POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Arbitrary order virtual element method for linear elastostatics

Project for the course
Advanced Programming for Scientific Computing

Author: **Elias Pescialli**

Student ID: 962695
Course held by: Prof. Luca Formaggia, Matteo Caldana
Supervisor: Prof. Paola F. Antonietti
Academic Year: 2022-23

Contents

Introduction

The present work focuses on a recently born technique named *virtual element method* (VEM, [6]), which allows to use general, possibly non-convex polygons/polyhedra and even elements with curved boundaries, support the embedding of hanging nodes and can easily be integrated in a standard FEM environment. In the following, a virtual element implementation for the elastic problem in three dimensions is proposed starting from a mixed variational formulation based on the three-field Hu-Washizu functional [20]. The mathematical tools of the virtual element method are subsequently presented and declined in the context of linear elastostatics. The last part is devoted to the c++ implementation, starting from the adopted algorithms, the organization of the code, and concluding with numerical h- and p-convergence tests and some displayed solutions.

1 | Elastic problem in 3D

1.1. Continuum mechanics for linear elasticity

This section briefly describes the continuum boundary value problem for solid mechanics and recalls its governing equations (see, e.g. [15]).
Infinitesimal strain theory applies, thus displacements are assumed much smaller than the relevant dimension of the deformable body and strains are much smaller than unity. The material is modelled through a linear elastic constitutive law.
Henceforth, the *Voigt notation* will be adopted ([17]), in view of the symmetric nature of the 2^{nd} order stress tensor $\boldsymbol{\sigma}$ and strain tensor $\boldsymbol{\varepsilon}$. Namely, the convention reduces the order of symmetric tensors and therefore allows to treat in 3D the aforementioned two quantities as $[6 \times 1]$ vectors and the 4^{th} order stiffness tensor coupling them as a $[6 \times 6]$ matrix.
The solid body represented in Figure 1.1 by the domain $\Omega \subset \mathbb{R}^3$ is set in a Cartesian reference system $Oxyz$ with $\boldsymbol{x} = \{x\ y\ z\}^{\mathrm{T}}$ being a generic position vector, and its sufficiently regular boundary $\partial\Omega$ is partitioned in a constrained part $\partial_u\Omega$ and a free part $\partial_p\Omega$, such that $\partial_u\Omega \cap \partial_p\Omega = \emptyset$. On each point of the boundary $\boldsymbol{x} \in \partial\Omega$ a local reference system $\boldsymbol{x}e_ue_vn$ is set, where $\boldsymbol{n}$ is the outward normal unit vector orthogonal to the surface and $\boldsymbol{e_u}$ and $\boldsymbol{e_v}$ are two mutually orthogonal arbitrarily chosen unit vectors tangent to the surface of Ω so that the triple represents a right-hand oriented coordinate system. On the constrained subset of the boundary imposed displacements $\overline{\boldsymbol{u}} = \overline{\boldsymbol{u}}(\boldsymbol{x})$ are assigned and on the free subset surface tractions $\boldsymbol{p} = \boldsymbol{p}(\boldsymbol{x})$ are applied. The volume described by Ω is subjected to body forces $\boldsymbol{b} = \boldsymbol{b}(\boldsymbol{x})$. The data of the problem is summarized below.

- body forces vector $\boldsymbol{b}$ in Ω

$$\boldsymbol{b} = \begin{Bmatrix} b_x(x,y,z) \\ b_y(x,y,z) \\ b_z(x,y,z) \end{Bmatrix}$$

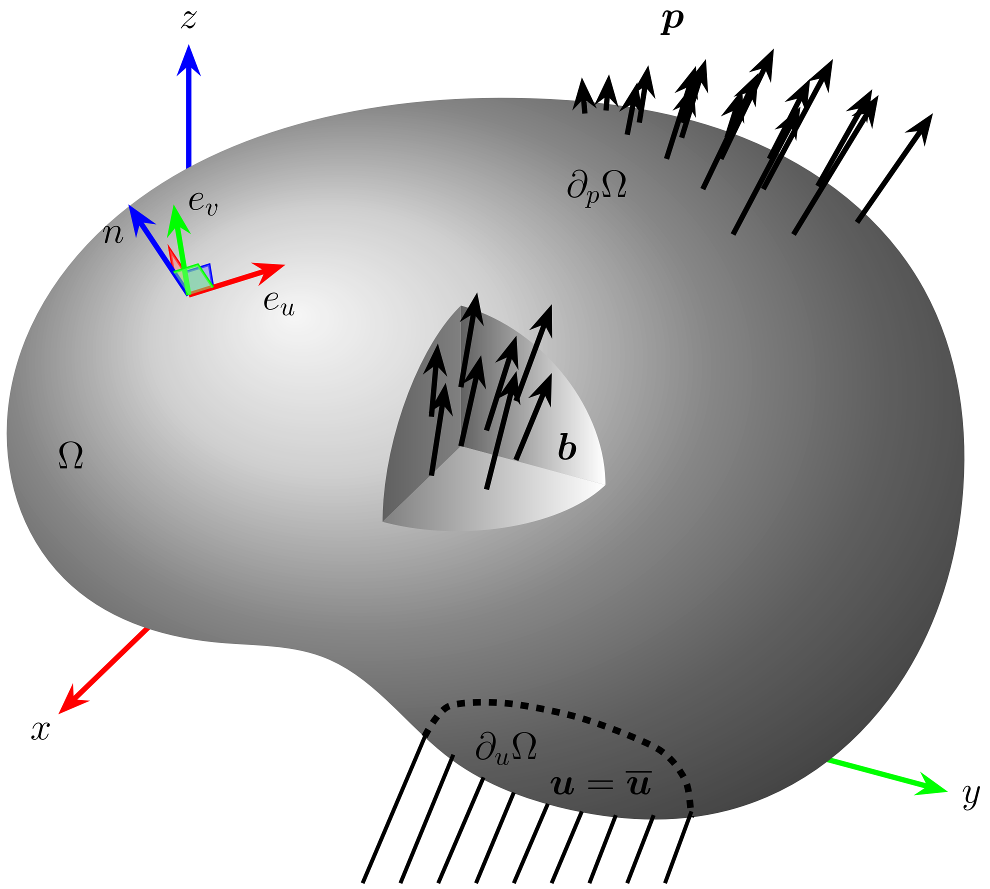


Figure 1.1: Elastic boundary value problem.

- surface tractions $\boldsymbol{p}$ on $\partial_p\Omega$

$$\boldsymbol{p} = \begin{Bmatrix} p_x(x,y,z) \\ p_y(x,y,z) \\ p_z(x,y,z) \end{Bmatrix}$$

- imposed displacements $\overline{\boldsymbol{u}}$ on $\partial_u\Omega$

$$\overline{\boldsymbol{u}} = \begin{Bmatrix} \overline{u}_x(x,y,z) \\ \overline{u}_y(x,y,z) \\ \overline{u}_z(x,y,z) \end{Bmatrix}$$

The unknowns for the problem are:

- displacement vector $\boldsymbol{u}$ in Ω

$$\boldsymbol{u} = \begin{Bmatrix} u_x(x,y,z) \\ u_y(x,y,z) \\ u_z(x,y,z) \end{Bmatrix}$$

- strains vector $\boldsymbol{\varepsilon}$ in Ω

$$\boldsymbol{\varepsilon} = \begin{Bmatrix} \varepsilon_x(x,y,z) \\ \varepsilon_y(x,y,z) \\ \varepsilon_z(x,y,z) \\ \gamma_{xy}(x,y,z) \\ \gamma_{yz}(x,y,z) \\ \gamma_{xz}(x,y,z) \end{Bmatrix}$$

- stresses vector $\boldsymbol{\sigma}$ in Ω

$$\boldsymbol{\sigma} = \begin{Bmatrix} \sigma_x(x,y,z) \\ \sigma_y(x,y,z) \\ \sigma_z(x,y,z) \\ \tau_{xy}(x,y,z) \\ \tau_{yz}(x,y,z) \\ \tau_{xz}(x,y,z) \end{Bmatrix}$$

The governing equations for the problem in matrix form are:

- indefinite equilibrium in Ω

$$\boldsymbol{S}^{\mathrm{T}}\boldsymbol{\sigma} + \boldsymbol{b} = \boldsymbol{0} \tag{1.1}$$

  with boundary conditions on $\partial\Omega$

$$\mathbb{N}\boldsymbol{\sigma} = \boldsymbol{p} \tag{1.2}$$

- kinematic compatibility in Ω

$$\boldsymbol{\varepsilon} = \boldsymbol{S}\boldsymbol{u} \tag{1.3}$$

  with boundary conditions on $\partial\Omega$

$$\boldsymbol{u} = \overline{\boldsymbol{u}} \tag{1.4}$$

- constitutive law for linear elasticity in Ω

$$\boldsymbol{\sigma} = \boldsymbol{D}\boldsymbol{\varepsilon} \tag{1.5}$$

In (1.3) the matrix $\boldsymbol{S}$ is the compatibility differential operator

$$\boldsymbol{S} = \begin{bmatrix} \partial_x & 0 & 0 \\ 0 & \partial_y & 0 \\ 0 & 0 & \partial_z \\ \partial_y & \partial_x & 0 \\ 0 & \partial_z & \partial_y \\ \partial_z & 0 & \partial_x \end{bmatrix} \tag{1.6}$$

and its transpose $\boldsymbol{S}^{\mathrm{T}}$ in (1.1) is the equilibrium differential operator

$$\boldsymbol{S}^{\mathrm{T}} = \begin{bmatrix} \partial_x & 0 & 0 & \partial_y & 0 & \partial_z \\ 0 & \partial_y & 0 & \partial_x & \partial_z & 0 \\ 0 & 0 & \partial_z & 0 & \partial_y & \partial_x \end{bmatrix} \tag{1.7}$$

where $\partial_{(\cdot)}$ represents the partial derivative with respect to $(\cdot)$.
The matrix $\mathbb{N}$ contains the direction cosines of the outward normal unit vector $\boldsymbol{n}$ so that the matrix product in (1.2) correctly represents the tensor counterpart $\sigma_{ij}n_j$.

$$\mathbb{N} = \begin{bmatrix} n_x & 0 & 0 & n_y & 0 & n_z \\ 0 & n_y & 0 & n_x & n_z & 0 \\ 0 & 0 & n_z & 0 & n_y & n_x \end{bmatrix} \tag{1.8}$$

where n_x, n_y and n_z are the three components of the outward normal unit vector $\boldsymbol{n} = \{n_x\ n_y\ n_z\}^{\mathrm{T}}$. In (1.5), for homogeneous isotropic media, the [6x6] material elastic stiffness matrix $\boldsymbol{D}$ represents in Voigt notation the tensor identity $\sigma_{ij} = 2\mu\varepsilon_{ij} + \lambda\delta_{ij}\varepsilon_{kk}$, where δ_{ij} is the Kronecker delta, and reduces to

$$\boldsymbol{D} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} \lambda+2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda+2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda+2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix} \tag{1.9}$$

where λ is the *first Lamé parameter* and μ the *second Lamé parameter*, equal to the shear modulus G.
Equivalently, the above material stiffness matrix can be expressed as a function of the

Young's modulus E and the Poisson's ratio ν, broadly used in engineering.

$$\boldsymbol{D} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \tag{1.10}$$

where the following identities have been applied

$$E = \frac{\mu(3\lambda + 2\mu)}{\lambda + \mu} \qquad \nu = \frac{\lambda}{2(\lambda + \mu)}$$

1.2. Mixed variational formulation of the continuous problem

In this section a mixed three-field variational formulation is presented based on the *Hu-Washizu functional* ([29]). The formulation is the most general one since it assumes the three unknown fields $\boldsymbol{u}$, $\boldsymbol{\varepsilon}$ and $\boldsymbol{\sigma}$ of the elastic problem to be independent. Under suitable assumptions, the formulation is equivalent to other reduced formulations, such as *Hellinger-Reissner* two-field formulation, which assumes only $\boldsymbol{u}$ and $\boldsymbol{\sigma}$, as shown in [13]. Eventually, under proper requirements based on energy conservation at the discrete level, the Hu-Washizu formulation can be brought back to the irreducible primal one-field displacement-based formulation, corresponding to the widely known *total potential energy* functional.

Definition 1.1 (Hu-Washizu functional)**.** *The Hu-Washizu functional* Π *is defined as*

$$\Pi : U \times \mathcal{E} \times \Sigma \to \mathbb{R}$$

$$\Pi(\boldsymbol{u}, \boldsymbol{\varepsilon}, \boldsymbol{\sigma}) = \frac{1}{2}\int_\Omega \boldsymbol{\varepsilon}^{\mathrm{T}} \boldsymbol{D} \boldsymbol{\varepsilon}\, d\Omega - \int_\Omega \boldsymbol{\sigma}^{\mathrm{T}} (\boldsymbol{\varepsilon} - \boldsymbol{S}\boldsymbol{u})\, d\Omega - \int_\Omega \boldsymbol{u}^{\mathrm{T}} \boldsymbol{b}\, d\Omega - \int_{\partial_p\Omega} \boldsymbol{u}^{\mathrm{T}} \boldsymbol{p}\, d\Sigma \tag{1.11}$$

where U*,* $\mathcal{E}$ *and* Σ *are suitable functional spaces defined in the domain* Ω*.*

The last integral in (1.11) is computed only on the free part $\partial_p\Omega$ of the boundary surface since on the constrained part $\partial_u\Omega$ it is assumed a priori that $\boldsymbol{u} = \overline{\boldsymbol{u}}$.

1.2.1. Stationarity of the continuous mixed functional

The following result allows to find the relationships between the three unknown fields and will be used to build a general finite element scheme, exploitable for a virtual element setting.

Theorem 1.1 (Hu-Washizu principle). *The true solution of the elastic problem amongst all those admissible is such that the* action *defined by the Hu-Washizu functional is stationary.*

Proof. Performing a variation of the functional and setting it to 0 yields to

$$\begin{aligned}\delta\Pi = \int_\Omega \boldsymbol{\delta\varepsilon}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{\varepsilon}\, d\Omega - \int_\Omega \boldsymbol{\sigma}^{\mathrm{T}}(\boldsymbol{\delta\varepsilon} - \boldsymbol{S}\boldsymbol{\delta u})\, d\Omega - \int_\Omega \boldsymbol{\delta\sigma}^{\mathrm{T}}(\boldsymbol{\varepsilon} - \boldsymbol{S}\boldsymbol{u})\, d\Omega + \\ - \int_\Omega \boldsymbol{\delta u}^{\mathrm{T}}\boldsymbol{b}\, d\Omega - \int_{\partial_p\Omega} \boldsymbol{\delta u}^{\mathrm{T}}\boldsymbol{p}\, d\Sigma = 0 \quad \forall \boldsymbol{\delta u}, \boldsymbol{\delta\varepsilon}, \boldsymbol{\delta\sigma}\end{aligned} \tag{1.12}$$

Integrating by parts the integral involving $\boldsymbol{\sigma}^T\boldsymbol{S}\boldsymbol{\delta u}$ and recalling that the variation of the displacement field $\boldsymbol{\delta u}$ on $\partial_u\Omega$ is null one has

$$\begin{aligned}\int_\Omega \boldsymbol{\sigma}^{\mathrm{T}}\boldsymbol{S}\boldsymbol{\delta u}\, d\Omega &= \int_{\partial\Omega=\partial_p\Omega\cup\partial_u\Omega} \boldsymbol{\delta u}^{\mathrm{T}}\mathbb{N}\boldsymbol{\sigma}\, d\Sigma - \int_\Omega \boldsymbol{\delta u}^{\mathrm{T}}\boldsymbol{S}^{\mathrm{T}}\boldsymbol{\sigma}\, d\Omega = \\ &= \int_{\partial_p\Omega} \boldsymbol{\delta u}^{\mathrm{T}}\mathbb{N}\boldsymbol{\sigma}\, d\Sigma - \int_\Omega \boldsymbol{\delta u}^{\mathrm{T}}\boldsymbol{S}^{\mathrm{T}}\boldsymbol{\sigma}\, d\Omega\end{aligned} \tag{1.13}$$

so that

$$\begin{aligned}\delta\Pi = \int_\Omega \boldsymbol{\delta\varepsilon}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{\varepsilon}\, d\Omega - \int_\Omega \boldsymbol{\sigma}^{\mathrm{T}}\boldsymbol{\delta\varepsilon}\, d\Omega + \int_{\partial_p\Omega} \boldsymbol{\delta u}^{\mathrm{T}}\mathbb{N}\boldsymbol{\sigma}\, d\Sigma - \int_\Omega \boldsymbol{\delta u}^{\mathrm{T}}\boldsymbol{S}^{\mathrm{T}}\boldsymbol{\sigma}\, d\Omega + \\ - \int_\Omega \boldsymbol{\delta\sigma}^{\mathrm{T}}(\boldsymbol{\varepsilon} - \boldsymbol{S}\boldsymbol{u})\, d\Omega - \int_\Omega \boldsymbol{\delta u}^{\mathrm{T}}\boldsymbol{b}\, d\Omega - \int_{\partial_p\Omega} \boldsymbol{\delta u}^{\mathrm{T}}\boldsymbol{p}\, d\Sigma = 0 \quad \forall \boldsymbol{\delta u}, \boldsymbol{\delta\varepsilon}, \boldsymbol{\delta\sigma}\end{aligned} \tag{1.14}$$

Finally, gathering the terms involving common variations leads to

$$\begin{aligned}\delta\Pi = &- \int_\Omega \boldsymbol{\delta u}^{\mathrm{T}}(\boldsymbol{S}^{\mathrm{T}}\boldsymbol{\sigma} + \boldsymbol{b})\, d\Omega + \int_\Omega \boldsymbol{\delta\varepsilon}^{\mathrm{T}}(\boldsymbol{D}\boldsymbol{\varepsilon} - \boldsymbol{\sigma})\, d\Omega + \\ &- \int_\Omega \boldsymbol{\delta\sigma}^{\mathrm{T}}(\boldsymbol{\varepsilon} - \boldsymbol{S}\boldsymbol{u})\, d\Omega + \int_{\partial_p\Omega} \boldsymbol{\delta u}^{\mathrm{T}}(\mathbb{N}\boldsymbol{\sigma} - \boldsymbol{p})\, d\Sigma = 0 \quad \forall \boldsymbol{\delta u}, \boldsymbol{\delta\varepsilon}, \boldsymbol{\delta\sigma}\end{aligned} \tag{1.15}$$

Equation (1.15) is the weak form of the elastic problem described in 1.1. By the *fundamental lemma of the calculus of variations*, the variation of the functional $\delta\Pi$ being null for *any* admissible variation of the three fields implies that the four quantities in round brackets above are null in a suitable sense in the domains described by their respective integrals, giving rise to four *Euler's equations* corresponding to (1.1), (1.5), (1.3) and (1.2). Hence the set ($\boldsymbol{u}$, $\boldsymbol{\varepsilon}$, $\boldsymbol{\sigma}$) satisfying the governing equations for the linear elastic problem makes the Hu-Washizu functional stationary. □

1.3. Finite element approximation

1.3.1. Discretization

The preprocessing phase, or *meshing*, in finite element methods consists of generating a partition of the domain $\mathcal{P}_h$ where the governing equations of the problem are being studied. The volume of the body then becomes

$$|\Omega| \approx |\mathcal{P}_h| = \sum_{e=1}^{n_e} |\Omega_e|$$

where n_e is the number of elements composing the mesh and e is the general index representing the element e.
By linearity of the integral operator, the definite integrals in the domain Ω become

$$\int_{\Omega} (\cdot)\, d\Omega \approx \sum_{e=1}^{n_e} \int_{\Omega_e} (\cdot)\, d\Omega \tag{1.16}$$

The same discretization procedure can be applied to the Hu-Washizu functional, so that

$$\Pi \approx \sum_{e=1}^{n_e} \Pi_e \tag{1.17}$$

where

$$\Pi_e : U_e \times \mathcal{E}_e \times \Sigma_e \to \mathbb{R}$$

$$\begin{aligned} \Pi_e(\boldsymbol{u}, \boldsymbol{\varepsilon}, \boldsymbol{\sigma}) = \frac{1}{2} \int_{\Omega_e} \boldsymbol{\varepsilon}^{\mathrm{T}}(\boldsymbol{\xi}) \boldsymbol{D} \boldsymbol{\varepsilon}(\boldsymbol{\xi})\, d\Omega - \int_{\Omega_e} \boldsymbol{\sigma}^{\mathrm{T}}(\boldsymbol{\xi}) \Big(\boldsymbol{\varepsilon}(\boldsymbol{\xi}) - \boldsymbol{S}\boldsymbol{u}(\boldsymbol{\xi}) \Big)\, d\Omega\, + \\ - \int_{\Omega_e} \boldsymbol{u}^{\mathrm{T}}(\boldsymbol{\xi}) \boldsymbol{b}(\boldsymbol{\xi})\, d\Omega - \int_{\partial_p \Omega_e} \boldsymbol{u}^{\mathrm{T}}(\boldsymbol{\xi}) \boldsymbol{p}(\boldsymbol{\xi})\, d\Sigma \end{aligned} \tag{1.18}$$

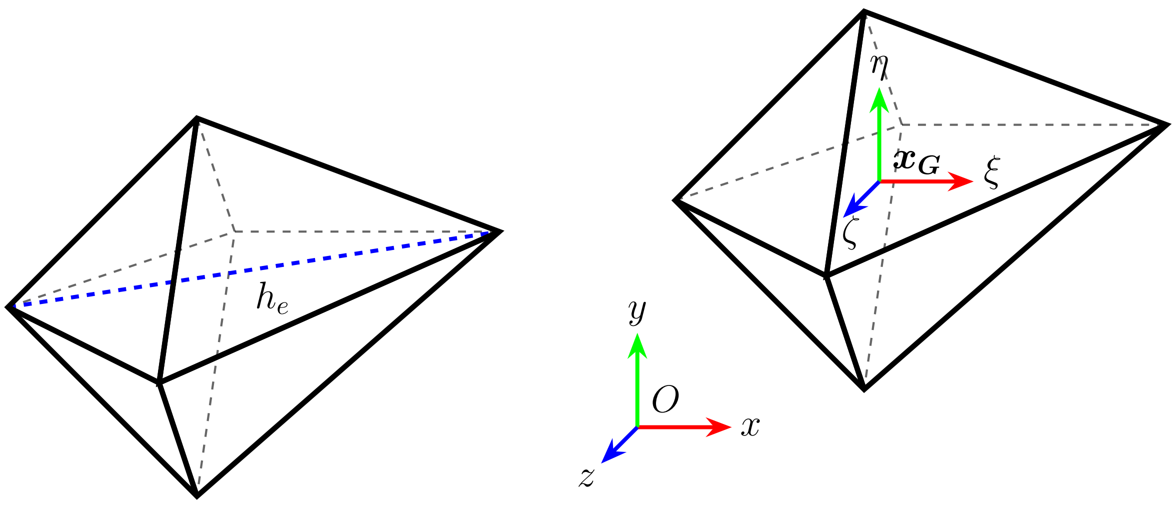
in which U_e, $\mathcal{E}_e$ and Σ_e are suitable functional spaces defined in the domain Ω_e and $\boldsymbol{\xi}$ is the vector of non-dimensional barycentric local coordinates of the element (Figure 1.2b), given by

$$\boldsymbol{\xi} = \begin{Bmatrix} \xi \\ \eta \\ \zeta \end{Bmatrix} = \frac{\boldsymbol{x} - \boldsymbol{x_G}}{h_e} = \begin{Bmatrix} \frac{x - x_G}{h_e} \\ \frac{y - y_G}{h_e} \\ \frac{z - z_G}{h_e} \end{Bmatrix} \tag{1.19}$$

with $\boldsymbol{x_G} = \{x_G \; y_G \; z_G\}^{\mathrm{T}}$ being the centroid of the element and h_e the *diameter* of the element, or *element size*, namely the maximum distance between to points belonging to the element boundary (Figure 1.2a)

$$h_e = \max_{\boldsymbol{x_1},\boldsymbol{x_2} \in \partial\Omega_e} \|\boldsymbol{x_1} - \boldsymbol{x_2}\|$$

The element size plays a fundamental role in convergence analysis of finite element methods.



(a) Element diameter. (b) Non-dimensional barycentric local coordinates.

Figure 1.2: Example of element diameter and barycentric local coordinates for a polyhedron.

The discretization (1.17) of the functional holds in view of (1.16) for the integrals of (1.18) over Ω and by global continuity of the displacement field $\boldsymbol{u}$ and equal modulus and opposite sign of the elements surface tractions $\boldsymbol{p}$ across the elements boundaries. More precisely,

$$\sum_{e=1}^{n_e} \int_{\partial_p\Omega_e} \boldsymbol{u}^{\mathrm{T}}(\boldsymbol{\xi})\boldsymbol{p}(\boldsymbol{\xi})\, d\Sigma = 0$$

An approximation of the three independent fields at elemental level $\boldsymbol{u}$, $\boldsymbol{\varepsilon}$ and $\boldsymbol{\sigma}$ is introduced ([4, 8, 30])

$$\boldsymbol{u}(\boldsymbol{\xi}) \approx \boldsymbol{u}^h(\boldsymbol{\xi}) = \boldsymbol{N}_u(\boldsymbol{\xi})\hat{\boldsymbol{u}} \tag{1.20}$$

$$\boldsymbol{\varepsilon}(\boldsymbol{\xi}) \approx \boldsymbol{\varepsilon}^h(\boldsymbol{\xi}) = \boldsymbol{N}_\varepsilon(\boldsymbol{\xi})\hat{\boldsymbol{\varepsilon}} \tag{1.21}$$

$$\boldsymbol{\sigma}(\boldsymbol{\xi}) \approx \boldsymbol{\sigma}^h(\boldsymbol{\xi}) = \boldsymbol{N}_\sigma(\boldsymbol{\xi})\hat{\boldsymbol{\sigma}} \tag{1.22}$$

where superscript h indicates the field is approximated and matrices $\boldsymbol{N}_{(\cdot)}$ map a discrete quantity $\hat{(\cdot)}$ into a continuous function and are known in the finite element literature as *shape functions*. More specifically, in 3D the three ansatze $\boldsymbol{N}_{\boldsymbol{u}}$, $\boldsymbol{N}_{\varepsilon}$, $\boldsymbol{N}_{\boldsymbol{\sigma}}$ have respectively dimensions $[3 \times n_u]$, $[6 \times n_\varepsilon]$ and $[6 \times n_\sigma]$, where $n_{(\cdot)}$ is the number of parameters required to describe the discrete field $\hat{(\cdot)}$. In standard primal FE, $\hat{\boldsymbol{u}}$ are nodal displacement values, while in a mixed formulation where also strains or stresses are unknowns, $\hat{\boldsymbol{\varepsilon}}$ and $\hat{\boldsymbol{\sigma}}$ are not necessarily nodal values and might loose physical meaning. All the maps $\boldsymbol{N}_{\boldsymbol{u}}$, $\boldsymbol{N}_{\varepsilon}$ and $\boldsymbol{N}_{\boldsymbol{\sigma}}$ are *locally* continuous, i.e. continuous in the element interior Ω_e, while only the first one is also *globally* continuous so that the approximate displacement field $\boldsymbol{u}^h \in [C^0(\Omega)]^3$. The last requirement is relaxed for discontinuous Galerking methods ([9]).
Henceforth, the superscript h will be omitted for the sake of conciseness when referring to the local approximated fields.
The assumed strain field must satisfy a particular case of *integrability conditions* undergoing the name of *internal compatibility*, namely the body must exhibit strains for which a continuous, single-valued displacement field is guaranteed ([26]). More specifically, solving the differential strain-displacement relations in the vector unknown u, given a strain field ε, is an overdetermined problem as it involves six independent equations and only three unknowns. Hence, additional equations for the strain field must be provided when trying to reconstruct the displacement field, which is the goal of finite element schemes. The mechanical reason behind this requirement lies in the fact that no overlapping nor tears of the constitutive material are allowed. The following theorem provides the above-mentioned differential equations that a generic strain field must satisfy in order to guarantee the existence of the corresponding continuous, single-valued displacement field. We shall briefly abandon Voigt notation when dealing with this theorem, where tensor calculus comes into help, so that ε is a 2nd order tensor. Moreover, to indicate partial derivatives $\frac{\partial}{\partial(\cdot)}(\diamond)$ the following notation will be adopted $(\diamond)_{,(\cdot)}$

Theorem 1.2 (Compatibility conditions)**.** *If the displacement field $\boldsymbol{u}$ is continuous and single-valued, then the following identity holds for the strain field $\boldsymbol{\varepsilon}$ in tensor form*

$$\nabla \times (\nabla \times \boldsymbol{\varepsilon}) = \boldsymbol{0} \tag{1.23}$$

or, using Einstein summation convention,

$$e_{ikr} e_{jls} \varepsilon_{kl,ij} = 0 \tag{1.24}$$

where e_{ijk} is the Levi-Civita *symbol.*
If the body is simply connected, the above conditions are also sufficient for the existence

of a continuous single-valued displacement field.

In (1.24) the *Levi-Civita*, or *permutation* symbol has been used, usually indicated by ε_{ijk} and here represented by e_{ijk} not to be confused with the strain symbol. In three dimensions, it is defined as

$$e_{ijk} = \begin{cases} 1 & \text{if } (i,j,k) \text{ is an even permutation of} (1,2,3) \\ -1 & \text{if } (i,j,k) \text{ is an odd permutation of } (1,2,3) \\ 0 & \text{if any index is repeated} \end{cases} \tag{1.25}$$

Proof. The proof for the necessary condition is reported here, while for the sufficient condition a detailed explanation can be found in [26], together with an exhaustive chapter on tensor calculus.
In the hypothesis of infinitesimal strains, the gradient of the displacements $\nabla \boldsymbol{u}$ can be decomposed in a symmetric part corresponding to the strains $\boldsymbol{\varepsilon}$ and a skew-symmetric part $\boldsymbol{\omega}$

$$\begin{gathered} \nabla \boldsymbol{u} = \boldsymbol{\varepsilon} + \boldsymbol{\omega} \\ \boldsymbol{\varepsilon} = \frac{1}{2}\Big(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^{\mathrm{T}}\Big) \qquad \boldsymbol{\omega} = \frac{1}{2}\Big(\nabla \boldsymbol{u} - (\nabla \boldsymbol{u})^{\mathrm{T}}\Big) \\ \varepsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}) \qquad \omega_{ij} = \frac{1}{2}(u_{i,j} - u_{j,i}) \end{gathered} \tag{1.26}$$

Taking the gradient of $\boldsymbol{\omega}$

$$\begin{aligned} \omega_{ij,k} = \frac{1}{2}(u_{i,j} - u_{j,i})_k = \frac{1}{2}(u_{i,jk} + u_{k,ij} - u_{k,ij} - u_{j,ik}) = \\ = (u_{i,kj} + u_{k,ij} - u_{k,ji} - u_{j,ki}) = \varepsilon_{ik,j} - \varepsilon_{jk,i} \end{aligned} \tag{1.27}$$

If $\boldsymbol{\omega}$ is continuously differentiable, differentiating another time and exploiting Schwarz theorem yields to

$$\omega_{ij,kl} = \omega_{ij,lk} \tag{1.28}$$

hence

$$\varepsilon_{ik,jl} - \varepsilon_{jk,il} - \varepsilon_{il,jk} + \varepsilon_{jl,ik} = 0 \tag{1.29}$$

which correspond to $3^4 = 81$ equations by letting the indices i, j, k, l vary within the values 1,2,3. These, however are not independent since the tensor $\boldsymbol{\varepsilon}$ is symmetric and reduce to the 9 equations in the free indices r and s of (1.24). The latter are, in turn, again symmetric in r and s and eventually give rise to 6 independent internal compatibility equations, now written in Voigt notation,

$$\begin{cases} \varepsilon_{x,yy} + \varepsilon_{y,xx} - \gamma_{xy} = 0 \\ \varepsilon_{y,zz} + \varepsilon_{z,yy} - \gamma_{yz} = 0 \\ \varepsilon_{z,xx} + \varepsilon_{x,zz} - \gamma_{xz} = 0 \\ (\gamma_{xy,z} - \gamma_{yz,x} + \gamma_{xz,y})_{,x} - 2\varepsilon_{x,yz} = 0 \\ (\gamma_{yz,x} - \gamma_{xz,y} + \gamma_{xy,z})_{,y} - 2\varepsilon_{y,xz} = 0 \\ (\gamma_{xz,y} - \gamma_{xy,z} + \gamma_{yz,x})_{,z} - 2\varepsilon_{z,xy} = 0 \end{cases} \tag{1.30}$$

Finally, the equivalence of (1.23) with the identity (1.24) is obtained applying twice the definition of curl of a 2nd order tensor

$$(\nabla \times \boldsymbol{\varepsilon})_{sk} = e_{jls}\varepsilon_{kl,j}$$

so that

$$\Big(\nabla \times (\nabla \times \boldsymbol{\varepsilon})\Big)_{sk} = e_{ikr}(\nabla \times \boldsymbol{\varepsilon})_{sk,i} = e_{ikr}e_{jls}\varepsilon_{kl,ji} = e_{ikr}e_{jls}\varepsilon_{kl,ij}$$

$\square$

It is important to remark that the governing equations of the elastic problem 1.1, 1.3 and 1.5 already close the problem setting in a displacement-based procedure without forcing internal compatibility conditions, which are actually automatically satisfied. However, following a mixed variational scheme, this is not necessarily the case and internal compatibility is required to correctly choose an admissible strain field.

The approximated strain and stress fields have to be chosen such that the energy given by their scalar product is conserved. To perform the scalar product we must have $n_\varepsilon = n_\sigma$ and hence

$$\hat{\boldsymbol{\sigma}}^{\mathrm{T}}\hat{\boldsymbol{\varepsilon}} = \int_{\Omega_e} \boldsymbol{\sigma}^{\mathrm{T}}\boldsymbol{\varepsilon}\, d\Omega = \hat{\boldsymbol{\sigma}}^{\mathrm{T}}\left(\int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{\sigma}}^{\mathrm{T}}\boldsymbol{N}_{\varepsilon}\, d\Omega\right)\hat{\boldsymbol{\varepsilon}} \tag{1.31}$$

so that

$$\int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{\sigma}}^{\mathrm{T}}\boldsymbol{N}_{\varepsilon}\, d\Omega = \boldsymbol{I} \tag{1.32}$$

where $\boldsymbol{I}$ is the $[n_\varepsilon \times n_\varepsilon]$ identity matrix. If (1.32) holds, the variables $\hat{\boldsymbol{\sigma}}$ and $\hat{\boldsymbol{\varepsilon}}$ are said to be *generalized variables*. Possible choices for the stress ansatz $\boldsymbol{N}_{\boldsymbol{\sigma}}$ so that (1.32) is satisfied are

- first possible choice

$$\boldsymbol{N}_{\boldsymbol{\sigma}} = \boldsymbol{D}\boldsymbol{N}_{\varepsilon}\left(\int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{\varepsilon}}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{N}_{\varepsilon}\, d\Omega\right)^{-1} = \boldsymbol{D}\boldsymbol{N}_{\boldsymbol{\varepsilon}}\boldsymbol{E}^{-1} \tag{1.33}$$

where the square $[n_\varepsilon \times n_\varepsilon]$ invertible *elastic matrix* $\boldsymbol{E}$ is given by

$$\boldsymbol{E} = \int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{\varepsilon}}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{N}_{\varepsilon}\, d\Omega \tag{1.34}$$

- second possible choice, proposed by Corradi in [10] for elasto-plasticity

$$\boldsymbol{N}_{\boldsymbol{\sigma}} = \boldsymbol{N}_{\varepsilon}\left(\int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{\varepsilon}}^{\mathrm{T}}\boldsymbol{N}_{\varepsilon}\, d\Omega\right)^{-1} = \boldsymbol{N}_{\boldsymbol{\varepsilon}}\boldsymbol{G}^{-1} \tag{1.35}$$

where the square $[n_\varepsilon \times n_\varepsilon]$ invertible matrix $\boldsymbol{G}$ is given by

$$\boldsymbol{G} = \int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{\varepsilon}}^{\mathrm{T}}\boldsymbol{N}_{\varepsilon}\, d\Omega \tag{1.36}$$

Exploiting the three ansatze for the unknown fields given by (1.20), (1.21) and (1.22), the discretized functional Π_e given in (1.18) can be approximated as the function

$$\Pi_e^h : \mathbb{R}^{n_u} \times \mathbb{R}^{n_\varepsilon} \times \mathbb{R}^{n_\sigma} \to \mathbb{R}$$

$$\begin{aligned}\Pi_e^h(\hat{\boldsymbol{u}}, \hat{\boldsymbol{\varepsilon}}, \hat{\boldsymbol{\sigma}}) = \frac{1}{2}\hat{\boldsymbol{\varepsilon}}^{\mathrm{T}}\left(\int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{\varepsilon}}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{N}_{\varepsilon}\, d\Omega\right)\hat{\boldsymbol{\varepsilon}} - \hat{\boldsymbol{\sigma}}^{\mathrm{T}}\left(\int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{\sigma}}^{\mathrm{T}}(\boldsymbol{N}_{\varepsilon}\hat{\boldsymbol{\varepsilon}} - \boldsymbol{S}\boldsymbol{N}_{\boldsymbol{u}}\hat{\boldsymbol{u}})\, d\Omega\right) + \\ - \hat{\boldsymbol{u}}^{\mathrm{T}}\left(\int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{u}}^{\mathrm{T}}\boldsymbol{b}\, d\Omega + \int_{\partial_p\Omega_e} \boldsymbol{N}_{\boldsymbol{u}}^{\mathrm{T}}\boldsymbol{p}\, d\Sigma\right)\end{aligned} \tag{1.37}$$

which, by (1.32), becomes

$$\Pi_e^h(\hat{\boldsymbol{u}}, \hat{\boldsymbol{\varepsilon}}, \hat{\boldsymbol{\sigma}}) = \frac{1}{2}\hat{\boldsymbol{\varepsilon}}^{\mathrm{T}}\boldsymbol{E}\hat{\boldsymbol{\varepsilon}} - \hat{\boldsymbol{\sigma}}^{\mathrm{T}}(\boldsymbol{\varepsilon} - \boldsymbol{C}\hat{\boldsymbol{u}}) - \hat{\boldsymbol{u}}^{\mathrm{T}}\boldsymbol{F}_e \tag{1.38}$$

where $\boldsymbol{E}$ is the $[n_\varepsilon \times n_\varepsilon]$ elastic matrix already defined in (1.34) and where the expressions below have been introduced

- $[n_\varepsilon \times n_u]$ compatibility matrix, obtained by the choice made in (1.35)

$$\boldsymbol{C} = \int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{\sigma}}^{\mathrm{T}}\boldsymbol{S}\boldsymbol{N}_{\boldsymbol{u}}\, d\Omega = \boldsymbol{G}^{-1}\int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{\varepsilon}}^{\mathrm{T}}\boldsymbol{S}\boldsymbol{N}_{\boldsymbol{u}}\, d\Omega = \boldsymbol{G}^{-1}\boldsymbol{A} \tag{1.39}$$

where the $[n_\varepsilon \times n_u]$ matrix $\boldsymbol{A}$ is

$$\boldsymbol{A} = \int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{\varepsilon}}^{\mathrm{T}} \boldsymbol{S} \boldsymbol{N}_{\boldsymbol{u}} \, d\Omega \tag{1.40}$$

- $[n_u \times 1]$ local equivalent nodal forces vector

$$\boldsymbol{F}_e = \int_{\Omega_e} \boldsymbol{N}_{\boldsymbol{u}}^{\mathrm{T}} \boldsymbol{b} \, d\Omega + \int_{\partial_p\Omega_e} \boldsymbol{N}_{\boldsymbol{u}}^{\mathrm{T}} \boldsymbol{p} \, d\Sigma \tag{1.41}$$

In (1.41) the term *nodal* is placed here to maintain consistency with standard FE nomenclature. However, as will be clearer in the following chapters, the vector of discrete unknowns $\hat{\boldsymbol{u}}$ (and hence $\boldsymbol{F}_e$) may not contain only nodal values of the corresponding field, but different quantities, not necessarily leading to a physical interpretation or simple visualization.

1.3.2. Stationarity of the discretized mixed functional

Enforcing the stationarity of the mixed discrete functional (1.38) with respect to the variables $\hat{\boldsymbol{u}}$, $\hat{\boldsymbol{\varepsilon}}$ and $\hat{\boldsymbol{\sigma}}$ one obtains the algebraic governing equations

- equilibrium

$$\partial_{\hat{\boldsymbol{u}}} \Pi_e^h = \boldsymbol{0} \implies \boldsymbol{C}^{\mathrm{T}} \hat{\boldsymbol{\sigma}} = \boldsymbol{F}_e \tag{1.42}$$

- constitutive law

$$\partial_{\hat{\boldsymbol{\varepsilon}}} \Pi_e^h = \boldsymbol{0} \implies \hat{\boldsymbol{\sigma}} = \boldsymbol{E} \hat{\boldsymbol{\varepsilon}} \tag{1.43}$$

- kinematic compatibility

$$\partial_{\hat{\boldsymbol{\sigma}}} \Pi_e^h = \boldsymbol{0} \implies \hat{\boldsymbol{\varepsilon}} = \boldsymbol{C} \hat{\boldsymbol{u}} \tag{1.44}$$

Replacing (1.44) in (1.43) and (1.43) in (1.42) the following local algebraic system is obtained

$$(\boldsymbol{C}^{\mathrm{T}} \boldsymbol{E} \boldsymbol{C}) \hat{\boldsymbol{u}} = \boldsymbol{F}_e \tag{1.45}$$

which in more compact form becomes

$$\boldsymbol{K}_e^c \hat{\boldsymbol{u}} = \boldsymbol{F}_e \tag{1.46}$$

where $\boldsymbol{K}_e^c$ is the $[n_u \times n_u]$ symmetric, positive-semi definite *local stiffness matrix consistent with the strain and displacement models*. Positive definiteness is not yet achieved since at this stage the matrix $\boldsymbol{K}_e^c$ is *at least* 6 times singular, corresponding to the 6 rigid motions a body can undergo in 3D. An analysis of the degree of singularity of the matrix $\boldsymbol{K}_e^c$ is needed to ensure the *stability* of the element. If $n_u - n_\varepsilon \leq 6$ and $\boldsymbol{C}$ has rank equal to $n_u - 6$, then $\boldsymbol{K}_e^c$ has the correct degree of singularity (6), corresponding to the six orthogonal rigid body modes in 3D. Conversely, if $n_u - n_\varepsilon > 6$, $\boldsymbol{K}_e^c$ has a surplus of rank deficiency $n_u - n_\varepsilon - 6$ and *zero-energy modes* can appear, known in the finite element community for structural problems as *hourglass modes* (see Figure 1.3), from the most common characteristic shape exhibited by 2D quadrilateral elements, and often appear in mixed finite element formulations. These modes are spurious deformations as they represent a configuration of indefinite displacements that the element can exhibit under null external forces. It is a trivial consequence of the eigenvalue problem

$$\boldsymbol{K}_e^c \boldsymbol{V}_e = \boldsymbol{\Lambda}_e \boldsymbol{F}_e$$

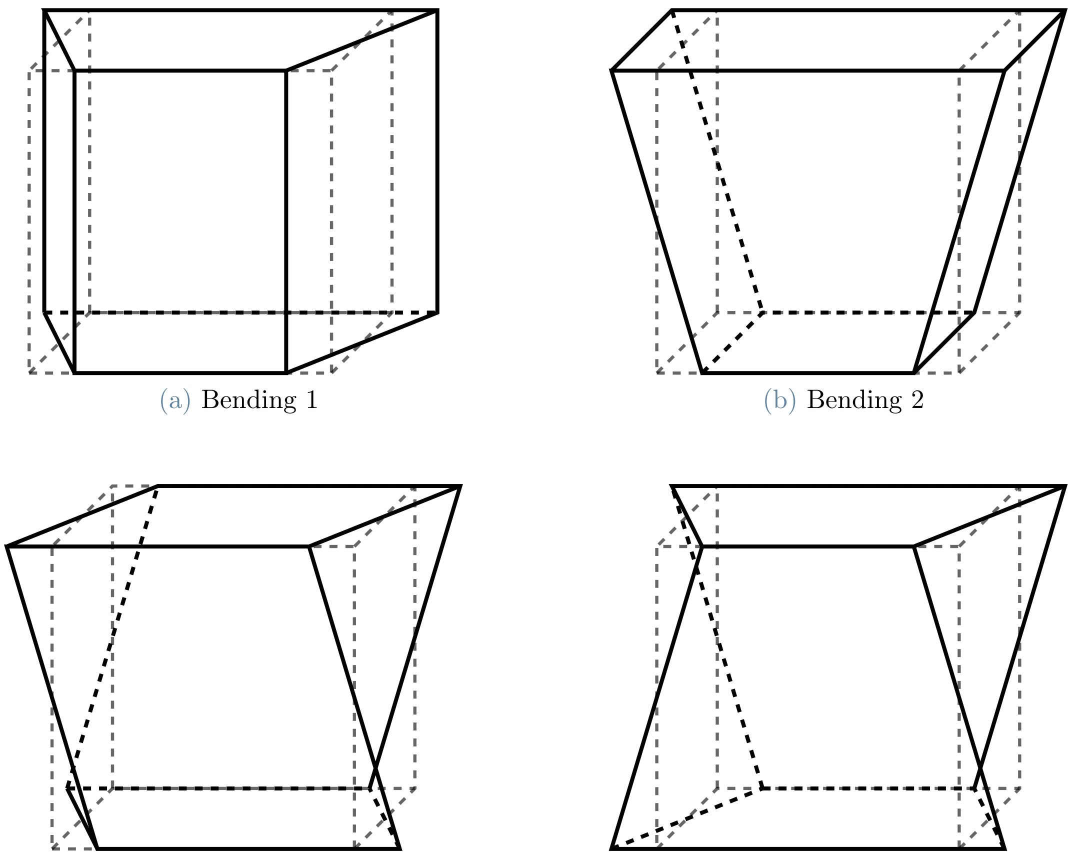
where the $n_u \times n_u$ matrix $\boldsymbol{V}_e$ collects the $n_u \times 1$ eigenvectors $\boldsymbol{v}_e^{(i)}$ corresponding to the eigenvalues $\lambda_e^{(i)}$ gathered on the diagonal matrix $\boldsymbol{\Lambda}_e$, with $i = 1, ..., n_u$. It is clear that an eigenvalue $\lambda_e^{(i)}$ being null implies that the load multiplier of the local equivalent nodal forces $\boldsymbol{F}_e$ is zero, producing a possibly non-zero indefinite displaced configuration described by the corresponding eigenvector $\boldsymbol{v}_e^{(i)}$ in the kernel of $\boldsymbol{K}_e^c$. If the null eigenvalues of the consistent matrix $\boldsymbol{K}_e^c$ are more than 6, the chosen strain field is *not rich enough* to correctly represent all the configurations of the chosen displacement field, and hence require some stabilizing technique to allow the element to be used.

1.3.3. Hourglass modes stabilization

The key to control hourglass modes and suppress the possible rise in the approximated solution of configurations such as those displayed in Figure 1.3 is to add a fictitious stiffness to the element. The mixed continuous functional (1.18) becomes

$$\begin{aligned}\Pi_e(\boldsymbol{u}, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}_H, \boldsymbol{\sigma}) = {} & \frac{1}{2}\int_{\Omega_e} \boldsymbol{\varepsilon}^{\mathrm{T}}(\boldsymbol{\xi})\boldsymbol{D}\boldsymbol{\varepsilon}(\boldsymbol{\xi})\, d\Omega + \frac{1}{2}\int_{\Omega_e} \boldsymbol{\varepsilon}_H^{\mathrm{T}}(\boldsymbol{\xi})\boldsymbol{D}_H\boldsymbol{\varepsilon}_H(\boldsymbol{\xi})\, d\Omega + \\ & - \int_{\Omega_e} \boldsymbol{\sigma}^{\mathrm{T}}(\boldsymbol{\xi})\Big(\boldsymbol{\varepsilon}(\boldsymbol{\xi}) - \boldsymbol{S}\boldsymbol{u}(\boldsymbol{\xi})\Big)\, d\Omega - \int_{\Omega_e} \boldsymbol{u}^{\mathrm{T}}(\boldsymbol{\xi})\boldsymbol{b}(\boldsymbol{\xi})\, d\Omega + \\ & - \int_{\partial_p\Omega_e} \boldsymbol{u}^{\mathrm{T}}(\boldsymbol{\xi})\boldsymbol{p}(\boldsymbol{\xi})\, d\Sigma\end{aligned} \tag{1.47}$$

where here $\boldsymbol{\varepsilon}$ is the deformation field obtained by removing the deformation induced by hourglass modes $\boldsymbol{\varepsilon}_H$ and $\boldsymbol{D}_H$ is a *hourglass fictitious material stiffness matrix.* A possible



(a) Bending 1 (b) Bending 2

(c) Warping (d) Non-physical

Figure 1.3: Some hourglass modes for a cubic element.

choice for $\boldsymbol{D}_H$ is

$$\boldsymbol{D}_H = \frac{1}{6}\mathrm{tr}(\boldsymbol{D}) \tag{1.48}$$

where $\mathrm{tr}(\cdot)$ stands for the trace operator of matrix $(\cdot)$. The discretized form of (1.47) becomes

$$\Pi_e^h(\hat{\boldsymbol{u}}, \hat{\boldsymbol{u}}_H, \hat{\boldsymbol{\varepsilon}}, \hat{\boldsymbol{\sigma}}) = \frac{1}{2}\hat{\boldsymbol{\varepsilon}}^{\mathrm{T}}\boldsymbol{E}\hat{\boldsymbol{\varepsilon}} + \frac{1}{2}\hat{\boldsymbol{u}}_H^{\mathrm{T}}\left(\int_{\Omega_e}\boldsymbol{B}^{\mathrm{T}}\boldsymbol{D}_H\boldsymbol{B}\,d\Omega\right)\hat{\boldsymbol{u}}_H - \hat{\boldsymbol{\sigma}}^{\mathrm{T}}(\boldsymbol{\varepsilon} - \boldsymbol{C}\hat{\boldsymbol{u}}) - \hat{\boldsymbol{u}}^{\mathrm{T}}\boldsymbol{F}_e \tag{1.49}$$

where the hourglass strains $\boldsymbol{\varepsilon}$ have been approximated with the $[n_H \times 1]$ vector of *unknown* hourglass discrete displacements $\hat{\boldsymbol{u}}_H$ through the standard compatibility operator in finite elements $\boldsymbol{B}$

$$\boldsymbol{\varepsilon}_H(\boldsymbol{\xi}) = \boldsymbol{S}\boldsymbol{N}_{\boldsymbol{u}}(\boldsymbol{\xi})\hat{\boldsymbol{u}}_H = \boldsymbol{B}(\boldsymbol{\xi})\hat{\boldsymbol{u}}_H \tag{1.50}$$

At this point the goal is to extract the vector $\hat{\boldsymbol{u}}_H$ from the vector $\hat{\boldsymbol{u}}$. By splitting the displacement parameters $\hat{\boldsymbol{u}}$ in a $[n_u \times 1]$ vector describing the deformations and rigid body motions $\hat{\boldsymbol{u}}_{D+R}$ and a $[n_u \times 1]$ vector describing the hourglass modes $\hat{\boldsymbol{u}}_H$, the displacement field can be expressed as

$$\boldsymbol{u}(\boldsymbol{\xi}) = \boldsymbol{N_u}(\boldsymbol{\xi})\hat{\boldsymbol{u}} = \boldsymbol{N_u}(\boldsymbol{\xi})(\hat{\boldsymbol{u}}_{D+R} + \hat{\boldsymbol{u}}_H) = \boldsymbol{N_u}(\boldsymbol{\xi})(\boldsymbol{T}_{D+R}\hat{\boldsymbol{p}}_{D+R} + \boldsymbol{T}_H\hat{\boldsymbol{p}}_H) \tag{1.51}$$

where the vectors $\hat{\boldsymbol{u}}_{D+R}$ and $\hat{\boldsymbol{u}}_H$ have been described by the *natural parameters* $[n_{D+R} \times 1]$ vector $\hat{\boldsymbol{p}}_{D+R}$ and $[n_H \times 1]$ vector $\hat{\boldsymbol{p}}_H$ through the $[n_u \times n_{D+R}]$ matrix $\boldsymbol{T}_{D+R}$ and $[n_u \times n_H]$ matrix $\boldsymbol{T}_H$ respectively. By matrix augmentation it is possible to obtain the square and invertible $[n_u \times n_u]$ matrix

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{T}_{D+R} & \boldsymbol{T}_H \end{bmatrix} \tag{1.52}$$

and gathering the natural parameters in the $[n_u \times 1]$ vector $\hat{\boldsymbol{p}}$

$$\hat{\boldsymbol{p}} = \begin{Bmatrix} \hat{\boldsymbol{p}}_{D+R} \\ \hat{\boldsymbol{p}}_H \end{Bmatrix} \tag{1.53}$$

so that

$$\hat{\boldsymbol{u}} = \boldsymbol{T}\hat{\boldsymbol{p}} \tag{1.54}$$

An important property of the decomposition described in (1.51) is the *orthogonality* between deformative or rigid body modes and hourglass modes

$$(\hat{\boldsymbol{u}}_{D+R})^{\mathrm{T}}\hat{\boldsymbol{u}}_H = \boldsymbol{0} \tag{1.55}$$

which translates into

$$(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_H = \boldsymbol{0} \tag{1.56}$$

Exploiting this last condition applied to (1.51) yields to

$$(\boldsymbol{T}_{D+R})^{\mathrm{T}}\hat{\boldsymbol{u}} = (\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\hat{\boldsymbol{p}}_{D+R} + (\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_H\hat{\boldsymbol{p}}_H = (\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\hat{\boldsymbol{p}}_{D+R} \tag{1.57}$$

so that the natural parameters corresponding to deformative or rigid body modes $\hat{\boldsymbol{p}}_{D+R}$ can be extracted

$$\hat{\boldsymbol{p}}_{D+R} = \left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-1}(\boldsymbol{T}_{D+R})^{\mathrm{T}}\hat{\boldsymbol{u}} \tag{1.58}$$

and plugged into (1.51) to obtain

$$\hat{\boldsymbol{u}} = \boldsymbol{T}_{D+R}\left\{\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-1}(\boldsymbol{T}_{D+R})^{\mathrm{T}}\hat{\boldsymbol{u}}\right\} + \boldsymbol{T}_H\hat{\boldsymbol{p}}_H \tag{1.59}$$

and hence, rearranging, $\hat{\boldsymbol{u}}_H$ is finally obtained

$$\hat{\boldsymbol{u}}_H = \boldsymbol{T}_H \hat{\boldsymbol{p}}_H = \left\{ \boldsymbol{I} - \boldsymbol{T}_{D+R} \left[(\boldsymbol{T}_{D+R})^{\mathrm{T}} \boldsymbol{T}_{D+R} \right]^{-1} (\boldsymbol{T}_{D+R})^{\mathrm{T}} \right\} \hat{\boldsymbol{u}} = \boldsymbol{H}\hat{\boldsymbol{u}} \tag{1.60}$$

In the above equation the $[n_u \times n_u]$ *hourglass matrix* $\boldsymbol{H}$ has been introduced

$$\boldsymbol{H} = \boldsymbol{I} - \boldsymbol{T}_{D+R} \left[(\boldsymbol{T}_{D+R})^{\mathrm{T}} \boldsymbol{T}_{D+R} \right]^{-1} (\boldsymbol{T}_{D+R})^{\mathrm{T}} \tag{1.61}$$

and it allows the computation of the extra term responsible for the stabilization contained in the functional (1.49) through the construction of matrix $\boldsymbol{T}_{D+R}$:

$$\Pi_e^h(\hat{\boldsymbol{u}}, \hat{\boldsymbol{\varepsilon}}, \hat{\boldsymbol{\sigma}}) = \frac{1}{2}\hat{\boldsymbol{\varepsilon}}^{\mathrm{T}} \boldsymbol{E} \hat{\boldsymbol{\varepsilon}} + \frac{1}{2}\hat{\boldsymbol{u}}^{\mathrm{T}} \boldsymbol{K}_e^s \hat{\boldsymbol{u}} - \hat{\boldsymbol{\sigma}}^{\mathrm{T}}(\boldsymbol{\varepsilon} - \boldsymbol{C}\hat{\boldsymbol{u}}) - \hat{\boldsymbol{u}}^{\mathrm{T}} \boldsymbol{F}_e \tag{1.62}$$

where $\boldsymbol{K}_e^s$ is the $[n_u \times n_u]$ *local stabilizing stiffness matrix*

$$\boldsymbol{K}_e^s = \boldsymbol{H}^{\mathrm{T}} \left(\int_{\Omega_e} \boldsymbol{B}^{\mathrm{T}} \boldsymbol{D}_H \boldsymbol{B} \, d\Omega \right) \boldsymbol{H} \tag{1.63}$$

The integral contained in round brackets has to be approximated since it contains the fictitious matrix $\boldsymbol{D}_H$ and the differential operator $\boldsymbol{B}$ which in turn includes the shape functions for the displacement field $\boldsymbol{N}_{\boldsymbol{u}}$, not always explicitly known, as will be in the case of the virtual element method. The stabilizing local stiffness matrix is required to scale with respect to the elements and meshes partitioning the domain the same way the local consistent matrix does ([7, 25]). Under a suitable choice of the degrees of freedom (so that they scale as 1, as will be clear in the following chapters), the above integral is hence required to scale according only to the problem at hands and the dimension it is embedded in. For second-order differential problems (as stationary elasticity) it has to scale as 1 in two dimensions, and as $\frac{1}{h}$ in three dimensions.

A possible choice to approximate such integral is the *scalar-based stabilization*, used in [3], and consisting to set the above to $\frac{1}{2}\mathrm{tr}(\boldsymbol{K}_e^c)\boldsymbol{I}$, so that

$$\boldsymbol{K}_e^s = \frac{1}{2}\mathrm{tr}(\boldsymbol{K}_e^c)\boldsymbol{H}^{\mathrm{T}}\boldsymbol{H} \tag{1.64}$$

Another choice (presented, e.g., in [23]) is to perform *diagonal matrix-based stabilization*, approximating the integral of (1.63) with a diagonal matrix $\boldsymbol{\Lambda}$, whose elements in row i and column j are obtained as follows

$$[\Lambda]_{ij} = \delta_{ij} \max \left\{ [\boldsymbol{K}_e^c]_{ij}, \frac{\alpha_0}{n_{\boldsymbol{D}}} \mathrm{tr}(\boldsymbol{D}) \right\} \tag{1.65}$$

where α_0 is a coefficient between 0 and 1 providing a lower bound for the diagonal matrix $\boldsymbol{\Lambda}$ and could be set as $\frac{1}{3}$ in 2D and $\frac{1}{9}$ in 3D, and $n_{\boldsymbol{D}}$ is the dimension of the matrix $\boldsymbol{D}$ (3 in 2D and 6 in 3D). The expression (1.63) then becomes

$$\boldsymbol{K}_e^s = \boldsymbol{H}^{\mathrm{T}}\boldsymbol{\Lambda}\boldsymbol{H} \tag{1.66}$$

Proposition 1.1. *The hourglass matrix* $\boldsymbol{H}$ *fulfills the property*

$$\boldsymbol{H} = \boldsymbol{H}^{\mathrm{T}}\boldsymbol{H} \tag{1.67}$$

Proof. By direct substitution

$$\begin{aligned}\boldsymbol{H}^{\mathrm{T}}\boldsymbol{H} = {} & \left\{\boldsymbol{I} - \boldsymbol{T}_{D+R}\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-1}(\boldsymbol{T}_{D+R})^{\mathrm{T}}\right\}^{\mathrm{T}}\left\{\boldsymbol{I} - \boldsymbol{T}_{D+R}\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-1}\right. \\ & \left.(\boldsymbol{T}_{D+R})^{\mathrm{T}}\right\} = \boldsymbol{I} - \boldsymbol{T}_{D+R}\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-1}(\boldsymbol{T}_{D+R})^{\mathrm{T}} + \\ & -\left\{\boldsymbol{T}_{D+R}\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-1}(\boldsymbol{T}_{D+R})^{\mathrm{T}}\right\}^{\mathrm{T}} + \left\{\boldsymbol{T}_{D+R}\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-1}\right. \\ & \left.(\boldsymbol{T}_{D+R})^{\mathrm{T}}\right\}^{\mathrm{T}}\boldsymbol{T}_{D+R}\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-1}(\boldsymbol{T}_{D+R})^{\mathrm{T}} = \\ & = \boldsymbol{I} - \boldsymbol{T}_{D+R}\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-1}(\boldsymbol{T}_{D+R})^{\mathrm{T}} - \boldsymbol{T}_{D+R}\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-\mathrm{T}} \\ & (\boldsymbol{T}_{D+R})^{\mathrm{T}} + \boldsymbol{T}_{D+R}\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-\mathrm{T}}(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-1} \\ & (\boldsymbol{T}_{D+R})^{\mathrm{T}} = \boldsymbol{I} - \boldsymbol{T}_{D+R}\left[(\boldsymbol{T}_{D+R})^{\mathrm{T}}\boldsymbol{T}_{D+R}\right]^{-1}(\boldsymbol{T}_{D+R})^{\mathrm{T}} = \boldsymbol{H}\end{aligned}$$

□

In view of Proposition 1.1, the choice for $\boldsymbol{K}_e^s$ described in (1.64) results particularly appealing as can be further simplified in

$$\boldsymbol{K}_e^s = \frac{1}{2}\mathrm{tr}(\boldsymbol{K}_e^c)\boldsymbol{H} \tag{1.68}$$

Enforcing the stationarity of the mixed discrete stabilized functional (1.62) with respect to $\hat{\varepsilon}$ and $\hat{\sigma}$ one obtains the same equations (1.43) and (1.3), while setting the derivative with respect to $\hat{\boldsymbol{u}}$ to 0 one obtains

$$\partial_{\hat{\boldsymbol{u}}}\Pi_e^h = \boldsymbol{0} \implies \boldsymbol{C}^{\mathrm{T}}\hat{\boldsymbol{\sigma}} + \boldsymbol{K}_e^s\hat{\boldsymbol{u}} = \boldsymbol{F}_e \tag{1.69}$$

which implies, recombining the three algebraic systems

$$(\boldsymbol{K}_e^c + \boldsymbol{K}_e^s)\hat{\boldsymbol{u}} = \boldsymbol{F}_e \tag{1.70}$$

where $\boldsymbol{K}_e = \boldsymbol{K}_e^c + \boldsymbol{K}_e^s$ is the *local stiffness matrix*, having the correct degree of singularity corresponding to the 6 rigid body motions in the three dimensional space.

1.3.4. Assembly of the global system

To obtain the solution of the discrete unknown field describing the approximated problem, an *assembly* procedure is required, gathering the contributions of the single finite elements, here referred to as *local* quantities, and building the *global* algebraic system to be passed to the solver. More precisely, the assembly operator is applied to the local stiffness matrix $\boldsymbol{K}_e$ to produce the global stiffness matrix $\boldsymbol{K}$

$$\boldsymbol{K} = \mathop{\mathsf{A}}_{e=1}^{n_e} \boldsymbol{K}_e \tag{1.71}$$

and to the local equivalent nodal forces vector $\boldsymbol{F}_e$ to produce the global equivalent nodal forces vector

$$\boldsymbol{F} = \mathop{\mathsf{A}}_{e=1}^{n_e} \boldsymbol{F}_e \tag{1.72}$$

A is the standard FE assembly operator, formally composed of a set of n_e $[n_u^e \times n_u]$ logical matrices $\boldsymbol{L}_e$ pre- and post-multiplying the local stiffness matrices as

$$\mathop{\mathsf{A}}_{e}(\cdot) = \boldsymbol{L}_e^{\mathrm{T}}(\cdot)\boldsymbol{L}_e$$

and mapping the vector of equivalent nodal forces as

$$\mathop{\mathsf{A}}_{e}(\cdot) = \boldsymbol{L}_e^{\mathrm{T}}(\cdot)\boldsymbol{L}_e$$

where the $\boldsymbol{L}_e$ non-zero entries correspond to the i^{th} row and j^{th} column if the local degree of freedom i is represented in the global degree of freedom vector at position j. The assembly operator can also take into account suitable transformations where rotations or other mappings between local and global kinematic descriptions are needed. In common practice the assembly operation is realized through incidence matrices, given the natural waste of memory and computational resources implied by the sparse matrices $\boldsymbol{L}_e$.

1.3.5. Enforcement of boundary conditions and solution

The assembled algebraic system in the n_u unknowns $\boldsymbol{U}$ reads

$$\boldsymbol{K}\boldsymbol{U} = \boldsymbol{F} \tag{1.73}$$

which is 6 times singular, as boundary conditions have not been imposed yet, and cannot be solved. To account for the assigned displacements $\overline{\boldsymbol{u}}$ on the constrained part of the boundary $\partial\Omega_u$ the most common approach is the *Guyan reduction* ([16]), also known as *static condensation*. The method reduces the number of degrees of freedom of the system by performing a partition of the vector $\boldsymbol{U}$ in a *free* part $\boldsymbol{U}_f$ and a *constrained* part $\boldsymbol{U}_c$ where displacements are prescribed by the problem. Formally, this coincides with the finite element counterpart of the usual operation of lifting the boundary datum performed for the continuum problem with non-homogeneous Dirichlet conditions. Indeed, lifting is achieved through functions whose support is limited to the only layer of elements of the partition that face the boundary. If the partition of the unknowns $\boldsymbol{U}$ is applied, (1.73) is equivalent to

$$\begin{bmatrix} \boldsymbol{K}_{ff} & \boldsymbol{K}_{fc} \\ \boldsymbol{K}_{cf} & \boldsymbol{K}_{cc} \end{bmatrix} \begin{Bmatrix} \boldsymbol{U}_f \\ \boldsymbol{U}_c \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F}_f \\ \boldsymbol{F}_c \end{Bmatrix} \tag{1.74}$$

where also the matrix $\boldsymbol{K}$ and the vector $\boldsymbol{F}$ are being partitioned accordingly. The sub-matrix $\boldsymbol{K}_{ff}$ is non-singular as rigid body motions are prevented, hence, it can be inverted and the free part of the unknowns $\boldsymbol{U}_f$ can be directly found by *condensing* the known displacements $\boldsymbol{U}_c$

$$\boldsymbol{U}_f = \boldsymbol{K}_{ff}^{-1}(\boldsymbol{F}_f - \boldsymbol{K}_{fc}\boldsymbol{U}_c) \tag{1.75}$$

Assembling back the vector $\boldsymbol{U}$ is straightforward by recomposing the partition and the reaction forces for the degrees of freedom belonging to the constrained boundary can be found by plugging $\boldsymbol{U}_f$ in the system given in the second row of (1.74)

$$\boldsymbol{F}_c = \boldsymbol{K}_{fc}^{\mathrm{T}}\boldsymbol{U}_f + \boldsymbol{K}_{cc}\boldsymbol{U}_c \tag{1.76}$$

where the identity $\boldsymbol{K}_{fc}^{\mathrm{T}} = \boldsymbol{K}_{cf}$ implied by the symmetry of the original matrix has been exploited.

1.3.6. Strains and stresses recovery

Once the unknown discrete field $\boldsymbol{U}$ is found, it is possible to recover the local degrees of freedom $\hat{\boldsymbol{u}}$ and reconstruct the local continuous field $\boldsymbol{u}(\boldsymbol{\xi})$ through the selected model for the displacements $\boldsymbol{N}_{\boldsymbol{u}}(\boldsymbol{\xi})$ (if available) by (1.20). The global field is then piecewise defined by the local fields after switching back to global coordinates $\boldsymbol{x}$. Recovering the local strain field $\boldsymbol{\varepsilon}(\boldsymbol{\xi})$ is similarly achieved exploiting (1.21) and (1.44), so that

$$\boldsymbol{\varepsilon}(\boldsymbol{\xi}) = \boldsymbol{N}_{\varepsilon}(\boldsymbol{\xi})\hat{\boldsymbol{\varepsilon}} = \boldsymbol{N}_{\varepsilon}(\boldsymbol{\xi})\boldsymbol{C}\hat{\boldsymbol{u}} \tag{1.77}$$

Analogously, the local stress field $\boldsymbol{\sigma}(\boldsymbol{\xi})$ is obtained by the condition prescribed by the generalized variables in (1.35), and the model (1.22) and identity in (1.43), so that

$$\boldsymbol{\sigma}(\boldsymbol{\xi}) = \boldsymbol{N}_{\sigma}(\boldsymbol{\xi})\hat{\boldsymbol{\sigma}} = \boldsymbol{N}_{\varepsilon}(\boldsymbol{\xi})\boldsymbol{G}^{-1}\hat{\boldsymbol{\sigma}} = \boldsymbol{N}_{\varepsilon}(\boldsymbol{\xi})\boldsymbol{G}^{-1}\boldsymbol{E}\boldsymbol{C}\hat{\boldsymbol{u}} \tag{1.78}$$

It is a common practice in FE programs to output the global displacement field as generated by the shape functions (if available) mapping the discrete nodal values. For strains and stresses which are often discontinuous across the elements boundaries, their values at a generic point of the computational domain $\boldsymbol{x}$ is obtained as an interpolation of the neighbouring sampling points. These latter are approximated by various averaging techniques as algebraic, volume, or strain energy mean.

1.3.7. Summary of the scheme

A concise list of the steps followed by the above-presented approach for the resolution of the elastostatic problem through the Hu-Washizu mixed finite element approximation is reported below.

- discretization of the domain Ω in $\mathcal{P}_h$
- definition of the strain $\boldsymbol{N}_{\varepsilon}$ and displacement $\boldsymbol{N}_{\boldsymbol{u}}$ models
- computation of matrices $\boldsymbol{A}$, $\boldsymbol{G}$ and the compatibility matrix $\boldsymbol{C}$
- computation of the local stiffness matrix consistent with the strain and displacement models $\boldsymbol{K}_e^c$
- computation of the transformation matrix $\boldsymbol{T}_{D+R}$, hourglass matrix $\boldsymbol{H}$ and local stabilizing stiffness matrix $\boldsymbol{K}_e^s$
- computation of the local equivalent nodal forces vector $\boldsymbol{F}_e$
- assembly of the global system from the local quantities
- enforcement of the Dirichlet boundary conditions

- solution of the global algebraic system

$$\boldsymbol{K}\boldsymbol{U} = \boldsymbol{F}$$

- recover of the strains and stresses parameters and reconstruction of the three continuous fields of the problem

2 | Tools of the virtual element method

The mixed-formulation described in Chapter 1 allows to implement very general element and schemes, amongst which the *virtual element method* (VEM). Before moving to applying it to the problem of linear elastostatics, a mathematical description of the tools employed by the method is necessary. The goal of this chapter is to present the key points of this technique: the polytopic mesh, the virtual element spaces and the projection operators.

2.1. Mesh

The *virtual element mesh* is one of the striking features this technique is mostly known for, as it can be of very general nature, including non-convex elements, aligned edges and faces, geometrically hanging nodes (Figure 2.1), and ridiculously complicated and diverse elements ([24]). In principle, the method is even suitable to support elements with curved edges ([11, 12]), allowing an incredibly accurate description of the geometries of the domain. However, in the subsequent sections only meshes containing straight edges and plane faces will be considered.

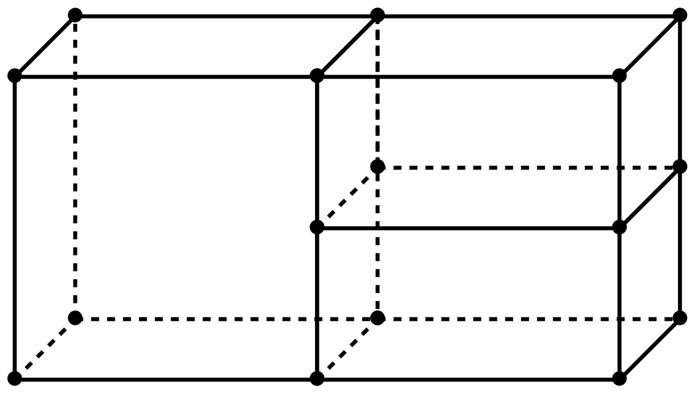


Figure 2.1: Hanging nodes situation for the element on the left.

The body represented in Figure 1.1 by the open bounded set $\Omega \subset \mathbb{R}^3$ can be partitioned in a finite collection $\mathcal{P}_h$ of n_p non-overlapping polyhedra P, such that

$$\Omega = \bigcup_{P \in \mathcal{P}_h} P$$

Each polyhedron P has a centroid $\boldsymbol{x}_P$, a diameter h_P, and a volume $|P|$. The polyhedron is described by a set of $n_{p,f}$ oriented planar polygons representing its facets F, i.e. its *2-dimensional faces* ([21]), from here on simply referred to as faces. The boundary of the polyhedron ∂P then becomes

$$\partial P = \bigcup_{F \in \partial P} F$$

Each face F has a centroid $\boldsymbol{x}_F$, a diameter h_F, and an area $|F|$. The face is defined by a sequence of $n_{p,f,e}$ edges representing its facets E, i.e. its *1-dimensional faces*, from now on simply called edges. The boundary of the face ∂F is

$$\partial F = \bigcup_{\in \partial F} E$$

Each edge E is then defined by the set of the two vertices V_1 and V_2 it is composed of, i.e. its *0-dimensional faces*. From the above construction it is evident how the polytopic virtual element mesh can handle very general situations.

Some remarks have to be specified concerning the description of the spatial entities. Local non-dimensional coordinates as described in (1.19) will be adopted through a simple linear mapping between the local and global reference systems. In VEM, there is no *parent element* as for *isoparametric FE*, where nonlinear maps are needed and are one of the requirements limiting the set of allowable shapes. Scaled monomials will be extensively used throughout the subsequent discussion. In 2D they are defined for polygons as

$$m_{\boldsymbol{\alpha}}(\boldsymbol{x}) := \left(\frac{\boldsymbol{x} - \boldsymbol{x}_F}{h_F} \right)^{\boldsymbol{\alpha}} \tag{2.1}$$

where $\boldsymbol{\alpha} = (\alpha_\xi, \alpha_\eta)$ is a multiindex, and $|\boldsymbol{\alpha}| = \alpha_\xi + \alpha_\eta$ its order. A one-to-one correspondence can be established between the scaled monomials of (2.1) and the indices $\alpha \in \mathbb{N}$, starting from index 1 corresponding to the constant monomial 1, and following each row, from left to right, of *Pascal's triangle* (Figure 2.2). Given an integer k the number of parameters n_k necessary to describe a polynomial in $\mathcal{P}_k(F)$ is given by the number of

$$\begin{array}{lccccccccccccc}
k=0 & & & & & & & 1 & & & & & & \\
k=1 & & & & & & \xi & & \eta & & & & & \\
k=2 & & & & & \xi^2 & & \xi\eta & & \eta^2 & & & & \\
k=3 & & & & \xi^3 & & \xi^2\eta & & \xi\eta^2 & & \eta^3 & & & \\
k=4 & & & \xi^4 & & \xi^3\eta & & \xi^2\eta^2 & & \xi\eta^3 & & \eta^4 & & \\
k=5 & & \xi^5 & & \xi^4\eta & & \xi^3\eta^2 & & \xi^2\eta^3 & & \xi\eta^4 & & \eta^5 & \\
k=6 & \xi^6 & \xi^5\eta & & \xi^4\eta^2 & & \xi^3\eta^3 & & \xi^2\eta^4 & & \xi\eta^5 & & & \eta^6
\end{array}$$

Figure 2.2: Pascal's triangle truncated at $k = 6$.

total elements up to row k of Pascal's triangle, that is

$$n_k = \dim \mathcal{P}_k(F) = \frac{(k+1)(k+2)}{2} \tag{2.2}$$

Therefore, the scaled monomials of degree less or equal to k can be gathered in the $[n_k \times 1]$ vector $\boldsymbol{m}_k$

$$\boldsymbol{m}_k := \begin{Bmatrix} 1 & \xi & \eta & \xi^2 & \dots & \eta^k \end{Bmatrix}^{\mathrm{T}} \tag{2.3}$$

whose elements form a basis for the polynomials $\mathcal{P}_k(F)$ of degree less or equal to k.

In 3D scaled monomials are defined for polyhedra as

$$\mu_{\boldsymbol{\alpha}}(\boldsymbol{x}) := \left(\frac{\boldsymbol{x} - \boldsymbol{x}_P}{h_P}\right)^{\boldsymbol{\alpha}} \tag{2.4}$$

where $\boldsymbol{\alpha} = (\alpha_\xi, \alpha_\eta, \alpha_\zeta)$ is a multiindex, and $|\boldsymbol{\alpha}| = \alpha_\xi + \alpha_\eta + \alpha_\zeta$ its order. Analogously, a one-to-one correspondence can be established between the scaled monomials of (2.4) and the indices $\alpha \in \mathbb{N}$, starting from index 1 corresponding to the constant monomial 1, and following each layer, read counterclockwise starting from the top, of *Pascal's pyramid* in Figure 2.3 and Figure 2.4. Given an integer k, the number of parameters ν_k necessary to describe a polynomial in $\mathcal{P}_k(P)$ is given by the number of total elements up to layer k of Pascal's pyramid, that is

$$\nu_k = \dim \mathcal{P}_k(P) = \frac{(k+1)(k+2)(k+3)}{6} \tag{2.5}$$

Therefore, once again, the scaled monomials of degree less or equal to k can be gathered in the $[\nu_k \times 1]$ vector $\boldsymbol{\mu}_k$

$$\boldsymbol{\mu}_k := \left\{1 \quad \xi \quad \eta \quad \zeta \quad \xi^2 \quad \xi\eta \quad \eta^2 \quad \eta\zeta \quad \zeta^2 \quad \xi\zeta \quad \xi^3 \quad \dots \quad \mu_k\right\}^{\mathrm{T}} \tag{2.6}$$

whose elements form a basis for the polynomials $\mathcal{P}_k(P)$ of degree less or equal to k.

$$\begin{tikzcd}
k=2 & & & \xi^2 \arrow[ddl] & & \\
k=1 & & & \xi \arrow[dddl] & & \\
& & \xi\eta \arrow[dddl] & & \xi\zeta & \\
k=0 & & & 1 & & \\
& & \eta \arrow[rr] & & \zeta & \\
& \eta^2 \arrow[rr] & & \eta\zeta \arrow[rr] & & \zeta^2 \arrow[uuul]
\end{tikzcd}$$

Figure 2.3: First three layers of Pascal's pyramid.

$$\begin{tikzcd}
k=4 & & & & & \xi^4 \arrow[ddl] & & & & \\
k=3 & & & & & \xi^3 \arrow[ddl] & & & & \\
& & & & \xi^3\eta \arrow[ddl] & & \xi^3\zeta \arrow[ddl] & & & \\
& & & & \xi^2\eta \arrow[ddl] & & \xi^2\zeta \arrow[ddl] & & & \\
& & & \xi^2\eta^2 \arrow[ddl] & & \xi^2\eta\zeta \arrow[ddl] & & \xi^2\zeta^2 \arrow[uul] & & \\
& & & \xi\eta^2 \arrow[ddl] & & \xi\eta\zeta & & \xi\zeta^2 \arrow[uul] & & \\
& & \xi\eta^3 \arrow[ddl] & & \xi\eta^2\zeta \arrow[rr] & & \xi\eta\zeta^2 & & \xi\zeta^3 \arrow[uul] & \\
& & \eta^3 \arrow[rr] & & \eta^2\zeta \arrow[rr] & & \eta\zeta^2 \arrow[rr] & & \zeta^3 \arrow[uul] & \\
& \eta^4 \arrow[rr] & & \eta^3\zeta \arrow[rr] & & \eta^2\zeta^2 \arrow[rr] & & \eta\zeta^3 \arrow[rr] & & \zeta^4 \arrow[uul]
\end{tikzcd}$$

Figure 2.4: Layers corresponding to $k = 3$ and $k = 4$ of Pascal's pyramid.

Given a polynomial of degree k embedded in $\mathbb{R}^3$, it is possible to express its restriction on a plane through a polynomial of degree k embedded in $\mathbb{R}^2$. Specifically, given a scaled monomial $\mu_k(\boldsymbol{\xi})$, one can map its value through the scaled barycentric coordinates of the face $\boldsymbol{\xi}_f$ according to the subsequent procedure, following Figure 2.5.

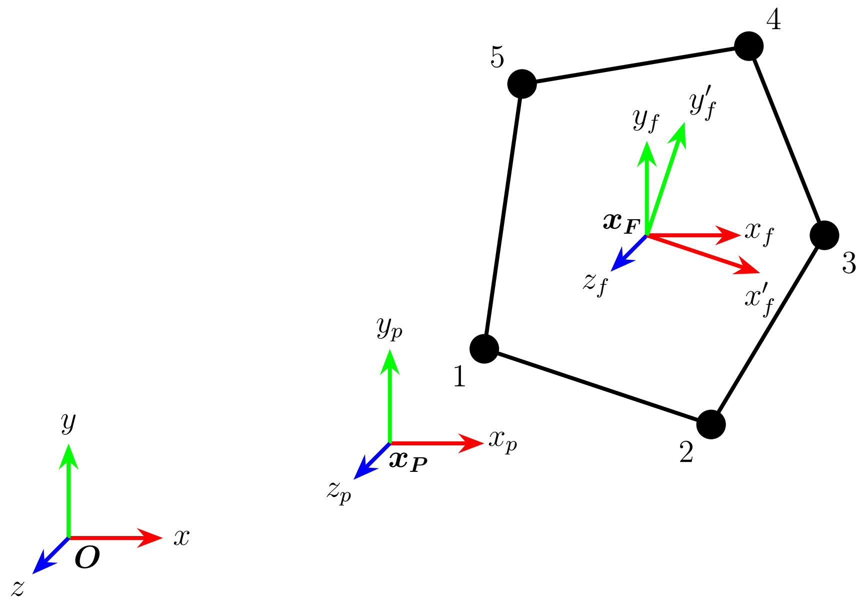


Figure 2.5: Coordinates transformation from 2D face barycentric scaled reference system to 3D scaled polyhedral reference system.

A polyhedron whose centroid lies in $\boldsymbol{x_P}$ has a pentagonal face depicted in Figure 2.5, whose centroid is in turn $\boldsymbol{x_F}$. The reference frames $Oxyz$, $x_P x_p y_p z_p$ and $x_F x_f y_f z_f$ are obtained with a translation only, since no rotation nor scaling is applied. A rotation is applied from reference system $x_F x_f y_f z_f$ to $x_F x'_f y'_f z'_f$ so that the latter's z'_f axis is orthogonal to the polygon, x'_f axis is parallel to the first edge, following the nodes numbering, and the y'_f axis is obtained with the right-hand rule. A scaling by h_P is further performed in $x_P x_p y_p z_p$ system, so to retrieve the scaled coordinates system $x_P \xi_p \eta_p \zeta_p$ (simply indicated with $x_P \xi\eta\zeta$) and by h_F in $x_F x'_f y'_f z'_f$ so to obtain the scaled coordinates $x_F \xi_f \eta_f \zeta_f$. Hence, the following relations hold

$$\boldsymbol{x}_p = \boldsymbol{x} - \boldsymbol{x_P} \qquad \boldsymbol{x}_f = \boldsymbol{x} - \boldsymbol{x_F} \tag{2.7}$$

$$\boldsymbol{\xi} = \frac{\boldsymbol{x}_p}{h_P} \qquad \boldsymbol{\xi}_f = \frac{\boldsymbol{x}'_f}{h_F} \tag{2.8}$$

$$\boldsymbol{x}'_f = \boldsymbol{R}\boldsymbol{x}_f = \begin{bmatrix} (\boldsymbol{e}_{x'_f} \cdot \boldsymbol{e}_{x_f}) & (\boldsymbol{e}_{x'_f} \cdot \boldsymbol{e}_{y_f}) & (\boldsymbol{e}_{x'_f} \cdot \boldsymbol{e}_{z_f}) \\ (\boldsymbol{e}_{y'_f} \cdot \boldsymbol{e}_{x_f}) & (\boldsymbol{e}_{y'_f} \cdot \boldsymbol{e}_{y_f}) & (\boldsymbol{e}_{y'_f} \cdot \boldsymbol{e}_{z_f}) \\ (\boldsymbol{e}_{z'_f} \cdot \boldsymbol{e}_{x_f}) & (\boldsymbol{e}_{z'_f} \cdot \boldsymbol{e}_{y_f}) & (\boldsymbol{e}_{z'_f} \cdot \boldsymbol{e}_{z_f}) \end{bmatrix} \boldsymbol{x}_f \tag{2.9}$$

where the orthogonal rotation matrix $\boldsymbol{R}$ in (2.9) contains the cosines directors of the coordinates axes. Exploiting the above relations and the orthogonality of $\boldsymbol{R}$, one has

$$\boldsymbol{\xi} = \frac{\boldsymbol{x} - \boldsymbol{x_P}}{h_P} = \frac{\boldsymbol{x}_f + \boldsymbol{x_F} - \boldsymbol{x_P}}{h_P} = \frac{R^{-1}\boldsymbol{x}'_f + \boldsymbol{x_F} - \boldsymbol{x_P}}{h_P} = \frac{h_F \boldsymbol{R}^{\mathrm{T}} \boldsymbol{\xi}_f + \boldsymbol{x_F} - \boldsymbol{x_P}}{h_P} \tag{2.10}$$

2.2. Local virtual element space in $\mathbb{R}^2$

In the following, focus is put on a single element P of the polytopic mesh $\mathcal{P}_h$ defined in 2.1, and will be referred to as the *virtual element*. As in FE, the approximating functions are sought in a suitable space, namely the *virtual element space*. One of the major complexities of dealing with VE embedded in $\mathbb{R}^3$ resides in the natural structure of the polyhedron, whose boundary is composed of polygons where a proper virtual element space embedded in $\mathbb{R}^2$ should be given.

Therefore, in order to properly tackle the 3D problem, it is necessary to first understand the 2D setting and:

- define the local virtual element space in $\mathbb{R}^2$
- find a uniquely-defying way of representing its elements (which will be the *degrees of freedom*)

Definition 2.1. *Local virtual element space embedded in $\mathbb{R}^2$. The local virtual element space $V_k(F)$ of order k, $k \in \mathbb{N}, k \geq 1$, for a polygon F is defined by functions v such that*

$$\begin{cases} \bullet\ v \text{ is a polynomial of degree } k \text{ on each edge } E \text{ of the polygon } F, \text{ i.e. } v|_E \in \mathcal{P}_k(E) \\ \bullet\ v \text{ is globally continuous on } \partial F, \text{ i.e. } v|_{\partial F} \in C^0(\partial F) \\ \bullet\ \Delta v \text{ is a polynomial of degree } k-2 \text{ in } F, \text{ i.e. } \Delta v \in \mathcal{P}_{k-2}(F) \end{cases}$$

In simple terms, the above definition includes in $V_k(F)$ all polynomials of order k (as usually required for standard FE) plus some additional functions whose restriction on an edge is still a polynomial of order k. These *additional functions* are one of the key points of VEM, as their explicit expression is unknown and never required to be computed, remaining *virtual* throughout the whole process, so that they lent the name to the method itself. The third condition is enforced to fix the dimension of the space, as will be clarified later. Since a polygon of order k satisfies all the three requirements of Definition 2.1, the following inclusion holds

$$\mathcal{P}_k(F) \subset V_k(F) \tag{2.11}$$

which is essential for convergence properties. As already anticipated, the next step is to uniquely identify an element of $V_k(F)$ through cleverly chosen parameters, the *local degrees of freedom* (DOFs) *for the 2D virtual element space* ([6]).

Proposition 2.1. *Local degrees of freedom for the virtual element space embedded in* $\mathbb{R}^2$*.*

An element v of the space $V_k(F)$ defined in Definition 2.1 *is uniquely identified by the DOFs*

$$\Xi : V_k(F) \to \mathbb{R}$$

grouped in the three following sets

- *the value of v at the vertices of F*
- *for each edge E of F, the value of v at the $k-1$ internal points of the $(k+1)$-point Gauss-Lobatto quadrature rule on E*
- *the n_{k-2} scaled moments up to order $k-2$ of v in F:*

$$\frac{1}{|F|}\int_F v m_\alpha, \qquad \alpha = 1, ..., n_{k-2} \tag{2.12}$$

where m_α are the scaled monomials defined in (2.1) *and n_{k-2} in* (2.2)

Proof. To prove the unisolvence of the chosen degrees of freedom for the space $V_k(F)$, we split the argument in two steps. First we prove that if the value of the function v on the boundary ∂F and the polynomial of degree $k-2$ Δv are known, then the element v is uniquely identified. Let us consider the following problem

$$\begin{cases} \Delta v = \displaystyle\sum_{\alpha=1}^{n_{k-2}} m_\alpha = p_{k-2} & \text{in } F \\ v = g & \text{on } \partial F \end{cases} \tag{2.13}$$

where the polynomial of degree $k-2$ p_{k-2} is composed of the scaled monomials m_α and g is the boundary datum, i.e., the value of v on the edges of the polygon. We claim that given suitable g and p_{k-2}, there exists a unique v. The corresponding weak form then reads

$$\begin{aligned} &\text{Given } p_{k-2} \in L^2(F),\ g \in H^{\frac{1}{2}}(\partial F), \text{ and } v_0 \in H^1(F) \text{ such that } v_0 = g \text{ on } \partial F, \\ &\text{find } v \in H^1(F) \text{ such that } v - v_0 \in H^1_0 \text{ and} \\ &\qquad \int_F \nabla v \nabla \phi \, d\Sigma = -\int_F p_{k-2}\phi \, d\Sigma \qquad \forall \phi \in H^1_0(F) \end{aligned} \tag{2.14}$$

Setting $w = v - v_0$ so that $w \in H^1_0(F)$, problem 2.14 becomes

Given $p_{k-2} \in L^2(F)$, $g \in H^{\frac{1}{2}}(\partial F)$, and $v_0 \in H^1(F)$ such that $v_0 = g$ on ∂F, find $w \in H^1_0(F)$ such that (2.15)

$$\int_F \nabla w \nabla \phi \, d\Sigma = -\int_F p_{k-2} \phi \, d\Sigma - \int_F \nabla v_0 \nabla \phi \, d\Sigma \qquad \forall \phi \in H^1_0(F)$$

Now, the following hypotheses for problem 2.15 hold:

- Continuity of the bilinear form $\int_F \nabla w \nabla \phi \, d\Sigma$

$$\int_F \nabla w \nabla \phi \, d\Sigma \leq \|\nabla w\|_{L^2(F)} \|\nabla \phi\|_{L^2(F)} \leq \|w\|_{H^1(F)} \|\phi\|_{H^1(F)} \tag{2.16}$$

  where Cauchy-Schwartz inequality and the definition of the H^1-norm have been applied;

- Coercivity of the bilinear form $\int_F \nabla w \nabla \phi \, d\Sigma$

$$\int_F \nabla w \nabla \phi \, d\Sigma = \|\nabla w\|^2_{L^2(F)} \geq \frac{1}{1 + C_F^2} \|w\|^2_{H^1(F)} = \alpha \|w\|^2_{H^1(F)} \tag{2.17}$$

  where H^1-norm definition and the below Poincaré inequality have been exploited, with C_F being a constant depending on the domain F

$$\|w\|_{L^2(F)} \leq C_F \|\nabla w\|_{L^2(F)} \tag{2.18}$$

- Continuity of the linear functional $-\int_F p_{k-2} \phi \, d\Sigma - \int_F \nabla v_0 \nabla \phi \, d\Sigma$

$$\begin{aligned} -\int_F p_{k-2} \phi \, d\Sigma - \int_F \nabla v_0 \nabla \phi \, d\Sigma &\leq \|p_{k-2}\|_{L^2(F)} \|\phi\|_{L^2(F)} + \|\nabla v_0\|_{L^2(F)} \|\nabla \phi\|_{L^2(F)} \leq \\ &\leq \left(\|p_{k-2}\|_{L^2(F)} + \|\nabla v_0\|_{L^2(F)} \right) \|\phi\|_{H^1(F)} \leq \\ &\leq C \|\phi\|_{H^1(F)} \end{aligned}$$

  where Cauchy-Schwartz, H^1-norm definition and the embedding $\mathcal{P}_{k-2}(F) \subset L^2(F)$ have been exploited;

In view of the above properties, Lax-Milgram lemma guarantees existence and uniqueness of a function $w \in H^1_0(F)$. However, the latter still depends on the choice of v_0, hence only existence for $v \in H^1(F)$ is guaranteed. To prove uniqueness, we argue by contradiction assuming there exist two solutions v_1 and v_2 to Problem 2.14, both coinciding with g on the boundary. Taking the difference of the resulting weak forms, the following is obtained

$$\int_F \nabla (v_1 - v_2) \nabla \phi \, d\Sigma = 0 \qquad \forall \phi \in H_0^1 \tag{2.19}$$

and testing with the particular choice of $(v_1 - v_2)$ for ϕ leads to

$$0 = \int_F \nabla (v_1 - v_2) \nabla (v_1 - v_2) \, d\Sigma = \| \nabla (v_1 - v_2) \|^2_{L^2(F)} \geq \| v_1 - v_2 \|^2_{L^2(F)} \geq 0 \tag{2.20}$$

where Poincaré inequality has been applied. Equation (2.20) implies that $v_1 = v_2$, and uniqueness for Problem 2.14 is also proved.

The second step of the proof consists in checking that the selected degrees of freedom of Proposition 2.1 uniquely identify the data of Problem 2.13. To this aim, we note that the first two sets of DOFs, corresponding to the value of v at the vertices V of the polygon F and at the $k-1$ internal points of each edge E, uniquely define the boundary datum g. In fact, for each edge, the polynomial of degree k prescribed by Definition 2.1 is described exactly by the $k+1$ points, vertices included, belonging to the edge. It only remains to check that if the internal DOFs are 0, together with the boundary datum, then the function v is identically null. Substituting p_{k-2} with the sum of the scaled monomials m_α in Problem 2.14, one obtains

$$\begin{aligned} \int_F \nabla v \nabla \phi \, d\Sigma &= - \int_F \left(\sum_{\alpha=1}^{n_{k-2}} m_\alpha \right) \phi \, d\Sigma = \\ &= - \sum_{\alpha=1}^{n_{k-2}} \left(\int_F m_\alpha \phi \, d\Sigma \right) \qquad \forall \phi \in H_0^1(F) \end{aligned} \tag{2.21}$$

Being the data $g = 0$, we can take $v \in H_0^1(F)$, and testing with the particular v in place of ϕ, the following holds

$$0 \leq \| v \|^2_{L^2(F)} \leq C_F^2 \| \nabla v \|^2_{L^2(F)} = - \sum_{\alpha=1}^{n_{k-2}} \left(\int_F m_\alpha \phi \, d\Sigma \right) \tag{2.22}$$

where Poincaré inequality has once again been exploited, justified by $v \in H_0^1(F)$. The quantities in brackets are exactly the internal moments: if these are zero, then the L^2-norm of v is bounded from below and above by 0, and therefore v is identically null, completing the proof. $\square$

As a consequence of the above proposition, the dimension of the space $V_k(F)$ coincides with the number of corresponding degrees of freedom N_{DOF}

$$\dim V_k(F) = N_{DOF} = N_V + N_V(k-1) + n_{k-2} = kN_V + \frac{(k-1)k}{2} \tag{2.23}$$

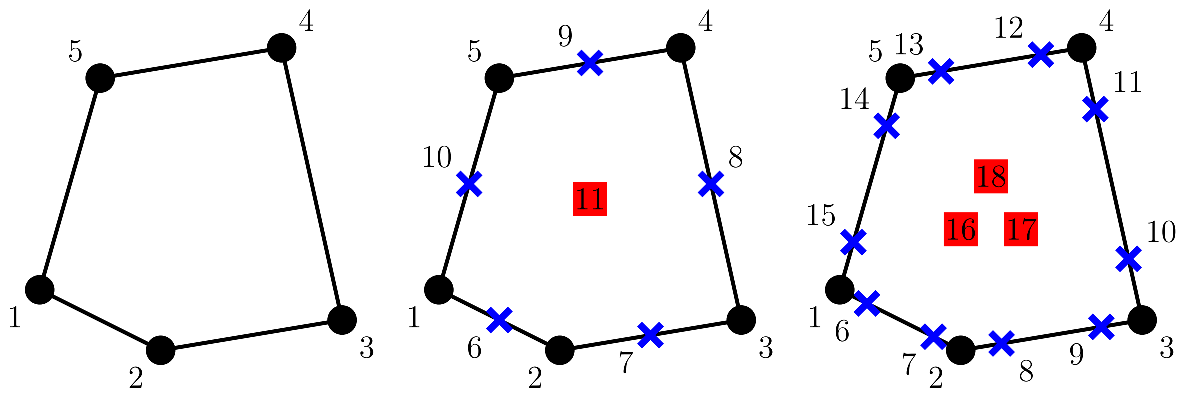
where N_V is the number of vertices (equal to the number of edges) belonging to the polygon F. The space $V_k(F)$ can be decomposed into its N_{DOF} *basis functions* (or *shape functions*), here denoted with φ

$$\Xi_i(\varphi_j) = \delta_{ij} \quad \forall i, j = 1, ..., N_{DOF} \tag{2.24}$$

so that any element of v can be expressed through the Lagrangian interpolation

$$v = \sum_{i=1}^{N_{DOF}} \Xi_i(v)\varphi_i \qquad \forall v \in V_k(F) \tag{2.25}$$

A visual representation of the DOFs described in Proposition 2.1 is depicted in Figure 2.6 for the first three virtual element spaces corresponding to $k = 1$, $k = 2$ and $k = 3$. It is important to remark that the choice for the DOFs made in Proposition 2.1 is not the only possible one in order to achieve unique identification of an element v of the space $V_k(F)$. One could take, for instance, $k-1$ non-coinciding randomly-picked internal points on each edge E and still achieve the same results. However, as will be clarified later, the sets of local DOFs have been specifically selected to ease computations and perform the steps with the minimum amount of information required to still guarantee convergence.



(a) $V_1(F)$ (b) $V_2(F)$ (c) $V_3(F)$

Figure 2.6: Local degrees of freedom for the three virtual element spaces $V_1(F)$, $V_2(F)$ and $V_3(F)$ in a pentagon F. The black dots correspond to vertices DOFs, blue crosses to edge DOFs, and red squares to internal DOFs. Note that while the first two sets match with the function evaluation at the precise location shown in the figure, the latter do not have a geometric punctual representation and are displayed inside the polygon for the sake of simplicity.

2.3. Projection operator

Given the unknown structure of the functions v of the space $V_k(F)$ described in 2.2, whose explicit form is in general known only after solving a boundary value problem, a projection operator has been introduced in the VEM literature. In this specific work, the projector will be explicitly used for the faces only, since for projecting operation applied for functions defined in polyhedra will already be incorporated in the definition of the strain field.

Definition 2.2. *Projection operator* $\Pi^{\nabla}_{F,k}$. *The projection operator* $\Pi^{\nabla}_{F,k}$ *maps an element of* $v \in V_k(F)$ *into a polynomial of order* k

$$\Pi^{\nabla}_{F,k} : V_k(F) \to \mathcal{P}_k(F)$$

and it is defined by the orthogonality condition

$$\int_F \nabla p_k \cdot \nabla \left(\Pi^{\nabla}_{F,k} v - v\right) \, d\Sigma = 0 \qquad \forall p_k \in \mathcal{P}_k(F) \tag{2.26}$$

and the conditions to fix the constants

$$\begin{cases} \dfrac{1}{N_V}\displaystyle\sum_{j=1}^{N_V}\left\{\left[\Pi^{\nabla}_{F,k}v\right](\boldsymbol{\xi}_{V_j}) - v(\boldsymbol{\xi}_{V_j})\right\} = 0 & \text{if } k = 1 \\ \dfrac{1}{|F|}\displaystyle\int_F \left(\Pi^{\nabla}_{F,k}v - v\right)\, d\Sigma = 0 & \text{if } k \geq 2 \end{cases} \tag{2.27}$$

where $\boldsymbol{\xi}_{V_j}$ are the coordinates of the vertex V_j.

To extract the polynomial projection of a virtual function, the following procedure is applied. We focus the attention on a virtual basis function φ_i instead of a virtual function v, in view of (2.24) and (2.25). From Definition 2.2, since $\varphi_i \in V_k(F)$ and $m_\alpha \in \mathcal{P}_k(F)$ we have

$$\int_F \nabla m_\alpha \cdot \nabla\left(\Pi^{\nabla}_{F,k}\varphi_i - \varphi_i\right)\, d\Sigma = 0 \qquad \forall\alpha \leq n_k, \quad \forall i = 1, ..., \text{dim}V_k(F) \tag{2.28}$$

Since $\Pi^{\nabla}_{F,k}\varphi_i \subset \mathcal{P}_k(F)$, we have

$$\Pi^{\nabla}_{F,k}\varphi_i = \sum_{\beta=1}^{n_k} s_i^\beta m_\beta \qquad \forall i = 1, ..., \text{dim}V_k(F)$$

which plugged into (2.28), after splitting the integral, leads to

$$\sum_{\beta=1}^{n_k} s_i^\beta \int_F \nabla m_\alpha \cdot \nabla m_\beta \, d\Sigma = \int_F \nabla m_\alpha \cdot \nabla\varphi_i \, d\Sigma \qquad \forall\alpha \leq n_k, \quad \forall i = 1, ..., \text{dim}V_k(F) \tag{2.29}$$

The right hand side of (2.29) can be integrated by parts, yielding to

$$\begin{aligned}\int_F \nabla m_\alpha \cdot \nabla\varphi_i \, d\Sigma &= \int_{\partial F} \nabla m_\alpha \cdot \boldsymbol{n}\varphi_i \, d\Gamma - \int_F \Delta m_\alpha \varphi_i \, d\Sigma = \\ &= \sum_{E\in\partial F}\left[\int_E \nabla m_\alpha \cdot \boldsymbol{n}\varphi_i \, d\Gamma\right] - \int_F \Delta m_\alpha\varphi_i \, d\Sigma \qquad \forall\alpha \leq n_k, \\ &\qquad \forall i = 1, ..., \text{dim}V_k(F)\end{aligned}$$

where $\boldsymbol{n}$ is the outward unit vector of the boundary ∂F of the polygon F. The n_k linear equations in the n_k unknowns of (2.29) can be gathered in an algebraic system. However, the first equation coming from $\alpha = 1$ is the trivial identity $0 \equiv 0$ and is therefore replaced with the equation corresponding to the correct order in (2.27), which read

$$\begin{cases} \displaystyle\sum_{\beta=1}^{n_k} s_i^\beta \left[\frac{1}{N_V} \sum_{j=1}^{N_V} m_\beta(\boldsymbol{\xi}_{V_j}) \right] = \frac{1}{N_V} \sum_{j=1}^{N_V} \varphi_i(\boldsymbol{\xi}_{V_j}) & \text{if } k = 1 \\ \displaystyle\sum_{\beta=1}^{n_k} s_i^\beta \left[\frac{1}{|F|} \int_F m_\beta \, d\Sigma \right] = \frac{1}{|F|} \int_F \varphi_i \, d\Sigma & \text{if } k \geq 2 \end{cases} \tag{2.30}$$

The linear algebraic system in the n_k unknowns s_i^β then reads

$$\sum_{\beta=1}^{n_k} [G_F]_{\alpha\beta} \, s_i^\beta = b_{F,i}^\alpha \qquad \forall \alpha = 1, \ldots, n_k \tag{2.31}$$

where the α^{th} row and β^{th} column entries of $[n_k \times n_k]$ matrix $\boldsymbol{G}_F$ are

$$\begin{cases} \displaystyle\frac{1}{N_V} \sum_{j=1}^{N_V} m_\beta(\boldsymbol{\xi}_{V_j}) & \text{if } \alpha = 1 \text{ and } k = 1 \\ \displaystyle\frac{1}{|F|} \int_F m_\beta \, d\Sigma & \text{if } \alpha = 1 \text{ and } k \geq 2 \\ \displaystyle\int_F \nabla m_\alpha \cdot \nabla m_\beta \, d\Sigma & \text{if } \alpha > 1 \end{cases} \tag{2.32}$$

and the α^{th} component of $[n_k \times 1]$ vector $\boldsymbol{b}_{F,i}$ is

$$\begin{cases} \displaystyle\frac{1}{N_V} \sum_{j=1}^{N_V} \varphi_i(\boldsymbol{\xi}_{V_j}) & \text{if } \alpha = 1 \text{ and } k = 1 \\ \displaystyle\frac{1}{|F|} \int_F \varphi_i \, d\Sigma & \text{if } \alpha = 1 \text{ and } k \geq 2 \\ \displaystyle\sum_{E \in \partial F} \left[\int_E \nabla m_\alpha \cdot \boldsymbol{n} \varphi_i \, d\Gamma \right] - \int_F \Delta m_\alpha \varphi_i \, d\Sigma & \text{if } \alpha > 1 \end{cases} \tag{2.33}$$

The quantities of matrix $\boldsymbol{G}_F$ in (2.32) are all computable as they involve integrations of polynomials over F or simple function evaluations. The quantities of the vector $\boldsymbol{b}_{F,i}$ are also computable as they involve, from top to bottom, evaluations of a basis function in vertices DOFs, evaluation of a basis function in a face DOF, integral of a polynomial times the basis function on the edge and evaluation of a basis function in a face DOF. To fix the ideas, the procedure applied to produce the algebraic systems to be solved to compute the coefficients of the projections $\Pi^\nabla_{F,1}\varphi_i$ and $\Pi^\nabla_{F,2}\varphi_i$ are explicitly reported below.

- $k = 1$

The $n_1 = 3$ scaled monomials, their gradients and laplacians are

$$\begin{array}{ccc} m_1 = 1 & m_2 = \xi & m_3 = \eta \\ \nabla m_1 = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} & \nabla m_2 = \begin{Bmatrix} 1 \\ 0 \end{Bmatrix} & \nabla m_3 = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \\ \Delta m_1 = 0 & \Delta m_2 = 0 & \Delta m_3 = 0 \end{array}$$

Lagragian-type interpolation implies

$$\varphi_i(\boldsymbol{\xi}_{V_j}) = \delta_{ij} \implies \sum_{j=1}^{N_V} \varphi_i(\boldsymbol{\xi}_{V_j}) = 1 \qquad \forall i = 1, \dots, \mathrm{dim} V_1(F) \equiv N_V$$

Since the shape function is a polynomial of degree 1 on the edges, we can apply 2-point Gauss-Lobatto quadrature with the two vertices of the edge

$$\begin{cases} \varphi_i(\boldsymbol{\xi}_{V_j}) = \delta_{ij} \\ \varphi_i|_E \in \mathcal{P}_1(E) \end{cases} \implies \begin{cases} \displaystyle\int_E \nabla m_2 \cdot \boldsymbol{n} \varphi_i \, d\Gamma = \begin{cases} \frac{1}{2}|E| n_{E,\xi} & \text{if } V_i \in E \\ 0 & \text{if } V_i \notin E \end{cases} \\ \displaystyle\int_E \nabla m_3 \cdot \boldsymbol{n} \varphi_i \, d\Gamma = \begin{cases} \frac{1}{2}|E| n_{E,\xi} & \text{if } V_i \in E \\ 0 & \text{if } V_i \notin E \end{cases} \end{cases}$$

where $\boldsymbol{n}_E = \{n_{E,\xi} \quad n_{E,\eta}\}^{\mathrm{T}}$ is the outward normal vector of ∂F in correspondence of edge E. Indicating the two edges shared by vertex i with E_+ and E_-, the final system solving for the polynomial coefficients $\boldsymbol{s}_i$ of $\Pi^{\nabla}_{F,1}\varphi_i$ is obtained

$$\begin{bmatrix} 1 & \frac{1}{N_V}\sum_{j=1}^{N_V} \xi_{V_j} & \frac{1}{N_V}\sum_{j=1}^{N_V} \eta_{V_j} \\ 0 & \int_F 1 & 0 \\ 0 & 0 & \int_F 1 \end{bmatrix} \begin{Bmatrix} s_i^1 \\ s_i^2 \\ s_i^3 \end{Bmatrix} = \begin{Bmatrix} \frac{1}{N_V} \\ \frac{1}{2}(|E_+| n_{E_+,\xi} + |E_-| n_{E_-,\xi}) \\ \frac{1}{2}(|E_+| n_{E_+,\eta} + |E_-| n_{E_-,\eta}) \end{Bmatrix} \tag{2.34}$$

- k=2

  The $n_2 = 6$ scaled monomials, their gradients and laplacians are

$$\begin{array}{cccccc} m_1 = 1 & m_2 = \xi & m_3 = \eta & m_4 = \xi^2 & m_5 = \xi\eta & m_6 = \eta^2 \\ \nabla m_1 = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} & \nabla m_2 = \begin{Bmatrix} 1 \\ 0 \end{Bmatrix} & \nabla m_3 = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} & \nabla m_4 = \begin{Bmatrix} 2\xi \\ 0 \end{Bmatrix} & \nabla m_5 = \begin{Bmatrix} \eta \\ \xi \end{Bmatrix} & \nabla m_6 = \begin{Bmatrix} 0 \\ 2\eta \end{Bmatrix} \\ \Delta m_1 = 0 & \Delta m_2 = 0 & \Delta m_3 = 0 & \Delta m_4 = 2 & \Delta m_5 = 0 & \Delta m_6 = 2 \end{array}$$

Lagrangian-type interpolation implies

$$\frac{1}{|F|}\int_F \varphi_i \, d\Sigma = \delta_{i,\dim V_2(F)} = \delta_{i,2N_V+1}$$

as there is only one face moment DOF. Since the shape function is a polynomial of degree 2 on the edges and the gradients of the scaled monomials are at most of degree 1, we can apply 3-point Gauss-Lobatto quadrature with the two vertices of the edge and its midpoint

$$\begin{cases} \varphi_i(\boldsymbol{\xi}_{V_j}) = \delta_{ij} \\ \varphi_i|_E \in \mathcal{P}_2(E) \\ m_\alpha|_E \in \mathcal{P}_1(E) \end{cases} \implies \begin{cases} \displaystyle\int_E \nabla m_2 \cdot \boldsymbol{n}\varphi_i \, d\Gamma = \begin{cases} \frac{1}{6}|E| n_{E,\xi} & \text{if } V_i \in E \\ \frac{2}{3}|E| n_{E,\xi} & \text{if } E_i \in E \\ 0 & \text{otherwise} \end{cases} \\ \displaystyle\int_E \nabla m_3 \cdot \boldsymbol{n}\varphi_i \, d\Gamma = \begin{cases} \frac{1}{6}|E| n_{E,\eta} & \text{if } V_i \in E \\ \frac{2}{3}|E| n_{E,\eta} & \text{if } E_i \in E \\ 0 & \text{otherwise} \end{cases} \\ \displaystyle\int_E \nabla m_4 \cdot \boldsymbol{n}\varphi_i \, d\Gamma = \begin{cases} \frac{1}{6}|E| 2\xi_{V_i} n_{E,\xi} & \text{if } V_i \in E \\ \frac{2}{3}|E| 2\xi_{E_i} n_{E,\xi} & \text{if } E_i \in E \\ 0 & \text{otherwise} \end{cases} \\ \displaystyle\int_E \nabla m_5 \cdot \boldsymbol{n}\varphi_i \, d\Gamma = \begin{cases} \frac{1}{6}|E| \left(\eta_{V_i} n_{E,\xi} + \xi_{V_i} n_{E,\eta}\right) & \text{if } V_i \in E \\ \frac{2}{3}|E| \left(\eta_{E_i} n_{E,\xi} + \xi_{E_i} n_{E,\eta}\right) & \text{if } E_i \in E \\ 0 & \text{otherwise} \end{cases} \\ \displaystyle\int_E \nabla m_6 \cdot \boldsymbol{n}\varphi_i \, d\Gamma = \begin{cases} \frac{1}{6}|E| 2\eta_{V_i} n_{E,\eta} & \text{if } V_i \in E \\ \frac{2}{3}|E| 2\xi_{V_i} n_{E,\eta} & \text{if } E_i \in E \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

where $\boldsymbol{n}_E = \{n_{E,\xi} \quad n_{E,\eta}\}^{\mathrm{T}}$ is the outward normal vector of ∂F in correspondence of edge E, V_i is the vertex where the vertex shape function φ_i assumes value 1 and E_i is the midpoint where the edge shape function φ_i assumes value 1. Indicating as before the two edges shared by vertex i with E_+ and E_-, the final system solving for the polynomial coefficients $\boldsymbol{s}_i$ of $\Pi^\nabla_{F,2}\varphi_i$ is obtained (the differential symbol $d\Sigma$

is omitted)

$$
\begin{bmatrix}
1 & \frac{1}{|F|}\int_F \xi & \frac{1}{|F|}\int_F \eta & \frac{1}{|F|}\int_F \xi^2 & \frac{1}{|F|}\int_F \xi\eta & \frac{1}{|F|}\int_F \eta^2 \\
0 & \int_F 1 & 0 & \int_F 2\xi & \int_F \eta & 0 \\
0 & 0 & \int_F 1 & 0 & \int_F \xi & \int_F 2\eta \\
0 & \int_F 2\xi & 0 & \int_F 4\xi^2 & \int_F 2\xi\eta & 0 \\
0 & \int_F \eta & \int_F \xi & \int_F 2\xi\eta & \int_F \xi^2+\eta^2 & \int_F 2\xi\eta \\
0 & 0 & \int_F 2\eta & 0 & \int_F 2\xi\eta & \int_F 4\eta^2
\end{bmatrix}
\begin{Bmatrix} s_i^1 \\ s_i^2 \\ s_i^3 \\ s_i^4 \\ s_i^5 \\ s_i^6 \end{Bmatrix} =
$$

$$
= \begin{cases}
\begin{Bmatrix}
0 \\
\frac{1}{6}(|E_+|n_{E_+,\xi} + |E_-|n_{E_-,\xi}) \\
\frac{1}{6}(|E_+|n_{E_+,\eta} + |E_-|n_{E_-,\eta}) \\
\frac{1}{6}2\xi_{V_i}(|E_+|n_{E_+,\xi} + |E_-|n_{E_-,\xi}) \\
\frac{1}{6}\begin{bmatrix} \eta_{V_i}\left(|E_+|n_{E_+,\xi} + |E_-|n_{E_-,\xi}\right) + \\ +\xi_{V_i}\left(|E_+|n_{E_+,\eta} + |E_-|n_{E_-,\eta}\right) \end{bmatrix} \\
\frac{1}{6}2\eta_{V_i}(|E_+|n_{E_+,\eta} + |E_-|n_{E_-,\eta})
\end{Bmatrix} & \text{if } i \leq N_V \\
\begin{Bmatrix}
0 \\
\frac{2}{3}|E|n_{E,\xi} \\
\frac{2}{3}|E|n_{E,\eta} \\
\frac{2}{3}2\xi_{E_i}|E|n_{E,\xi} \\
\frac{2}{3}\left(\eta_{E_i}|E|n_{E,\xi} + \xi_{E_i}|E|n_{E,\eta}\right) \\
\frac{2}{3}2\eta_{E_i}|E|n_{E,\eta}
\end{Bmatrix} & \text{if } N_V < i \leq 2N_V \\
\begin{Bmatrix}
1 \\ 0 \\ 0 \\ -2|F| \\ 0 \\ -2|F|
\end{Bmatrix} & \text{if } i = 2N_V + 1
\end{cases}
\tag{2.35}
$$

The above right hand side entries depend on whether the shape function φ_i corresponds to a vertex DOF, edge DOF, or internal moment face DOF.

2.4. Enhanced local virtual element space in $\mathbb{R}^2$

In this section the enhanced local virtual element space $W_k(F)$ embedded in $\mathbb{R}^2$ is presented. The arguments on the existence of this space and the proofs of its properties are shown in [1]. The enhanced local virtual element space $W_k(F)$ is specifically built from the local space $V_k(F)$ defined in Definition 2.1, so that

- $w \in W_k(F)$ is still a polynomial of degree k on each edge E of the polygon F
- $\mathcal{P}_k(F) \subset W_k(F)$
- the DOFs described in Proposition 2.1 for $V_k(F)$ can still be used for the space $W_k(F)$

and suitably modified to enjoy the property of the following

Theorem 2.1. *Enhanced virtual element space property. Given $w \in W_k(F)$, where $W_k(F)$ is the enhanced local virtual element space embedded in $\mathbb{R}^2$, the following holds*

$$\int_F w m_{\boldsymbol{\alpha}} \, d\Sigma = \int_F \Pi^{\nabla}_{F,k} w m_{\boldsymbol{\alpha}} \, d\Sigma \qquad |\boldsymbol{\alpha}| = k-1, k \tag{2.36}$$

As will be clear in the following chapter, the property given in 2.1 is essential to implement the virtual element method in 3D, to deal with the equivalent nodal forces vector for $k \geq 3$ and to tackle dynamic problems where mass matrices appear. A precise definition of the space $W_k(F)$ is reported here for completeness.

Definition 2.3. *Local enhanced virtual element space embedded in $\mathbb{R}^2$. The local virtual element space $W_k(F)$ of order k, $k \in \mathbb{N}, k \geq 1$, for a polygon F is defined by functions w such that*

$$\begin{cases} \bullet \ w \text{ is a polynomial of degree } k \text{ on each edge } E \text{ of the polygon } F\text{, i.e. } w|_E \in \mathcal{P}_k(E) \\ \bullet \ w \text{ is globally continuous on } \partial F\text{, i.e. } w|_{\partial F} \in C^0(\partial F) \\ \bullet \ \Delta w \text{ is a polynomial of degree } k \text{ in } F\text{, i.e. } \Delta w \in \mathcal{P}_k(F) \\ \bullet \ \text{the enhanced property holds} \\ \qquad \displaystyle\int_F w m_{\boldsymbol{\alpha}} \, d\Sigma = \int_F \Pi^{\nabla}_{F,k} w m_{\boldsymbol{\alpha}} \, d\Sigma \qquad |\boldsymbol{\alpha}| = k-1, k \end{cases}$$

2.5. Local virtual element space in $\mathbb{R}^3$

Having in mind the local virtual element space in 2D, the projector operator, and the enhanced space, it is now possible to extend the setting to three dimensions.

Definition 2.4. *Local virtual element space embedded in $\mathbb{R}^3$. The local virtual element space $V_k(P)$ of order k, $k \in \mathbb{N}, k \geq 1$, for a polyhedron P is defined by functions v such* *that*

$$\begin{cases} \bullet \ v \text{ is a polynomial of degree } k \text{ on each edge } E \text{ of the polyhedron } P, \text{ i.e. } v|_E \in \mathcal{P}_k(E) \\ \bullet \ v \text{ is globally continuous on } \partial P, \text{ i.e. } v|_{\partial P} \in C^0(\partial P) \\ \bullet \ \Delta v \text{ is a polynomial of degree } k-2 \text{ in } P, \text{ i.e. } \Delta v \in \mathcal{P}_{k-2}(P) \\ \bullet \text{ for every face } F \text{ in } \partial P, \ v|_F \in W_k(F) \end{cases}$$

The first three conditions are the three-dimensional equivalent of Definition 2.1. Conversely, the 4th condition is added as the boundary of a polyhedron is made of the set of its faces, where information is available for moments up to order $k-2$ only. This statement will be much clearer in the following chapter, where VEM will be applied to elastostatics. Once again, a polyhedron of order k satisfies all of the above requirements in Definition 2.4, so that the following inclusion holds

$$\mathcal{P}_k(P) \subset V_k(P) \tag{2.37}$$

which is essential for convergence properties. The corresponding *local degrees of freedom for the 3D virtual element space* are given by the following.

Proposition 2.2. *Local degrees of freedom for the virtual element space embedded in* $\mathbb{R}^3$.
An element v *of the space* $V_k(P)$ *defined in* Definition 2.4 *is uniquely identified by the DOFs*

$$\Xi : V_k(P) \to \mathbb{R}$$

grouped in the four following sets

- *the value of v at the vertices of P*
- *for each edge E of P, the value of v at the $k-1$ internal points of the $(k+1)$-point Gauss-Lobatto quadrature rule on E*
- *for each face F of P, the n_{k-2} moments up to order $k-2$ of v in F*

$$\frac{1}{|F|}\int_F v m_\alpha, \quad \alpha = 1, ..., n_{k-2} \tag{2.38}$$

  where m_α are the scaled monomials defined in (2.1) *and n_{k-2} in* (2.2)
- *the ν_{k-2} scaled moments up to order $k-2$ of v in P:*

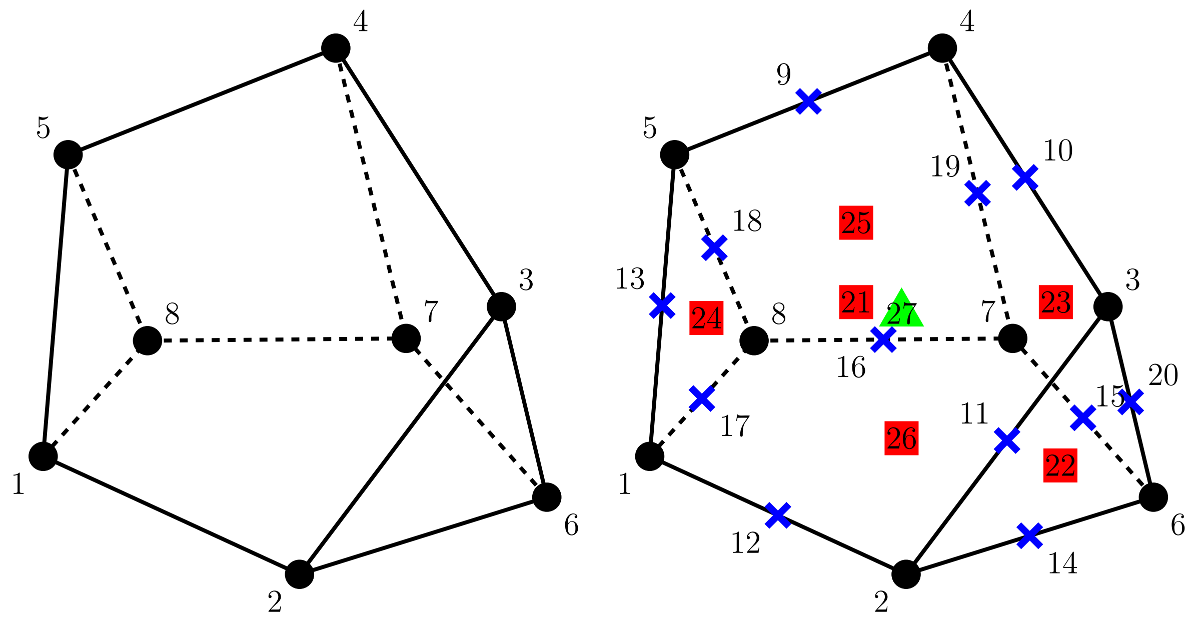
$$\frac{1}{|P|}\int_F v \mu_\alpha, \quad \alpha = 1, ..., \nu_{k-2} \tag{2.39}$$

  where μ_α are the scaled monomials defined in (2.4) *and ν_{k-2} in* (2.5)

The proof of Proposition 2.2 can be found in [1]. The dimension of the space $V_k(P)$ coincides with the number of corresponding degrees of freedom N_{DOF}

$$\text{dim}V_k(P) = N_{DOF} = N_V + N_E(k-1) + N_F n_{k-2} + \nu_{k-2} \qquad (2.39)$$

where N_V, N_E and N_F are respectively the number of vertices, edges and faces belonging to the polyhedron P. The space decomposition of the space $V_k(P)$ into its basis functions φ through Lagrangian interpolation obviously still applies in an equivalent manner as described in (2.24) and (2.25) for the 2D case. A visual representation of the degrees of freedom defined in Proposition 2.2 is shown in Figure 2.7



(a) $V_1(F)$ (b) $V_2(F)$

Figure 2.7: Local degrees of freedom for the two virtual element spaces $V_1(P)$ and $V_2(P)$ in a pentagonal wedge P. The black dots correspond to vertices DOFs, blue crosses to edge DOFs, red squares to internal face DOFs and green triangles to internal volume DOFs. Note that while the first two sets match with the function evaluation at the precise location shown in the figure, the last two do not have a geometric punctual representation and are displayed respectively inside the faces and the polyhedron for the sake of simplicity.

3 | Virtual elements for elastostatics in 3D

With the tools presented in 2 it is now possible to tackle the three-dimensional linear elastostatic problem with the virtual element method. Many choices, including the use of scaled monomials, the selection of the degrees of freedom to describe the virtual element space and the use of the direct projection operator for faces only will be clear as the scheme presented in Subsection 1.3.7 is followed. The polytopic mesh described in Section 2.1 and its conventions to characterize the entities will be adopted.

3.1. Definition of the displacement and strain models

Each of the three components of the unknown local displacement vector field will be approximated by the virtual element space presented in Section 2.5. As described in Subsection 1.3.1, we recall that it is possible to collect the basis functions, here represented with the symbol N^u to cope with usual structural mechanics notation, in the following $[3 \times n_u]$ matrix $\boldsymbol{N_u}$

$$\boldsymbol{N_u} = \begin{bmatrix} N_1^u & 0 & 0 & N_2^u & 0 & 0 & \dots & N_{N_{DOF}}^u & 0 & 0 \\ 0 & N_1^u & 0 & 0 & N_2^u & 0 & \dots & 0 & N_{N_{DOF}}^u & 0 \\ 0 & 0 & N_1^u & 0 & 0 & N_2^u & \dots & 0 & 0 & N_{N_{DOF}}^u \end{bmatrix} \tag{3.1}$$

where the first N_V $[3 \times 3]$ blocks correspond to vertex nodal DOFs, the second $N_E(k-1)$ $[3 \times 3]$ blocks to edge DOFs, the third $N_F n_{k-2}$ $[3 \times 3]$ blocks to face moment DOFs and the last ν_{k-2} $[3 \times 3]$ blocks are associated with volume moment internal DOFs. Matrix $\boldsymbol{N_u}$ realizes the map between the local DOFs described in Proposition 2.2, gathered in the vector $\hat{\boldsymbol{u}}$, and the continuous displacement field inside the element P. Once again, we stress that the shape functions N_i^u are not explicitly known, unless the corresponding partial differential equation is solved, and will remain so even after the solution of the virtual element approximation scheme. As anticipated in Section 2.3, the projection operator for the space $V_k(P)$ will be encapsulated in the definition of the strain field, being this the symmetric gradient of the displacement. Following the general procedure for mixed finite elements of Chapter 1, the virtual element projection operator is embodied by the compatibility matrix $\boldsymbol{C}$, *projecting* the displacement field (which contains polynomials of degree k plus other additional functions) into the strain field (made of polynomials up to degree $k-1$). Namely, matrix $\boldsymbol{C}$ projects the local displacement DOFs $\hat{\boldsymbol{u}}$ into the parameters $\hat{\boldsymbol{\varepsilon}}$ of the polynomial strain field and it is the discrete counterpart of

$$\Pi_P : V_k(P) \rightarrow \mathcal{P}_{k-1}(P) \tag{3.2}$$

In simple terms, $\boldsymbol{C}$ encapsulates the gradient operator and the projection operator in one single matrix mapping the discrete parameters. In view of the above considerations, the local strain field is *a priori* defined as it contains a polynomial of order $k-1$. Therefore, following (2.5), matrix $\boldsymbol{N}_\varepsilon$ has dimensions $[6 \times 6\nu_{k-1}]$, as it is composed by flanking ν_{k-1} $[6 \times 6]$ diagonal blocks with the entries of the $[\nu_{k-1} \times 1]$ vector $\boldsymbol{\mu}_k$ defined in (2.6)

$$\boldsymbol{N}_\varepsilon = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \xi & 0 & 0 & 0 & 0 & 0 & \dots & \mu_{\nu_{k-1}} & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & \xi & 0 & 0 & 0 & 0 & \dots & 0 & \mu_{\nu_{k-1}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \xi & 0 & 0 & 0 & \dots & 0 & 0 & \mu_{\nu_{k-1}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \xi & 0 & 0 & \dots & 0 & 0 & 0 & \mu_{\nu_{k-1}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \xi & 0 & \dots & 0 & 0 & 0 & 0 & \mu_{\nu_{k-1}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \xi & \dots & 0 & 0 & 0 & 0 & 0 & \mu_{\nu_{k-1}} \end{bmatrix} \tag{3.3}$$

If the lowest order $k = 1$ is adopted, the strain model reduces to the $[6 \times 6]$ identity matrix.

3.2. Local stiffness matrix

3.2.1. Consistent part of the local stiffness matrix

Having defined the strain and displacement model, it is possible to tackle the assembly of the consistent part of the local stiffness matrix, starting from the computation of $\boldsymbol{C}$. We recall from (1.39) that we need the two matrices $\boldsymbol{G}$ and $\boldsymbol{A}$. The first one, given in (1.36), expands to the $[\nu_{k-1} \times \nu_{k-1}]$

$$\boldsymbol{G} = \int_P \boldsymbol{N}_\varepsilon^{\mathrm{T}} \boldsymbol{N}_\varepsilon \, d\Omega = \int_P \begin{bmatrix} \boldsymbol{I} \\ \xi\boldsymbol{I} \\ \eta\boldsymbol{I} \\ \zeta\boldsymbol{I} \\ \vdots \\ \mu_{\nu_{k-1}}\boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} & \xi\boldsymbol{I} & \eta\boldsymbol{I} & \zeta\boldsymbol{I} & \dots & \mu_{\nu_{k-1}}\boldsymbol{I} \end{bmatrix} d\Omega =$$

$$= \int_P \begin{bmatrix} \boldsymbol{I} & \xi\boldsymbol{I} & \eta\boldsymbol{I} & \zeta\boldsymbol{I} & \dots & \xi^{k-2}\zeta\boldsymbol{I} \\ \xi\boldsymbol{I} & \xi^2\boldsymbol{I} & \xi\eta\boldsymbol{I} & \xi\zeta\boldsymbol{I} & \dots & \xi\mu_{\nu_{k-1}}\boldsymbol{I} \\ \eta\boldsymbol{I} & \xi\eta\boldsymbol{I} & \eta^2\boldsymbol{I} & \eta\zeta\boldsymbol{I} & \dots & \eta\mu_{\nu_{k-1}}\boldsymbol{I} \\ \zeta\boldsymbol{I} & \xi\zeta\boldsymbol{I} & \eta\zeta\boldsymbol{I} & \zeta^2\boldsymbol{I} & \dots & \zeta\mu_{\nu_{k-1}}\boldsymbol{I} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mu_{\nu_{k-1}}\boldsymbol{I} & m_{\nu_{k-1}}\xi\boldsymbol{I} & \mu_{\nu_{k-1}}\eta\boldsymbol{I} & \mu_{\nu_{k-1}}\zeta\boldsymbol{I} & \dots & \mu_{\nu_{k-1}}^2\boldsymbol{I} \end{bmatrix} d\Omega \tag{3.4}$$

where $\boldsymbol{I}$ is the $[6 \times 6]$ identity matrix. The integrals in (3.4) are computable *exactly* (up to machine precision) by means of numerical integration techniques over polyhedral domains. A brief discussion of the available methods is due and reported here.

- A straightforward approach is to *sub-tetrahedralize* the polyhedron, adopt Gaussian integration over each tetrahedron and sum the result of the integrals. This however, requires a considerable amount of computational effort as the number of Gauss points increases with cubic power of the order of the method k. In fact, n-point Gaussian quadrature in one-dimensional domains guarantees exact integration of polynomials of order $2n - 1$.

$$k = 2n - 1 \implies n = \left\lceil \frac{k+1}{2} \right\rceil \tag{3.5}$$

Exploiting tensor product, the rule is able to integrate exactly polynomials up to order k for d-dimensional domains if $\left\lceil \frac{k+1}{2} \right\rceil^d$ points are used, where $\lceil(\cdot)\rceil$ is the ceiling operator applied to the quantity $(\cdot)$

$$n = \left\lceil \frac{k+1}{2} \right\rceil^d \tag{3.6}$$

Assuming the polyhedron has N_F faces, a simple tratrahedralization (only valid for star-shaped[1] polyhedra) splits the faces in triangles and generates the corresponding tetrahedra with an internal chosen common point, the center of the star. We indicate with N_Δ the number of triangles resulting from face triangulations, given by

$$N_\Delta = \sum_{f=1}^{N_F} (N_{f,E} - 2)$$

where $N_{f,E}$ are the numbers of edges of the face indexed with f. As an example, a hexahedron, whose 6 faces are quadrilaterals, has $N_\Delta = \sum_{f=1}^{6}(4-2) = 12$, hence its faces will be decomposed in 12 triangles. Following the above notation, the number of points required to exactly integrate a polynomial of order k described in (3.6) becomes, for a polyhedral domain,

$$n = N_\Delta \left\lceil \frac{k+1}{2} \right\rceil^3 \tag{3.7}$$

Now, a generic $[n \times n]$ symmetric matrix has $\frac{n(n+1)}{2}$ independent entries. The $[6\nu_{k-1} \times 6\nu_{k-1}]$ matrix $\boldsymbol{G}$ is block-symmetric and thus requires to compute in general the $\frac{\nu_{k-1}(\nu_{k-1}+1)}{2}$ independent components. These are monomials up to order $2(k-1)$ and hence require to compute

$$n = N_\Delta \left\lceil \frac{2(k-1)+1}{2} \right\rceil^3 = N_\Delta \left\lceil \frac{2k-1}{2} \right\rceil^3 = N_\Delta k^3 \qquad \forall k \in \mathbb{N} \tag{3.8}$$

Gauss points for each polyhedron P and for each set of these points they require to perform $\frac{\nu_{k-1}(\nu_{k-1}+1)}{2}$ function evaluations to fill the matrix $\boldsymbol{G}$. To better understand this, we apply (3.8) to a hexahedron and compute the number of Gauss points required following the above-presented method in Table 3.1.

	Independent entries of $\boldsymbol{G}$	Gauss points	Function evaluations
$k = 1$	1	12	12
$k = 2$	10	96	960
$k = 3$	55	324	17820
$k = 4$	210	768	161280

Table 3.1: Computational cost to produce the entries of matrix G using Gauss sub-tetrahedralization for a hexahedron as a function of the order k of the VEM.

[1]A set Ω in the Euclidean space $\mathbb{R}^d$ is a star-shaped domain (or radially convex set) if there exists an $\boldsymbol{x}_0 \in \Omega$, the center of the star, such that for all $\boldsymbol{x} \in \Omega$, the line segment $\overline{\boldsymbol{x}_0\boldsymbol{x}}$ lies in Ω.

Already with $k = 2$, the number of function evaluations to populate one single matrix $\boldsymbol{G}$ sensibly increases, underlying the weakness of Gaussian quadrature for this purpose.

- Some variants of the above technique manage to reduce the number of points needed for the integration over tetrahedra (e.g. [19]), though still needing to process sub-tetrahedralization to handle polyhedra. Other rules avoid the need of subdivision of the integration domain in tetrahedra, as presented in [22] and in the brand new [27], or belong to the *compressed* integration techniques, which manage to reduce the number of evaluation points (as in [5]). However, the latter are mostly available for 2D domains, and the computational cost required to produce such schemes is not negligible.
- The choice adopted in this work follows an algorithm presented in [2], a technique developed *ad hoc* to build stiffness matrices for discontinuous Galerkin methods on general polytopic meshes.

The second matrix involved in the computation of $\boldsymbol{C}$ is the $[6\nu_{k-1} \times n_u]$ matrix $\boldsymbol{A}$, defined in (1.40). Performing by-parts integration, one obtains

$$\begin{aligned}\boldsymbol{A} = \int_P \boldsymbol{N}_{\boldsymbol{\varepsilon}}^{\mathrm{T}} \boldsymbol{S} \boldsymbol{N}_{\boldsymbol{u}}\, d\Omega &= \int_{\partial P} (\mathbb{N}_P \boldsymbol{N}_{\boldsymbol{\varepsilon}})^{\mathrm{T}} \boldsymbol{N}_{\boldsymbol{u}}\, d\Sigma - \int_P (\boldsymbol{S}^{\mathrm{T}} \boldsymbol{N}_{\boldsymbol{\varepsilon}})^{\mathrm{T}} \boldsymbol{N}_{\boldsymbol{u}}\, d\Omega = \\ &= \sum_{F \in \partial P} \left[\int_F (\mathbb{N}_F \boldsymbol{N}_{\boldsymbol{\varepsilon}})^{\mathrm{T}} \boldsymbol{N}_{\boldsymbol{u}}\, d\Sigma \right] - \int_P (\boldsymbol{S}^{\mathrm{T}} \boldsymbol{N}_{\boldsymbol{\varepsilon}})^{\mathrm{T}} \boldsymbol{N}_{\boldsymbol{u}}\, d\Omega \end{aligned} \tag{3.9}$$

where $\mathbb{N}_P$ is the $[3 \times 6]$ matrix collecting the direction cosines of the outward normal unit vector on the surface ∂P, and $\mathbb{N}_F$ the corresponding one for a single face F. It is convenient to separate the two quantities $\boldsymbol{A}$ has been split into, and analyze their properties.

$$\boldsymbol{A}_1 = \sum_{F \in \partial P} \left[\int_F (\mathbb{N}_F \boldsymbol{N}_{\boldsymbol{\varepsilon}})^{\mathrm{T}} \boldsymbol{N}_{\boldsymbol{u}}\, d\Sigma \right] \tag{3.10}$$

$$\boldsymbol{A}_2 = -\int_P (\boldsymbol{S}^{\mathrm{T}} \boldsymbol{N}_{\boldsymbol{\varepsilon}})^{\mathrm{T}} \boldsymbol{N}_{\boldsymbol{u}}\, d\Omega \tag{3.11}$$

Each term in the summation of $\boldsymbol{A}_1$ is a matrix made of integrals *over a polygon* of a polynomial of at most degree $k-1$ multiplied with the unknown basis functions. This is one of the key differences in implementing VE in 3D with respect to a two-dimensional setting, where the boundary of the polygonal element is made by its *edges*, where the basis functions are *explicitly known* from the degrees of freedom. Conversely, in three dimensions, only the integrals of the shape functions multiplied by a polynomial up to order $k-2$ can be directly addressed by the internal face DOFs. To tackle the problem for the remaining $(k-1)$-order monomials, we can draw from Theorem 2.1

$$\int_F m_{\boldsymbol{\alpha}} N_i^u \, d\Sigma = \int_F m_{\boldsymbol{\alpha}} \Pi_{F,k}^{\nabla} N_i^u \, d\Sigma \qquad \forall \boldsymbol{\alpha} : |\boldsymbol{\alpha}| = k-1 \tag{3.12}$$

and we can explicitly compute the integrals through the projection $\Pi_{F,k}^{\nabla} N_i^u$, whose coefficients come from solving the linear algebraic system (2.31). Expanding $\boldsymbol{A}_1$ leads to

$$\begin{aligned}
\boldsymbol{A}_1 &= \sum_{F\in\partial P} \left[\int_F (\mathbb{N}_F \boldsymbol{N}_\varepsilon)^{\mathrm{T}} \boldsymbol{N}_{\boldsymbol{u}} \, d\Sigma \right] = \\
&= \sum_{F\in\partial P} \left[\int_F \begin{bmatrix} \mathbb{N}_F \\ \xi\mathbb{N}_F \\ \eta\mathbb{N}_F \\ \zeta\mathbb{N}_F \\ \vdots \\ \mu_{\nu_{k-1}}\mathbb{N}_F \end{bmatrix} \begin{bmatrix} N_1^u \boldsymbol{I} & N_2^u \boldsymbol{I} & \dots & N_{N_{DOF}}^u \boldsymbol{I} \end{bmatrix} d\Sigma \right] = \\
&= \sum_{F\in\partial P} \left[\int_F \begin{bmatrix} \mathbb{N}_F N_1^u & \mathbb{N}_F N_2^u & \dots & \mathbb{N}_F N_{N_{DOF}}^u \\ \xi\mathbb{N}_F N_1^u & \xi\mathbb{N}_F N_2^u & \dots & \xi\mathbb{N}_F N_{N_{DOF}}^u \\ \eta\mathbb{N}_F N_1^u & \eta\mathbb{N}_F N_2^u & \dots & \eta\mathbb{N}_F N_{N_{DOF}}^u \\ \zeta\mathbb{N}_F N_1^u & \zeta\mathbb{N}_F N_2^u & \dots & \zeta\mathbb{N}_F N_{N_{DOF}}^u \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{\nu_{k-1}}\mathbb{N}_F N_1^u & \mu_{\nu_{k-1}}\mathbb{N}_F N_2^u & \dots & \mu_{\nu_{k-1}}\mathbb{N}_F N_{N_{DOF}}^u \end{bmatrix} d\Sigma \right]
\end{aligned} \tag{3.13}$$

where $\boldsymbol{I}$ is the $[3\times 3]$ identity matrix. Each $[6\times 3]$ submatrix in between rows $6(i-1)+1$ and $6i$ and between columns $3(j-1)+1$ and $3j$ contains the direction cosines of the outward normal unit vector of the face multiplied by the i^{th} scaled monomial m_i and j^{th} shape function N_j^u. Explicitly expanded, each block reads

$$\mu_i \mathbb{N}_F N_1^u = \begin{bmatrix} \mu_i n_x N_j^u & 0 & 0 \\ 0 & \mu_i n_y N_j^u & 0 \\ 0 & 0 & \mu_i n_z N_j^u \\ \mu_i n_y N_j^u & \mu_i n_x N_j^u & 0 \\ 0 & \mu_i n_z N_j^u & \mu_i n_y N_j^u \\ \mu_i n_z N_j^u & 0 & \mu_i n_x N_j^u \end{bmatrix}$$

According to the order k of the VEM and the type of shape function being integrated, the entries of $\boldsymbol{A}_1$ can be directly computed from the virtual DOFS or require a projection operation through (3.12). First we focus on vertex-type, edge-type and face-type DOFS for the entries of $\boldsymbol{A}_1$. For each face, the $[6 \times 3]$ blocks up to row $6\nu_{k-2}$ contain restrictions on polygons of monomials up to degree $k-2$ in $\mathbb{R}^3$ multiplied by shape functions, which can be transformed into a combination of the virtual degrees of freedom through the change of coordinates described in (2.10). Lagrangian interpolation property ensures that in these first $6\nu_{k-2}$ rows, only the columns corresponding to face-type DOFS have non-zero entries. For the remaining rows, from $6\nu_{k-2}+1$ to $6\nu_{k-1}$, the projection operation described in Section 2.3 has to be applied to find the coefficients of the monomials up to order k of the polynomial $\Pi^{\nabla}_{F,k} N^u_j$ appearing in the identity (3.12). Once these are found, these last entries of $\boldsymbol{A}_1$ are obtained by integrating the resulting polynomial of degree $(k-1)+k = 2k-1$. The polyhedron-type DOFS were left out in $\boldsymbol{A}_1$ and indeed their contribution is zero. In fact, by inspecting the right hand side (2.33) of the projection linear algebraic system, one notices that for a polyhedron-type shape function N^u_j

$$\frac{1}{|F|}\int_F N^u_j \, d\Sigma = 0$$

thanks to Lagrangian interpolation property, which ensures that the above quantity is 1 if and only if j corresponds to the so-defined face-type DOF. Similarly, after performing the change of coordinates described in (2.10), the quantity

$$\int_F \Delta m_{\boldsymbol{\alpha}} N^u_j \, d\Sigma = 0 \qquad \forall \boldsymbol{\alpha} : |\boldsymbol{\alpha}| = k-1$$

again due to Lagrangian interpolation property with respect to face-type DOFS. Lastly,

$$\int_E \nabla m_{\boldsymbol{\alpha}} \cdot \boldsymbol{n} N^u_j \, d\Gamma = 0$$

due to Lagrangian interpolation property with respect to vertex-type and edge-type DOFS, and since the shape functions are *exactly* polynomials on the edges of the polyhedron.

Matrix $\boldsymbol{A}_2$ appears only when the order of the VEM $k \geq 2$, since the derivatives of the constant strain field vanish for $k = 1$. The divergence of the strain field in $[3 \times 6\nu_{k-1}]$

matrix form $\boldsymbol{S}^{\mathrm{T}}\boldsymbol{N}_{\varepsilon}$ expands to

$$\begin{aligned}
\boldsymbol{S}^{\mathrm{T}}\boldsymbol{N}_{\varepsilon} &= \begin{bmatrix} \partial_x & 0 & 0 & \partial_y & 0 & \partial_z \\ 0 & \partial_y & 0 & \partial_x & \partial_z & 0 \\ 0 & 0 & \partial_z & 0 & \partial_y & \partial_x \end{bmatrix} \begin{bmatrix} \boldsymbol{I} & \xi\boldsymbol{I} & \eta\boldsymbol{I} & \zeta\boldsymbol{I} & \dots & \mu_{\nu_{k-1}}\boldsymbol{I} \end{bmatrix} = \\
&= \frac{1}{h_P}\begin{bmatrix} 0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0 \dots \partial_\xi\mu_{\nu_{k-1}} & 0 & 0 & \partial_\eta\mu_{\nu_{k-1}} & 0 & \partial_\zeta\mu_{\nu_{k-1}} \\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0 \dots \quad 0 & \partial_\eta\mu_{\nu_{k-1}} & 0 & \partial_\xi\mu_{\nu_{k-1}} & \partial_\zeta\mu_{\nu_{k-1}} & 0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1 \dots \quad 0 & 0 & \partial_\zeta\mu_{\nu_{k-1}} & 0 & \partial_\eta\mu_{\nu_{k-1}} & \partial_\xi\mu_{\nu_{k-1}} \end{bmatrix} = \\
&= \frac{1}{h_P}\tilde{\boldsymbol{M}} \qquad (3.14)
\end{aligned}$$

where $\boldsymbol{I}$ is the $[6 \times 6]$ identity matrix. The $[3 \times 6\nu_{k-1}]$ matrix $\tilde{\boldsymbol{M}}$ is constant for all the elements of the virtual element program and can also be expressed as a combination of the scaled monomials $\boldsymbol{\mu}_{k-2}$

$$\tilde{\boldsymbol{M}} = \sum_{i=1}^{\nu_{k-2}} \tilde{\boldsymbol{M}}_i \mu_i \qquad (3.15)$$

where $\tilde{\boldsymbol{M}}_i$ gathers the coefficients of the scaled monomial μ_i. Substituting (3.15) in (3.14) and plugging back in (3.11) one obtains

$$\begin{aligned}
\boldsymbol{A}_2 &= -\int_P (\boldsymbol{S}^{\mathrm{T}}\boldsymbol{N}_{\varepsilon})^{\mathrm{T}}\boldsymbol{N}_{\boldsymbol{u}}\, d\Omega = -\frac{1}{h_P}\int_P \sum_{i=1}^{\nu_{k-2}} \left[\tilde{\boldsymbol{M}}_i^{\mathrm{T}}\mu_i\right] \boldsymbol{N}_{\boldsymbol{u}}\, d\Omega = \\
&= -\frac{1}{h_P}\sum_{i=1}^{\nu_{k-2}} \left[\tilde{\boldsymbol{M}}_i^{\mathrm{T}} \left(\int_P \mu_i \boldsymbol{N}_{\boldsymbol{u}}\, d\Omega\right)\right] \qquad (3.16)
\end{aligned}$$

By Lagrangian interpolation property, the quantities in round brackets assume values equal to the volume $|P|$ of the element if the shape function refers to the corresponding internal polyhedron-type DOF, and zero otherwise.

Having assembled $\boldsymbol{G}$ and $\boldsymbol{A}$, the $[6\nu_{k-1}\times n_u]$ compatibility matrix $\boldsymbol{C}$ is computed according to (1.39), and the $[n_u\times n_u]$ local consistent part of the stiffness matrix $\boldsymbol{K}_e^c$, following (1.45), requires the $[\nu_{k-1} \times \nu_{k-1}]$ elastic matrix $\boldsymbol{E}$, which expands to the fully computable

$$\boldsymbol{E} = \int_P \boldsymbol{N}_\varepsilon^{\mathrm{T}} \boldsymbol{D} \boldsymbol{N}_\varepsilon \, d\Omega = \int_P \begin{bmatrix} \boldsymbol{I} \\ \xi \boldsymbol{I} \\ \eta \boldsymbol{I} \\ \zeta \boldsymbol{I} \\ \vdots \\ \mu_{\nu_{k-1}} \boldsymbol{I} \end{bmatrix} \boldsymbol{D} \begin{bmatrix} \boldsymbol{I} & \xi \boldsymbol{I} & \eta \boldsymbol{I} & \zeta \boldsymbol{I} & \dots & \mu_{\nu_{k-1}} \boldsymbol{I} \end{bmatrix} d\Omega =$$

$$= \int_P \begin{bmatrix} \boldsymbol{D} & \xi \boldsymbol{D} & \eta \boldsymbol{D} & \zeta \boldsymbol{D} & \dots & \xi^{k-2} \zeta \boldsymbol{D} \\ \xi \boldsymbol{D} & \xi^2 \boldsymbol{D} & \xi \eta \boldsymbol{D} & \xi \zeta \boldsymbol{D} & \dots & \xi \mu_{\nu_{k-1}} \boldsymbol{D} \\ \eta \boldsymbol{D} & \xi \eta \boldsymbol{D} & \eta^2 \boldsymbol{D} & \eta \zeta \boldsymbol{D} & \dots & \eta \mu_{\nu_{k-1}} \boldsymbol{D} \\ \zeta \boldsymbol{D} & \xi \zeta \boldsymbol{D} & \eta \zeta \boldsymbol{D} & \zeta^2 \boldsymbol{D} & \dots & \zeta \mu_{\nu_{k-1}} \boldsymbol{D} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mu_{\nu_{k-1}} \boldsymbol{D} & m_{\nu_{k-1}} \xi \boldsymbol{D} & \mu_{\nu_{k-1}} \eta \boldsymbol{D} & \mu_{\nu_{k-1}} \zeta \boldsymbol{D} & \dots & \mu_{\nu_{k-1}}^2 \boldsymbol{D} \end{bmatrix} d\Omega \tag{3.17}$$

3.2.2. Stabilizing part of the local stiffness matrix

The computation of the stabilizing matrix requires the assembly of the $[n_u \times n_u]$ hourglass matrix $\boldsymbol{H}$, described in (1.61), which in turn reduces to the computation of the $[n_u \times n_{D+R}]$ matrix $\boldsymbol{T}_{D+R}$. In the context of the virtual element method, the rank deficiency of the consistent local stiffness matrix $\boldsymbol{K}_e^c$ is due to the extra non-polynomial functions allowable in the displacement field. If the displacement model was made of polynomials up to degree k only, then the $k-1$ polynomial strain field would capture all possible deforming modes and no stabilization would be required as hourglass modes would not arise. Therefore, the number of hourglass modes is equal to $n_u - 3\nu_k$, where indeed the number of parameters n_u (coinciding with the dimension of the virtual space for the displacement field) has been subtracted by the dimension $3\nu_k$ of the space spanned by the monomials up to degree k in three dimensions. Following the reasoning above, the approximate displacement field purified from the hourglass modes can be expressed as

$$\boldsymbol{u}_{D+R}(\boldsymbol{\xi}) = \boldsymbol{N}_k(\boldsymbol{\xi}) \hat{\boldsymbol{p}}_{D+R} = \boldsymbol{N}_{\boldsymbol{u}}(\boldsymbol{\xi}) \hat{\boldsymbol{u}}_{D+R} \tag{3.18}$$

where the $[3 \times 3\nu_k]$ matrix $\boldsymbol{N}_k$ gathers the scaled monomials $\boldsymbol{\mu}_k$

$$\boldsymbol{N}_k = \begin{bmatrix} 1 & 0 & 0 & \xi & 0 & 0 & \dots & \mu_{\nu_k} & 0 & 0 \\ 0 & 1 & 0 & 0 & \xi & 0 & \dots & 0 & \mu_{\nu_k} & 0 \\ 0 & 0 & 1 & 0 & 0 & \xi & \dots & 0 & 0 & \mu_{\nu_k} \end{bmatrix} \tag{3.19}$$

Recalling from (1.51) that

$$\hat{\boldsymbol{u}}_{D+R} = \boldsymbol{T}_{D+R}\hat{\boldsymbol{p}}_{D+R}$$

the identity (3.18) becomes

$$\boldsymbol{N}_k(\boldsymbol{\xi})\hat{\boldsymbol{p}}_{D+R} = \boldsymbol{N}_{\boldsymbol{u}}(\boldsymbol{\xi})\boldsymbol{T}_{D+R}\hat{\boldsymbol{p}}_{D+R} \tag{3.20}$$

Applying each one of the maps of the local degrees of freedom described in Proposition 2.2 to the above identity, the entries of matrix $\boldsymbol{T}_{D+R}$ are found. In other words, the procedure exploits the evaluation of each of the DOFS at the coordinates contained in matrix $\boldsymbol{N}_k$ and exploits the Lagrangian interpolation property to assess the entries of matrix $\boldsymbol{N}_{\boldsymbol{u}}$. Explicitly, if the first DOF of vertex-type is evaluated, indicating with $\boldsymbol{I}$ the $[3\times3]$ identity matrix, one has

$$\begin{bmatrix}\boldsymbol{I} & \xi_1\boldsymbol{I} & \dots & \mu_{\nu_k 1}\boldsymbol{I}\end{bmatrix}\hat{\boldsymbol{p}}_{D+R} = \begin{bmatrix}\boldsymbol{I} & 0\boldsymbol{I} & \dots & 0\boldsymbol{I}\end{bmatrix}\boldsymbol{T}_{D+R}\hat{\boldsymbol{p}}_{D+R}$$

since $N_1^u(\boldsymbol{\xi}_j) = \delta_{1j} \quad \forall j = 1,\dots,N_{N_{DOF}}$. Similarly, evaluating a generic vertex-type or edge-type DOF labelled with j, one has

$$\begin{bmatrix}\boldsymbol{I} & \xi_j\boldsymbol{I} & \dots & \mu_{\nu_k j}\boldsymbol{I}\end{bmatrix}\hat{\boldsymbol{p}}_{D+R} = \begin{bmatrix}0\boldsymbol{I} & \dots & \boldsymbol{I} & \dots & 0\boldsymbol{I}\end{bmatrix}\boldsymbol{T}_{D+R}\hat{\boldsymbol{p}}_{D+R} \tag{3.21}$$

where the only three non-zero entries in the matrix on the right hand side lie in j^{th} $[3\times3]$ block. Evaluating a face-type DOF j yields to

$$\begin{bmatrix}\fint_F m_j\boldsymbol{I} & \fint_F \xi m_j\boldsymbol{I} & \dots & \fint_F \mu_{\nu_k} m_j\boldsymbol{I}\end{bmatrix}\hat{\boldsymbol{p}}_{D+R} = \begin{bmatrix}0\boldsymbol{I} & \dots & \boldsymbol{I} & \dots & 0\boldsymbol{I}\end{bmatrix}\boldsymbol{T}_{D+R}\hat{\boldsymbol{p}}_{D+R} \tag{3.22}$$

where the only three non-zero entries in the matrix on the right hand side lie in j^{th} $[3\times3]$ block after the blocks corresponding to vertex-type and edge-type and the symbol $\fint_\Omega(\cdot)$ stands for the averaged integral of $(\cdot)$

$$\fint_\Omega(\cdot) = \frac{1}{|\Omega|}\int_\Omega(\cdot)\,d\Omega$$

Finally, if a polyhedron-type DOF j is evaluated[2], the following identity is obtained

$$\begin{bmatrix}\fint_P \mu_j\boldsymbol{I} & \fint_P \xi\mu_j\boldsymbol{I} & \dots & \fint_P \mu_{\nu_k}\mu_j\boldsymbol{I}\end{bmatrix}\hat{\boldsymbol{p}}_{D+R} = \begin{bmatrix}0\boldsymbol{I} & \dots & \boldsymbol{I} & \dots & 0\boldsymbol{I}\end{bmatrix}\boldsymbol{T}_{D+R}\hat{\boldsymbol{p}}_{D+R} \tag{3.23}$$

[2]In the example j is not the last DOF of the collection, otherwise in the right hand side the last $[3\times3]$ block is the one containing the only non-zero entries.

where again the only three non-zero entries in the matrix on the right hand side lie in j^{th} $[3 \times 3]$ block after the blocks corresponding to face-type DOFS. In order to satisfy (3.21), (3.22) and (3.23) for every corresponding DOF, the matrix $\boldsymbol{T}_{D+R}$ has to contain the respective evaluation of $\boldsymbol{N}_k$ displayed on the left hand sides of the identities at the corresponding j^{th} row-block. Hence, the $[n_u \times 3\nu_k]$ matrix $\boldsymbol{T}_{D+R}$ explicitly becomes

$$\boldsymbol{T}_{D+R} = \begin{bmatrix} \boldsymbol{I} & \xi_1\boldsymbol{I} & \eta_1\boldsymbol{I} & \zeta_1\boldsymbol{I} & \dots & \mu_{\nu_k 1}\boldsymbol{I} \\ \boldsymbol{I} & \xi_2\boldsymbol{I} & \eta_2\boldsymbol{I} & \zeta_2\boldsymbol{I} & \dots & \mu_{\nu_k 2}\boldsymbol{I} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ ⨍_F \boldsymbol{I} & ⨍_F \xi\boldsymbol{I} & ⨍_F \eta\boldsymbol{I} & ⨍_F \zeta\boldsymbol{I} & \dots & ⨍_F \mu_{\nu_k}\boldsymbol{I} \\ ⨍_F \xi_f\boldsymbol{I} & ⨍_F \xi_f\xi\boldsymbol{I} & ⨍_F \xi_f\eta\boldsymbol{I} & ⨍_F \xi_f\zeta\boldsymbol{I} & \dots & ⨍_F \xi_f\mu_{\nu_k}\boldsymbol{I} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ ⨍_P \boldsymbol{I} & ⨍_P \xi\boldsymbol{I} & ⨍_P \eta\boldsymbol{I} & ⨍_P \zeta\boldsymbol{I} & \dots & ⨍_P \mu_{\nu_k}\boldsymbol{I} \\ ⨍_P \xi\boldsymbol{I} & ⨍_P \xi^2\boldsymbol{I} & ⨍_P \xi\eta\boldsymbol{I} & ⨍_P \xi\zeta\boldsymbol{I} & \dots & ⨍_P \xi\mu_{\nu_k}\boldsymbol{I} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \end{bmatrix} \tag{3.24}$$

where $\boldsymbol{I}$ is again the $[3 \times 3]$ identity matrix. Computing the hourglass matrix $\boldsymbol{H}$ requires the inversion of $(\boldsymbol{T}_{D+R})^{\mathrm{T}}\,\boldsymbol{T}_{D+R}$, which however is computationally acceptable considering the inversion has to be performed once per element, the matrices are sparse and their dimension is relatively small.

3.3. Equivalent nodal forces vector

Recalling the expression of the equivalent nodal forces vector reported in (1.41) clearly establishes how the explicit computation is unfeasible. A suitable projection has to be performed to tackle both the body forces and the surface tractions, when these are applied. For the latter

$$\boldsymbol{F}_e^p = \int_{\partial_p P} \boldsymbol{N}_{\boldsymbol{u}}^{\mathrm{T}} \boldsymbol{p}\, d\sigma \tag{3.25}$$

where $\boldsymbol{p}$ is the $[3 \times 1]$ vector gathering the surface tractions, the computation can be done by exploiting the projections of $\boldsymbol{N}_{\boldsymbol{u}}$ already computed for the entries of $\boldsymbol{A}_1$. For the former, we recall the definition

$$\boldsymbol{F}_e^b = \int_P \boldsymbol{N}_{\boldsymbol{u}}^{\mathrm{T}} \boldsymbol{b}\, d\Omega \tag{3.26}$$

where $\boldsymbol{b}$ is the $[3 \times 1]$ vector gathering the body forces per unit volume. Different choices are available to approximate the above integral. In [1] it is shown that optimal error estimate in the H^1 and L^2 norm is obtained if the shape functions N_i^u are projected into the space of polynomials of degree k through the projector $\Pi^0_{P,k}$[3]. In the same [1] it is also proved that optimal convergence rates are still achieved if the projector $\Pi^0_{P,k-1}$ is applied to the displacement basis N_i^u for VEM with order $k = 1, 2$ and if the projector $\Pi^0_{P,k-2}$ is applied for order $k > 2$. In this work the approximation of the equivalent nodal forces vector is distinguished in the cases $k = 1$ and $k \geq 1$.

3.3.1. Approximation of body forces for first order VEM

If $k = 1$, the approximation of the body forces can be achieved by projecting the shape functions onto the space of constants. The technique is exactly the same as the one introduced in the first condition of (2.27) for the computation of the right hand side of the algebraic system for $\Pi^\nabla_{F,1}$. Given the $[3 \times 1]$ vector $\boldsymbol{f}$

$$\boldsymbol{f} = \begin{Bmatrix} f_x \\ f_y \\ f_z \end{Bmatrix} = \int_P \begin{Bmatrix} b_x \\ b_y \\ b_z \end{Bmatrix} d\Omega = \int_P \boldsymbol{b}\, d\Omega$$

The equivalent nodal forces vector for the body components can be computed as

$$\begin{aligned} \boldsymbol{F}_e^b = \int_P \boldsymbol{N}_{\boldsymbol{u}}^{\mathrm{T}} \boldsymbol{b}\, d\Omega &\approx \int_P \left[\Pi^0_{P,0} \boldsymbol{N}_{\boldsymbol{u}}\right]^{\mathrm{T}} \boldsymbol{b}\, d\Omega = \\ &= \int_P \left[\tfrac{1}{N_V}\boldsymbol{I} \quad \tfrac{1}{N_V}\boldsymbol{I} \quad \dots \quad \tfrac{1}{N_V}\boldsymbol{I}\right]^{\mathrm{T}} \boldsymbol{b}\, d\Omega = \\ &= \frac{1}{N_V} \begin{Bmatrix} f_x & f_y & f_z & \dots & f_x & f_y & f_z \end{Bmatrix}^{\mathrm{T}} \end{aligned} \tag{3.27}$$

where $\boldsymbol{I}$ is the $[3 \times 3]$ identity matrix and N_V the number of vertices of the element. In other words, the body forces are uniformly distributed to the vertices of the polyhedron.

3.3.2. Approximation of body forces for higher order VEM

If $k \geq 2$ the body forces $\boldsymbol{b}$ are projected in the space of polynomials of degree $k - 2$, so that the approximated body forces $\boldsymbol{b}^h$ can be expressed as

$$\boldsymbol{b}^h = \Pi^0_{P,k-2} \boldsymbol{b} = \sum_{i=1}^{\nu_{k-2}} \mu_i \hat{\boldsymbol{b}}_i^h \tag{3.28}$$

[3]The projection $\Pi^0_{P,k}$ is similarly achieved as described in Section 2.3 for the operator $\Pi^\nabla_{F,k}$, but applied so that the orthogonality conditions hold between the function and the monomial instead of between their gradients, and the domain is polyhedral. Moreover, the enhance property of Theorem 2.1 is exploited to compute the right hand side vector of the system leading to the coefficients of the polynomial projection.

where $\hat{\boldsymbol{b}}_i^h$ is the $[3 \times 1]$ vector gathering the coefficients for the monomial μ_i. These can be computed in a similar manner as presented in Section 2.3. The orthogonality condition in this case reads

$$\int_P \mu_j \left(\boldsymbol{b}^h - \boldsymbol{b}\right) \, d\Omega = \boldsymbol{0} \qquad \forall j \leq \nu_{k-2} \tag{3.29}$$

which, after substituting (3.28) in place of $\boldsymbol{b}^h$, translates into

$$\int_P \mu_j \left[\sum_{i=1}^{\nu_{k-2}} \left(\mu_i \hat{\boldsymbol{b}}_i^h\right) - \boldsymbol{b}\right] d\Omega = \boldsymbol{0} \qquad \forall j \leq \nu_{k-2}, \quad \forall i = 1, ..., \mathrm{dim} V_k(F) \tag{3.30}$$

and rearranging leads to

$$\sum_{i=1}^{\nu_{k-2}} \left[\left(\int_P \mu_j \mu_i \, d\Omega\right) \hat{\boldsymbol{b}}_i^h\right] = \int_P \mu_j \boldsymbol{b} \, d\Omega \qquad \forall j \leq \nu_{k-2} \tag{3.31}$$

which is an algebraic linear system in the unknown $[3\nu_{k-2} \times 1]$ vector $\hat{\boldsymbol{b}}^h$. Explicitly, the above linear system reads

$$\begin{bmatrix} \left(\int_P 1\right) \boldsymbol{I} & \left(\int_P \xi\right) \boldsymbol{I} & \dots & \left(\int_P \mu_{\nu_{k-2}}\right) \boldsymbol{I} \\ \left(\int_P \xi\right) \boldsymbol{I} & \left(\int_P \xi^2\right) \boldsymbol{I} & \dots & \left(\int_P \xi \mu_{\nu_{k-2}}\right) \boldsymbol{I} \\ \vdots & \vdots & \ddots & \vdots \\ \left(\int_P \mu_{\nu_{k-2}}\right) \boldsymbol{I} & \left(\int_P \xi \mu_{\nu_{k-2}}\right) \boldsymbol{I} & \dots & \left(\int_P \mu_{\nu_{k-2}}^2\right) \boldsymbol{I} \end{bmatrix} \begin{Bmatrix} \hat{\boldsymbol{b}}_1^h \\ \hat{\boldsymbol{b}}_2^h \\ \vdots \\ \hat{\boldsymbol{b}}_{\nu_{k-2}}^h \end{Bmatrix} = \begin{Bmatrix} \int_P \boldsymbol{b} \\ \int_P \xi \boldsymbol{b} \\ \vdots \\ \int_P \mu_{\nu_{k-2}} \boldsymbol{b} \end{Bmatrix} \tag{3.32}$$

where $\boldsymbol{I}$ is the $[3 \times 3]$ identity matrix and the differentials $d\Omega$ have been omitted for conciseness. Solving for $\hat{\boldsymbol{b}}^{\boldsymbol{h}}$ allows to find the direct coefficients to associate with the eqivalent nodal forces vector. Indeed, plugging the approximation (3.28) in (3.26) leads to

$$\boldsymbol{F}_e^b = \int_P \boldsymbol{N}_{\boldsymbol{u}}^{\mathrm{T}} \boldsymbol{b} \, d\Omega \approx \int_P \boldsymbol{N}_{\boldsymbol{u}}^{\mathrm{T}} \boldsymbol{b}^h \, d\Omega = \sum_{i=1}^{\nu_{k-2}} \left[\left(\int_P \mu_i \boldsymbol{N}_{\boldsymbol{u}}^{\mathrm{T}} \, d\Omega\right) \hat{\boldsymbol{b}}_i^h\right] \tag{3.33}$$

The quantities in round brackets are known directly from the polyhedron-type internal DOFS, available if the VEM order is $k \geq 2$. Specifically, by the Lagrangian interpolation property, the vector of local equivalent nodal forces, contains all zeros up to the previous index corresponding to the first polyhedron-type DOF, and then it contains exactly the entries of $\hat{\boldsymbol{b}}^h$. Explicitly

$$\boldsymbol{F}_e^b = \left\{0 \quad 0 \quad 0 \quad \dots \quad \hat{\boldsymbol{b}}_1^h \quad \hat{\boldsymbol{b}}_2^h \quad \dots \quad \hat{\boldsymbol{b}}_{\nu_{k-2}}^h\right\}^{\mathrm{T}} \tag{3.34}$$

4 | Algorithms

In this chapter a discussion on the adopted algorithms is reported.

4.1. Gauss-Lobatto quadrature rule

The virtual element method of order k in three dimensions requires the projection onto the space of polynomials of degree k over the faces of the mesh of the DOFS, as described in Section 2.3. The right hand side of the algebraic system to perform such operation requires the integration of polynomials up to order $2k-1$ over the skeleton of the polyhedra, made up by the edges E. Gauss-Lobatto quadrature rule is chosen because it matches with the 2 vertex-type plus $k-1$ edge-type DOFS completely describing the polynomial of order k representing the unknown field on the boundary, and allow to integrate exactly (up to machine precision) a polynomial of order $(k-1)+k = 2k-1$, resulting from the product of the polynomial restriction on the edges of the shape functions and the derivatives of a scaled monomial of order k.

The $n-2$ internal points of n^{th}-point Gauss-Lobatto quadrature rule over the standard integration interval $[-1,1]$ can be found as the stationary points of the $(n-1)^{\text{th}}$ *Legendre polynomials.* These functions can be defined recursively as

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ &\vdots \\ P_n(x) &= \frac{2n-1}{n} x P_{n-1}(x) - \frac{n-1}{n} P_{n-2}(x) \end{aligned} \tag{4.1}$$

The library `boost` already provides the definition of the Legendre polynomial function in of order n

```
boost::math::legendre_p(n, x)
```

and can be used to construct the derivatives of the Legendre polynomials recursively as

$$P_n'(x) = \frac{nxP_n(x) - nP_{n-1}(x)}{x^2 - 1} \tag{4.2}$$

To find the stationary points X_i of the Legendre polynomials in (4.1) one can easily set to zero (4.2). The weights W_i are then computed according to

$$W_i = \frac{2}{n(n-1)\left[P_{n-1}(x_i)\right]^2} \tag{4.3}$$

Computationally, to find all the zeros of the function in (4.2) it is convenient to call the solver $n-2$ times with bracketing points corresponding to the $(n-1)$-point Gauss-Lobatto rule. Indeed, it is easy to check that the stationary points of each Legendre polynomial are contained in between the intervals generated by the stationary points of the 1-order-less Legendre polynomial. Moreover, since the points are symmetric with respect to 0, the solver can find the Gauss-Lobatto points on the interval $[0, 1]$ and then mirror the results in the other $[-1, 0]$ interval. The call is therefore recursive but its not expensive as it is computed once for the reference interval $[-1, 1]$ and then stored in a `class` functioning as `cache` for easy access.

To find the Gauss-Lobatto integration points and weights for an edge, introducing the simple linear map

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}\sum_{i=1}^{n}\left[f\left(\frac{b-a}{2}X_i + \frac{a+b}{2}\right)W_i\right] \tag{4.4}$$

where a and b are the two extreme points of the edge, one can recognize the weights as

$$w_i = \frac{b-a}{2} \tag{4.5}$$

and the points as

$$x_i = \frac{b-a}{2}X_i + \frac{a+b}{2} \tag{4.6}$$

4.2. Outward normal unit vector and area of a polygon

Given a planar polygon embedded in $\mathbb{R}^3$ it is possible to compute the area and the outward normal vector by employing *Newell's algorithm.* This technique produces a vector which is normal to the polygon, follwoing the counter-clockwise orientation of its vertices, and whose magnitude is twice the area of the polygon itself. Newell's formula reads

$$\tilde{\boldsymbol{n}} = \frac{1}{2}\sum_{i=1}^{N_V}(\boldsymbol{V}_i \times \boldsymbol{V}_{i+1}) \tag{4.7}$$

where N_V are the number of vertices of the polygon, $\boldsymbol{V}_i$ are the coordinates of the i^{th} vertex, starting with index 1, and $\boldsymbol{V}_{i+1}$ the coordinates of the successive vertex following vertex i, so that for the last vertex, $\boldsymbol{V}_{i+1}$ refers again to vertex 1. To compute the outward unit normal, it is enough to normalize $\tilde{\boldsymbol{n}}$

$$\boldsymbol{n} = \frac{\tilde{\boldsymbol{n}}}{\|\tilde{\boldsymbol{n}}\|} \tag{4.8}$$

and to retrieve the area

$$A = \|\tilde{\boldsymbol{n}}\| \tag{4.9}$$

4.3. Ordered monomials

To allow the execution of the VEM program with a virtually arbitrary input order, the 1-to-1 mapping between natural numbers and scaled monomials presented in Section 2.1 has to be established. Specifically, the ordering presented by *Pascal's triangle* of Figure 2.2 and *Pascal's pyramid* of Figure 2.3 and Figure 2.4 will be followed respectively for scaled monomials embedded in $\mathbb{R}^2$ and $\mathbb{R}^3$.

4.3.1. Ordered monomials embedded in $\mathbb{R}^2$

Including every row of Pascal's triangle of Figure 2.2 up to order k, a complete polynomial of order k is described. The 1-to-1 correspondence described in (2.3) can be achieved with the following double `for` loop

Algorithm 4.1 Construction of ordered monomials in $\mathbb{R}^2$ up to degree k

```
1: push back monomial $x^0 y^0 (= 1)$;
2: for kk=1, 2, . . . , k do
3:     for i=kk, kk-1, . . . , 1 do
4:         j=kk-i;
5:         push back monomial $x^i y^j$;
6:     end for
7:     push back monomial $x^0 y^{kk} (= y^{kk})$;
8: end for
```

where `monomials_ordered` is a vector collecting the monomials, represented by the indices of the exponents and the coefficient (set to `1.0` in this case). In the program, this is required to be computed only once and stored in a cache-like structure for easy access. The structure has been optimized so that if higher degree monomials are required and the cache already contains some lower degree monomials, only the remaining ones are added.

4.3.2. Ordered monomials embedded in $\mathbb{R}^3$

Including every layer of Pascal's pyramid of Figure 2.3 up to layer k, a complete polynomial of order k is described. Analogously as for the 2D case, the 1-to-1 correspondence described in (2.6) can be achieved with this triple `for` loop

Algorithm 4.2 Construction of ordered monomials in $\mathbb{R}^3$ up to degree k

```
1: push back monomial $x^0 y^0 z^0 (= 1)$;
2: for kk=1, 2, ..., k do
3:     for r=kk, kk+1, ..., floor(kk/3)+1 do
4:         for i=kk-2r, kk-2r-1, ..., r+1 do
5:             j=kk-r-i;
6:             push back monomial $x^i y^j z^r$;
7:         end for
8:         for i=kk-2r, kk-2r-1, ..., r+1 do
9:             j=kk-r-i;
10:            push back monomial $x^r y^i z^j$;
11:        end for
12:        for i=kk-2r, kk-2r-1, ..., r+1 do
13:            j=kk-r-i;
14:            push back monomial $x^j y^r z^i$;
15:        end for
16:    end for
17:    if mod(kk/3)=0 then
18:        push back monomial $x^{\frac{kk}{3}} y^{\frac{kk}{3}} z^{\frac{kk}{3}}$;
19:    end if
20: end for
```

The same remarks apply as described in the previous subsection for the 2D case.

4.4. Integration of monomials over polytopic domains

Since the method allows to deal with very general, possibly non-convex, polygons and polyhedra, and a substantial part of the implementation is devoted to integration. Given the great amount of integrals of polynomials which needs to be computed, both for polygons concerning the projections over the faces described in Section 2.3 and for polyhedrons, needed to assembly both the consistent and stabilizing parts of the stiffness matrix and the right hand side projection, a *quadrature-free*, efficient technique presented in [2] has been implemented. The algorithm is a recursive call with a-priory exponential-time complexity with respect to the order of the monomial integrand, depending also on the number of lower order N-polytopes (faces, edges and points for polyhedra, edges and points for polygons). However, a clever choice of the local reference frame for each N-polytope, as described in [2], allows to drastically reduce computation times. A scheme of the pseudo-algorithm is reported below, while for all the details we remind to the original paper.

Algorithm 4.3 Integration of a monomial over a polytopic domain

```
$\mathcal{I}(N, \mathcal{E}, k_1, \dots, k_d) = \int_{\mathcal{E}} x_1^{k_1} \dots x_d^{k_d} \, d\sigma_N(x_1, \dots, x_d)$
1: if $N = 0$   $\big(\mathcal{E} = (v_1, \dots, v_d) \in \mathbb{R}^d$ is a point$\big)$ then
2:     return $\mathcal{I}(N, \mathcal{E}, k_1, \dots, k_d) = v_1^{k_1} \dots v_d^{k_d}$
3: else if $1 \leq N \leq d-1$   ($\mathcal{E}$ is a point if $d = 1$ or an edge if $d = 2$ or a face if $d = 3$)
   then
4:     return $\mathcal{I}(N, \mathcal{E}, k_1, \dots, k_d) = \frac{1}{N+\sum_{n=1}^{d} k_n} \left(\sum_{i=1}^{m} d_i \mathcal{I}(N-1, \mathcal{E}_i, k_1, \dots, k_d) + \right.$
           $+ x_{0,1} k_1 \mathcal{I}(N, \mathcal{E}, k_1 - 1, \dots, k_d) +$
           $+ \dots +$
           $\left. + x_{0,d} k_d \mathcal{I}(N, \mathcal{E}, k_1, \dots, k_d - 1)\right)$
5: else if $N = d$   ($\mathcal{E}$ is an interval if $d = 1$ or a polygon if $d = 2$ or a polyhedron if $d = 3$)
   then
6:     return $\mathcal{I}(N, \mathcal{E}, k_1, \dots, k_d) = \frac{1}{N+\sum_{n=1}^{d} k_n} \left(\sum_{i=1}^{m} b_i \mathcal{I}(N-1, \mathcal{E}_i, k_1, \dots, k_d)\right)$
7: end if
```

where $\boldsymbol{x}_0 = (x_{0,1}, \dots, x_{0,d})$ is the origin of the reference system for the N-polytope $\mathcal{E}$ called as input parameter by the algorithm, and d_i is the algebraic distance between the sub (N-1)-polytope $\mathcal{E}_i$ and $\boldsymbol{x}_0$. The choice of $\boldsymbol{x}_0$ is critical as if selected such that one or more of its coordinates are 0, then the recursion calls for those components are not needed. In the implemented version this is extensively exploited, considering also the degree of the variables making up the monomial being integrated: the coordinate of $\boldsymbol{x}_0$ being selected to 0 is always looked for when the corresponding exponent of the monomial is the highest. The situations in which faces or edges are parallel to the global axes are also taken into account, through a proper tolerance: in these cases there is no arbitrarity in the choice of $\boldsymbol{x}_0$.

Given the great amount of call to the face projections integrals, these are stored in a `class`, functioning as a cache, for fast access.

4.4.1. Centroid of polygons

While the area of polygons is computed with Newell's algorithm, exploiting the technique to retrive the outward normal vector, the computation of their centroids $\boldsymbol{x}_{\boldsymbol{F}}$ relies on the method presented above, simply integrating the first order monomial x, y and z and dividing by the area.

$$\boldsymbol{x_F} = \begin{Bmatrix} x_F \\ y_F \\ z_F \end{Bmatrix} = \frac{1}{A_F} \begin{Bmatrix} \int_F x\, d\Sigma \\ \int_F y\, d\Sigma \\ \int_F z\, d\Sigma \end{Bmatrix} \tag{4.10}$$

4.4.2. Volume and centroid of polyhedra

To compute the volume of polyhedra, an integration of the monomial 1 is performed

$$V_P = \int_P d\Omega \tag{4.11}$$

and similarily, to compute their centroids $\boldsymbol{x_P}$

$$\boldsymbol{x_P} = \begin{Bmatrix} x_P \\ y_P \\ z_P \end{Bmatrix} = \frac{1}{V_P} \begin{Bmatrix} \int_P x\, d\Omega \\ \int_P y\, d\Omega \\ \int_P z\, d\Omega \end{Bmatrix} \tag{4.12}$$

The algorithm presented in 4.3 has also been adapted to accept parameters describing the change of reference system described in (2.10), so to support integration of scaled monomials.

4.5. Integration of a generic function over polyhedra

To construct the equivalent nodal forces vector as described in Section 3.3, in particular to find the components of (3.27) and the right hand side vector of (3.32), an integration rule over polyhedra is required. The implementation adopted in the following work relies on *subtetrahedralization* of the polyhedron, obtained by *subtriangulation* of each face and joining with the centroid of the polyhedron. While this technique has some limitations, as it cannot work, in general, with all polyhedra, it is guaranteed to produce correct results for any *star-shaped* polyhedron with respect to its centroid, having convex faces. In each tetrahedron generated a Gauss-type quadrature is then performed. For efficiency reasons, Gauss points and weights are saved in a `constexpr struct GaussData` for fast access, and transformed with a change of variable for each tetrahedron. The implementation exploits the library `Eigen` for matrix and vectors manipulations and is inspired by the MATLAB function [28].

5 | Implementation

In the following, the implementation aspects for the numerical solution of the linear elastic problem given in Chapter 1, according to the virtual element method presented in Chapter 3 is given. The program is written in C++, exploiting to their best the features of an object-oriented language. `Makefiles` have been written to automate the compilation process, and allow to test and see the different components of the code by simply appending to `make` the corresponding `main` file with the features being tested (e.g. by calling in bash `make main_monomial` a main file specifically built to show the feature of the code contained in `monomial.hpp` is compiled). Moreover the source code has been written so that a full documentation can be automatically generated with Doxygen.

5.1. General outlook of the program

The program runs exploiting 19 base classes, 9 derived ones, and 1 struct. Modern features of *template programming* have been extensively used, especially for the `geometry` namespace, where the geometrical entites and their methods have been declared and defined in the header files, so that substitution of the specific types is performed at compile time. The memory layout has been implemented so to avoid unnecessary copies through the use of `references` where object ownership was logically not needed and `pointers`, when necessary, have been implemented as smart pointers so that their lifetime is automatically managed and memory leaks are avoided. The code can be divided into 7 main components, called by the main class `Problem` which solves the linear elastostatic problem defined in Section 1.

```
    class Problem
{
public:
    // Constructor
    Problem(const char *parametersFilename = "parameters.dat", bool
   printError = false);
};
```

Listing 5.1: problem.hpp

- **Parameters**
  This part defines the parameters of the problem that can be read from a file thanks to `GetPot`, so compilation is not required when some of the data is changed, including the mesh file.

- **Geometrical entities**
  This part deals with the geometry of the problem and the mesh, defining the objects `Point`, `Edge`, `Polygon`, `Polyhedron` and `Mesh` in their respective classes and is entirely implemented following template programming paradigms. An ad-hoc method to interact with Gmsh [14] is also implemented.

- **Monomials and polynomials**
  Being the monomials one of the major cores of the virtual element method, a suite for symbolic calculus for handling generic monomials and polynomials has been defined, together with their derived classes to deal with the specialized versions embedded in $\mathbb{R}^2$ and $\mathbb{R}^3$.

- **Integration**
  This part contains all the integration techniques adopted in the code, ranging from Gauss-Lobatto quadrature, quadrature-free integration of monomials over polytopic domains and Gauss quadrature exploiting subtetrahedralization over polyhedra, all grouped in their respective `namespaces` to avoid conflicts. Moreover, `structs` and `classes` functioning as cache are defined to ease the access of quadrature nodes and points and integrals, where necessary.

- **Virtual space**
  This part is the core of the virtual element program as it contains the definition of the virtual degrees of freedom and all the polynomial projections required. The first subgroup strongly relies on *polymorphism*, so that specialization of the base class `VirtualDof` into the derived classes `VertexDof`, `EdgeDof`, `FaceDof` and `PolyhedronDof` is achieved. Shared pointers are extensively used to allow access from the local degrees of freedom to the global ones, where all the information is stored. The second subgroup computes the face projections, the polyhedron projections and the local matrices needed prior the assembly procedure, strongly relying on the library `Eigen`.

- **Solver**
  The base class `Solver` is designed to be as general as possible, so to allow the implementation of other problems, with also different boundary conditions than Dirichlet. The derived class `SolverVEM` realizes the assembly of the global linear

system and the enforcing of the Dirichlet boundary conditions.

- **Visualization of the results**
  This part is devoted to exporting the results in a `.vtk` format, which can be easily opened with Paraview and manipulated to visualize displacements, strains and stresses in both the undeformed and deformed domain, as well as perform cuts and slices to see the interior.

5.2. Geometrical entities

The declarations and definitions of this part are entirely written in header files, according to *template programming* paradigms, here reported in brief for all the 5 classes `Point`, `Edge`, `Polygon`, `Polyhedron` and `Mesh`. For each class, a main file called `main_[class].cpp` has been written to show the feature of the corresponding class. Simply calling from `bash`, e.g.

```
make main_point
```

will compile `main_point.cpp` and show some feature of the class `Point`.

5.2.1. Point

This `class` handles the points of the mesh. It has been written to deal with very general N-dimensional points and defines an abundant number methods to perform operations between points. It also has a policy to manage auto numbering of the ids, so that when a point is deleted, another point can be instantiated with the freed id. Additionally a method to express a point embedded in R^3 in a 2D coordinate system is given, which will be useful to deal with scaled monomials for the faces.

```
template <typename... Args>
class Point
{
private:
    static IndexType lastId; // current available id
    static std::set<IndexType> freeIds;
    IndexType id;
    std::array<real, sizeof...(Args)> coordinates;

public:
    // Default constructor to initialize coordinates to 0
    Point() : coordinates{0};

```

```
        // Construct a new Point object
        Point(Args... args) : coordinates{static_cast<real>(args)...};

        // Destructor to free the id when the point goes out of scope
        ~Point();

        constexpr std::size_t getDimension() const;

        // Set id
        void setId(IndexType _id);

        // Get id
        const IndexType &getId() const;

        // Get coordinates
        const std::array<real, sizeof...(Args)>
        getCoordinates() const;

        // Access operator
        const real &operator[](std::size_t index) const;

        // Overloaded * operator to compute the scalar multiplication of a point (point*scalar)
        auto operator*(const real &scalar) const;

        // Overloaded * operator to support scalar * point multiplication
        friend auto operator*(const real &scalar, const Point &point);

        // scalar multiplication of a point
        template <size_t... Indices>
        auto multiplyByScalar(real scalar, std::index_sequence<Indices...>) const;

        // Overloaded / operator to compute the scalar division of a point (point/scalar)
        auto operator/(const real &scalar) const;

        // Overloaded + operator to compute the sum of two points
        template <typename... OtherArgs>
        auto operator+(const Point<OtherArgs...> &other) const;

        // Overloaded - operator to compute the difference of two points
        template <typename... OtherArgs>
```

```cpp
    auto operator-(const Point<OtherArgs...> &other) const;

    // Piecewise multiplication of two points
    template <typename... OtherArgs>
    auto piecewiseMultiply(const Point<OtherArgs...> &other) const;

    // Binary operation in class
    template <typename... OtherArgs, size_t... Indices, typename Operation>
    auto binaryOperation(const Point<OtherArgs...> &other, Operation operation, std::index_sequence<Indices...>) const;

    // Dot product in the form point1.dot(point2)
    template <typename... OtherArgs>
    auto dot(const Point<OtherArgs...> &other) const;

    // In-class cross product calculation (3D points only) in the form point1.cross(point2)
    template <typename... OtherArgs>
    auto cross(const Point<OtherArgs...> &other) const;

    // Euclidean distance
    template <typename... OtherArgs>
    auto distance(const Point<OtherArgs...> &other) const;

    // Norm
    auto norm() const;

    // Normalize coordinates
    auto normalize() const;

    // Define the comparison function based on edge Ids
    bool operator<(const Point<Args...> &other) const;

    // Custom definition of operator== for Point
    template <typename... OtherArgs>
    bool operator==(const Point<OtherArgs...> &other) const;

    // Output stream operator to stream coordinates
    friend std::ostream &operator<<(std::ostream &os, const Point<Args...> &point);
};
```

Listing 5.2: Point.hpp

5.2.2. Edge

This class handles the edges of the mesh. The implementation considers the edges as *half-edges*, meaning they have a direction given by the ordering of the points, stored in a reference wrapper. This is useful to construct polygons as an ordered sequence of edges.

```cpp
template <typename PointType>
class Edge
{
private:
    static IndexType lastId; // current available id
    IndexType id;
    bool flipped; // if edge must be read the other way around
    std::array<std::reference_wrapper<const PointType>, 2> points;
    real length;
    PointType direction; // unit vector of the direction of the edge
    std::shared_ptr<Edge<PointType>> otherHalfEdge; // Pointer to store the other half-edge

public:
    // Constructor with two points
    Edge(const PointType &point1,
         const PointType &point2,
         bool _flipped = false) : id(lastId++),
                                  flipped(_flipped),
                                  points({point1, point2});

    // Setter method to set the other half-edge
    void setOtherHalfEdge(const Edge<PointType> &otherEdge);

    // Getter method to get the other half-edge
    Edge<PointType> &getOtherHalfEdge() const;

    // Update the properties of the edge.
    // When modifying one point, the edge autmatically sees the change as they are references
    // It is not the case for other data structures
    void update();

    // Set id
    void setId(IndexType _id);

    // Get id
    const IndexType &getId() const;

```

```
        // Get length
        const real &getLength() const;

        // Get direction
        const PointType getDirection() const;

        // Constant getter
        const PointType &operator[](IndexType index) const;

        // Define the comparison function based on edge Ids
        bool operator<(const Edge<PointType> &other) const;

        // Equality operator for edges
        bool operator==(const Edge<PointType> &other) const;

        // Stream output operator for the Edge class
        friend std::ostream &operator<<(std::ostream &os, const Edge<
    PointType> &edge);
    };
```

Listing 5.3: Edge.hpp

5.2.3. Polygon

This class handles polygons as an ordered sequence of edges. It provides methods to compute the diameter h_F and the outward unit normal to the polygon $\tilde{\boldsymbol{n}}$ according to (4.7), serving to compute the area as well. Moreover, methods to validate polygons are also implemented, such as checking the ordering of the vertices given by the edges and comparison based on the sequence of edges and polygons' ids.

```
template <typename EdgeType>
    class Polygon
    {
    private:
        static IndexType lastId;
        IndexType id;
        bool orientation; // true if the edges ordering are consistent
    with the outward normal
        std::vector<std::reference_wrapper<const EdgeType>> edges;
        std::shared_ptr<Polygon<EdgeType>> otherPolygon; // Pointer to
    store the other polygon
        Point3D outwardNormalArea;
        real diameter = 0.0;

    public:
        // Constructor to initialize the polygon, prior edge insertion
        Polygon() : Polygon({});

        // Constructor taking individual edges
        Polygon(const std::initializer_list<EdgeType> &edges_,
                bool _orientation = false) : id(lastId++),
                                             orientation(_orientation);

        // Setter method to set the other polygon
        void setOtherPolygon(const Polygon<EdgeType> &otherPolygon_);

        // Getter method to get the other polygon
        Polygon<EdgeType> &getOtherPolygon() const;

        // Set orientation
        void setOrientation(bool _orientation);

        // Add an edge and its direction to the polygon
        void addEdge(const EdgeType &edge);

        // Set id
        void setId(IndexType _id);

        // Get id
        const IndexType &getId() const;

        // Get the number of edges in the polygon
        std::size_t numEdges() const;

        // Get an edge by index
        const EdgeType &getEdge(std::size_t index) const;

        // Access edges through []
        const EdgeType &operator[](IndexType index) const;

        // Get original Edge
        const EdgeType &getPositiveEdge(std::size_t index) const;

        // Check if the edges are stored consistently
        bool areEdgesConsistent() const;

        // Compute properties
        void computeProperties();

        // Compute outward normal unit vector
        void computeOutwardNormalArea();

        // Get outward normal unit vector
        const Point3D getOutwardNormal() const;

        // Get first local axis e_x
        const Point3D get_e_x() const;

        // Get second local axis e_y
        const Point3D get_e_y() const;

        // Get area
        real getArea() const;

        // Compute diameter
        void computeDiameter();

        // Get diameter
        real getDiameter() const;

        // Define the comparison function based on polygon Ids
        bool operator<(const Polygon<EdgeType> &other) const;

        // Comparison operator for polygons (==)
        bool operator==(const Polygon<EdgeType> &other) const;

        // Stream output operator for the Polygon class
        friend std::ostream &operator<<(std::ostream &os, const Polygon<
   EdgeType> &polygon);
    };
```

Listing 5.4: Polygon.hpp

5.2.4. Polyhedron

This class handles polyhedra as a collection of polygons. In this case the order in which polygons are stored does not matter, this notwithstanding the faces are saved in a `std::vector` for efficiency reasons.

```
        template <typename PolygonType>
    class Polyhedron
    {
    private:
```

```cpp
        static std::size_t lastId;
        std::size_t id;
        std::vector<std::reference_wrapper<const PolygonType>> polygons;
        real diameter = 0.0;

    public:
        // Constructor taking individual polygons
        Polyhedron(const std::initializer_list<PolygonType> &
    polygonsWithoutDirection) : id(lastId++);

        // Add a polygon and its direction to the polyhedron
        void addPolygon(const PolygonType &polygon);

        // Set id
        void setId(std::size_t _id);

        // Get id
        const std::size_t &getId() const;

        // Method to get the number of polygons in the Polyhedron
        std::size_t numPolygons() const;

        // Method to get a polygon by index
        const PolygonType &getPolygon(std::size_t index) const;

        // Access operator [] to get a polygon by index
        const PolygonType &operator[](std::size_t index) const;

        // Compute diameter
        void computeDiameter();

        // Get diameter
        real getDiameter() const;

        // Stream output operator for the Polyhedron class
        friend std::ostream &operator<<(std::ostream &os, const
    Polyhedron<PolygonType> &polyhedron);
    };
```

Listing 5.5: Polyhedron.hpp

5.2.5. Mesh

The class `Mesh` allows to read the geometrical entities defined above from a `.geo` file, such as those handled by Gmsh [14], and retrieve them through proper getters.

```cpp
template <typename PointType,
          typename EdgeType,
          typename PolygonType,
          typename PolyhedronType>
class Mesh
{
private:
    std::map<std::size_t, PointType> points;
    std::map<IndexType, EdgeType> edges;
    std::map<IndexType, PolygonType> polygons;
    std::map<std::size_t, std::size_t> PlaneSurfaceMap;
    std::map<std::size_t, PolyhedronType> polyhedra;
    real h; // size of the mesh

public:
    // Default constructor
    Mesh() = default;

    // Constructor reading entities from Gmsh .geo file
    Mesh(const std::string &filename);

    // Method to get the number of vertices
    std::size_t numVertices() const;

    // Method to get the number of edges
    std::size_t numEdges() const;

    // Method to get the number of polygons
    std::size_t numPolygons() const;

    // Method to get the number of polyhedra
    std::size_t numPolyhedra() const;

    // Getter for a vertex
    const PointType &getVertex(std::size_t index) const;

    // Getter for an edge
    const EdgeType &getEdge(IndexType index) const;

    // Getter for a polygon
    const PolygonType &getPolygon(IndexType index) const;

    // Getter for a polyhedron
    const PolyhedronType &getPolyhedron(std::size_t index) const;

    // Getter for the map of vertices
    const std::map<std::size_t, PointType> &getVertices() const;

    // Getter for the map of edges
    const std::map<IndexType, EdgeType> getEdges() const;

    // Getter for the map of polygons
    const std::map<IndexType, PolygonType> getPolygons() const;

    // Getter for the map of polyhedra
    const std::map<std::size_t, PolyhedronType> &getPolyhedra() const;

    // Getter for the average diameter
    const real &getSize() const;

    // Print mesh information
    void print() const;
};
```

Listing 5.6: Mesh.hpp

5.3. Monomials and polynomials

This feature supports monomial and polynomial symbolic calculus, thanks to the templated base class `Monomial`, the two derived specializations `Monomial2D` and `Monomial3D`, the templated class `Polynomial` and its derived class `LinearTrinomialPower`. The first one allows to instantiate a monomial through the integer exponents of its variables and the real coefficient. Methods to compute the derivative of a monomial with respect to one of its variables, the product of two monomials, and to allow the evaluation of a monomial in a point are provided and extensively used throughout the code. The derived classes `Monomial2D` and `Monomial3D` implement methods to ease the call of the derivative method with, e.g.,

```
monomial.dx();
```

instead of the general

```
monomial.derivative(0);
```

However, the main purpose of the two derived classes is to define and store the static vectors ordering the monomials up to a given degree and their laplacians for the 2D version, and the monomials up to a given degree and their gradients for the 3D version. This provides an efficient way of accessing the correct monomial when populating the matrices. The class polynomial has been written to better handle projections, as in higher order VEM these are no longer constants and can become more complex. Moreover, the specialized class `LinearTrinomialPower` handles a polynomial generated by the n^{th} power of a trinomial of the kind $ax + by + c$, useful when one wants to pass from the local polyhedron reference frame, to the face reference frame.

```cpp
template <unsigned int Dimension>
class Monomial
{
private:
    std::vector<unsigned int> exponents;
    real coefficient;

public:
    // Default constructor
    Monomial() : exponents(Dimension, 0), coefficient(0.0) {}

    // Constructor
    Monomial(std::vector<unsigned int> exponents, real coefficient) :
   exponents(std::move(exponents)), coefficient(coefficient) {}

    // Getter for the exponents
    const std::vector<unsigned int> &getExponents() const;

    // Getter for the coefficient
    real getCoefficient() const;

    // Setter for the coefficient
    void setCoefficient(real coeff);

    // Compute the product of two monomials
    Monomial<Dimension> operator*(const Monomial<Dimension> &other)
   const;

    // Compute the derivative with respect to a variable
    Monomial<Dimension> derivative(unsigned int variableIndex) const;

    // Evaluate the monomial at a point
    template <typename PointType>
    real evaluate(const PointType &point) const;

    // Get the monomial order
    unsigned int getOrder() const;

    // Output stream operator
    friend std::ostream &operator<<(std::ostream &os, const Monomial &
   monomial);
};

class Monomial2D : public Monomial<2>
{
private:
    static unsigned int order;
    static std::vector<Monomial2D> monomials_ordered;
    static std::vector<std::pair<std::pair<real, std::size_t>,
                                 std::pair<real, std::size_t>>>
        laplacians_to_monomials_ordered;

public:
    // Default constructor
    Monomial2D() : Monomial<2>() {}

    // Constructor with exponents and coefficient
    Monomial2D(unsigned int expX, unsigned int expY, double coeff) :
   Monomial<2>({expX, expY}, coeff) {}

    // Constructor for implicit conversion from Monomial<2> to
   Monomial2D
    Monomial2D(const Monomial<2> &monomial)
        : Monomial<2>(monomial.getExponents(), monomial.getCoefficient()
   ) {}

    // Method to compute the monomials up to a given degree
    static void computeMonomialsUpToOrder(unsigned int order_);

    // Method to get the monomials up to a given degree
    static const std::vector<Monomial2D> getMonomialsOrdered(unsigned
   int order_);

    // Method to compute the laplacians up to a given degree
    static void computeLaplaciansToMonomialsOrdered(unsigned int order_)
   ;

```

```
    // Method to get the laplacians up to a given degree
    static const std::vector<std::pair<std::pair<real, std::size_t>,
                                       std::pair<real, std::size_t>>>
    getLaplaciansToMonomialsOrdered(unsigned int order_);

    // Method to compute the product of two monomials and return a new
   Monomial2D instance
    Monomial2D operator*(const Monomial2D &other) const;

    // Method to compute the derivative with respect to a variable and
   return a new Monomial2D instance
    Monomial2D derivative(unsigned int variableIndex) const;

    // Compute the derivative with respect to x
    Monomial2D dx() const;

    // Compute the derivative with respect to y
    Monomial2D dy() const;

    // Overriding the output stream operator to print "x, y" for
   Monomial2D
    friend std::ostream &operator<<(std::ostream &os, const Monomial2D &
   monomial);
};

class Monomial3D : public Monomial<3>
{
private:
    static unsigned int order;
    static std::vector<Monomial3D> monomials_ordered;
    static std::vector<std::array<std::pair<real, std::size_t>, 3>>
        gradients_to_monomials_ordered;
    static std::vector<std::array<std::pair<real, std::size_t>, 3>>
        laplacians_to_monomials_ordered;

public:
    // Default constructor
    Monomial3D() : Monomial<3>() {}

    // Constructor with exponents and coefficient
    Monomial3D(unsigned int expX, unsigned int expY, unsigned int expZ,
   double coeff) : Monomial<3>({expX, expY, expZ}, coeff) {}

    // Constructor for implicit conversion from Monomial<3> to
```

```
Monomial3D
    Monomial3D(const Monomial<3> &monomial)
        : Monomial<3>(monomial.getExponents(), monomial.getCoefficient()) {}

    // Method to compute the monomials up to a given degree
    static void computeMonomialsUpToOrder(unsigned int order_);

    // Method to get the monomials up to a given degree
    static const std::vector<Monomial3D>
    getMonomialsOrdered(unsigned int order_);

    // Method to compute the gradients up to a given degree
    static void computeGradientsToMonomialsOrdered(unsigned int order_);

    // Method to get the gradients up to a given degree
    static const std::vector<std::array<std::pair<real, std::size_t>, 3>>
    getGradientsToMonomialsOrdered(unsigned int order_);

    // Method to compute the laplacians up to a given degree
    static void computeLaplaciansToMonomialsOrdered(unsigned int order_);

    // Method to get the laplacians up to a given degree
    static const std::vector<std::array<std::pair<real, std::size_t>, 3>>
    getLaplaciansToMonomialsOrdered(unsigned int order_);

    // Method to compute the product of two monomials and return a new Monomial2D instance
    Monomial3D operator*(const Monomial3D &other) const;

    // Method to compute the derivative with respect to a variable and return a new Monomial2D instance
    Monomial3D derivative(unsigned int variableIndex) const;

    // Compute the derivative with respect to x
    Monomial3D dx() const;

    // Compute the derivative with respect to y
    Monomial3D dy() const;

    // Compute the derivative with respect to z
    Monomial3D dz() const;

    // Overriding the output stream operator to print "x, y, z" for Monomial3D
    friend std::ostream &operator<<(std::ostream &os, const Monomial3D &monomial);
};

template <unsigned int Dimension>
class Polynomial
{
protected:
    // Map of monomials
    std::map<std::vector<unsigned int>, Monomial<Dimension>> polynomial;
    unsigned int order = 0;

public:
    // Default constructor
    Polynomial() = default;

    // Constructor
    Polynomial(const std::vector<Monomial<Dimension>> &monomials);

    // Method to add a monomial to the polynomial
    void addMonomial(const Monomial<Dimension> &monomial);

    // Get the polynomial order
    unsigned int getOrder() const;

    // Overload * operator to compute the product of two polynomials
    Polynomial<Dimension> operator*(const Polynomial<Dimension> &other) const;

    // Overload * operator to compute the product of a polynomial and a monomial
    Polynomial<Dimension> operator*(const Monomial<Dimension> &other) const;

    // Getter for the number of monomials making up the polynomial
    size_t size() const;

    // Getter
    const std::map<std::vector<unsigned int>, Monomial<Dimension>> &getPolynomial() const;
```

```
};

class LinearTrinomialPower : public Polynomial<2>
{
public:
    // Constructor
    LinearTrinomialPower(const real &a, const real &b, const real &c,
    const unsigned int &power);
};
```

Listing 5.7: monomial.hpp

5.4. Integration

This part contains all the numerical techniques to perform integration, grouped in three `namespaces`: `GaussLobatto`, `IntegrationMonomial` and `Gauss`. The first one computes the points and weight over the standard interval $[-1, 1]$ of the Gauss-Lobatto quadrature rule, according to the procedure presented in Section 4.1, as well as the methods to map them through (4.6) and (4.5) for the general interval. It exploits the solver `toms748_solve` from the `boost` library to find the zeros of the derivatives of Legendre polynomials in (4.2). The namespace `IntegrationMonomial` contains the templated function `integrateMonomial`, which allows to integrate a `Monomial2D` or `Monomial3D` over a 2D or 3D domain. It also supports integration of a `Monomial3D` over a polygon embedded in $\mathbb{R}^3$ and accepts parameters to perform change of coordinates. The method exploits the structure of the geometrical entities implemented and the flexibility of the class `Monomial`. The namespace `Gauss` contains the points and weights for the N^3-point Gauss quadrature over the standard tetrahedron whose vertices coordinates are

$$\begin{aligned}
\boldsymbol{V_1} &= (0, 0, 0);\\
\boldsymbol{V_2} &= (1, 0, 0);\\
\boldsymbol{V_3} &= (0, 1, 0);\\
\boldsymbol{V_4} &= (0, 0, 1);
\end{aligned}$$

and allows for exact integration of monomials up to degree $2N - 1$. Currently it stores integration points and weights up to $N = 6$, therefore allowing for exact integration of monomials up to degree 11. The purpose of this method is however to allow for integration of general functions, as it is used to construct the equivalent nodal forces vector of (3.26). Methods to map the points and weights from the standard tetrahedron to a general tetrahedron are implemented, as well as integrating over polyhedral domains, through *subtetrahedralization*. In this latter case, it is recalled that the method works only for star-shaped polyhedra with respect to their centroids, having all convex faces.

```
namespace GaussLobatto
{
    class GaussLobattoCache
    {
    private:
        // Stores the abscissas and weights for n-points Gauss-Lobatto
   rule in [-1,1]
        static std::map<unsigned int,
                        std::pair<std::vector<real>, std::vector<real>>>
            XW;

    public:
        // Initialize the cache
        static void initialize(unsigned int n);

        // Get the cache
        static const std::pair<std::vector<real>,
                               std::vector<real>> &
        getCache(unsigned int n);
    };

    // Function to compute the derivative of the Legendre polynomial of
   order n
    real legendreDerivative(unsigned int n,
                            real x);

    // Returns the floor((n-4)/2+1) abscissas of n-points Gauss Lobatto
   inside (0,1) for the standard interval [-1,1]
    std::vector<real> computeOneSideGaussLobattoInternalX(unsigned int n
   );

    // Returns the n abscissas and weights of n-points Gauss Lobatto for
    the standard interval [-1,1]
    std::pair<std::vector<real>,
              std::vector<real>>
    computeGaussLobattoXW(unsigned int n);

    // Method to compute n-points Gauss-Lobatto points and weights on an
    edge
    std::pair<std::vector<Point3D>,
              std::vector<real>>
    computeGaussLobattoPointsWOnEdge(const Edge3D &edge,
                                     unsigned int n);
}

namespace IntegrationMonomial
{
    class MonomialsFaceIntegralsCache
    {
    private:
        // Stores the integrals of the monomials for the faces
        static std::map<std::size_t, std::vector<real>>
   monomials_face_integrals;

    public:
        // Initialize the cache for the whole faces in the mesh
        static void initialize(const Mesh<Point3D, Edge3D, Polygon3D,
   Polyhedron<Polygon3D>> &mesh, unsigned int order);

        // Initialize the cache for a face
        static void initialize(const Polygon3D &F, unsigned int order);

        // Get the integrals of monomials up to a given order over a
   face
        static std::vector<real> &getCacheMonomials(const Polygon3D &F,
                                                    unsigned int order);

        // Get a the integral of a monomial over a face
        static real &getCacheMonomial(const Polygon3D &F,
                                      const Monomial2D &m);
    };

    // Integrate a (scaled) 2D or 3D monomial respectively over a non-
   scaled 2D or 3D domain.
    // To get the integral of a (scaled) monomial over the scaled domain
    simply multiply the result by h^d.
    // Can also perform integration of a (scaled) monomial in 3D over a
   2D domain.
    template <typename DomainType, unsigned int d>
    real integrateMonomial(const unsigned int &N,
                           const DomainType &E,
                           const Monomial<d> &monomial,
                           const Point3D &O,
                           const real &h,
                           const Point3D &ex,
                           const Point3D &ey);

    // Get a polygon's centroid
    Point3D getPolygonCentroid(const Polygon3D &F);

    // Get a polyhedron's volume
    real getPolyhedronVolume(const Polyhedron<Polygon3D> &P);

    // Get a polyhedron's centroid
    Point3D getPolyhedronCentroid(const Polyhedron<Polygon3D> &P);

    // Integrate the restriction on a plane of a monomial embedded in R3
     times a monomial embedded in R2
    real integrateMonomial3DRestrictedMonomial2D(const Point3D &X_P,
   const real &h_P, const Polygon3D &F, const Monomial3D &m3D, const
   Monomial2D &m2D);

    // Integrate the restriction on a plane of a monomial embedded in R3
     times a polynomial embedded in R2
    real integrateMonomial3DRestrictedPolynomial2D(const Point3D &X_P,
   const real &h_P, const Polygon3D &F, const Monomial3D &m3D, const
   Polynomial<2> &p2D, const Point3D &X_F = Point3D());
}

namespace Gauss
{
    // Computes the Gauss points and weights for a tetrahedron
    std::vector<std::array<real, 4>> gaussPointsTetrahedron(const
   Point3D &v1, const Point3D &v2, const Point3D &v3, const Point3D &v4,
     const unsigned int &N_);

    // Computes the Gauss points and weights for a polyhedron,
   exploiting subtetrahedralization
    std::vector<std::array<real, 4>> gaussPointsPolyhedron(const
   Polyhedron<Polygon3D> &P, const unsigned int &N_);

    // Integrate a generic function over a tetrahedron
    real integrateFunctionOverTetrahedron(const std::function<real(real,
     real, real)> &func,
                                          const Point3D &v1,
                                          const Point3D &v2,
                                          const Point3D &v3,
                                          const Point3D &v4,
                                          unsigned int N_);

    // Integrate a generic function over a polyhedron
    real integrateFunctionOverPolyhedron(const std::function<real(real,
   real, real)> &func,
                                         const Polyhedron<Polygon3D> &P,
                                         const unsigned int &N_);
}
```

Listing 5.8: integration.hpp

5.5. Virtual space

This part is devoted to the creation of the virtual degrees of freedom and the computation of the virtual projections, respectively grouped in the headers `virtualDofs.hpp` and `virtualProjections.hpp` and their corresponding source files.

5.5.1. Virtual degrees of freedom

The first one heavily exploits *polymorphism*, following the natural feature of the VEM of having different types of degrees of freedom, namely vertex-type, edge-type, face-type and polyhedron-type DOFs. While the first two refer to pointwise values, having a precise meaning linked to the location they are defined, it has been preferred to separate them, under two different specialization of the base virtual class `VirtualDof`: `VertexDof` and `EdgeDof`. The first one provides methods to access the vertex the DOF refers to and its id, the second one provides methods to access the Gauss-Lobatto point the DOF refers to, the corresponding weight, and the id of the edge. The last two DOF types refer to internal moments for the faces and polyhedra, and allow to retrieve the corresponding monomials. The degrees of freedom are stored in a specific class `VirtualDofsCollection`, thanks to a `vector` of shared pointers to VirtualDof object. The constructor of this class is responsible of adding the correct DOFS specialization in the collection, given the mesh and the required order of the VEM. To correctly construct the stiffness local matrices, a class `LocalVirtualDof` allows to handle the mapping from global to local DOFS, and vice versa, information needed in the assembly process. Shared pointers are once again used so that storing duplicates in memory is avoided. Lastly, a class `LocalVirtualDofsCollection` groups all the LocalVirtualDof objects, for clean and organized access of them.

```
// Base class for degrees of freedom
class VirtualDof
{
private:
    std::size_t id;
    static std::size_t last_id;

public:
    // Constructor
    VirtualDof() : id(last_id++){};

    // Destructor
    virtual ~VirtualDof() = default;

    // Pure virtual output stream operator
    virtual std::ostream &operator<<(std::ostream &os) const = 0;

    // Output stream operators
    friend std::ostream &operator<<(std::ostream &os, const VertexDof &
   vDof);
    friend std::ostream &operator<<(std::ostream &os, const EdgeDof &
   eDof);
    friend std::ostream &operator<<(std::ostream &os, const FaceDof &
   fDof);
    friend std::ostream &operator<<(std::ostream &os, const
   PolyhedronDof &pDof);

    // Getter for the VirtualDof Id
    virtual std::size_t getId() const;
};

// Derived class for VertexDof
class VertexDof : public VirtualDof
{
private:
    std::size_t vertexId;
    Point3D vertex;

public:
    // Constructor
    VertexDof(std::size_t id, const Point3D &vertex_) : vertexId(id),
   vertex(vertex_) {}

    // Getter fot the id of the vertex
    std::size_t getId() const override;

    // Getter for the vertex
    const Point3D &getVertex() const;
```

```cpp

    // Output stream operator
    std::ostream &operator<<(std::ostream &os) const override;
};

// Derived class for EdgeDof
class EdgeDof : public VirtualDof
{
private:
    std::size_t edgeId;
    Point3D gaussLobattoPoint;
    real weight;

public:
    // Constructor
    EdgeDof(std::size_t id, const Point3D &gaussLobattoPoint_, const
   real &weight_)
        : edgeId(id), gaussLobattoPoint(gaussLobattoPoint_), weight(
   weight_) {}

    // Getter for the id of the Edge DOF
    std::size_t getId() const override;

    // Getter for the Gauss-Lobatto point
    const Point3D &getGaussLobattoPoint() const;

    // Getter for the weight of the Gauss-Lobatto point
    const real &getWeight() const;

    // Output stream operator
    std::ostream &operator<<(std::ostream &os) const override;
};

// Derived class for FaceDof
class FaceDof : public VirtualDof
{
private:
    std::size_t faceId;
    Monomial2D monomial;

public:
    // Constructor
    FaceDof(std::size_t id, const Monomial2D &monomial_) : faceId(id),
   monomial(monomial_) {}
```

```cpp

    // Getter for the id of the face
    std::size_t getId() const override;

    // Getter for the monomial of the Face DOF
    const Monomial2D &getMonomial() const;

    // Output stream operator
    std::ostream &operator<<(std::ostream &os) const override;
};

// Derived class for PolyhedronDof
class PolyhedronDof : public VirtualDof
{
private:
    std::size_t polyhedronId;
    Monomial3D monomial;

public:
    // Constructor
    PolyhedronDof(std::size_t id, const Monomial3D &monomial_) :
   polyhedronId(id), monomial(monomial_) {}

    // Getter for the id of the polyhedron
    std::size_t getId() const override;

    // Getter for the monomial of the polyhedron DOF
    const Monomial3D &getMonomial() const;

    // Output stream operator
    std::ostream &operator<<(std::ostream &os) const override;
};

class VirtualDofsCollection
{
private:
    static unsigned int order;
    std::vector<std::shared_ptr<VirtualDof>> dofs;
    std::size_t numVdofs = 0;
    std::size_t numEdofs = 0;
    std::size_t numFdofs = 0;
    std::size_t numPdofs = 0;

public:
    // Virtual default destructor
    virtual ~VirtualDofsCollection() = default;

    // Constructor to create VirtualDofs from a Mesh and order
    VirtualDofsCollection(const Mesh<Point3D, Edge3D, Polygon3D,
   Polyhedron<Polygon3D>> &mesh, unsigned int order_);

    // Get the order of the virtual dofs collection
    static unsigned int getOrder();

    // Get the number of vertex-type dofs
    std::size_t getnumVdofs() const;

    // Get the number of edge-type dofs
    std::size_t getnumEdofs() const;

    // Get the number of face-type dofs
    std::size_t getnumFdofs() const;

    // Get the number of polyhedron-type dofs
    std::size_t getnumPdofs() const;

    // Get the total number of dofs
    std::size_t getnumDofs() const;

    // Method to get the corresponding specialized dof to a given id
    template <typename DofType>
    std::shared_ptr<DofType> getDof(std::size_t id) const;

    // Method to get the corresponding dof to a given id
    std::shared_ptr<VirtualDof> getDof(std::size_t id) const;

    // Output stream operator for VirtualDofsCollection
    friend std::ostream &operator<<(std::ostream &os, const
   VirtualDofsCollection &dofsCollection);

    // Print VirtualDofsCollection information
    void print() const;
};

class LocalVirtualDofs
{
private:
    std::vector<std::shared_ptr<VirtualDof>> dofs; // map local dofs to global DOFS
    std::map<std::size_t, std::size_t> V_map;
    std::map<std::size_t, std::vector<std::size_t>> E_map;
    std::map<std::size_t, std::vector<std::size_t>> F_map;
    std::vector<std::size_t> P_vector;

public:
    // Constructor
    LocalVirtualDofs(const Polyhedron<Polygon3D> &P, const VirtualDofsCollection &DOFS);

    // Method to get the global id of the corresponding local dof id
    std::size_t getID(std::size_t id) const;

    // Method to get the corresponding specialized dof to a given id
    template <typename DofType>
    std::shared_ptr<DofType> getDof(std::size_t id) const;

    // Get the corresponding local dof for vertex-type global dof
    std::size_t VToLocalId(const std::size_t &ID) const { return V_map.at(ID); }

    // Get the corresponding local dof for edge-type global dof
    std::vector<std::size_t> EToLocalId(const std::size_t &ID) const { return E_map.at(ID); }

    // Get the corresponding local dof for face-type global dof
    std::vector<std::size_t> FToLocalId(const std::size_t &ID) const { return F_map.at(ID); }

    // Get the corresponding local dof for polyhedron-type global dof
    std::size_t PToLocalId(const std::size_t &ID) const { return P_vector[ID]; }

    // Get the number of vertex-type dofs
    std::size_t getnumVdofs() const;

    // Get the number of edge-type dofs
    std::size_t getnumEdofs() const;

    // Get the number of face-type dofs
    std::size_t getnumFdofs() const;

    // Get the number of polyhedron-type dofs
    std::size_t getnumPdofs() const;

    // Get the total number of local dofs
    std::size_t getnumDofs() const;

    // Output stream operator for LocalVirtualDofs
    friend std::ostream &operator<<(std::ostream &os, const
   LocalVirtualDofs &dofsCollection);
};

class LocalVirtualDofsCollection
{
private:
    std::vector<LocalVirtualDofs> Pdofs;

public:
    // Constructor
    LocalVirtualDofsCollection(const Mesh<Point3D, Edge3D, Polygon3D,
   Polyhedron<Polygon3D>> &mesh, const VirtualDofsCollection &DOFS);

    // Get the LocalVirtualDofs of the element Id
    const LocalVirtualDofs &getLocalDofs(const std::size_t &Id) const;

    // Get the number of LocalVirtualDofs in the collection
    size_t numLocalDofsCollection() const;
};
```

Listing 5.9: virtualDofs.hpp

5.5.2. Virtual projections

This second part deals with the construction of the projection matrices for the faces $\boldsymbol{G}_F$, defined in Section 2.3, the local matrices $\boldsymbol{G}$, $\boldsymbol{C}$, $\boldsymbol{E}$, $\boldsymbol{T}_{\boldsymbol{D}+\boldsymbol{R}}$ and the right hand sides projections $\hat{\boldsymbol{b}}^h$ described in Chapter 3. It relies on the library `Eigen`, making use of the `SparseMatrix` type and storing only matrices' upper parts whenever these are symmetric. The file `virtualProjection.hpp` and its source code develops two classes: `VirtualFaceProjections` and `VirtualPolyhedronProjections`. The first one computes all the projections onto the space of polynomials for the vertex-type, edge-type and face-type over the faces of the mesh, needed to populate the matrices $\boldsymbol{A}_1$ (3.13). The constructor also accepts an optional `bool` parameter to check if the projections have been computed correctly, by exploiting the identity $\boldsymbol{G}_F = \boldsymbol{B}_F \boldsymbol{D}_F$ [7] and checking the Frobenius norms of the difference between the two computed matrices $\boldsymbol{G}_F$. The second class computes the elemental matrices needed for the stiffness consistent and stabilizing matrix and the projections for the local equivalent nodal forces vector.

```
class VirtualFaceProjections
{
private:
    static constexpr double threshold = 1e-12; // entries for matrix G
    std::map<std::size_t, std::vector<Polynomial<2>>> faceProjections;

public:
    // Constructor
    VirtualFaceProjections(const VirtualDofsCollection &dofs, const Mesh
   <Point3D, Edge3D, Polygon3D, Polyhedron<Polygon3D>> &mesh, const
   unsigned int &order);

    // Compute the projections of the basis functions corresponding to
   the dofs defined on the face
    std::vector<Polynomial<2>>
    computeFaceProjection(const VirtualDofsCollection &dofs, const
   Polygon3D &face, const unsigned int &order, bool checkConsistency =
   false);

    // Get the face projections
    const std::vector<Polynomial<2>> &getFaceProjection(const std::
   size_t &Id) const;
};

class VirtualPolyhedronProjections
{
private:
    unsigned int order;
    real youngs_mod;
    real poisson_ratio;
    static constexpr real threshold = 1e-12;
           // entries for matrix G
    std::map<std::size_t, Eigen::SparseMatrix<real>>
   polyhedronProjections;    // stores matrices C
    std::map<std::size_t, Eigen::SparseMatrix<real>> elastic_matrices;
           // stores matrices E, upper part
    std::map<std::size_t, Eigen::SparseMatrix<real>>
   deformation_RBM_matrices; // stores matrices Tdr
    std::map<std::size_t, Eigen::VectorXd> forcingProjections;
           // stores polynomials b_hat*V_P

```

```
public:
    // Constructor with the materials parameters and the forcing function
    VirtualPolyhedronProjections(const real &E, const real &nu, const VirtualFaceProjections &faceProjections, const LocalVirtualDofsCollection &dofs, const Mesh<Point3D, Edge3D, Polygon3D, Polyhedron<Polygon3D>> &mesh, const std::function<real(real, real, real)> &funcx, const std::function<real(real, real, real)> &funcy, const std::function<real(real, real, real)> &funcz, const unsigned int &order);

    // Constructor taking an instance of parameters
    VirtualPolyhedronProjections(const Parameters &parameters, const VirtualFaceProjections &faceProjections, const LocalVirtualDofsCollection &dofs, const Mesh<Point3D, Edge3D, Polygon3D, Polyhedron<Polygon3D>> &mesh);

    // Compute the projections over the polyhedron
    Eigen::SparseMatrix<real> computePolyhedronProjections(const VirtualFaceProjections &faceProjections, const LocalVirtualDofs &dofs, const Polyhedron<Polygon3D> &polyhedron, const std::function<real(real, real, real)> &funcx, const std::function<real(real, real, real)> &funcy, const std::function<real(real, real, real)> &funcz, const unsigned int &order);

    // Get the projections C
    const std::map<std::size_t, Eigen::SparseMatrix<real>> &getPolyhedronProjections() const;

    // Get the elastic matrices E
    const std::map<std::size_t, Eigen::SparseMatrix<real>> &getElasticMatrices() const;

    // Get the deformation and rigid body motion matrices Tdr
    const std::map<std::size_t, Eigen::SparseMatrix<real>> &getDeformationRBMMatrices() const;

    // Get the forcing projections
    const std::map<std::size_t, Eigen::VectorXd> &getforcingProjections() const;

    // Get the order
    const unsigned int &getOrder() const;
```

```
};
```

Listing 5.10: virtualProjections.hpp

5.6. Solver

This part of the code handles the assembly step of the local matrices into the global system, the enforcement of the boundary conditions and the solution of the linear algebraic system. A base class `Solver` has been written to handle general problems and to allow to interface with different methods, e.g. the finite element method instead of the VEM. A specialized solver `SolverVEM` is implemented as a derived class from `Solver`, being responsible of the assembly and the imposition of Dirichlet boundary conditions. The assembly process has been parallelized thanks to the *multi-threading* interface `OpenMP`. Currently homogeneous conditions can be enforced, but a method to handle more general situations can be easily implemented and integrated in the class. The adopted technique first establishes the constrained DOFs, and then, during the assembly process, it checks if the DOF i being assembled in the global matrix is constrained and, if so, it skips the assembly for that DOF by setting the only nonzero term in the stiffness matrix at position (i, i) to 1, and the corresponding right hand side entry to 0. The class also provides a method to compute the strain L^2-norm, defined as

$$\|e_{\varepsilon}\|_{L^2} = \sqrt{\sum_{P\in\mathcal{P}} \int_P \|\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^h\|^2 \, d\Omega} \tag{5.1}$$

which will be used for the convergence tests. The solution vector is found exploiting the built-in solver `Eigen::ConjugateGradient`, exploiting the symmetric positive-definiteness of the stiffness matrix. An `IncompleteCholesky` preconditioner is adopted to speed up convergence.

```
    // Indexing (built in Eigen 3.4, not yet available in previous
   versions)
template <typename T, typename T2>
Eigen::Matrix<typename T2::Scalar, T::RowsAtCompileTime, T::
   ColsAtCompileTime, T::Options>
extract(const Eigen::DenseBase<T2> &full, const Eigen::DenseBase<T> &ind
   );

class Solver
{
protected:
    Eigen::SparseMatrix<real> K; // System matrix, upper part
    Eigen::VectorXd F; // Right-hand side vector
    Eigen::VectorXd U; // solution
    std::vector<bool> isConstrained; // vector storing true if dof is
   constrained
    std::vector<std::size_t> unconstrainedDofs; // vector storing
   uncosntrained dofs
    std::vector<std::size_t> constrainedDofs; // vector storing
   constrained dofs
    std::vector<real> constrainedDofsValues; // vector storing
   constrained dofs

public:
    // Default constructor
    Solver() {}

    // Constructor
    Solver(const Eigen::SparseMatrix<real> &K, const Eigen::VectorXd &F)
    : K(K), F(F) {}

    // Solve the system KU=F
    void solve();

    // Getter for F
    const Eigen::VectorXd &getRightHandSide() const;

    // Getter for U as Eigen::Vector
    const Eigen::VectorXd &getSolutionDisplacementsEig() const;

    // Getter for U as std::vector
    std::vector<real> getSolutionDisplacements() const;
};

class SolverVEM : public Solver
{
private:
    static constexpr real tolerance = 1e-12; // for Dirichlet boundary
   conditions

public:
    // Constructor, assembles K and F
    SolverVEM(const Parameters &parameters,
              const Mesh<Point3D, Edge3D, Polygon3D, Polyhedron<
   Polygon3D>> &mesh,
              const VirtualDofsCollection &DOFS,
                const LocalVirtualDofsCollection &dofs,
                const VirtualPolyhedronProjections &vp);

    // Enforce homogeneous Dirichlet boundary conditions
    void enforceHomogeneousDirichletBC(const Mesh<Point3D, Edge3D,
    Polygon3D, Polyhedron<Polygon3D>> &mesh,
                                        const VirtualDofsCollection &DOFS
    ,
                                        const std::vector<std::function<
    real(real, real, real)>> &constraintFunctions);

    // Compute the error in the L2 strain norm
    real computeStrainError(const Mesh<Point3D, Edge3D, Polygon3D,
    Polyhedron<Polygon3D>> &mesh,
                            const LocalVirtualDofsCollection &dofs,
                            const VirtualPolyhedronProjections &vp,
                            const std::function<std::array<real, 6>(real
    , real, real)> &EpsEx_func);
};
```

Listing 5.11: solver.hpp

5.7. Visualization of the results

This last section of the code is devoted to export the results, so they can be visualized, e.g., in Paraview. A snippet of the output format is given below, for a unitary cube domain.

```
# vtk DataFile Version 2.0
output.vtk
ASCII
DATASET UNSTRUCTURED_GRID
POINTS 8 double
0 0 0
1 0 0
0 1 0
1 1 0
0 0 1
1 0 1
0 1 1
1 1 1

CELLS 26 82
1 0
```

```
1 1
1 2
1 3
1 4
1 5
1 6
1 7
2 1 3
2 3 2
2 2 0
2 0 1
2 1 5
2 5 7
2 7 3
2 0 4
2 4 5
2 2 6
2 6 4
2 7 6
4 1 3 2 0
4 1 5 7 3
4 1 0 4 5
4 2 6 4 0
4 2 3 7 6
4 4 6 7 5

CELL_TYPES 26
1
1
1
1
1
1
1
1
3
3
3
3
3
3
3
3
3
```

```
3
3
3
7
7
7
7
7
7

POINT_DATA 8
VECTORS Displacement double
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0

FIELD FieldData 2
Strain 6 8 double
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

Stresses 6 8 double
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

Listing 5.12: output.vtk

```
    namespace plotting
```

```
{
    // Function to write the output file in the .vtk format
    void export_results(Mesh<Point3D, Edge3D, Polygon3D, Polyhedron<
   Polygon3D>> mesh,
                         std::vector<real> solution,
                         const char *filename = "output.vtk");
};
```

Listing 5.13: export_results.hpp

A possible way of visualizing the results allowed by the output format is depicted in Figure 5.1 and Figure 5.2.

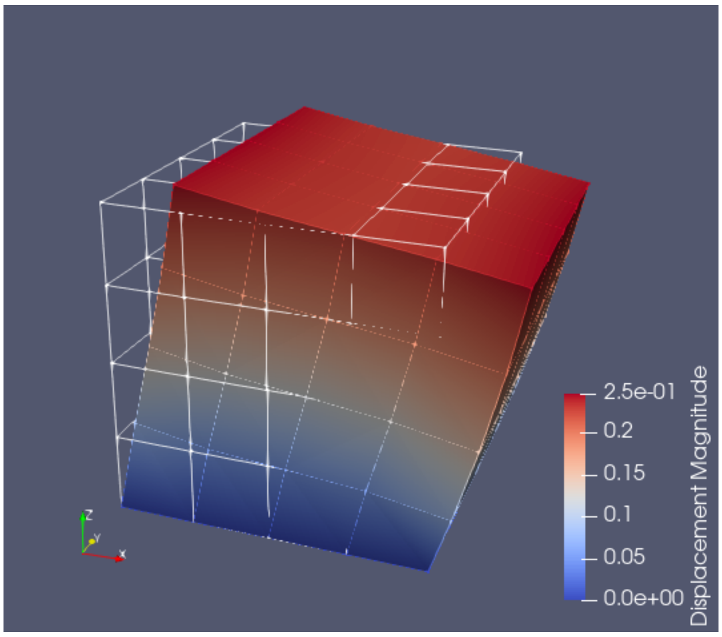


Figure 5.1: Magnitude of the displacement vector field displayed on the deformed body through a 64-element, 4th-order VEM, under constant body force field along x $f_x = 0.2$, $\lambda = 1$, $\mu = 1$, homogeneous Dirichlet boundary conditions applied on the plane $z = 0$. The white wireframe corresponds to the undeformed mesh of the $[1 \times 1 \times 1]$ cubic domain.

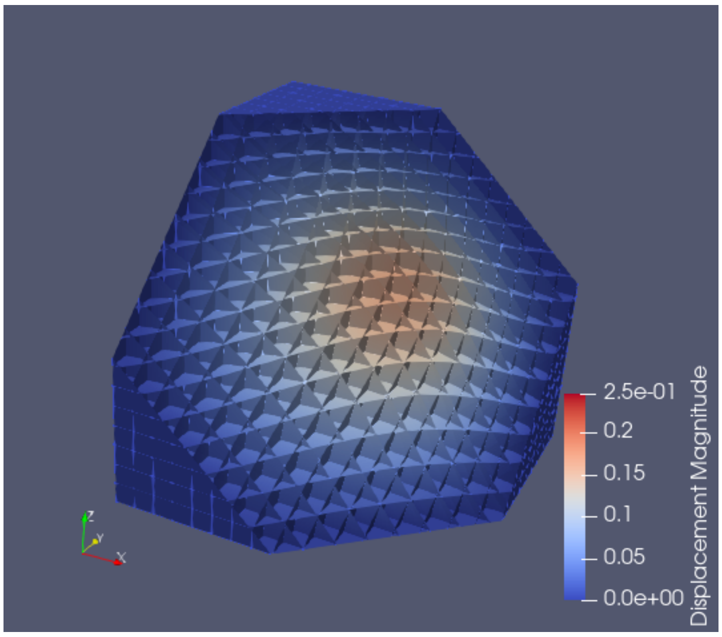


Figure 5.2: Cut through the body to see the magnitude of the displacement vector field displayed on the deformed body through a 4096-element, 1st-order VEM, under trigonometric body force field, $\lambda = 1$, $\mu = 1$, homogeneous Dirichlet boundary conditions applied on all the 6 faces.

6 | Numerical tests of the virtual element method for 3D elastostatics

In the following, the program presented in Chapter 5 is tested, performing h- and p-convergences analyses. Hereafter, units of measurement are not specified, and can be chosen provided they are consistent with eachother (e.g. N/mm^3 for body forces; N/mm^2 for surface tractions, stresses, Young's modulus and Lamé constants; mm for lengths). The tests are run on a cubic domain of dimensions $[1 \times 1 \times 1]$.

6.1. Data of the problem

The following problem is numerically solved

$$\begin{cases} -\nabla \cdot [\boldsymbol{D}\boldsymbol{\varepsilon}(\boldsymbol{u})] = \boldsymbol{f} & \text{in } \Omega = (0,1)^3 \\ \boldsymbol{u} = \boldsymbol{0} & \text{on } \partial\Omega \end{cases} \tag{6.1}$$

where

$$\boldsymbol{\varepsilon}(\boldsymbol{u}) = \frac{\nabla \boldsymbol{u} + \nabla^{\mathrm{T}} \boldsymbol{u}}{2}$$

A trigonometric field for the displacements $\boldsymbol{u}$ is selected (Figure 6.1), so that it satisfies the boundary conditions of problem (6.1)

$$\boldsymbol{u}(x,y,z) = C \sin(\pi x) \sin(\pi y) \sin(\pi z) \begin{Bmatrix} 1 \\ 1 \\ 1 \end{Bmatrix}$$

where C is a constant. The displacement above corresponds to the following exact strains, which will be used for the computation of the L^2 strain norm of the error, according to

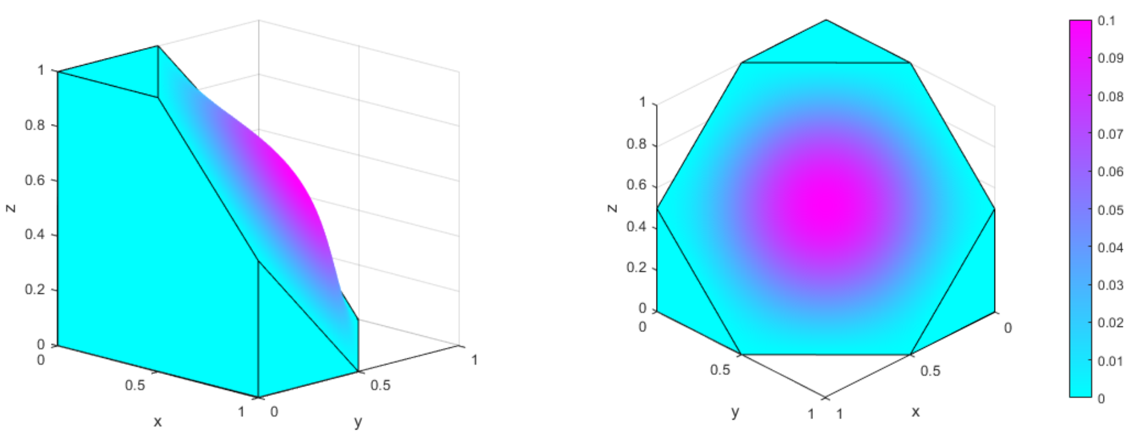


Figure 6.1: Exact displacements solution for the elastic problem in (6.1).

(5.1).

$$\boldsymbol{\varepsilon}(x,y,z) = C \left\{ \begin{matrix} \pi\cos(\pi x)\sin(\pi y)\sin(\pi z) \\ \pi\sin(\pi x)\cos(\pi y)\sin(\pi z) \\ \pi\sin(\pi x)\sin(\pi y)\cos(\pi z) \\ \pi\left[\sin(\pi x)\cos(\pi y)\sin(\pi z) + \cos(\pi x)\sin(\pi y)\sin(\pi z)\right] \\ \pi\left[\sin(\pi x)\sin(\pi y)\cos(\pi z) + \sin(\pi x)\cos(\pi y)\sin(\pi z)\right] \\ \pi\left[\sin(\pi x)\sin(\pi y)\cos(\pi z) + \cos(\pi x)\sin(\pi y)\sin(\pi z)\right] \end{matrix} \right\}$$

Hence, plugging in (6.1), the body forces $\boldsymbol{f}$ to be given to the solver are found

$$\boldsymbol{f}(x,y,z) = C \left\{ \begin{matrix} -\pi^2\left[(\lambda+\mu)\cos(\pi x)\sin(\pi y+\pi z) - (\lambda+4\mu)\sin(\pi x)\sin(\pi y)\sin(\pi z)\right] \\ -\pi^2\left[(\lambda+\mu)\cos(\pi y)\sin(\pi x+\pi z) - (\lambda+4\mu)\sin(\pi x)\sin(\pi y)\sin(\pi z)\right] \\ -\pi^2\left[(\lambda+\mu)\cos(\pi z)\sin(\pi x+\pi y) - (\lambda+4\mu)\sin(\pi x)\sin(\pi y)\sin(\pi z)\right] \end{matrix} \right\}$$

In the following tests, the constant C is set to 0.1. Hexahedral meshes and generic polyhedral meshes are generated with progressively smaller elements, exploiting Voronoi tessellation with seeds placed respectively on a specific pattern, and randomly.

6.2. h-refinement

In the section, h-refinement test results are presented for the first three order of the virtual element method. The error in the L^2 strain norm computed according to (5.1) can be estimated with the following

$$\|e_{\varepsilon}\|_{L^2(\Omega)} = C(\boldsymbol{u},k)h^p \tag{6.2}$$

where $C(\boldsymbol{u}, k)$ is a constant depending only on the solution and on the order k of the VEM, h is the average element size of the mesh and p is the order of convergence. This section aims at finding the values of p under the refinement of the mesh, i.e., progressively reducing the size of the elements for a fixed order k. Taking (6.2) for two different sizes h_1 and h_2 leads to

$$\left.\begin{aligned}\|e_\varepsilon(h_1)\|_{L^2(\Omega)} &= C(\boldsymbol{u}, k)h_1^p \\ \|e_\varepsilon(h_2)\|_{L^2(\Omega)} &= C(\boldsymbol{u}, k)h_2^p\end{aligned}\right\} \implies \frac{\|e_\varepsilon(h_2)\|_{L^2(\Omega)}}{\|e_\varepsilon(h_2)\|_{L^2(\Omega)}} = \frac{C(\boldsymbol{u}, k)h_2^p}{C(\boldsymbol{u}, k)h_1^p} = \left(\frac{h_2}{h_1}\right)^p$$

and taking the logarithm leads to

$$p = \log_{\left(\frac{h_2}{h_1}\right)}\left[\frac{\|e_\varepsilon(h_2)\|_{L^2(\Omega)}}{\|e_\varepsilon(h_2)\|_{L^2(\Omega)}}\right] = \frac{\log\left(\frac{\|e_\varepsilon(h_2)\|_{L^2(\Omega)}}{\|e_\varepsilon(h_2)\|_{L^2(\Omega)}}\right)}{\log\left(\frac{h_2}{h_1}\right)}$$

Therefore, it is convenient to plot the L^2 strain errors in a log-log plane as a function of the mesh size h, and check if the points align on a straight line whose slope corresponds to the order of the method k.

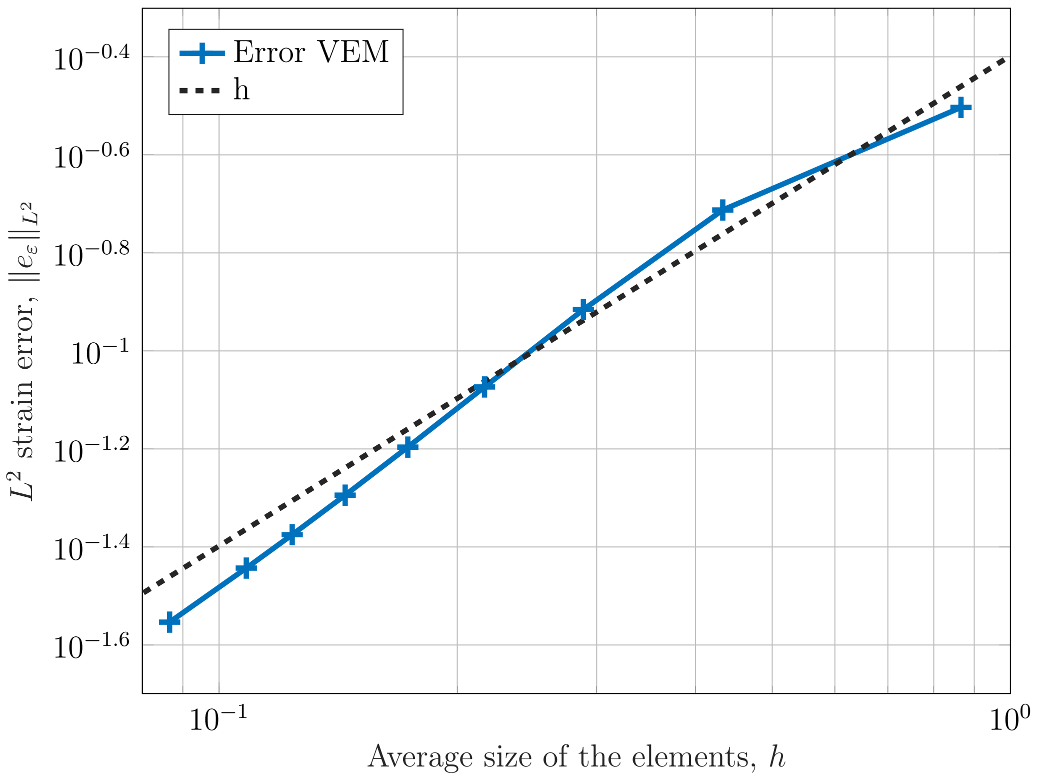


Figure 6.2: h-convergence test for 1st order VEM.

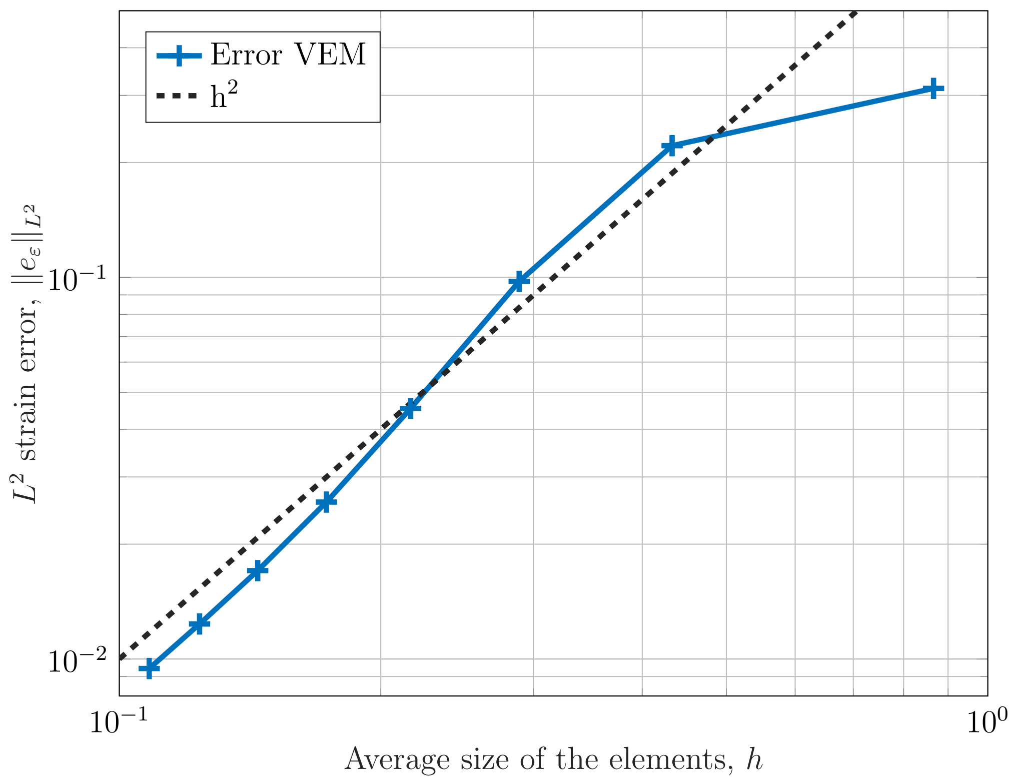


Figure 6.3: h-convergence test for 2nd order VEM.

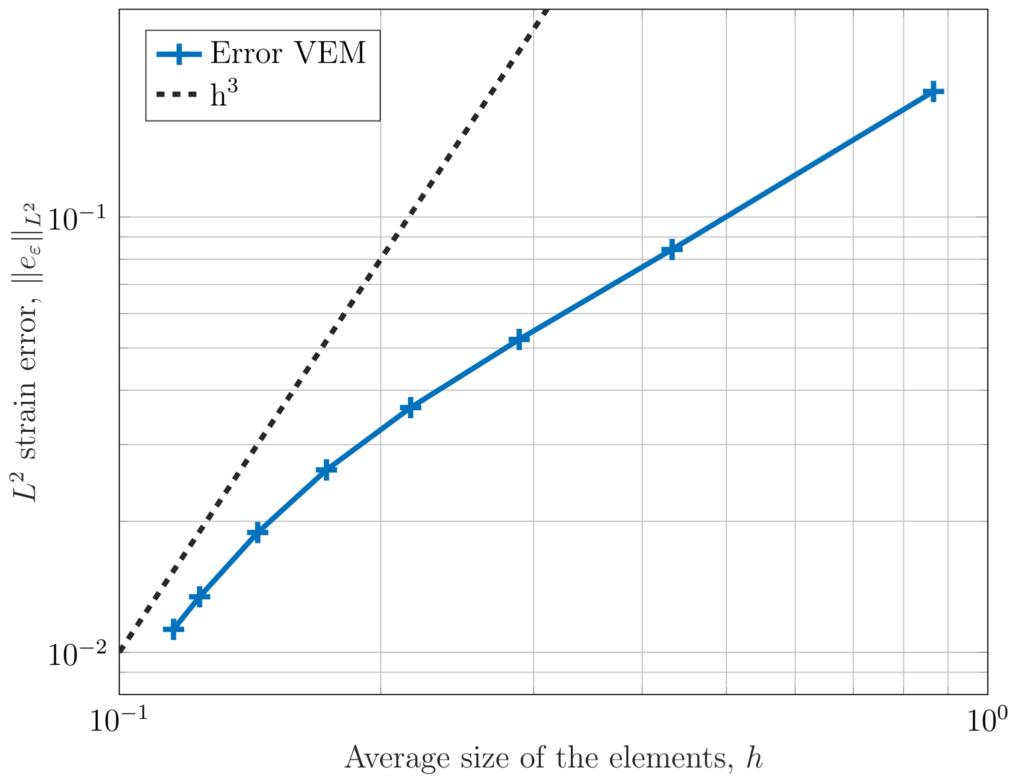


Figure 6.4: h-convergence test for 3rd order VEM.

For $k = 1$, the error in the L^2 strain norm shown in Figure 6.2 clearly follows the corresponding line representing the expected behaviour under h-refinement, dashed in black. Good convergence is already achieved with very coarse meshes, with elements of diameter of $\frac{1}{4}$ of the size of the domain. Further refining leads to an actual slope which seems to provide even better convergence results. This should be attributed to the symmetry of the problem and the data, still remembering that the estimate (6.2) provides an upper bound for the error.

For $k = 2$, the graph in Figure 6.3 displays a quadratic convergence, as expected, following the black dashed line.

For $k = 3$, the graph in Figure 6.4 initially seems not to follow the expected cubic convergence. However, under mesh refinement, one notices that the errors slowly align towards the correct line. This behaviour of slow approaching towards the expected convergence rate could be attributed by the high degree of constraint that the problem is subjected to. Indeed, Dirichlet boundary conditions are imposed on all the six faces, effectively restraining all the related vertex-type, edge-type and face-type degrees of freedom, and can strongly affect a finer solution provided by a higher order degree VEM with respect to, e.g., the linear or quadratic version. This behavior is even more evident with higher orders VEM such as 4 or 5. A further refinement would confirm the correct degrees of convergence also for these higher order VEM, but the number of degrees of freedom rapidly grows for $k > 3$ (for a hexahedral mesh of 3375 elements a VEM of 4th order already has almost half a million of DOFS) and a lot of RAM and computational power is required.

6.3. p-refinement

In this section, p-refinements results are presented, given a fixed mesh size h of 0.866 0.433, 0.289, 0.217.

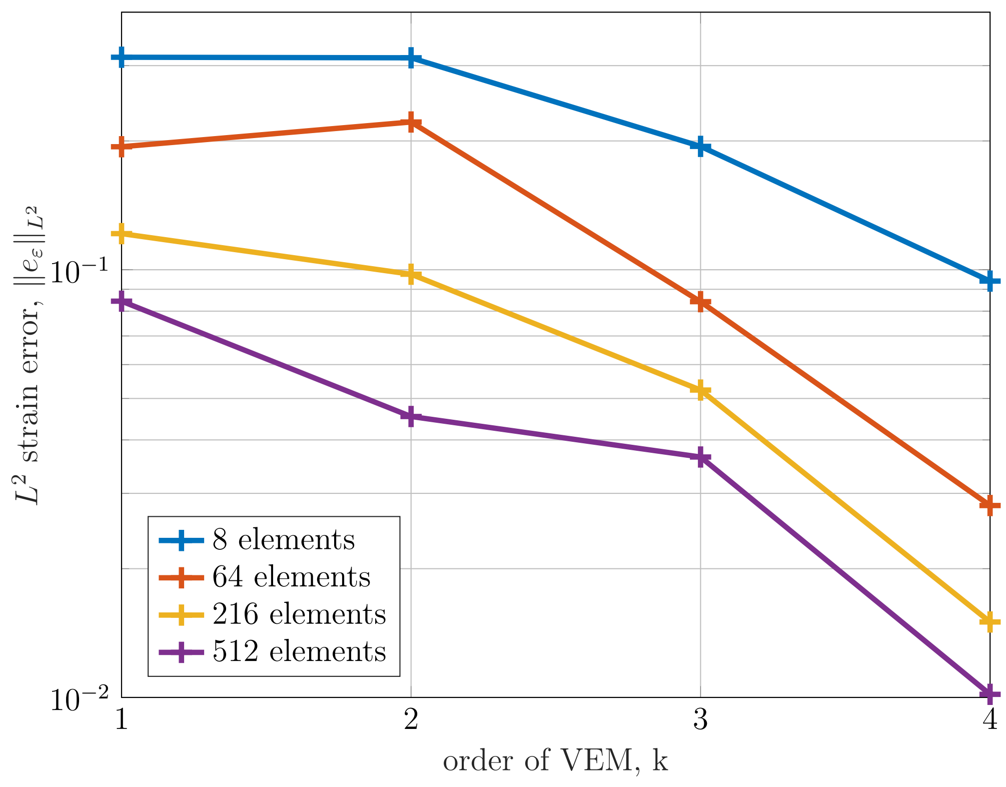


Figure 6.5: p-convergence test of 1st to 4th order VEM for meshes of 8, 64, 216 and 512 elements.

As shown in Figure 6.5, for a given mesh, the computed L^2 strain error in general decreases as the VEM order k increases, proving the superior accuracy of higher order methods. However, some anomalies appear, being attributable to the coarse mesh of the cases, the specific regularity of the problem and the strong boundary conditions, as already discussed for h-convergence. To visualize the improved performance of adopting a higher order method, a corner of the domain $\Omega = (0,1)^3$ undergoing the elastic deformation of (6.1), this time under homogeneous Dirichlet conditions imposed only on the $z = 0$ face, and under a constant body force in the x direction $f_x = 0.2$, is displayed in Figure 6.6. Despite the coarse mesh, increasing the VEM order k rapidly leads to a gain in the accuracy of the solution.

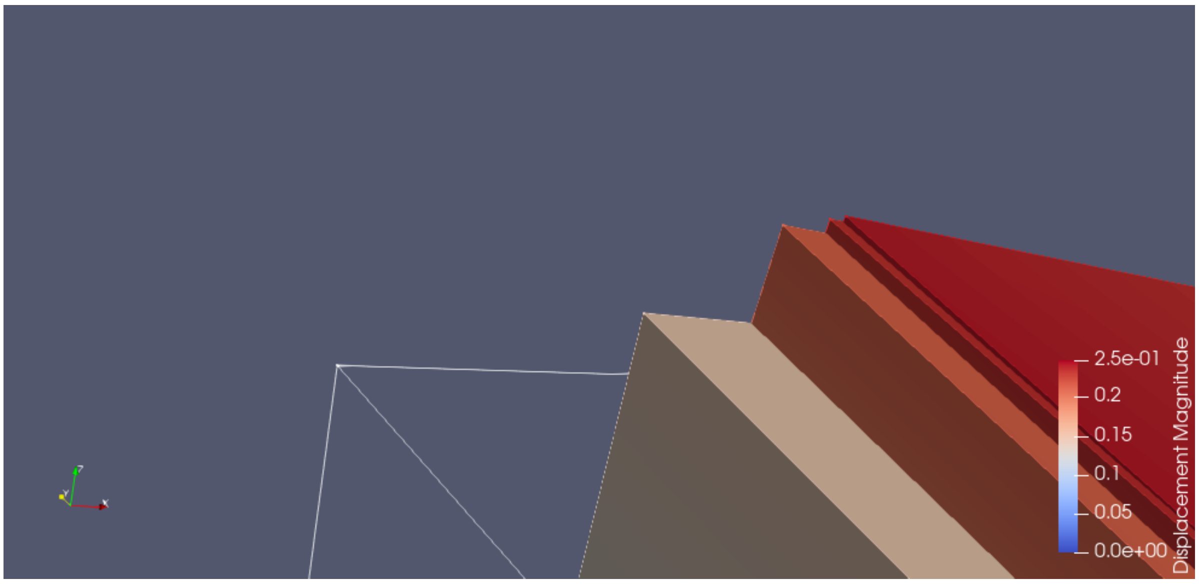


Figure 6.6: Magnitude of the displacement vector field displayed on the deformed body through a 8-element, 1st to 4th order VEM, under constant body force field along x $f_x = 0.2$, $\lambda = 1$, $\mu = 1$, homogeneous Dirichlet boundary conditions applied on the plane $z = 0$. The white edges represent the undeformed configuration and the colored corners, from left to right, the solution found through 1st, 2nd, 3rd and 4th order VEM.

7 | Conclusions and future developments

This project has developed the virtual element method of arbitrary order in the context of linear elastostatics, providing the theoretical background and the computer implementation in the object-oriented programming language c++. An extensive introduction on the elastic problem has been given to provide a good setting for the problem at hand, and a mixed-finite element formulation from the three-field Hu-Washizu functional has been presented and followed to construct the virtual element method. The mathematical key aspects of the VEM have been presented and applied. The implementation choices have been explained and documented, together with the adopted algorithms, amongst which an innovative numerical integration technique to perform quadrature of monomials over general polytopes. Numerical tests, in the form of h- and p-convergence analyses, were developed to show the capabilities of the method and the robustness of the code, together with visualization results.

The present work can be further extended, remaining in the context of elasticity, to account for dynamic effects or material nonlinearities. From the computer program point of view, these features could be implemented exploiting the modularity of the present code, and adapting the solver to handle the iterative nature of the above improvements. Additionally, more complex and efficient algorithms could be adopted to support integration of general functions, needed in the computation of the equivalent nodal forces, over all the polyhedra allowed by the virtual element method.

Bibliography

[1] B. Ahmad, A. Alsaedi, F. Brezzi, L. D. Marini, and A. Russo. Equivalent projectors for virtual element methods. *Computers & Mathematics with Applications*, 66(3): 376–391, 2013.

[2] P. F. Antonietti, P. Houston, and G. Pennesi. Fast numerical integration on polytopic meshes with applications to discontinuous galerkin finite element methods. *Journal of Scientific Computing*, 77(3):1339–1370, 2018.

[3] E. Artioli, L. Beirão da Veiga, C. Lovadina, and E. Sacco. Arbitrary order 2d virtual elements for polygonal meshes: part i, elastic problem. *Computational Mechanics*, 60:355–377, 2017.

[4] F. Auricchio, L. B. da Veiga, F. Brezzi, and C. Lovadina. Mixed finite element methods. *Encyclopedia of Computational Mechanics Second Edition*, pages 1–53, 2017.

[5] B. Bauman, A. Sommariva, and M. Vianello. Compressed algebraic cubature over polygons with applications to optical design. *Journal of Computational and Applied Mathematics*, 370:112658, 2020.

[6] L. Beirão da Veiga, F. Brezzi, A. Cangiani, G. Manzini, L. D. Marini, and A. Russo. Basic principles of virtual element methods. *Mathematical Models and Methods in Applied Sciences*, 23(01):199–214, 2013.

[7] L. Beirão da Veiga, F. Brezzi, L. D. Marini, and A. Russo. The hitchhiker's guide to the virtual element method. *Mathematical models and methods in applied sciences*, 24(08):1541–1573, 2014.

[8] D. Boffi, F. Brezzi, M. Fortin, et al. *Mixed finite element methods and applications*, volume 44. Springer, 2013.

[9] B. Cockburn, G. E. Karniadakis, and C.-W. Shu. *Discontinuous Galerkin methods: theory, computation and applications*, volume 11. Springer Science & Business Media, 2012.

[10] L. Corradi. A displacement formulation for the finite element elastic-plastic problem. *Meccanica*, 18(2):77–91, 1983.

[11] L. B. Da Veiga, A. Russo, and G. Vacca. The virtual element method with curved edges. *ESAIM: Mathematical Modelling and Numerical Analysis*, 53(2):375–404, 2019.

[12] F. Dassi, A. Fumagalli, D. Losapio, S. Scialò, A. Scotti, and G. Vacca. The mixed virtual element method on curved edges in two dimensions. *Computer Methods in Applied Mechanics and Engineering*, 386:114098, 2021.

[13] J. Djoko, B. Lamichhane, B. Reddy, and B. Wohlmuth. Conditions for equivalence between the hu–washizu and related formulations, and computational behavior in the incompressible limit. *Computer methods in applied mechanics and engineering*, 195(33-36):4161–4178, 2006.

[14] Geuzaine, Christophe and Remacle, Jean-Francois. Gmsh. URL `http://gmsh.info/`.

[15] M. E. Gurtin. *An introduction to continuum mechanics*. Academic press, 1982.

[16] R. J. Guyan. Reduction of stiffness and mass matrices. *AIAA journal*, 3(2):380–380, 1965.

[17] P. Helnwein. Some remarks on the compressed matrix representation of symmetric second-order and fourth-order tensors. *Computer methods in applied mechanics and engineering*, 190(22-23):2753–2770, 2001.

[18] T. M. Inc. Matlab version: 9.13.0 (r2022b), 2022. URL `https://www.mathworks.com`.

[19] J. Jaśkowiec and N. Sukumar. High-order cubature rules for tetrahedra. *International Journal for Numerical Methods in Engineering*, 121(11):2418–2436, 2020.

[20] A. Lamperti. Virtual elements based on the hu-washizu variational principle and their application in linear elastostatics and elastodynamics. Master's thesis, Politecnico di Milano, 12 2021.

[21] J. Matousek. *Lectures on discrete geometry*, volume 212. Springer Science & Business Media, 2013.

[22] S. Mousavi and N. Sukumar. Numerical integration of polynomials and discontinuous functions on irregular convex polygons and polyhedrons. *Computational Mechanics*, 47:535–554, 2011.

[23] K. Park, H. Chi, and G. H. Paulino. Numerical recipes for elastodynamic virtual element methods with explicit time integration. *International Journal for Numerical Methods in Engineering*, 121(1):1–31, 2020.

[24] G. H. Paulino and A. L. Gain. Bridging art and engineering using escher-based virtual elements. *Structural and Multidisciplinary Optimization*, 51:867–883, 2015.

[25] A. Russo and N. Sukumar. Quantitative study of the stabilization parameter in the virtual element method. *arXiv preprint arXiv:2304.00063*, 2023.

[26] W. S. Slaughter. *The linearized theory of elasticity.* Springer Science & Business Media, 2012.

[27] A. Sommariva and M. Vianello. Tetrafreeq: tetrahedra-free quadrature on polyhedral elements. *Applied Numerical Mathematics*, 2023.

[28] G. von Winckel. Gauss quadrature for tetrahedra, 2005. URL `https://www.mathworks.com/matlabcentral/fileexchange/9389-gauss-quadrature-for-tetrahedra`.

[29] K. Washizu. *Variational Methods in Elasticity and Plasticity.* Division 1 : solid and structural mechanics. Elsevier Science & Technology, 1974. ISBN 9780080176536. URL `https://books.google.it/books?id=jpseAQAAIAAJ`.

[30] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The finite element method: its basis and fundamentals.* Elsevier, 2005.

List of Figures

Acknowledgements

Special thanks for this project are due to my Master Thesis advisor, Prof. Massimiliano Cremonesi, as the present c++ arbitrary order implementation of the three dimensional virtual element method is an improved development of the low-order MATLAB [18] program I wrote for the same problem, under his supervision. A heartfelt thanks goes to Prof. Franco Dassi, for the precious suggestions on the object-oriented implementation.