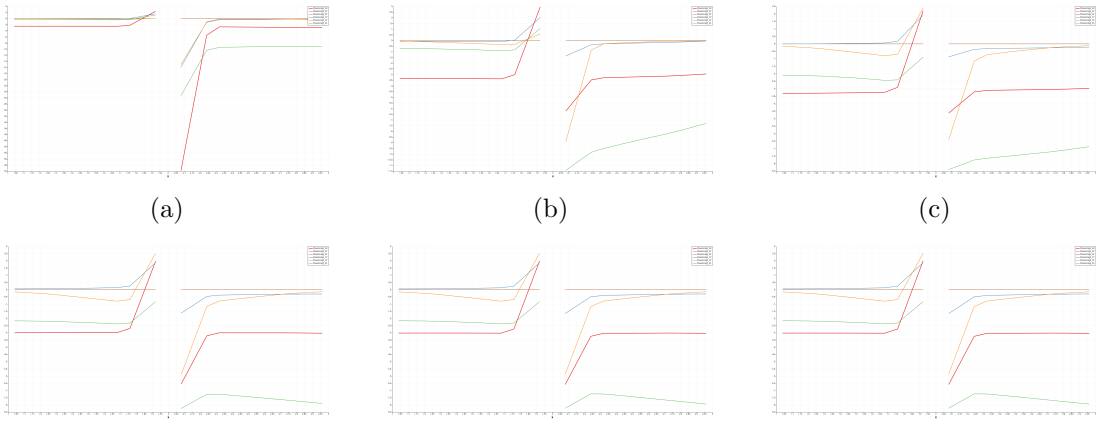
However, even with very small values of γ_N, the Newton solver has some convergence troubles. For $\gamma_N = 1.e\{5, 6\}$, the solver needs some nonphysical initial timesteps to start converging. A good compromise between accurate contact enforcement and nonlinear solver stability is given by values comprised between $\gamma_N = 1.e4$ and $\gamma_N = 1.e5$, in agreement with the results obtained in Chouly, Fabre, et al. 2016.



(a) (b) (c)

(d) (e) (f)

Figure 3.3: Continuity of the displacement (u_x in red, u_y in blue, u_z in green) along the x-axis at t=9.5: (a) $\gamma_N = 1.0e1$, (b) $\gamma_N = 1.0e2$, (c) $\gamma_N = 1.0e3$, (d) $\gamma_N = 1.0e4$, (e) $\gamma_N = 1.0e5$, (f) $\gamma_N = 1.0e6$.



(a) (b) (c)

(d) (e) (f)

Figure 3.4: Continuity of the stress (S_{xx} in red, S_{yy} in blue, S_{zz} in green, S_{yz} in orange, S_{xz} in brown, S_{xy} in purple), along the x-axis at t = 9.5: (a) $\gamma_N = 1.0e1$, (b) $\gamma_N = 1.0e2$, (c) $\gamma_N = 1.0e3$, (d) $\gamma_N = 1.0e4$, (e) $\gamma_N = 1.0e5$, (f) $\gamma_N = 1.0e6$.

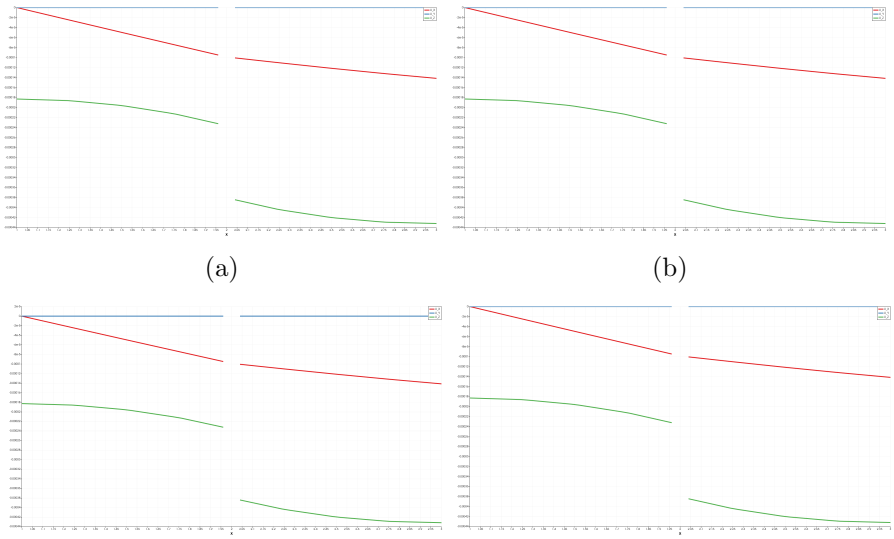
Considering the case $\gamma_N = 1.e5$, using $\theta = 0, 1, -1$ we get similar results in terms of convergence of the Newton solver, begin the case $\theta = 0$ more robust w.r.t. the initial solution:

```
...                              ...                              ...
--method: nitsche                --method: nitsche                --method: nitsche
--theta = 0                      --theta = 1                      --theta = -1
...                              ...                              ...
Solving the problem...           Solving the problem...           Solving the problem...
t = 0   converged in 25 iter     t = 0   Warning:  did not        t = 0   Warning:  did not
                                          converge in 50 iter              converge in 50 iter
t = 0.5 converged in 1 iter      t = 0.5 converged in 28 iter     t = 0.5 converged in 28 iter
t = 1   converged in 1 iter      t = 1   converged in 1 iter      t = 1   converged in 1 iter
t = 1.5 converged in 1 iter      t = 1.5 converged in 1 iter      t = 1.5 converged in 1 iter
t = 2   converged in 1 iter      t = 2   converged in 2 iter      t = 2   converged in 2 iter
t = 2.5 converged in 3 iter      t = 2.5 converged in 3 iter      t = 2.5 converged in 3 iter
t = 3   converged in 3 iter      t = 3   converged in 3 iter      t = 3   converged in 3 iter
t = 3.5 converged in 2 iter      t = 3.5 converged in 2 iter      t = 3.5 converged in 2 iter
t = 4   converged in 3 iter      t = 4   converged in 3 iter      t = 4   converged in 3 iter
t = 4.5 converged in 3 iter      t = 4.5 converged in 3 iter      t = 4.5 converged in 3 iter
t = 5   converged in 3 iter      t = 5   converged in 4 iter      t = 5   converged in 4 iter
t = 5.5 converged in 3 iter      t = 5.5 converged in 3 iter      t = 5.5 converged in 3 iter
t = 6   converged in 3 iter      t = 6   converged in 4 iter      t = 6   converged in 4 iter
t = 6.5 converged in 3 iter      t = 6.5 converged in 3 iter      t = 6.5 converged in 3 iter
t = 7   converged in 3 iter      t = 7   converged in 4 iter      t = 7   converged in 4 iter
t = 7.5 converged in 3 iter      t = 7.5 converged in 3 iter      t = 7.5 converged in 3 iter
t = 8   converged in 4 iter      t = 8   converged in 4 iter      t = 8   converged in 4 iter
t = 8.5 converged in 5 iter      t = 8.5 converged in 5 iter      t = 8.5 converged in 5 iter
t = 9   converged in 2 iter      t = 9   converged in 3 iter      t = 9   converged in 3 iter
t = 9.5 converged in 2 iter      t = 9.5 converged in 2 iter      t = 9.5 converged in 2 iter
```

Figure 3.5: Newton iterations for $\theta = 0$ (left), $\theta = 1$ (center), $\theta = -1$ (right).

3.3 Augmented Lagrange Multipliers

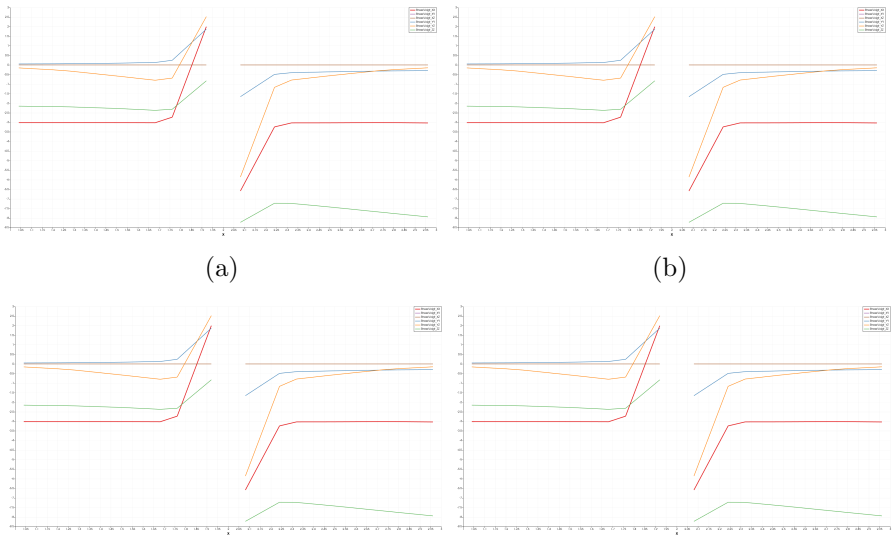
Using as γ_L the same values as before, for $\gamma_L = 1.e1, 2$ the Newton solver has difficulty to converge, even performing several step control with a line search algorithm. Moreover, for all the values of γ_L, it is very sensitive to the initial solution, needing for a proper initialization (e.g. with the solution at $t = 0$ coming from the Nitsche method). For γ_P, Newton converges quadratically after at most 3 iterations, while for $\gamma_P = 1.4{,}5$, it performs some line search steps in order to reach convergence. For $\gamma_L = 1.e6$, the solver has a similar behaviour to those of the other methods, needing some initial unphysical timesteps before starting converging. Whenever it converges, it displays excellent conservation properties, being the displacement and stress continuity along x preserved throughout the whole simulation.



(a) (b)

(c) (d)

Figure 3.6: Continuity of the displacement (u_x in red, u_y in blue, u_z in green) along the x-axis at t = 9.5: (a) γ_L = 1.0e3, (b) γ_L = 1.0e4, (c) γ_L = 1.0e5, (d) γ_L = 1.0e6.



(a) (b)

(c) (d)

Figure 3.7: Continuity of the stress (S_{xx} in red, S_{yy} in blue, S_{zz} in green, S_{yz} in orange, S_{xz} in brown, S_{xy} in purple), along the x-axis at t = 9.5: (a) γ_L = 1.0e3, (b) γ_L = 1.0e4, (c) γ_L = 1.0e5, (d) γ_L = 1.0e6.

3.4 Simulation test

To give an idea of the simulation, the parameters in 3.1 produce the following result:

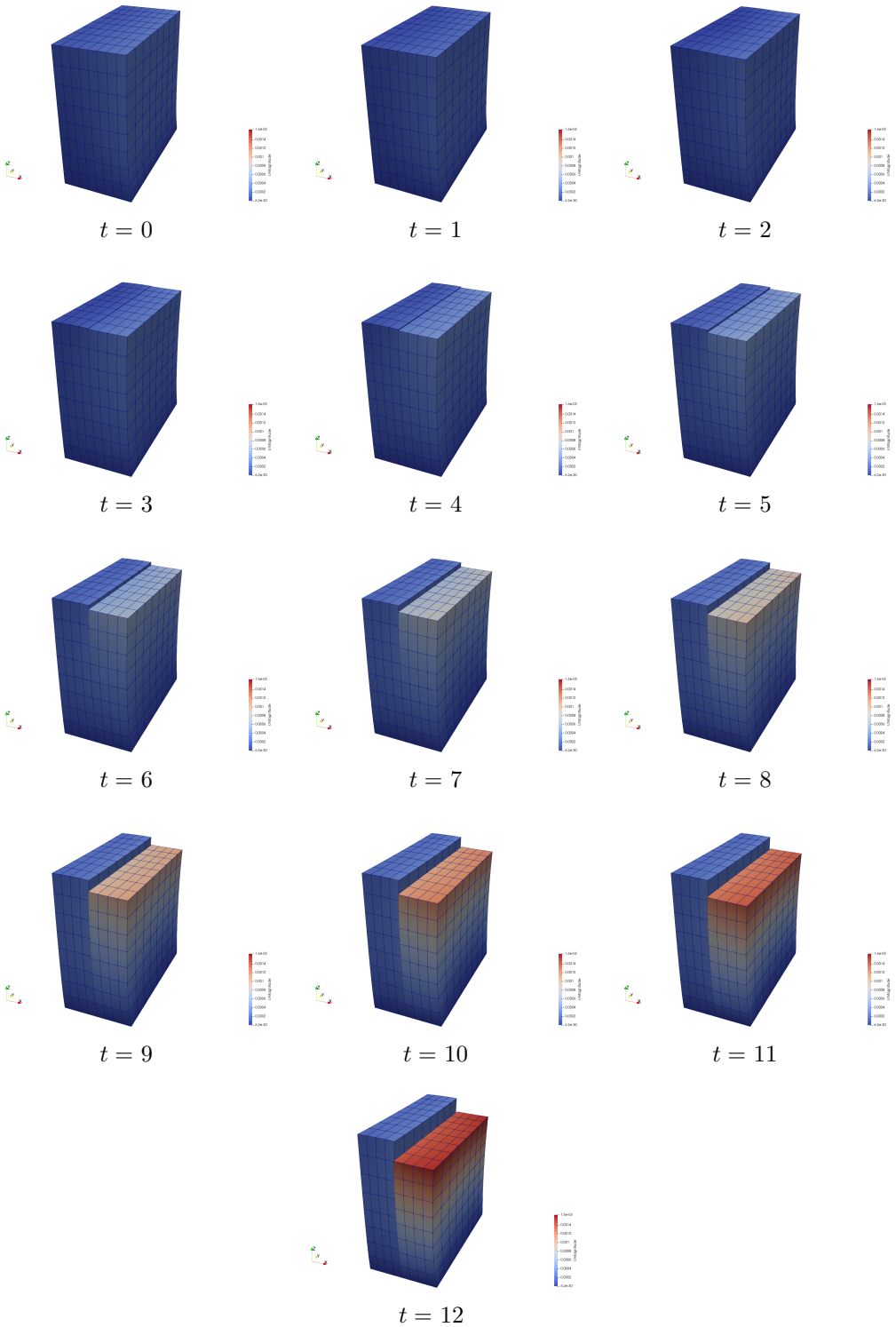


Figure 3.8: Output of the simulation with the parameters given in Fig. 3.1.

For the first timesteps (up to $t = 3$), the tangential load is not strong enough w.r.t.

the compressive traction, thus the fault remains *stick* between the two bulk regions. In the next timestep of the simulation, the right bulk starts slipping due to the increase of tangential load, and the displacement starts displaying a discontinuity at $x = 0$ (corresponding to the fault interface) in the z-component. The left bulk portion is pushed down as well due to the effect of friction.

Conclusion

The work presented in this project culminated in the development of a compact, modular library built on top of GetFEM++ for simulating three-dimensional contact mechanics problems in linear elasticity. The library is designed to handle simple domains entirely intersected by a fault and supports general configurations, including inclined faults and unstructured grids, thanks to the optional integration with Gmsh.

A central design goal was extensibility. To that end, the code leverages the strategy design pattern, enabling the seamless addition of new boundary condition types, numerical methods, or alternative mesh generation tools. Each module follows the single responsibility principle, ensuring clarity, modularity, and ease of maintenance.

At runtime, the code supports a range of configuration options through command-line flags, allowing it to be used both for numerical simulation and testing purposes. In particular, one can initialize a solution from a previous simulation, facilitating continuation strategies or addressing sensitivity issues in Newton's method. While performance is adequate for basic benchmark scenarios, tackling more complex and large-scale systems would necessitate parallelization, which falls beyond the scope of this project.

The numerical experiments conducted confirm the strengths and limitations of the three considered methods:

- The **penalty method** is robust with respect to both the initial guess and parameter tuning. Although it exhibits poor enforcement of the contact constraints, it converges rapidly, making it an excellent candidate for generating initial conditions for more accurate solvers.
- The **Nitsche method** achieves a better balance, offering improved enforcement of the contact conditions and better conservation properties, while retaining convergence behavior similar to the penalty method.
- The **augmented Lagrange multiplier method**, while highly accurate in imposing constraint, is sensitive to the initial condition and often requires multiple

line search steps to achieve convergence.

3.5 Future developments

Future directions for this work include the parallelization of the current implementation to enable large-scale simulations, the integration of additional numerical methods for contact enforcement, and the extension of the current framework to handle more complex geometries and physics.

Bibliography

Ballard, P. and F. Iurlano (2023). *Optimal existence results for the 2d elastic contact problem with Coulomb friction*. arXiv: 2303.16298 [math.AP]. URL: https://arxiv.org/abs/2303.16298.

Bathe, K. J. (1996). *Finite Element Procedures*. Englewood Cliffs: Prentice-Hall.

Beaude, L. et al. (2024). "Mixed and Nitsche's Discretizations of Coulomb Frictional Contact-Mechanics for Mixed Dimensional Poromechanical Models". In: *Computer Methods in Applied Mechanics and Engineering*. In press. URL: https://hal.science/hal-03949272.

Burman, E. and P. Zunino (2011). "Numerical Approximation of Large Contrast Problems with the Unfitted Nitsche Method". In: *Frontiers in Numerical Analysis - Durham 2010*. Ed. by J. Blowey and M. Jensen. Vol. 85. Lecture Notes in Computational Science and Engineering. Springer, Berlin, Heidelberg. DOI: 10.1007/978-3-642-23914-4_4. URL: https://doi.org/10.1007/978-3-642-23914-4_4.

Cerroni, D., L. Formaggia, and A. Scotti (2021). "A control problem approach to Coulomb's friction". In: *Journal of Computational and Applied Mathematics* 385, p. 113196. ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2020.113196. URL: https://www.sciencedirect.com/science/article/pii/S0377042720304878.

Chouly, F., M. Fabre, et al. (Jan. 2016). "An Overview of Recent Results on Nitsche's Method for Contact Problems". In: *UCL Workshop 2016*. UCL (University College London). London, United Kingdom, pp. 93–141. DOI: 10.1007/978-3-319-71431-8_4. URL: https://hal.science/hal-01403003v2.

Chouly, F. and P. Hild (2013). "A Nitsche-based Method for Unilateral Contact Problems: Numerical Analysis". In: *SIAM Journal on Numerical Analysis* 51.2, pp. 1295–1307.

Chouly, F., P. Hild, and Y. Renard (2015). "A Nitsche Finite Element Method for Dynamic Contact: 1. Semi-Discrete Problem Analysis and Time-Marching Schemes". In: *ESAIM: Mathematical Modelling and Numerical Analysis* 49.2, pp. 481–502. DOI: 10.1051/m2an/2014041. URL: https://hal.science/hal-00958695.

Chouly, F., P. Hild, and Y. Renard (2023a). *Finite Element Approximation of Contact and Friction in Elasticity*. 1st ed. Advances in Mechanics and Mathematics. Mathematics and Statistics eBook Package; 294 pages; 1 b/w illustration. Birkhäuser Cham. ISBN: 978-3-031-31423-0. DOI: 10.1007/978-3-031-31423-0. URL: https://doi.org/10.1007/978-3-031-31423-0.

– (2023b). *Lagrangian and Nitsche Methods for Frictional Contact*. Preprint. URL: https://hal.science/hal-04242349.

Chouly, F., R. Mlika, and Y. Renard (2018). "An Unbiased Nitsche's Approximation of the Frictional Contact Between Two Elastic Structures". In: *Numerische Mathematik* 139, pp. 593–631. DOI: 10.1007/s00211-018-0950-x. URL: https://doi.org/10.1007/s00211-018-0950-x.

Eck, C., J. Jarusek, and M. Krbec (2005). *Unilateral Contact Problems: Variational Methods and Existence Theorems*. 1st ed. CRC Press. DOI: 10.1201/9781420027365. URL: https://doi.org/10.1201/9781420027365.

Franceschini, A., N. Castelletto, et al. (2020). "Algebraically stabilized Lagrange multiplier method for frictional contact mechanics with hydraulically active fractures". In: *Computer Methods in Applied Mechanics and Engineering* 368, p. 113161. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2020.113161. URL: https://www.sciencedirect.com/science/article/pii/S0045782520303467.

Franceschini, A., M. Ferronato, et al. (2016). "A novel Lagrangian approach for the stable numerical simulation of fault and fracture mechanics". In: *Journal of Computational Physics* 314, pp. 503–521. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2016.03.032. URL: https://www.sciencedirect.com/science/article/pii/S0021999116001844.

Hild, P. (2003). "An example of nonuniqueness for the continuous static unilateral contact model with Coulomb friction". In: *Comptes Rendus Mathematique* 337.10, pp. 685–688. ISSN: 1631-073X. DOI: https://doi.org/10.1016/j.crma.2003.10.010. URL: https://www.sciencedirect.com/science/article/pii/S1631073X03004552.

Kikuchi, N. and J. T. Oden (1988). *Contact Problems in Elasticity*. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9781611970845. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611970845. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611970845.

Mlika, R. (2018). "Nitsche Method for Frictional Contact and Self-Contact: Mathematical and Numerical Study". Analysis of PDEs [math.AP]; English. PhD Thesis. Institut National des Sciences Appliquées de Lyon. URL: https://theses.hal.science/tel-01764646.

Nitsche, J. A. (1971). "Über ein Variationsprinzip zur Lösung von Dirichlet-Problemen bei Verwendung von Teilräumen, die keinen Randbedingungen unterworfen sind". In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 36. Published online: 26 November 2013, pp. 9–15. DOI: 10.1007/BF02995904.

Renard, Y. (2013). "Generalized Newton's methods for the approximation and resolution of frictional contact problems in elasticity". In: *Computer Methods in Applied Mechanics and Engineering* 256, pp. 38–55. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2012.12.008. URL: https://www.sciencedirect.com/science/article/pii/S0045782512003787.

Rockafellar, R. T. and R. J. B. Wets (1998). *Variational Analysis*. 1st ed. Grundlehren der mathematischen Wissenschaften. Springer Book Archive; XII, 736 pages. Topics: Calculus of Variations and Optimal Control; Optimization, Systems Theory, Control. Springer Berlin, Heidelberg. ISBN: 978-3-540-62772-2. DOI: 10.1007/978-3-642-02431-3. URL: https://doi.org/10.1007/978-3-642-02431-3.

Wohlmuth, B. (2011). "Variationally consistent discretization schemes and numerical algorithms for contact problems". In: *Acta Numerica* 20, pp. 569–734. DOI: 10.1017/S0962492911000079.