Politecnico di Milano

Advanced Programming for Scientific Computing
Project Report

Discontinuous Galerkin approximation of elastic wave propagation phenomena on polygonal grids

Author:
Nicola Melas

Supervisors:
Luca Formaggia
Carlo De Falco
Paola F. Antonietti
Ilario Mazzieri

February 6, 2020

Contents

1 Introduction

The aim of this project is to write a *C++* program about fully discrete discontinuous Galerkin (DG) methods applied to an elastodynamics problem on computational meshes made by polygonal elements. From the numerical point of view, it is important to obtain a good compromise between these three features of the numerical scheme:

- *accuracy;*
- *geometric flexibility;*
- *scalability.*

So in this project a `C++` code will be developed to solve this kind of problem with polytopic elements, that are a good choice to have a good accuracy with a geometric flexibility that, in general, triangles may not have. Developing a solver in `C++` could need more effort with respect to other programming languages but it guarantees, in average, a good result in term of computational time.

The first part of the report focuses on the problem, explaining its main features, then the main classes of the code are analyzed and explained toghether with an example about how to use it. Finally, before the conclusions, some numerical results to verify the code are shown.

2 The Elastodynamics Equation

2.1 The Model

In the report, the standard notation for the Sobolev spaces $H^m(\Omega)$, $m \geq 0$, endowed with the usual norm $\|\cdot\|_{H^m(\Omega)}$ is used.

Sobolev spaces of vector-valued functions and symmetric tensors are:

- $\mathbf{H^m(\Omega)} = [H^m(\Omega)]^d$
- $\boldsymbol{\mathcal{H}^m}(\Omega) = [H^m(\Omega)]^{d\times d}_{sym}$

with $d = 2$.

For $m = 0$, it is the $L^2(\Omega)$ norm.

Let $\Omega \subset \mathbb{R}^d$, $d = 2$ as above, be an open, bounded, convex region with Lipschitz boundary $\partial\Omega$ and outward normal unit vector $\mathbf{n}$.
The boundary $\partial\Omega$ can be subdivided in two disjoint portions, Γ_D and Γ_N, $\Gamma_D \neq \emptyset$ and $\Gamma_D \cap \Gamma_N = 0$.

Given:

- $\mathbf{f} \in L^2((0,\mathrm{T}]; \mathbf{L}^2(\Omega))$: external load;
- $\mathbf{g} \in C^1((0,\mathrm{T}]; \mathbf{H}^{\frac{1}{2}}(\Omega))$: boundary datum;
- $\mathbf{u}_0 \in \mathbf{H}^1_{0,\Gamma_D}(\Omega)$ and $\mathbf{u}_1 \in \mathbf{L}^2(\Omega)$: suitable initial conditions;

the mathematical model of linear elastodynamics is:

$$\begin{cases} \rho\ddot{\mathbf{u}} - \nabla\cdot\boldsymbol{\sigma} = \mathbf{f}, & \text{in } \Omega\times(0,\mathrm{T}], \\ \mathbf{u} = \mathbf{0}, & \text{on } \Gamma_D\times(0,\mathrm{T}], \\ \sigma\mathbf{n} = \mathbf{g}, & \text{on } \Gamma_N\times(0,\mathrm{T}], \\ \mathbf{u} = \mathbf{u}_0, & \text{in } \Omega\times\{0\}, \\ \dot{\mathbf{u}} = \mathbf{u}_1, & \text{in } \Omega\times\{0\}. \end{cases} \tag{1}$$

where:

- $\mathbf{u}$: $\Omega\times[0,T] \to \mathbb{R}^d$ is the displacement vector in the time interval [0,T], with $\mathrm{T} > 0$;
- $\boldsymbol{\sigma} : \Omega\times[0,T] \to \mathbb{S}$ is the stress tensor, modeled by the generalized Hooke's law, i.e.,
$$\boldsymbol{\sigma}(\mathbf{u}) = \mathcal{D}\boldsymbol{\epsilon}(\mathbf{u}) \tag{2}$$
where $\mathcal{D} : \mathbb{S} \to \mathbb{S}$, forth order stiffness tensor, is defined as:

$$\mathcal{D}\boldsymbol{\tau} = 2\mu\boldsymbol{\tau} + \lambda tr(\boldsymbol{\tau})\mathbb{1} \quad \forall \boldsymbol{\tau} \in \mathbb{S}.$$

In the definition of $\boldsymbol{\sigma}$, $\mathbb{S}$ is the space of symmetric, $d \times d$, real-valued tensorial functions and $\boldsymbol{\epsilon}(\mathbf{u})$ is the symmetric gradient of $\mathbf{u}$, i.e., $\boldsymbol{\epsilon}(\mathbf{u}) = \frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^T)$.

Some assumptions on the data have been made:

- the Lamé parameters $\lambda, \mu \in L^\infty(\Omega)$ and positive;
- $\rho \in L^\infty(\Omega)$ and positive;
- $\mathcal{D}$ is symmetric, definite positive and uniformly bounded over Ω, i.e., there exist D_*, $D^* > 0$ such that

$$0 < D_*(\boldsymbol{\tau}, \boldsymbol{\tau})_\Omega \leq (\mathcal{D}\boldsymbol{\tau}, \boldsymbol{\tau})_\Omega \leq D^*(\boldsymbol{\tau}, \boldsymbol{\tau})_\Omega \quad \forall \boldsymbol{\tau} \in \mathbb{R}^{d\times d}, \quad \boldsymbol{\tau} \neq 0.$$

Then, a possible weak formulation of the problem (1) is:

$\forall t \in (0, T]$ *find* $\mathbf{u}(t) \in \mathbf{H}^1_{0,\Gamma_D}(\Omega)$ *such that,* $\forall \mathbf{v} \in \mathbf{H}^1_{0,\Gamma_D}(\Omega)$

$$\begin{cases} \int_\Omega \rho\ddot{\mathbf{u}} \cdot \mathbf{v}\ \mathrm{d}\mathbf{x} + \int_\Omega \mathcal{D}\boldsymbol{\epsilon}(\mathbf{u}) : \boldsymbol{\epsilon}(\mathbf{v})\ \mathrm{d}\mathbf{x} = \int_\Omega \mathbf{f} \cdot \mathbf{v}\ \mathrm{d}\mathbf{x} + \int_{\Gamma_N} \mathbf{g} \cdot \mathbf{v}\ \mathrm{d}\mathbf{s} \\ \mathbf{u} = \mathbf{u}_0 \\ \dot{\mathbf{u}} = \mathbf{u}_1 \end{cases} \tag{3}$$

The problem (3) is well posed and it is known that the solution is unique and $\mathbf{u} \in C((0,T]; \mathbf{H}^1_{0,\Gamma_D}(\Omega)) \cap C^1((0,T]; \mathbf{L}^2(\Omega))$. [7]

2.2 Space Discretization and Technical Tools

2.2.1 Mesh Partition

Let $\mathcal{T}$ be a partition of the domain Ω made by open disjoint polygonal elements $\mathcal{K}$ of diameter h_K such that $\overline{\Omega} = \cup_{\mathcal{K}\in\mathcal{T}}\overline{\mathcal{K}}$, where $\mathcal{K}$ is the affine image of a fixed master element $\widehat{\mathcal{K}}$.
In two dimensions, the *faces* of the mesh are the straight intersection of the one dimensional facets of neighboring elements, i.e. line segments.

Let $\mathcal{F}$ be the union of all interior and boundary faces, i.e., $\mathcal{F} = \mathcal{F}_\mathcal{I} \cup \mathcal{F}_\mathcal{B}$ and then set $\mathcal{F}_\mathcal{B} = \mathcal{F}_\mathcal{D} \cup \mathcal{F}_\mathcal{N}$ where $\mathcal{F}_\mathcal{D} = \{F \in \mathcal{F}_\mathcal{B} : F \subseteq \Gamma_D\}$ and $\mathcal{F}_\mathcal{N} = \{F \in \mathcal{F}_\mathcal{B} : F \subseteq \Gamma_N\}$. Notice that any $\gamma \in \mathcal{F}_\mathcal{B}$ belongs to $\mathcal{F}_\mathcal{D}$ or $\mathcal{F}_\mathcal{N}$ but not to both.

It is now possible to define, $s \geq 1$, the two *broken* Sobolev space:

- $\mathbf{H}^s(\mathcal{T}) = \{\mathbf{v} \in \mathbf{L}^2(\Omega) \text{ such that } \mathbf{v}|_\mathcal{K} \in \mathbf{H}^s(\mathcal{K}) \quad \forall \mathcal{K} \in \mathcal{T}\}$
- $\boldsymbol{\mathcal{H}}^s(\mathcal{T}) = \{\boldsymbol{\tau} \in \boldsymbol{\mathcal{L}}^2(\Omega) \text{ such that } \boldsymbol{\tau}|_\mathcal{K} \in \boldsymbol{\mathcal{H}}^s(\mathcal{K}) \quad \forall \mathcal{K} \in \mathcal{T}\}$

2.2.2 Trace Operators

Let $\mathcal{K}^{\pm}$ two elements of $\mathcal{T}$ that share the same interior face $\mathrm{F} \in \mathcal{F}_h^I$ and let $\mathbf{n}^{\pm}$ be the normal unit vectors on F pointing outward $\mathcal{K}^{\pm}$.
Considering $(\mathbf{v}, \boldsymbol{\tau}) \in \mathbf{H}^1(\mathcal{T}) \times \boldsymbol{\mathcal{L}}^2(\mathcal{T})$, $(\mathbf{v}^{\pm}, \boldsymbol{\tau}^{\pm})$ denotes the trace of $(\mathbf{v}, \boldsymbol{\tau})$ on F.

Then, it is possible to define the *mean* and *jump* operators as:

- *Mean*: $\{\mathbf{v}\} = \frac{1}{2}(\mathbf{v}^+ + \mathbf{v}^-)$ $\qquad$ $\{\boldsymbol{\tau}\} = \frac{1}{2}(\boldsymbol{\tau}^+ + \boldsymbol{\tau}^-)$,
- *Jump*: $[[\mathbf{v}]] = \mathbf{v}^+ \odot \mathbf{n}^+ + \mathbf{v}^- \odot \mathbf{n}^-$ $\qquad$ $[[\boldsymbol{\tau}]] = \boldsymbol{\tau}^+\mathbf{n}^+ + \boldsymbol{\tau}^-\mathbf{n}^-$.

Observe that, on $\mathrm{F} \in \mathcal{F}_h^B$, $\quad \{\mathbf{v}\} = \mathbf{v}, \quad \{\boldsymbol{\tau}\} = \boldsymbol{\tau}, \quad [[\mathbf{v}]] = \mathbf{v} \odot \mathbf{n}, \quad [[\boldsymbol{\tau}]] = \boldsymbol{\tau}\mathbf{n}$.

2.2.3 Finite Element Spaces

The finite elements spaces used to approximate the continuous problem are:

- $V_h = \{v \in L^2(\Omega) : v|_{\mathcal{K}} \in \mathcal{P}_{p_{\mathcal{K}}} \quad \forall \mathcal{K} \in \mathcal{T}\}$,
- $\boldsymbol{V}_h = [V_h]^d$,
- $\boldsymbol{\mathcal{V}}_h = [V_h]^{d\times d}_{sym}$,

with $d = 2$, where $\mathcal{P}_{p_{\mathcal{K}}}(\mathcal{K})$ is the space of polynomials of maximum degree $p_{\mathcal{K}} \geq 1$ on the element $\mathcal{K}$ of the mesh $\mathcal{T}$.

2.3 The DG Family

To analyze the family of DG methods, it is useful to rewrite problem 1 in a more general way, as follows:

$$\begin{cases} \rho\ddot{\mathbf{u}} - \nabla \cdot \boldsymbol{\sigma} = \mathbf{f}, & \text{in } \Omega \times (0,\mathrm{T}], \\ \mathcal{A}\boldsymbol{\sigma} - \boldsymbol{\epsilon} = \mathbf{0}, & \text{in } \Omega \times (0,\mathrm{T}], \\ \mathbf{u} = \mathbf{0}, & \text{on } \Gamma_D \times (0,\mathrm{T}], \\ \sigma\mathbf{n} = \mathbf{g}, & \text{on } \Gamma_N \times (0,\mathrm{T}], \\ \mathbf{u} = \mathbf{u}_0, & \text{in } \Omega \times \{0\}, \\ \dot{\mathbf{u}} = \mathbf{u}_1, & \text{in } \Omega \times \{0\}. \end{cases} \tag{4}$$

where:

- all the hypothesis on the data, done in *Section 2.1*, are kept;
- the compliance tensor $\mathcal{A} = \mathcal{A}(x) : \mathbb{S} \to \mathbb{S}$ is a bounded, symmetric and uniformly positive definite operator such that, provided it is invertible, $\mathcal{A}^{-1}\boldsymbol{\epsilon} = \mathcal{D}\boldsymbol{\epsilon}$.

So, $\mathcal{A}$ is defined as:

$$\mathcal{A}\boldsymbol{\sigma} = \frac{1}{2\mu}\left(\boldsymbol{\sigma} - \frac{\lambda}{3\lambda+2\mu} tr(\boldsymbol{\sigma})\mathbb{1}\right) \quad \forall \boldsymbol{\sigma} \in \mathbb{S}. \tag{5}$$

Then, to simplify the notation, $\mathbf{L}^2(\mathcal{T})$ and $\mathbf{L}^2(\mathcal{F})$ scalar products have been introduced:

$$(\boldsymbol{\phi}, \boldsymbol{\psi})_{\mathcal{T}} = \sum_{\mathcal{K}\in\mathcal{T}} (\boldsymbol{\phi}, \boldsymbol{\psi})_{\mathcal{K}} \quad \langle \boldsymbol{\phi}, \boldsymbol{\psi} \rangle_{\mathcal{F}} = \sum_{F\in\mathcal{F}} (\boldsymbol{\phi}, \boldsymbol{\psi})_{F}$$

Now, given $\mathbf{u}_h(0)$ and $\dot{\mathbf{u}}_h(0)$, approximations of the initial data, the general semi-discrete variational formulation for DG method is:

$\forall t \in (0, T]$ *find* $\mathbf{u}_h \in C^2((0,T]; \mathbf{V}_h)$ and $\boldsymbol{\sigma}_h \in C^0((0,T]; \boldsymbol{\mathcal{V}}_h)$ *such that*, $\forall \mathbf{v} \in \mathbf{V}_h$, $\forall \boldsymbol{\tau} \in \boldsymbol{\mathcal{V}}_h$:

$$\left(\rho\ddot{\mathbf{u}}_h, \mathbf{v}\right)_{\mathcal{T}} + \left(\boldsymbol{\sigma}_h, \boldsymbol{\epsilon}(\mathbf{v})\right)_{\mathcal{T}} - \langle \{\widehat{\boldsymbol{\sigma}}\}, [[\mathbf{v}]] \rangle_{\mathcal{F}} - \langle [[\widehat{\boldsymbol{\sigma}}]], \{\mathbf{v}\} \rangle_{\mathcal{F}_{\mathcal{I}}} = (\mathbf{f}, \mathbf{v})_{\mathcal{T}} \tag{6}$$

$$\left(\mathcal{A}\boldsymbol{\sigma}_h, \boldsymbol{\tau}\right)_{\mathcal{T}} - \left(\boldsymbol{\epsilon}(\mathbf{u}_h), \boldsymbol{\tau}\right)_{\mathcal{T}} - \left\langle \{\widehat{\mathbf{u}} - \mathbf{u}_h\}, [[\boldsymbol{\tau}]] \right\rangle_{\mathcal{F}_{\mathcal{I}}} - \left\langle [[\widehat{\mathbf{u}} - \mathbf{u}_h]], \{\boldsymbol{\tau}\} \right\rangle_{\mathcal{F}} = 0 \tag{7}$$

where:

- $\widehat{\mathbf{u}} = \widehat{\mathbf{u}}(\mathbf{u}_h, \boldsymbol{\sigma}_h) \in \mathbf{L}^2(\mathcal{F})$
- $\widehat{\boldsymbol{\sigma}} = \widehat{\boldsymbol{\sigma}}(\mathbf{u}_h, \boldsymbol{\sigma}_h) \in \boldsymbol{\mathcal{L}}^2(\mathcal{F})$

are the numerical fluxes, whose choice identifies a particular DG method.

Notice that numerical fluxes have to be set in such a way that the boundary conditions are satisfied ($\mathbf{c}_{11}, \mathbf{c}_{22}$ will be defined later):

- $\widehat{\mathbf{u}} = \mathbf{0}$ on $F \in \mathcal{F}_D$, $\quad \widehat{\mathbf{u}} = \mathbf{u}_h - \mathbf{c}_{22}(\boldsymbol{\sigma}_h\mathbf{n} - \mathbf{g})$ on $F \in \mathcal{F}_N$
- $\widehat{\boldsymbol{\sigma}}\mathbf{n} = \boldsymbol{\sigma}_h\mathbf{n} - \mathbf{c}_{11}\mathbf{u}_h$ on $F \in \mathcal{F}_D$, $\quad \widehat{\boldsymbol{\sigma}}\mathbf{n} = \mathbf{g}$ on $F \in \mathcal{F}_D$

The DG formulation chosen in this project is the *displacement* formulation, where $\widehat{\boldsymbol{\sigma}}$ is defined in the following way:

$$\widehat{\boldsymbol{\sigma}} = \begin{cases} \{\mathcal{D}\boldsymbol{\epsilon}(\mathbf{u}_h)\} - \gamma[[\mathbf{u}_h]] & F \in \mathcal{F}_I, \\ \mathcal{D}\boldsymbol{\epsilon}(\mathbf{u}_h) - \gamma\mathbf{u}_h\mathbf{n} & F \in \mathcal{F}_D \end{cases} \tag{8}$$

and

- $\mathbf{c}_{11}$ has been replaced by γ,

- $\gamma = C_\gamma h_F^{-1} k^2\{\mathcal{D}\} \quad \forall F \in \mathcal{F}_\mathcal{I} \cup \mathcal{F}_\mathcal{D}$ (9)
- $\mathbf{c}_{22} = \mathbf{0}$.

Setting $\boldsymbol{\tau} = \mathcal{D}\boldsymbol{\epsilon}(\mathbf{v})$ in the variational formulation (7), after some calculations (see [1]), it becomes:

find $\mathbf{u}_h \in C^2((0,T];\mathbf{V}_h)$ *such that* $\forall \mathbf{v} \in \mathbf{V}_h$

$$\Big(\rho\ddot{\mathbf{u}}_h, \mathbf{v}\Big)_\mathcal{T} + \Big(\boldsymbol{\epsilon}(\mathbf{u}_h), \mathcal{D}\boldsymbol{\epsilon}(\mathbf{v})\Big)_\mathcal{T} + \langle\{\widehat{\mathbf{u}} - \mathbf{u}_h\}, [[\mathcal{D}\boldsymbol{\epsilon}(\mathbf{v})]]\rangle_{\mathcal{F}_\mathcal{I}} + \langle[[\widehat{\mathbf{u}} - \mathbf{u}_h]], \{\mathcal{D}\boldsymbol{\epsilon}(\mathbf{v})\}\rangle_{\mathcal{F}_\mathcal{I}\cup\mathcal{F}_\mathcal{D}}$$
$$- \langle\{\mathcal{D}\boldsymbol{\epsilon}(\mathbf{u}_h)\}, [[\mathbf{v}]]\rangle_{\mathcal{F}_\mathcal{I}\cup\mathcal{F}_\mathcal{D}} + \langle\gamma[[\mathbf{u}_h]], [[\mathbf{v}]]\rangle_{\mathcal{F}_\mathcal{I}\cup\mathcal{F}_\mathcal{D}} = (\mathbf{f}, \mathbf{v})_\mathcal{T} + \langle\mathbf{g}, \mathbf{v}\rangle_{\mathcal{F}_\mathcal{N}} \quad (10)$$

The problem (10) represents the family of classical Interior Penalty methods (IP). The choice of $\widehat{\mathbf{u}}$ corresponds to a particular IP method.

The method largely used in this project is the Symmetric Interior Penalty method (SIP), where

$$\widehat{\mathbf{u}} = \{\mathbf{u}_h\}.$$

The SIP formulation of the problem is:

find $\mathbf{u}_h \in C^2((0,T];\mathbf{V}_h)$ *such that* $\forall \mathbf{v} \in \mathbf{V}_h$

$$\Big(\rho\ddot{\mathbf{u}}_h, \mathbf{v}\Big)_\mathcal{T} + a(\mathbf{u}_h, \mathbf{v}) = (\mathbf{f}, \mathbf{v})_\mathcal{T} + \langle\mathbf{g}, \mathbf{v}\rangle_{\mathcal{F}_\mathcal{N}} \quad (11)$$

where $a(\cdot,\cdot)$ is the bilinear form defined as:
$a(\mathbf{u}, \mathbf{v}) = (\boldsymbol{\epsilon}(\mathbf{u}), \mathcal{D}\boldsymbol{\epsilon}(\mathbf{v}))_\mathcal{T} - \langle\{\mathcal{D}\boldsymbol{\epsilon}(\mathbf{u})\}, [[\mathbf{v}]]\rangle_{\mathcal{F}_\mathcal{I}\cup\mathcal{F}_\mathcal{D}} - \langle[[\mathbf{u}]], \{\mathcal{D}\boldsymbol{\epsilon}(\mathbf{v})\}\rangle_{\mathcal{F}_\mathcal{I}\cup\mathcal{F}_\mathcal{D}} + \langle\gamma[[\mathbf{u}]], [[\mathbf{v}]]\rangle_{\mathcal{F}_\mathcal{I}\cup\mathcal{F}_\mathcal{D}}$

2.4 Error Bounds

Theorem
Let the mesh be sufficiently regular, let $\mathbf{u}_h \in C^2((0,T];\mathbf{V}_h)$ be the DG solution to the elastodynamics problem (11) solved with a sufficiently large C_γ and let $\mathbf{u}$ be the solution to the continuous problem (4) such that it is sufficiently regular. See [2] for the proofs.
Then it holds:

$$\sup_{0<t\leq T} \|\mathbf{u}(t) - \mathbf{u}_h(t)\|_{\mathbf{L}^2(\Omega)} \lesssim \frac{h^s}{k^{m-1}}\Big(\mathcal{N}(\mathbf{u}) + \int_0^t \mathcal{N}(\dot{\mathbf{u}})\mathrm{d}\tau + \frac{h}{k^2}\|\boldsymbol{\sigma}(\mathbf{u})\|_{\mathcal{H}^m(\Omega)}\Big)$$

where h is the space discretization size, k is the chosen polynomial degree, $s = \min(k+1, m)$ and

$$\mathcal{N}(\mathbf{u}) = \Big(\|\mathbf{u}\|^2_{\mathbf{H}^m(\Omega)} + \frac{h^2}{p^3}\|\dot{\mathbf{u}}\|^2_{\mathbf{H}^m(\Omega)} + \frac{h^2}{p^3}\|\boldsymbol{\sigma}(\mathbf{u})\|^2_{\mathcal{H}^m(\Omega)}\Big)^{\frac{1}{2}}$$

2.5 Semidiscrete Algebraic Formulation

Let the domain Ω be subdivided in N_{el} disjoint polytopic elements $\mathcal{K}_j,\ j = 1 : N_{el}$ and define n_{p_k} as $\dim(\mathbb{P}_{p_k})$.
Then, setting $N_{dof} = \sum_{r=1}^{N_{el}} n_{p_k}$ dimension of each component of a function in $\mathbf{V}_h$, let's indroduce a modal basis $\mathbf{\Phi}_i^s(\mathbf{x})$ such that:

- $\mathbf{\Phi}_i^s(\mathbf{x}) = (0, ..., \Phi_i^s(\mathbf{x}), ..., 0)^T$
- $\mathbf{u}_h(\mathbf{x}, t) = \sum_{s=1}^{d} \sum_{j=1}^{N_{dof}} \mathbf{\Phi}_j^s(\mathbf{x})\ U_j^s(t)$.

At this point, choosing $\mathbf{v} = \mathbf{\Phi}_i^s(\mathbf{x}) \in \mathbf{V}_h$ in (11), $s = 1 : 2$, it holds:

$$M\ddot{\mathbf{U}}(t) + A\mathbf{U}(t) = \mathbf{F}(t) \quad \forall t \in (0, T) \tag{12}$$

where

- M and A are the Mass and Stiffness matrices, respectively;
- $\mathbf{U}$(t) is the unknown displacement, time dependent;
- $\mathbf{F}(t)$ is the external applied load.

2.6 Time Integration

Subdivide the interval (0,T] in n_T subintervals of amplitude $\Delta t = T/n_T$ and denote $t_i = i\Delta t$.
To perform the time integration, the leap-frog scheme is used (see [4], [5]).
It is second-order accurate, explicit and conditionally stable; the method follows the *Cfl* condition: $\Delta t \leq C_{CFL} h$.
In the leap-frog scheme, the equations to update the displacement and its derivatives are:

- $\ddot{\mathbf{u}}(t_i) = G(t_i)$
- $\dot{\mathbf{u}}(t_{i+\frac{1}{2}}) = \dot{\mathbf{u}}(t_{i-\frac{1}{2}}) + \Delta t\ \ddot{\mathbf{u}}(t_i)$
- $\mathbf{u}(t_{i+1}) = \mathbf{u}(t_i) + \Delta t\ \dot{\mathbf{u}}_{\mathbf{i+\frac{1}{2}}}(t_i)$

From these equations, the second order accurate approximation is:

$$\ddot{\mathbf{u}}(t_i) = \frac{\mathbf{u}(t_{i+1}) - 2\mathbf{u}(t_i) + \mathbf{u}(t_{i-1})}{\Delta t^2}$$

2.7 Fully Discrete Approximation and Error Estimates

Subdivide the interval (0,T] as in Section 4.2, then it is possible to rewrite the system (12) as:

$$M\mathbf{U}(t_{i+1}) = [2M - \Delta t^2 A]\ \mathbf{U}(t_i) - M\mathbf{U}(t_{i-1}) + \Delta t^2 \mathbf{F}(t_i) \quad \forall i \in 1, ..., n_T$$

The method is a 2-step method and so, having already the solution at the initial step t_0, it requires a little modification (see [2]) to be applied to find the solution at $t = t_1$. Indeed

$$M\mathbf{U}(t_1) = [M - \frac{\Delta t^2}{2}A]\mathbf{U}(t_0) + \Delta t M\dot{\mathbf{U}}(t_0) + \frac{\Delta t^2}{2}\mathbf{F}(t_0)$$

where $\dot{\mathbf{U}}(t_0) = \mathbf{u}_1$.

Fully Discrete L^2 Error Estimate
The assumptions made to prove the semi-discrete error bounds still hold.
Then, considering the leap-frog scheme to discretize the second order time derivative, it holds:

$$\max_{n=0\ldots n_T} \|\mathbf{u}(t_n) - \mathbf{U}(t_n)\|_{\mathbf{L}^2(\Omega)} \lesssim \frac{h^s}{p^{m-1}} + \Delta t^2$$

where s and m are defined as before.
Notice that, to verify the space convergence, Δt must be sufficiently small such that Δt^2 is negligible with respect to h^s.

3 Implementation Details

3.1 Basis Function

The basis functions can be chosen in many different ways; the usual approach, as it is done in the classical finite element method, is to employ standard polynomial bases on a reference element and then map them to physical elements; generally this reduces costs and is effective. However, in order to deal with polygonal elements, we follow a different strategy ([6]), that works directly on the physical element $\forall k \in \mathcal{K}$. First we construct a Cartesian bounding box $B_k = I_1 \times I_2$ $\forall k \in \mathcal{T}$ such that $\overline{k} \subseteq \overline{B_k}$. Then on every bounding box B_k we define a standard polynomial space $\mathbb{P}_r(B_k)$, spanned by a set of basis functions $\{\phi_{i,k}\}_{i=1}^{dim(\mathbb{P}_r(B_k))}$. We use tensor-product (scaled) Legendre polynomials i.e., denoting by $L_r(x)$ the Legendre polynomial of degree r defined on the interval $[-1, 1]$, the corresponding scaled Legendre polynomial on the interval $I_b = [x_1 , x_2]$ may be defined by:

$$L_r^{[I_b]}(x) = \frac{1}{\sqrt{h_b}} L_r\Big(\frac{x - m_b}{h_b}\Big)$$

where $h_b = (x_2 - x_1)/2$ and $m_b = (x_1 + x_2)/2$. Then the basis on the box B_k is given by:

$$\phi_{i,k}(\mathbf{x}) = L_{r_1}^{[1]}(x) L_{r_2}^{[2]}(y), \quad r_1 + r_2 \leq r, \quad r_k > 0, \quad k = 1, 2.$$

Finally the polynomial basis over the polygonal element $\mathcal{K}$ is defined simply restricting the support of $\phi_{i,k}(\mathbf{x}), \quad i = 1, ..., \mathbb{P}_r(B_k)$ to k, i.e. choosing $\phi_{i,k}\|_{\mathcal{K}}(\mathbf{x})$, $i = 1, ..., \mathbb{P}_r(B_k)$.

3.2 Matrix Composition

The two matrices used to solve the system are M and A, respectively the Mass matrix and the Stiffness matrix. In particular,

$$A = V + S - IT - IT^T$$

where:

$$V_{ij} = \sum_{\mathcal{K} \in \mathcal{T}} \int_{\mathcal{K}} \nabla\phi_j \cdot \nabla\phi_i \, dx, \quad IT_{ij} = \sum_{\mathcal{F}_\mathcal{I} \cup \mathcal{F}_\mathcal{D}} \int_F [\![\phi_j]\!] \cdot \{\nabla\phi_i\} \, ds, \quad S_{ij} = \sum_{\mathcal{F}_\mathcal{I} \cup \mathcal{F}_\mathcal{D}} \gamma \int_F [\![\phi_j]\!] \cdot [\![\phi_i]\!] \, ds$$

for $i, j = 1, ..., N$. Let us analyse their pattern:

- V is a diagonal matrix. Indeed, if $i \neq j$, ϕ_i and ϕ_j supports intersection is empty and $V_{ij} = 0$.
- IT (and IT^T) has a block-diagonal pattern, with in addition some blocks different from zero out of the diagonal, those referring to basis functions of

neighbouring elements. Indeed, using the definitions of jump and average:

$$\int_F [\![\phi_j]\!] \cdot \{\nabla \phi_i\} = \frac{1}{2}\int_F (\nabla {\phi_i}^+ \cdot \mathbf{n}^+)\phi_j^+ + \frac{1}{2}\int_F (\nabla {\phi_i}^- \cdot \mathbf{n}^-)\phi_j^- - \frac{1}{2}\int_F (\nabla {\phi_i}^+ \cdot \mathbf{n}^+)\phi_j^- - \frac{1}{2}\int_F (\nabla {\phi_i}^- \cdot \mathbf{n}^-)\phi_j^+$$

for $i, j = 1, \ldots, N$, where $F \in \Gamma_h$ is the face shared by κ^+ and κ^-. We see that only the first two contributions fall on blocks along the diagonal. If $F \in \mathcal{F}_D$, we use definition of jump and average at the boundary, obtaining:

$$\int_F [\![\phi_j]\!] \cdot \{\nabla \phi_i\} = \int_F (\nabla \phi_i \cdot \mathbf{n})\phi_j, \quad F \in \mathcal{F}_D, \quad i, j = 1, \ldots, N.$$

- S has the same structure of I, indeed we have:

$$\gamma \int_F [\![\phi_j]\!] \cdot [\![\phi_i]\!] = \gamma \int_F \phi_i^+ \phi_j^+ + \gamma \int_F \phi_i^- \phi_j^- - \gamma \int_F \phi_i^+ \phi_j^- - \gamma \int_F \phi_i^- \phi_j^+,$$

for $i, j = 1, \ldots, N$, $F \in \mathcal{F}_I$, F is shared by $\mathcal{K}^+$ and $\mathcal{K}^-$, and

$$\gamma \int_F [\![\phi_j]\!] \cdot [\![\phi_i]\!] = \gamma \int_F \phi_i \phi_j,$$

for $i, j = 1, \ldots, N$, $F \in \mathcal{F}_D$.

Hence we globally obtain a symmetric sparse matrix A.
Let now $\mathbf{g}$ and $\mathbf{h}$ the Dirichlet and Neumann boundary data respectively. The right hand side $\mathbf{f}$ reads as follows:

$$\mathbf{f} = \mathbf{f}_{\mathrm{V}} + \mathbf{f}_{\mathrm{N}} - \mathbf{f}_{\mathrm{IT}} + \mathbf{f}_{\mathrm{S}},$$

where

$$\mathbf{f}_{\mathrm{V},i} = \sum_{\mathcal{K}\in\mathcal{T}} \int_{\mathcal{K}} f\phi_i, \quad \mathbf{f}_{\mathrm{N},i} = \sum_{F\in\Gamma_N} \int_F h\phi_i, \quad \mathbf{f}_{\mathrm{IT},i} = \sum_{F\in\Gamma_D} \int_F g(\nabla\phi_i \cdot \mathbf{n}), \quad \mathbf{f}_{\mathrm{S},i} = \sum_{F\in\Gamma_D} \gamma \int_F g\phi_i$$

for $i = 1, \ldots, N$.

3.3 Quadrature rules

Quadrature over general polygonal elements is taken on constructing a triangulation, followed by the exploitation of standard integration schemes over simplexes. For every polygonal element $\mathcal{K} \in \mathcal{T}$ we first construct a non-overlapping sub-triangulation $\mathcal{K}_\mathcal{S} = \{\tau_\kappa\}$ made of triangular elements, then we map quadrature nodes from a reference simplex to each triangle τ_κ. For example:

$$\int_{\mathcal{K}} u \cdot v = \sum_{\tau_\mathcal{K}\in\mathcal{K}_\mathcal{S}} u \cdot v \approx \sum_{\tau_\mathcal{K}\in\mathcal{K}_\mathcal{S}} \sum_{i=1}^{q} u(M_\mathcal{K}(\xi_i)) \cdot v(M_\mathcal{K}(\xi_i)) det(J_{M_\mathcal{K}}(\xi_i)) w_i,$$

where $M_{\mathcal{K}} : \hat{\mathcal{K}} \to \tau_{\mathcal{K}}$ is the map from the reference simplex $\hat{\mathcal{K}}$ to $\tau_{\mathcal{K}}$, $J_{F_{\mathcal{K}}}$ is its Jacobian and $(\xi_i, w_i)_{i=1}^{q}$ denote the quadrature nodes and weights over $\hat{\kappa}$. Note that, in the case of integral with gradients, the gradient operator is not transformed, as it happens when integrals are computer on the reference simplex.
Integrals over edges of polygonal elements can be computed in an analogous manner, mapping quadrature points from a reference interval to each edge. Of course it is useful for the implementation if each edge of a polygon $\mathcal{K}$ coincides with an edge of a triangle $\tau_{\mathcal{K}}$ of the sub-triangulation $\mathcal{K}_{\mathcal{S}}$.

3.4 Mesh structures

In the implementation polygonal meshes are generated using the Polymesher algorithm ([8]). The starting point of the algorithm is the definition of a signed distance between a point and the domain. It is needed during the generation of n random points to be sure that the points live inside the domain. Then, given m random points, a Voronoi Diagram is built. It has the properties of being bounded and of giving as result a tessellation of m convex polygon, called Voronoi Cell $\mathbf{V_y}$, where $\mathbf{y}$ is one of the random points. The true key point of a Voronoi cell $\mathbf{V_y}$ is that it contains all the points that are closer to $\mathbf{y}$ and not to any other point generating the diagram. The main difficulty is to represent in a good way the boundary. Indeed, before proceeding in building the diagram, the points that are close to the boundary are reflected with respect to the closer boundary. In this way $m = n + n_{refl}$. Thanks to this trick, the common edge between $\mathbf{V_y}$ and $\mathbf{V_{Refl(y)}}$ will be a good representation of the boundary. Then, to have a more homogeneous mesh, the algorithm focuses on Centroidal Voronoi Tessellation, where the Lloyd algorithm([8]) is used as a fixed point iteration:

- Computing reflected points P_R of P;
- Computing Voronoi diagram of $[P; P_R]$;
- Computing the centroids P_C of these polygons;
- Computing the error. It is zero if centroids coincide with the points generating the diagram;
- If error is sufficiently big, $P = P_C$ and repeat the algorithm.

Finally, three functions are used to improve the mesh quality:

- *PolyMshr_ExtrNds*, function that extract the first n polygons from the Voronoi result. These are the polygons inside the domain;
- *PolyMshr_CllpsEdgs*, function that delete the edges that are sufficiently small to be neglected without affecting the homogeneity of the mesh in a relevant way;
- *PolyMshr_RsqsNds*, function that reorder the nodes to reduce the bandwith of the stiffness matrix. Reverse Cuthill-Mckee ordering is used inside this

function (`Boost` library).

On the other hand, also triangular meshes are built. In this case, it is constructed simply adding points in an emphirical way (horizontally), creating an homogeneous mesh. The used approach works only for rectangular mesh. It has been done in this way because it's not the main goal of the project.

4 The C++ code

The code has been developed to test the presented method in two dimensional domains and it can be subdivided in four main parts:

- The `MeshGenerator` class that provides an idea about how the mesh is built, with particular attention to the polygonal case.
- The `Mesh` class that stores the geometrical entities that are used in the method.
- The `FeSpace` class that represents the discretization of the domain
- The `BoundaryCondition` class that stores the information about conditions to impose on the boundary.
- The `ElastodynamicsProblem` class that assembles and solves the system.

The `Eigen` Library has been used for the linear algebra, and some other libraries are used to compute some particular non trivial tasks. These are: `CGAL, Boost` and `Qhull`[[9],[11],[12]].

4.1 The MeshGenerator class

Listing 1: *MeshGenerator.hpp*

```
class MeshGenerator {

protected:

  // number of element
  std::size_t ne;

public:
  // Constructor
  MeshGenerator(const std::vector<double> & domain_, std::size_t ne_) : domain(domain_),
     ne(ne_){};

  //Generation function.
  virtual void generate(Mesh &) = 0;

  //...

};
```

It's the father class. The core of the class is the `generate` function. It receives by reference a `Mesh` object whose elements will be created (if the `Mesh` is defaulted) or edited properly. Note that it is a pure virtual function because the generator is different if you choose to build a polygonal mesh rather that a triangular one.

Listing 2: *PolyGenerator.hpp*

```
class PolyGenerator : public MeshGenerator {

private:

   // PolyMesher Tools
   Points PolyMshr_RndPtSet() const;
   Points PolyMshr_Rflct(const Points & P, double alpha) const;
   std::pair<Points, VectorD> PolyMshr_CntrdPly(....) const;
   std::pair<Points,VecVecXi> PolyMshr_ExtrNds(....) const;
   std::pair<Points,VecVecXi> PolyMshr_CllpsEdgs(....) const;
   std::pair<Points,VecVecXi> PolyMshr_RsqsNds(....) const;
   std::pair<Points,VecVecXi> PolyMshr_RbldLists(....) const;

  // Construct Voronoi Diagram from a starting set of points (Using Qhull)
   std::pair<Points,VecVecXi> construct_voronoi(const Points & generators) const;

   // Reordering algorithm (Using Boost)
   std::vector<int> reverse_cuthill(const Eigen::SparseMatrix<int> & A) const;

public:

   //...

   // Generation of the mesh
   void generate(Mesh &) override;
};
```

The Polygonal generator is based on the Voronoi Tessellation, that represents the core of the algorithm. Since the PolyMesher ([8]) library is originally written in `MATLAB`, the same reference library has been used, QHull. Its C++ interface is not very complete but sufficient to obtain the expected result, compared with the Matlab one. It needs in input the coordinates as list of doubles and the Qhull function compute the Voronoi Diagram from the given points. On the other hand, to reduce the bandwidth of the stiffness matrix, the Reverse Cuthill–McKee algorithm relies on the boost library and it is more elaborate. To use the RCM boost algorithm it is necessary to construct a Graph that contains pairs of row and column indexes of nonzero elements of the SparseMatrix. Then setting the degree and the indexes of the graph, the boost RCM is ready to be used. It stores the new order in a reverse iterator (because it is the reverse Cuthill-McKee).

Listing 3: *TriaGenerator.hpp*

```
class TriaGenerator : public MeshGenerator {

  std::size_t ne_x,ne_y;

public:
  // The TriaGenerator has to double the number of elements
  TriaGenerator(....) : MeshGenerator(...), ne_x(ne_x_), ne_y(ne_y_) {ne *=2;};

  // Generation of the mesh
  void generate(Mesh &) override;

  //...
};
```

The triangular generator is quite simple. The project is about polygonal elements but, since the code can work properly also with simplexes, a basic generator has been implemented. Note that in the constructor, given the number of x and y intervals, the total number of elements has to be doubled with respect to the polygonal mesh concept.
How the generators work is already explained in section 3.4. So now let's see the `Mesh` structure.

4.2 The Mesh class

4.2.1 The main class

Listing 4: *Mesh.hpp*

```
class Mesh {

private:

  std::vector<double> domain; //domain
  std::size_t  ne;   // number of elements
  Points  coord;// all the points
  std::vector<std::shared_ptr<AbstractPolygon>>  polygons; //Pointers to polygons
  VecVecXi  connectivity;   // Connectivity matrix
  double diameter; // maximum diameter

  //...

public:
  //Constructor for rectangular domain
  Mesh(const Point2D & p1, const Point2D & p2, std::size_t ne_x, std::size_t ne_y, std::
    string mesh_type = "Polygon");

  //... Getter methods, for example the diameter hmax, the area, the bounding box

  // Print info and export function
  void print_mesh(std::ostream & out=std::cout, bool print_polygons = 0) const;
  void printvtk(std::string fileName) const;

  friend class MeshGenerator;
  friend class PolyGenerator;
  friend class TriaGenerator;

};
```

The variable `coord` contains all the vertexes of the Mesh. The connectivity stores information about how elements are close or not, setting values less than 0 to edges that are on the boundary, since they are not close to any other element of the mesh. The variable `coords_element` is a `std::vector` of all the pointers to the class AbstractPolygon. It's an abstract class and using pointers is useful to manage all its children. The constructor takes two points to create a rectangular mesh. The type of mesh, triangular or polygonal, has to be defined passing the `mesh_type` string. The default one is the polygonal mesh. The Mesh constructor calls the correct generator coherently with the chosen `mesh_type`. Note that, since

the generators has to act directly on a mesh, they have been put as friend classes. Moreover, the geometrical information about the mesh are stored in the Polygons.

4.2.2 The Polygon class

Listing 5: *AbstractPolygon.hpp*

```
1
class AbstractPolygon
{
public:

6   // Constructor taking vertices as Points
    AbstractPolygon(Points const & v, bool check=true);

    // Intersection between polygons
    std::vector<std::shared_ptr<AbstractPolygon>> polyintersection(const std::shared_ptr<
      AbstractPolygon> polygon) const ;
11
    // Getter methods
    Point2D compute_centroid() const;
    double x_min() const;
    double x_max() const;
16  double y_min() const;
    double y_max() const;
    std::vector<double> get_x() const;
    std::vector<double> get_y() const;
    inline Point2D get_vertex(std::size_t i) const;
21  std::vector<Edge> get_edges() const;
    Points get_normals() const;
    std::vector<double> get_edges_length() const;
    std::vector<double> get_BBox() const ;

26  //Pure virtual methods
    virtual double area() const=0;
    virtual std::vector<double> get_max_kb() const = 0;
    virtual void addvertex(const Point2D & vertex) = 0;

31  //! Outputs some info on the polygon
    virtual void showMe(std::ostream & out=std::cout) const;

protected:
    Points vertexes;
36
    //..
};
```

All the important geometrical information can be obtained from the AbstractPolygon class. It is the father class of `Polygon` and `Triangle`. The `polyintersection(..)` method relies on the `boost` library. It's not a trivial function, in particular if the polygons are not convex (but we are not interested in this case). After constructing the two boost polygon, it receives in input the two polygons and a vector of boost polygon to fill with the intersection/s. Furthermore, it is possible to get the edges and the normal vectors to them. Edges are not stored in the Polygon because `vertexes` are able to give all the necessary information. The `add_vertex(..)` method is pure virtual since the triangle needs the check on the number of already existent vertexes. Moreover, also the `area()` method is pure virtual. In the

triangular case it's easier to compute it, as for `get_max_kb()`.

Listing 6: *Polygon.hpp*

```
class Polygon: public AbstractPolygon
{
public:
  //...

  //Specialised versions
  virtual double area() const override;
  virtual void showMe(std::ostream & out=std::cout) const override;
  std::vector<double> get_max_kb() const override;
  inline void addvertex(const Point2D & vertex) override;

};
```

Listing 7: *Triangle.hpp*

```
class Triangle final: public AbstractPolygon
{
public:
  //...

  // Constructor given three points
  Triangle(Point2D const &,Point2D const &,Point2D const &);

  //Specialised versions
  virtual double area() const override;
  inline virtual void addvertex(const Point2D & vertex) override;
  virtual void showMe(std::ostream & out=std::cout) const override;
  std::vector<double> get_max_kb() const override;

  //function about triangle orientation and vertexes ordering
  bool clockwise();
  bool counterclockwise();
  int orientation() const;

};
```

Both classes contain inside the specialised versions of some function. The `addvertex()` function needs the check on the number of vertexes in the triangular case and `area()` function is more difficult in the polygonal case, where divergence theorem is used. Moreover the class `Triangle` has one constructor more, giving in input three vertexes, and three more functions: `clockwise()` reorder the vertex in a clockwise manner, `counterclockwise()` does the opposite and `orientation()` return info about how vertexes are oriented. Note that almost all the getter methods of a `Mesh` element rely on `AbstractPolygon` routines (e.g. area of an element, the bounding box, the normals to the edges, etc.).

4.3 The FeSpace class

4.3.1 The main class

Listing 8: *fespace.hpp*

```
class FeSpace {

private:

  const Mesh & region;
  Neighbour & neighbour;
  std::vector<FeElement> FeElements; //contains all the info
  std::size_t fem_degree;
  std::size_t nln; // local degree of freedom
  std::size_t ndof; // total  degrees of freedom
  std::size_t nqn; // number of quadrature nodes

  //! Computing the vector of FeElements with all the important information
  void compute_FElements();

public:
  //Constructor
  FeSpace(const Mesh & region, Neighbour& neighbour, std::size_t fem_degree);

  //! Return the Mesh (as const Reference)
  inline const Mesh & get_mesh() const {return region;}
  inline Neighbour & get_Neighbour() const {return neighbour;}

};
```

To define a Finite Element Space, it's usually needed, as input, the mesh and the polynomial degree. Due to the use of a Discontinuos Galerkin method, also the information about all the adiancency elements are important and so, a `Neighbour` element is given. Note that the `Neighbour` element is passed/returned only by reference to the constructor/from the getter method and not as const reference, due to the need to allow the function `Neighbour::set_boundary_tag()` to modify the tags of the boundary. The main effort of the costructor is in the definition of the `std::vector<FeElement>`. For each element of the mesh, a `FeElement` is defined to contain all the information about values of the basis function and the integration coefficients computed with respect to the integration points defined by a quadrature rule. In this project the Gauss quadrature rule has been chosen.

4.3.2 The FeElement class

Listing 9: *FeElement.hpp*

```
class FeElement {

private:

  std::size_t id; //id

  //volumetric information about triangulation of element id
  std::size_t number_tria;
```

```
  std::vector<std::shared_ptr<Triangle>> triangulation;
  std::vector<std::vector<double>> dx_trias;
  std::vector<MatrixXd> phi_trias;
  std::vector<VecMatrixXd> Grad_trias;

  //edges information about polygon edges
  std::vector<double> penalty_edges;
  std::vector<std::vector<double>> ds_edges_poly;
  std::vector<MatrixXd> phi_edges;
  std::vector<VecMatrixXd> Grad_edges;
  Points normal_edges;

  //physical quadrature points
  std::vector<Points> physical_points_tria;
  std::vector<Points> physical_points_edge;

  //evaluation of polynomial basis on the quadrature nodes
  std::pair<MatrixXd,VecMatrixXd> evalshape2D(const std::shared_ptr<AbstractPolygon>
    polygon, const Points & Nodes) const;

public:

  FeElement(std::size_t nqn, std::size_t fem_degree_, std::size_t id_, const std::vector
    <std::shared_ptr<AbstractPolygon>> & polygons, const Neighbour & neighbour);

  // all the getter methods

};
```

The class can be divided in two blocks. The first that contains the variables about the volume integration, in particular:

- `triangulation` is a `std::vector<std::shared_ptr<Triangle>>` that represent the triangles that subdivide the element;
- `dx_tria` is a `std::vector<std::vector<double>>` containing, for each triangle, `nqn` integration coefficients to be applied during numerical integration;
- `phi_tria` and `Grad_trias` contain for each polynomial basis function, its value or the value of the gradient in the 2D quadrature nodes. Note that in the case of the gradient there are 2 components.

On the other hand the second block is about 1D geometrical elements, in particular these variables are saved:

- `penalty_edges` is a `std::vector<double>` containing the coefficients used to compute the S matrix;
- `ds_edges_poly` has the same goal of `dx_trias` but works for 1D quadrature rule;
- `phi_edges` and `Grad_edges` contain for each polynomial basis function, its value or the value of the gradient in the 1D quadrature nodes. Note that in the case of the gradient there are 2 components;
- `normal_edges` are represented through points and they are the normalized

vector normal to the edges. They are used to assemble matrices where the jump operator is present.

All this variable are stored in the memory to avoid to repeat the computation each time that they are needed. Indeed, these variables are used also in the imposition of boundary condition rather that in the evaluation of a function on that `FeSpace`. So, to be more efficient, all the getter methods of the variables stored in a FeElement return them by const reference.
Let's now analyze the constructor. It receive in input:

- the number of quadrature nodes `nqn` to know how to build the nqn quadrature points;
- the number of polynomial degree to know how the basis is;
- the id to identify it with its index;
- the `std::vector` with all the elements of the mesh because there are objects that depends on the properties of the neigh element;
- the `Neighbour` to know which are the elements close to that `FeElement`.

So, at this point, the `Mesh` has been defined through the `Meshgenerator` with all the elements, the `FeSpace` has been constructed saving all the information that will be needed to assemble the problem in a quicker way. The last object that is missing to define the `ElastodynamicsProblem` is the `BoundaryCondition`.

4.4 The BoundaryCondition class

4.4.1 The main class

Listing 10: *BoundaryCondition.hpp*

```
class BoundaryCondition {


protected:
  // Tag to identify the boundary
  int tag;

  // Function returning true when evaluated on the points where to apply BC
  std::function<bool(Point2D)> inside_domain;

  // components source
  muParserXInterface<3, std::array<double, 3>> & g1;
  muParserXInterface<3, std::array<double, 3>> & g2;

public:
  BoundaryCondition(std::function<bool(Point2D)> inside, muParserXInterface<3, std::
    array<double, 3>> & g1_,muParserXInterface<3, std::array<double, 3>> & g2_, int
    tag_)
  : tag(tag_), inside_domain(inside), g1(g1_), g2(g2_) {};

  // BC is applied to the rhs vector defined using FeSpace. If the BC source is time
    dependent, give also the time
```

```
20    virtual void apply(VectorD & rhs, const FeSpace &, double time = 0.0) const = 0;

      //! Return the BC identifier of the BC type, 1 for Dirichlet, 2 for Neumann
      virtual int get_type() const = 0;
};
```

The `BoundaryCondition` presented in the project needs 3 concepts:

- a `tag`, that must be a negative integer, to identify which are the boundary edges where the BC has to be applied;
- a `std::function<bool(Point2D)>` that return a boolean that is true if the given Point is inside the 1D domain where to apply BC;
- the source of the boundary condition.

Note that the source has been defined using the `muParserx` library ([13]), that allows a runtime evaluation of a math expression.
Two functions are declared pure virtual, the `apply()` function, due to the differences between Neumann and Dirichlet boundary conditions, and `get_type()`, function that will be useful in the basic distinction between Neumann and Dirichlet edges, because some matrices are affected only by Dirichlet boundaries contributions.

Let's see the differences between the children classes.

4.4.2 DirichletBC and NeumannBC classes

Listing 11: *DirichletBC.hpp*

```
1
class DirichletBC : public BoundaryCondition {

private:
  const Coefficients & dati;
6
public:

  //! Constructor taking a function, the sources and the tag
  DirichletBC(std::function<bool(Point2D)> inside, muParserXInterface<3, std::array<
    double, 3>> & g1_,muParserXInterface<3, std::array<double, 3>> & g2_, int tag_,
    const Coefficients & dati_)
11  : BoundaryCondition(inside, g1_,g2_,tag_), dati(dati_) {};

  //Specialised version for Dirichlet Booundary Condition
  void apply(VectorD &rhs,const FeSpace &, double time = 0.0) const override;

16  //Specialised version for DirichletBC, return 1
  inline int get_type() const override;
};
```

Listing 12: *NeumannBC.hpp*

```
2 class NeumannBC : public BoundaryCondition {
```

```
public:
  //Constructor
  NeumannBC(std::function<bool(Point2D)> inside, muParserXInterface<3, std::array<double
    , 3>> & g1_,muParserXInterface<3, std::array<double, 3>> & g2_, int tag_)
  : BoundaryCondition(inside, g1_,g2_,tag_) {};

  // Specialised version for NeumannBC
  void apply(VectorD &rhs,const FeSpace &, double time = 0.0) const override;

  // Specialised version for NeumannBC, return 2
  int get_type() const override {return 2;};

};
```

First of all, note that the `DirichletBC` class needs one parameter more in the constructor, because in its `apply()` function λ, μ are used. To save memory not copying data and also to not allocate useless memory, the `Coefficients` are passed by const reference. The two classes differ in this way ($\mathbf{g}$ is the boundary datum):

- The Dirichlet b.c. modifies the right hand side enforcing penalization:

$$f(i) = \int_{\mathcal{K}} f \cdot \phi_i dx - \int_{\mathcal{F}_D} \mathbf{g} \cdot \nabla \phi_i ds + \int_{\mathcal{F}_D} \gamma_D \mathbf{g} \cdot \phi_i ds$$

- The Neumann b.c. modifies the right hand side in the usual way:

$$f(i) = \int_{\mathcal{K}} f \cdot \phi_i dx + \int_{\mathcal{F}_N} \mathbf{g} \cdot \phi_i ds$$

Note that the Neumann `apply()` function doesn't need the penalty parameter and also the evaluations of the gradients at the physical nodes.

All the variables needed to impose the boundary condition are called by const reference first from the `FeSpace` and then from the `std::vector<FeElement>`. Also in this case we can see the advantages of previously storing all the variables in the `FeElement` class.

The parameter `time` may be not passed if the source doesn't depend on time. If it is done, the time won't be considered because in the muparserx evaluation the variable `x[2]` doesn't exist (The muparserx sintax is: x is x[0], y is x[1] and t is x[2]).

Now everything is ready and an `ElastodynamicsProblem` object can be created.

4.5 The ElastodynamicsProblem class

Listing 13: *ElastodynamicsProblem.hpp*

```
class ElastodynamicsProblem {

private:

  SparseMatrixXd M; // Mass Matrix \int_{@Omega} (u . v ) dx}
  SparseMatrixXd S; // \int_{E_h} penalty  h_e^(-1) [v].[u] ds
  SparseMatrixXd IT; // \int_{E_h} {sigma(v)} * [u]ds
  SparseMatrixXd V; // Matrix given by \int_{\Omega} (sigma(u) eps(v) dx
  SparseMatrixXd A; //Stiffness Matrix A = M + V - IT - IT^T
  VectorD f; //rhs

  FeSpace & femregion;
  const Coefficients & dati;

  //BC info
  std::vector<std::shared_ptr<BoundaryCondition>> BCs;
  std::vector<int> Diri_tags;

  //Solver
  Eigen::SparseLU<SparseMatrixXd> solver;
  bool solver_not_define = 1;

  //assembling helpers
  void finalize_matrix(SparseMatrixXd & M, const SparseMatrixXd & M1, const
    SparseMatrixXd & M2,
                                           const SparseMatrixXd & M3,const
    SparseMatrixXd & M4);
  void assemble_neigh(SparseMatrixXd& M,const VectorSt& row, const VectorI& neight,const
     VecSMatrixXd& M1,std::size_t nln,std::size_t n_edge);

public:

  //! Constructor.
  ElastodynamicsProblem(FeSpace & fespace, const Coefficients & coeff, std::vector<std::
    shared_ptr<BoundaryCondition>> BCs);

  //! Assemble all the matrices and the rhs of the Elastodynamics problem
  void assemble(double time = 0.0);

  //! Update the system at the given time
  void update_rhs(double time);

  //...getter methods, all by const reference

  // Return the L2 error
  double L2_error(const VectorD & u_h,  MuParserInterface::muParserXInterface<3, std::
    array<double, 3>> & u_x,  MuParserInterface::muParserXInterface<3, std::array<
    double, 3>> & u_y, double time = 0.0) const;

  // Solve the system
  void solve(VectorD & u_h, const VectorD & u1old, const VectorD & u2old);

};
```

To store this problem the construct receive as input:

- a `FeSpace`, the space where the problem lives;

- a set of `Coefficients`, containing the physical parameters of the material, the fixed part of the penalty coefficient and the source of the right hand side;
- a `std::vector` containing shared pointers to BoundaryCondition.

The constructor consists in modifying the Neighbour inside the FeSpace object (stored by reference only) to tag in a proper way the edges where to impose each boundary condition. This will also be useful in the assembling part when it is needed to distinguish Dirichlet boundaries from Neumann ones. Indeed, a `std::vector<int> Diri_tags` is stored, to contain only tags of Dirichlet boundaries. Concerning the solver, we can see that there is a `bool solver_not_define`. If it is 1, it means that the matrix that will be used to solve has not been assigned and factorized yet. This concept is checked in the function `solve()` that, after verifying it, solve the system using the Leap-Frog formulation. The matrices doesn't depend on time and, to have a more efficient computation, the factorizing part is done only once. But the biggest effort done inside this class is in the `assemble()` function. It recalls by const reference, through the `FeSpace`, all the info contained in the `std::vector<FeElement>`. Then all the matrices are assembled in 4 blocks, depending on how the basis functions are taken. In particular, note that there are also matrices that depends on the near elements, as explained in section 3.2. The function `assemble_neigh()` has the task to add this contribution to the correct reference matrix.
To save a lot of memory, all the matrices are stored in a sparse way (using Eigen) and M, V, IT and S are filled through the Eigen function `setFromTriplets()`. Finally, the last part of the `assemble()` function consists in the evaluation of the right hand side, evaluating the source and applying the boundary condition in a proper way, thanks to the tags set in the constructor.

4.6 Solving a problem (small tutorial)

Listing 14: *Solve_Example.cpp*

```
int main(int argc, char** argv) {

GetPot file(argc, argv);

std::string filename = file("filename", "Solve_example.txt");
GetPot g2(filename.c_str());

//Reading parameters using GetPot

//definition of sources
muParserXInterface<3> fx;
muParserXInterface<3> fy;
muParserXInterface<3> diri_x;
muParserXInterface<3> diri_y;
muParserXInterface<3> neum_x;
muParserXInterface<3> neum_y;
//... set expression add define constant to these expressions

//Define the data to describe the problem
```

```
Coefficients coefficients(rho, mu, lam, dt, fem_degree, penalty_coeff, fx, fy);

//Defining a polygonal domain and its Neighbours
Point2D p1,p2;
p1 << 0.0,0.0;
p2 << 1.0,1.0;
Mesh Th(p1,p2,ne,1,"Polygon");
Neighbour neighbour(Th);

//Define the Fe Space
FeSpace femregion(Th, neighbour, coefficients.get_fem());

//Defining BCs
double tol = 1e-9;
DirichletBC diri1([](Point2D x)-> bool {return x(0) < tol;},exact_sol_x, exact_sol_y,
    -1, coefficients);
DirichletBC diri2([](Point2D x)-> bool {return x(1) < tol;},exact_sol_x, exact_sol_y,
    -2, coefficients);
DirichletBC diri3([](Point2D x)-> bool {return x(0) > 1-tol;},exact_sol_x, exact_sol_y,
    -3, coefficients);
NeumannBC neum1([](Point2D x)-> bool {return x(1) > 1-tol;},neum_x, neum_y, -4);

std::vector<std::shared_ptr<BoundaryCondition>> bcs;
//.. inserting BC

//Defining the problem
ElastodynamicsProblem problem(femregion, coefficients, bcs);
problem.assemble();

//..create the two initial solution u1old and u2old

std::size_t nstep = T/dt;
VectorD solution;
//updating in time the rhs
problem.update_rhs(time);

time+=dt;
std::cout << "Computing time: " << time << std::endl;
problem.solve(solution, u1old, u2old);

return 0;

}
```

Let's now see step-by-step how to use the library:

- Reading through GetPot all the parameters and all the sources that describe the problem;
- Loading in the muParser interface all the sources as string and then setting the expressions contained inside the source(e.g. μ, λ);
- Defining the `Coefficients` needed to describe the problem;
- Generating and constructing a `Mesh` object and its `Neighbour`;
- Defining over this mesh a Finite Element Space, `FeSpace`;
- Defining all the boundary conditions with their constructor and inserting

them in a `std::vector`;

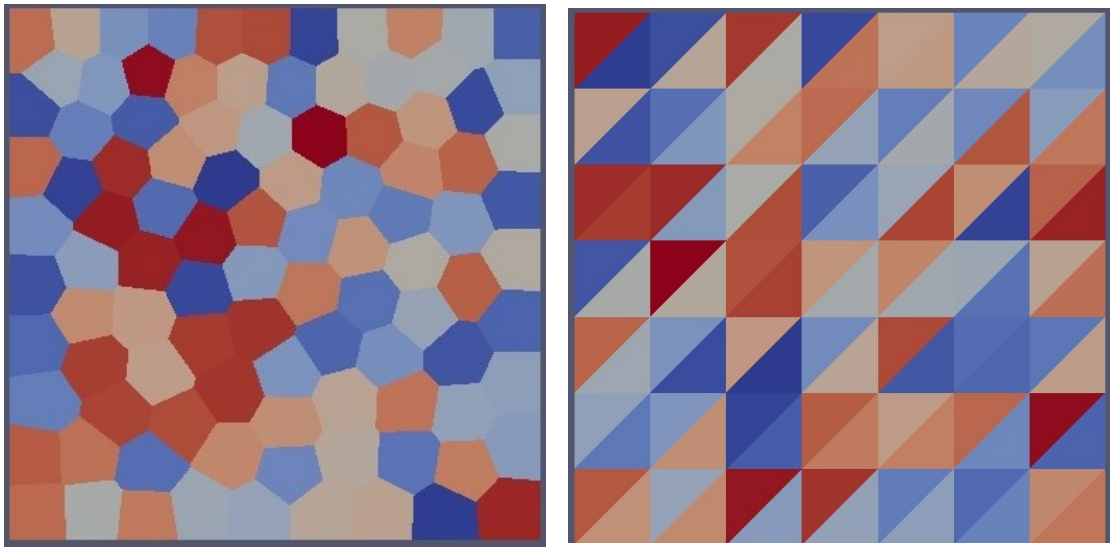
- Creating the problem and assembling it;
- Creating the initial guesses and then solving the problem.

Note that it's not necessary to use `GetPot` but it can be useful to avoid to recompile every time the user want to solve the same problem with different sources. `Muparserx` helps the user that can write the function as strings. A small remark about the starting solutions: it's not sufficient to evaluate the function at the given time step but it's necessary to project it over the FeSpace. To do it, calling $\mathbf{u1}$ = "function computed at the dofs", the operation that will be solved is $\mathbf{Mx} = \rho\mathbf{u1}$ (remember that ρ is inside the Mass Matrix).
The `solve()` function will use a `Eigen::SparseLU` solver. It's also possible to use other solvers, inside the Eigen Library rather than external ones.

5 Numerical Result

5.1 The generation of the Mesh

First of all, I started checking if the generation of the Mesh worked properly. In the `Polymesh_example.cpp` and `Triamesh_example.cpp` files Polygonal and Triangular Meshes are generated and then exported to be opened in Paraview (`https://www.paraview.org/`). Results are shown in *Figure 1*.



(a) (b)

Figure 1: *Representation of the mesh (a) Polygonal (b) Triangular*

The Polygonal Mesh is perfectly comparable with the ones created using the original Polymesher tool in MATLAB. Moreover we can observe that the Mesh, as expected, is homogeneous, thanks to the Lloyd Algorithm, remembering that the generation started from a set of random points.

5.2 The solution of the Test Problem

Let $\Omega = (0,1)^2$, $\mu = \rho = \lambda = 1$. Let's fix the analytical solution

$$\mathbf{u}(\mathbf{x},t) = \begin{bmatrix} \sin(\sqrt{2}\pi t)\sin(\pi x)^2\sin(2\pi y) \\ -\sin(\sqrt{2}\pi t)\sin(2\pi x)\sin(\pi y)^2 \end{bmatrix}.$$

Boundary conditions, initial conditions and $\mathbf{f}$ have been set accordingly.

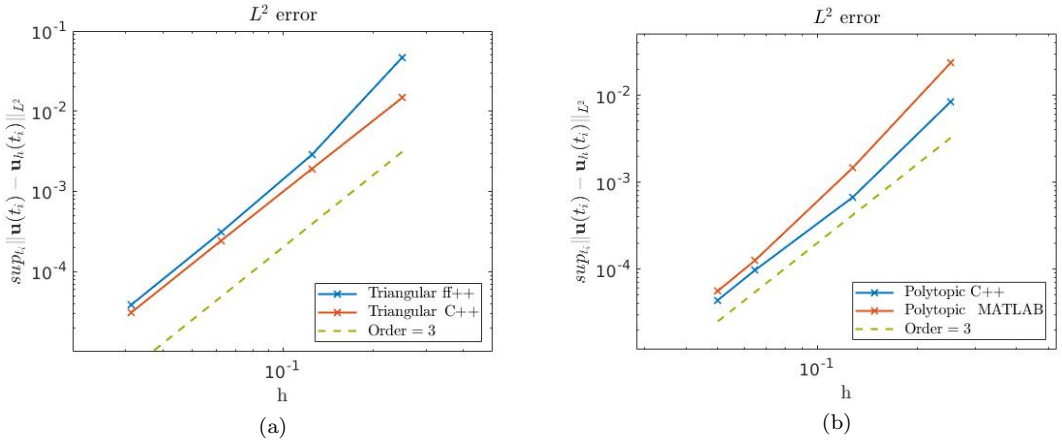
The computations done with the C++ code have been compared with two other solution obtained through:

- Triangular Mesh, using `FreeFem++`
- Polytopic Mesh, using `MATLAB`.

In the first case h = [0.25, 0.125, 0.0625, 0.03125] have been set.
In the second one, only the number of elements has been set, in particular N = [50, 225, 800, 1200].
The DG method used in the codes is the SIP one, fixing the polynomial approximation degree k to 2, 3 or 4, T to 0.15 and dt, time discretization, to 1e-4. The problem has been solved and then exported to be opened in Paraview.

5.2.1 Same Mesh Comparison

Let's compare the results obtained with the `C++` code and the `FreeFem++` one with triangular elements and then compare the same using polytopic elements in `MATLAB` and `C++` again. Note that the simulations are not very quick, because the problem is time dependent, the time discretization is space size dependent and at each time the right hand side must be updated. Indeed, even if the final time has been set to only 0.15s, most of the computational efforts of all the process are not in the assembling part but in the solving procedure. Let's now analyze the results in *Figure 2*.



(a) (b)

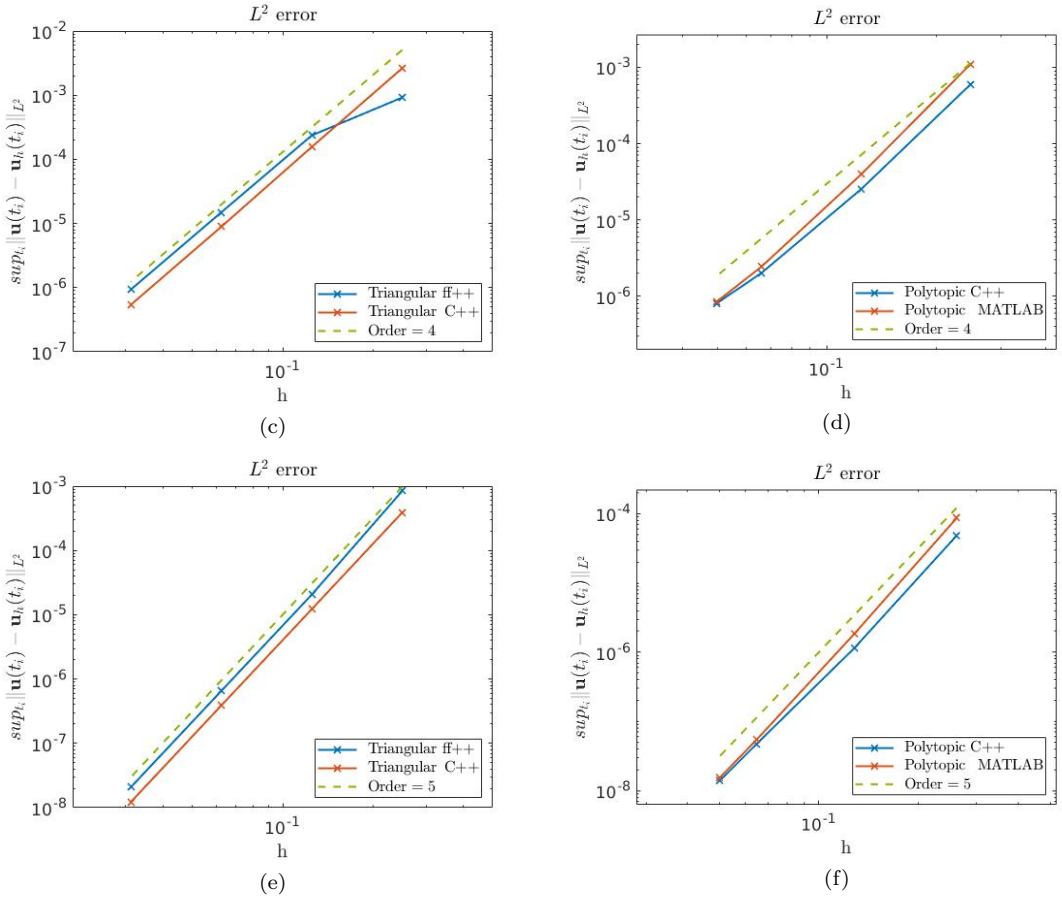


Figure 2: *Plots of comparisons of errors with respect to mesh-type, Triangular on the left with polynomial degree 2, 3, 4 [(a), (c), (e)], Polygonals on the right, with the same polynomial degree. [(b), (d), (f)]*

The result with polynomial degree equal to 1 are omitted because it's known that they cannot give, in general, good results in the elastodynamics equation. Observing the plots we can see that the C++ code give us consistent results with respect the classic codes written in `MATLAB` and `Freefem++`. Indeed, in *Table 1* and *Table 2*, comparison between convergence order are done. They are consistent with respect to what we can expect from the theory.

k	**Tria C++**	**Tria FF++**
2	2.9947	3.0322
3	4.0406	3.9987
4	4.9875	4.9724

Table 1: L^2 *Triangles rates of convergence.*

k	**Poly C++**	**Poly MAT**
2	3.0885	3.1276
3	3.9409	3.7301
4	4.7962	4.9105

Table 2: *L^2 Polygons rates of convergence.*

5.2.2 Different Mesh comparison

We expected that, since in the previous section everything has worked well, the errors computed with the `C++` code with triangular and polygonal elements are comparable. These result are shown in *Figure 3*.

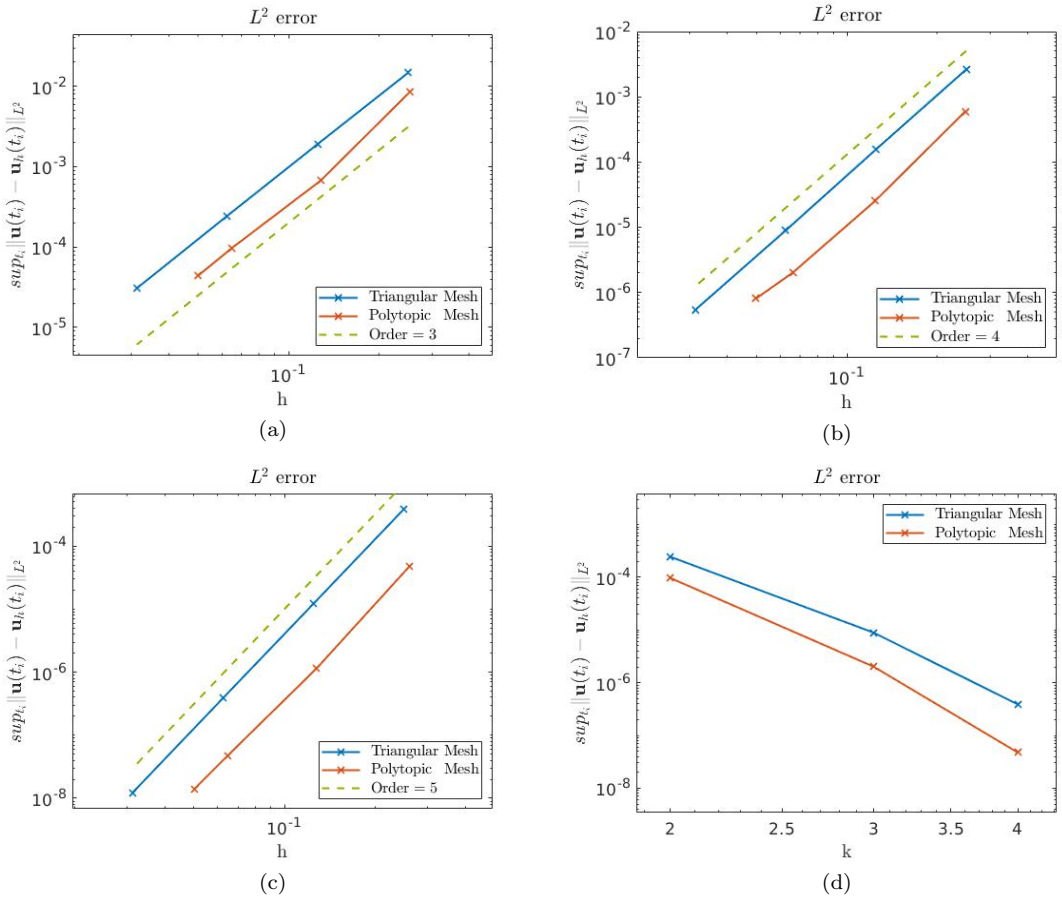


Figure 3: *Plots of L^2 errors computed with the code. Comparison between the two mesh type [(a), (b), (c)] and behaviour of the error increasing the polynomial degree [(d)].*

The code works well and everything is good. The expected order, $k+1$, is satisfied in the computation of the L^2-convergence rate and we can also see that, fixed the number of polygonal elements to 800 and the space discretization of triangular elements to 0.0625, the error decreases with respect to an increasing polynomial degree.

In the project, there are also other examples to partially test the library, such as tests on the transformation map, on the Boundary conditions rather than on the time spent in the solving procedure. Indeed in the `Solve_Example_trick.cpp` we can see that if the temporal part of the right hand side can be isolated and considered only as a multiplier, the solving procedure can be very efficient and quicker.

6 Conclusions

The elastodynamics equation has been introduced in this report. Then a C++ code to solve this problem has been written. The starting point constists in the construction of a Mesh generator that fill the mesh information that will be available to create a Finite element space. The problem and the other needed objects will be created and it can be solved quite quickly. It is significantly quicker than the MATLAB analogous solver. The code is consistent with the theoretical results and so, it allows to exploit the advantages of polytopic elements.

References

[1] P.F. Antonietti, B. Ayuso de Dios, I. Mazzieri and A. Quarteroni Stability Analysus of Discontinuous Galerkin Approximations to the Elastodynamicd Problem *J. Sci. Comput.* 68(2016) 143-170.

[2] P.F. Antonietti, I. Mazzieri High-order Discontinuous Galerkin methods for the elastodynamics equation on polygonal and polyhedral meshes *Compit. Methods Appl. Mech. Engrg.* 342(2018) 414-437.

[3] B. Rivière Discontinuos Galerkin Methods for Solving Elliptic and Parabolic Equations: Theory and Implementation *Society for Industrial and Applied Mathematics*, 2008

[4] A. Quarteroni Numerical Models for Differential Problems, volume 8 of MS&A Modeling, Simulation and Applications. *Springer-Verlag Italia*, Milan(2014)

[5] A. Quarteroni, R. Sacco, F. Saleri(2007) Numerical Mathematics *Texts in Applied Mathematics, vol. 37, 2nd edn. Berlin: Springer.*

[6] A. Cangiani, E. H. Georgoulis, P. Houston, hp-Version discontinuous Galerkin methods on polygonal and polyhedral meshes, *Math. Models Methods Appl. Sci.* 24 (10) (2014) 2009 – 2041.

[7] P.-A. Raviart, J.-M. Thomas, Introduction a l'analyse numerique des equations aux derivees partielles, *Masson* 1983.

[8] C. Talishi, G.H. Paulino, A. Pereira, I. Menezes, PolyMesher: a general-purpose mesh generator for polygonal elements written in MATLAB, *Springer-Verlag* 2011.

[9] The Computational Geometry Algorithms Library (CGAL lib) https://www.cgal.org/documentation.html.

[10] The Eigen Library, https://eigen.tuxfamily.org/dox/.

[11] The Boost Library, https://www.boost.org/.

[12] The Qhull Library, http://www.qhull.org/html/index.htm.

[13] The muParserx Library, https://beltoforion.de/article.php?a=muparserx.