Politecnico di Milano - MSc in Mathematical Engineering
Under the supervision of Prof. M. Verani and Prof. P. Antonietti

Virtual Element Method

Stefano Savarè
stefano.savare@mail.polimi.it
Ondine Chanon
ondine.chanon@mail.polimi.it

Course project of Advanced Numerical Analysis of Partial Differential Equations (Prof. S. Perotto), and Advanced Programming for Scientific Computing (Prof. L. Formaggia)

Fall 2016

Abstract

This project deals with the Virtual Element Method (VEM), both from the theoretical and the implementation points of view. The theoretical basis of VEM on Laplace equation with Dirichlet boundary conditions is given, in 1, 2 and 3 dimensions, together with the algorithms used for its implementation in `C++`. An explanation of our implementation of VEM is also given. Finally, some numerical results about convergence are reported.

Contents

Introduction

The Virtual Element Method (VEM in short) is an advanced numerical method that generalizes the Finite Element Method (FEM in short). Indeed, the underlying virtual element space is the same as the finite element space, together with some suitable non-polynomial functions. Non polynomial functions are harder to create and to handle. This is why VEM works in such a way that, in order to compute the stiffness matrix of the problem, we only need to compute the value of those non-polynomial functions on well-chosen degrees of freedom, without actually computing the functions.

Since the local stiffness matrix actually comes form a bilinear form, we want to be able to exactly compute every entry that corresponds to the result of the bilinear form with at least one polynomial argument. While when none of the arguments of the bilinear form are polynomials, we only require a result with the right order of magnitude (k-consistency) and with some stability properties. An advantage of doing so comes from the fact that VEM can then deal with more complex element geometries, such that non-necessarily convex polygons.

In this project, we will concentrate on Laplace equation with Dirichlet boundary conditions on a polygonal domain Ω embedded in a d dimensional space. The general problem writes as follows: find a solution u of

$$-\Delta u = f \text{ in } \Omega, \text{ and } u = g \text{ on } \Gamma = \partial\Omega,$$

where f and g are given functions defined respectively in a d-dimensional domain Ω and on Γ, the boundary of Ω. We will treat this problem both in theoretical terms, and with a regard towards its `C++` implementation for 2D and 3D domains, and with linear polynomial basis.

The first chapter will concentrate on the theory of VEM and on its application on Laplace equation. It will then be followed by a chapter explaining the algorithms used for the implementation of VEM, and by another chapter on its implementation itself. Some results, in particular on the numerical convergence of the method, and a conclusion will close this report.

1 Theoretical background

1.1 Variational problem

We recall the strong formulation of the general problem: on a polygonal domain $\Omega \subset \mathbb{R}^d$, find a solution u of

$$-\Delta u = f \text{ in } \Omega, \text{ and } u = g \text{ on } \Gamma = \partial\Omega, \tag{1}$$

where f and g are given functions defined respectively in Ω and on Γ.

Suppose that $f \in L^2(\Omega)$, $g \in H^{1/2}(\Gamma)$ and $d = 1, 2$ or 3. We multiply equation (1) by a test function v, and we integrate the strong formulation of the problem. Thanks to Green formula, we obtain:

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\Omega - \int_\Gamma (\nabla u \cdot \vec{n}) v \, \mathrm{d}\Gamma = \int_\Omega f v \, \mathrm{d}\Omega, \tag{2}$$

where $\vec{n}$ is the exterior normal vector to the surface boundary Γ. Suppose there exist a continuous lifting of the boundary data $R_g \in H^1(\Omega)$ such that $R_g|_\Gamma = g$. We set $\tilde{u} = u - R_g$, so that $\tilde{u}|_\Gamma = u|_\Gamma - R_g|\Gamma = g - g = 0$, and $\nabla\tilde{u} = \nabla u - \nabla R_g$. Consequently, equation (2) becomes:

$$\int_\Omega \nabla\tilde{u} \cdot \nabla v \, \mathrm{d}\Omega = \int_\Omega f v \, \mathrm{d}\Omega - \int_\Omega \nabla R_g \cdot \nabla v \, \mathrm{d}\Omega.$$

In the following, we omit the tilde on $\tilde{u}$ for a matter of simplicity. The variational formulation of our problem is thus now the following: find u in $V = H_0^1(\Omega)$ such that for all $v \in V$,

$$a(u, v) = F(v), \tag{3}$$

where $a : V \times V \to \mathbb{R}$ is such that $a(w, v) = \int_\Omega \nabla w \cdot \nabla v \, \mathrm{d}\Omega$ for all $w, v \in V$, and $F : V \to \mathbb{R}$ is such that for all $v \in V$, $F(v) = \int_\Omega f v \, \mathrm{d}\Omega - a(R_g, v)$.

We choose the semi-norm of $H^1(\Omega)$ to be the norm we will use for functions in $H_0^1(\Omega) = V$, thanks to Poincaré inequality. In V equipped with this norm, it is easy to show that a is a continuous and coercive bilinear form, and that F is a linear and continuous functional. We prove here some of the main properties: for all $v, w \in V$,

$$\begin{aligned}
|F(v)| &= \left| \int_\Omega f v \, \mathrm{d}\Omega - \int_\Omega \nabla R_g \cdot \nabla v \, \mathrm{d}\Omega \right| \\
&\leq \left| \int_\Omega f v \, \mathrm{d}\Omega \right| + \left| \int_\Omega \nabla R_g \cdot \nabla v \, \mathrm{d}\Omega \right| \\
&\leq \|f\|_{L^2(\Omega)} \|v\|_{L^2(\Omega)} + \|\nabla R_g\|_{L^2(\Omega)} \|\nabla v\|_{L^2(\Omega)} \\
&\leq \left(\|f\|_{L^2(\Omega)} + \|\nabla R_g\|_{L^2(\Omega)} \right) \|v\|_{H_0^1(\Omega)};
\end{aligned}$$

$$\begin{aligned}
a(v, v) &= \int_\Omega \nabla v \cdot \nabla v \, \mathrm{d}\Omega = \|v\|^2_{H_0^1(\Omega)} \\
&\geq 1 \cdot \|v\|^2_{H_0^1(\Omega)}; \qquad (4) \\
a(w, v) &= \int_\Omega \nabla w \cdot \nabla v \, \mathrm{d}\Omega \\
&\leq 1 \cdot \|w\|_{H_0^1(\Omega)} \|v\|_{H_0^1(\Omega)}. \qquad (5)
\end{aligned}$$

Cauchy-Schwarz inequality has been used to show both continuities. Thus by Lax-Milgram theorem, there exists a unique solution to the weak problem expressed in equation (3).

1.2 Discrete problem

1.2.1 Domain decomposition

Consider a sequence of decompositions $\{T_h\}_h$ of Ω, where h corresponds to the maximum of the diameters h_K of the elements K of the decomposition T_h. Note that since Ω is a polygonal domain, then for every h, $T_h = \Omega$. Let us recall the definition of the diameter of an element:

Definition 1.1 (Diameter) The *diameter* of an element K of a domain decomposition is defined as $h_K = \max_{x,y\in K} |x-y|$.

Since we only consider polygonal domains Ω, then exact decompositions with simple polygons exist.

Definition 1.2 (Simple polygon) A *simple polygon* is an open simply connected set whose boundary is a non-intersecting line made of a finite number of straight line segments.

From now on, we suppose that every considered decomposition T_h is made of a finite number of simple polygons. Note that a simple polygon is not necessarily convex, and this is what mainly characterizes VEM instead of FEM. We now introduce the following notation:

Notation 1.3 For any set of functions W, we note $W|_K$ the set of functions in W restricted to K, that is $W|_K = \{f|_K : f \in W\}$.

Let us define the bilinear form a reduced to any sub-domain D of Ω as follows: for all $w, v \in V$,

$$a^D(w,v) := \int_D \nabla w \cdot \nabla v \,\mathrm{d}\Omega.$$

Since the restriction of any function of $H_0^1(\Omega)$ on a sub-domain D of Ω is in $H_0^1(\Omega) \cap H^1(D)$ but is not necessarily in $H_0^1(D)$, we note that a^D is also a continuous bilinear form on $V \times V$, but it is not necessarilz coercive. Moreover, if D is some element K of T_h, for all h and for all $w, v \in V$,

$$a(w,v) = \sum_{K\in T_h} a^K(w,v). \tag{6}$$

In the same way, since $H_0^1(\Omega)|_K = V|_K \subset H^1(K)$, we define for all $v \in V$

$$|v|_{1,K} := |(v|_K)|_{H^1(K)} = \left(a^K(v,v)\right)^{1/2}, \tag{7}$$

$$\text{so that } \|v\|_{H_0^1(\Omega)} = \left(\sum_{K\in T_h} |v|_{1,K}^2\right)^{1/2}. \tag{8}$$

Furthermore, for every h, we will denote E_h the set of edges of the decomposition T_h.

1.2.2 Towards space discretization

For every domain decomposition T_h, the aim of VEM is to come up with a finite subspace V_h of V such that:

- we can obtain a discretized version of problem (3),

- the discretized problem has a unique solution,
- the unique solution of the discretized problem has good approximation properties with respect to the exact solution.

To do so, we first need to introduce a notation:

Notation 1.4 For every domain D and every non-negative integer k, $\mathbb{P}_k(D)$ is the set of polynomials in D of degree at most k, and $\mathbb{P}_{-1}(D) = \{0\}$.

To figure out what we exactly need to build, we state and prove the following theorem:

Theorem 1.5 *For each h, suppose that*

1. *on one hand, we have*
   (a) *a suitable finite dimensional subspace V_h of V;*
   (b) *a symmetric bilinear form $a_h : V_h \times V_h \to \mathbb{R}$, and for all K in T_h a bilinear form $a_h^K : V_h|_K \times V_h|_K \to \mathbb{R}$, such that for all $w_h, v_h \in V_h$,*
   $$a_h(w_h, v_h) = \sum_{K \in T_h} a_h^K(w_h|_K, v_h|_K).$$
   For simplicity, we write $a_h^K(w_h, v_h)$ instead of $a_h^K(w_h|_K, v_h|_K)$;
   (c) *an element $f_h \in V_h'$ where V_h' is the dual of V_h;*
2. *on the other hand, there exists an integer $k \geq 1$ such that for all $K \in T_h$,*
   (a) *$\mathbb{P}_k(K) \subset V_h|_K$, that is our virtual space restricted to any element of the domain decomposition contains the set of polynomials of order at most k;*
   (b) *(k-consistency) for all $p \in \mathbb{P}_k(K)$, for all $v_h \in V_h|_K$, the relation $a_h^K(p, v_h) = a^K(p, v_h)$ is verified;*
   (c) *(stability) there exist $\alpha^-, \alpha^+ \geq 0$ two constants independent of h and of K such that for all $v_h \in V_h|_K$,*
   $$\alpha^- a^K(v_h, v_h) \leq a_h^K(v_h, v_h) \leq \alpha^+ a^K(v_h, v_h).$$

Under these hypothesis,

1. *the following discrete problem: find $u_h \in V_h$ such that for all $v_h \in V_h$,*
   $$a_h(u_h, v_h) = \langle f_h, v_h \rangle, \tag{9}$$
   admits a unique solution;
2. *for every approximations $u_I \in V_h$ and u_π that is piecewise in $\mathbb{P}_k$, of the exact solution u of problem (3),*
   $$\|u - u_h\|_{H_0^1(\Omega)} \leq C \left(\|u - u_I\|_{H_0^1(\Omega)} + \|u - u_\pi\|_{H_0^1(\Omega)} + \mathcal{F}_h \right),$$
   where $C = C(\alpha^-, \alpha^-)$, and $\mathcal{F}_h$ is the smallest constant such that for all $v \in V_h$,
   $$F(v) - \langle f_h, v \rangle \leq \mathcal{F}_h \|v\|_{H_1^0(\Omega)}. \tag{10}$$

Remark 1.6 Note that here, for $f_h \in V_h'$ and $v_h \in V_h$, $\langle f_h, v_h \rangle$ designs the action of f_h on v_h. Moreover, u_π of statement 2 is piecewise in $\mathbb{P}_k$ and $\mathbb{P}_k(K)$ is included in $V_h|_K$ for all $K \in T_h$ (from hypothesis 2.a), so u_π belongs to $\Pi_{K \in V_h} \mathbb{P}_k(K) \subset \Pi_{K \in V_h} V_h|_K = V_h \subset H_0^1(\Omega)$. This is why the norm of $u - u_\pi$ can be the $H_0^1(\Omega)$ norm.

Let us prove this theorem:

Proof. We will prove the two statements in order. The abbreviation CS is used to say that we use the Cauchy-Schwarz inequality.

1. We want to use Lax-Milgram theorem.

   - Since V_h is a finite dimensional subspace of the Hilbert space V, then it is a closed subspace of an Hilbert space, so that V_h is also an Hilbert space. We still consider the $H^1(\Omega)$-semi-norm as the $H_0^1(\Omega)$-norm, i.e. the V-norm, and thus as the V_h-norm as well.
   - By hypothesis 1.b and 1.c, a_h is a bilinear form and f_h is a linear and continuous functional.
   - Continuity of a_h: first, for all $w, v \in V_h|_K$,

     $$\begin{aligned} a_h^K(w,v) &\leq \left(a_h^K(w,w)\right)^{1/2}\left(a_h^K(v,v)\right)^{1/2} && \text{(CS)} \\ &\leq \alpha^+ \left(a^K(w,w)\right)^{1/2}\left(a^K(v,v)\right)^{1/2} && \text{(stability)} \\ &= \alpha^+ |u|_{1,K} |v|_{1,K}. && \text{(definition)} \end{aligned} \tag{11}$$

     Consequently, for all $w_h, v_h \in V_h$,

     $$\begin{aligned} a_h(w_h,v_h) &= \sum_{K \in T_h} a_h^K(w_h,v_h) && \text{(hypothesis 1.}b\text{)} \\ &\leq \alpha^+ \sum_{K \in T_h} |w_h|_{1,K}|v_h|_{1,K} && \text{(inequality 11)} \\ &\leq \alpha^+ \left(\sum_{K \in T_h} |w_h|_{1,K}^2\right)^{1/2}\left(\sum_{K \in T_h} |v_h|_{1,K}^2\right)^{1/2} \\ & && \text{(discrete CS)} \\ &= \alpha^+ \|w_h\|_{H_0^1(\Omega)} \|v_h\|_{H_0^1(\Omega)}. && \text{(equation 8)} \end{aligned}$$

   - Coercivity of a_h: for all $v_h \in V_h$,

     $$\begin{aligned} a_h(v_h,v_h) &= \sum_{K \in T_h} a_h^K(v_h,v_h) && \text{(hypothesis 1.}b\text{)} \\ &\geq \alpha^- \sum_{K \in T_h} a^K(v_h,v_h) && \text{(stability)} \\ &= \alpha^- a(v_h,v_h) && \text{(equation 6)} \\ &= \alpha^- \|v_h\|_{H_0^1(\Omega)}^2. && \text{(equation 4)} \end{aligned}$$

Consequently, by Lax-Milgram theorem, the discrete problem (9) admits one unique solution u_h.

2. Let $u_I \in V_h$ and u_π piecewise in $\mathbb{P}_k$ such that they are approximations of the exact solution u of problem (3). Let $\delta_h = u_h - u_I \in V_h$. Then,

$$\begin{aligned}
\alpha^- \|\delta_h\|^2_{H^1_0(\Omega)} &\leq a_h(\delta_h, \delta_h) \text{ (coercivity of } a_h) \\
&= a_h(u_h, \delta_h) - a_h(u_I, \delta_h) \\
&\qquad\qquad\qquad (a_h \text{ linear and definition of } \delta_h) \\
&= \langle f_h, \delta_h \rangle - \sum_{K \in T_h} a_h^K(u_I, \delta_h) \\
&\qquad\qquad\qquad \text{(problem (9) and hypothesis 1.} b) \\
&= \langle f_h, \delta_h \rangle - \sum_{K \in T_h} \Big(a_h^K(u_I - u_\pi, \delta_h) + a_h^K(u_\pi, \delta_h) \Big) \\
&\qquad\qquad\qquad (a_h \text{ linear and } \pm u_\pi) \\
&= \langle f_h, \delta_h \rangle - \sum_{K \in T_h} \Big(a_h^K(u_I - u_\pi, \delta_h) + a^K(u_\pi, \delta_h) \Big) \\
&\qquad\qquad\qquad (k\text{-consistency}) \\
&= \langle f_h, \delta_h \rangle - \sum_{K \in T_h} \Big(a_h^K(u_I - u_\pi, \delta_h) + a^K(u_\pi - u, \delta_h) \Big) \\
&\qquad - a(u, \delta_h) \qquad (\pm u, a^K \text{linear and equation 6}) \\
&= \langle f_h, \delta_h \rangle - \sum_{K \in T_h} \Big(a_h^K(u_I - u_\pi, \delta_h) + a^K(u_\pi - u, \delta_h) \Big) \\
&\qquad - F(\delta_h) \qquad\qquad\qquad \text{(problem (3))} \\
&\leq |F(\delta_h) - \langle f_h, \delta_h \rangle| \\
&\quad - \sum_{K \in T_h} \Big(a_h^K(u_I - u_\pi, \delta_h) + a^K(u_\pi - u, \delta_h) \Big) \\
&\leq \mathcal{F}_h \|\delta_h\|_{H^1_0(\Omega)} - a_h(u_I - u_\pi, \delta_h) - a(u_\pi - u, \delta_h) \\
&\qquad\qquad \text{(inequalities 10, 6, and hypothesis 1.} b) \\
&\leq \|\delta_h\|_{H^1_0(\Omega)} \Big(\mathcal{F}_h + \alpha^+ \|u_I - u_\pi\|_{H^1_0(\Omega)} + \|u_\pi - u\|_{H^1_0(\Omega)} \Big) \\
&\qquad\qquad\qquad \text{(continuity of } a_h \text{ and of } a) \\
&\leq \|\delta_h\|_{H^1_0(\Omega)} \max\{\alpha^+, 1\} \Big(\mathcal{F}_h + \|u_I - u_\pi\|_{H^1_0(\Omega)} \\
&\qquad\qquad\qquad + \|u_\pi - u\|_{H^1_0(\Omega)} \Big).
\end{aligned}$$

Now, let $\tilde{C} := \max\{\alpha^+, 1\} > 0$, then:

$$\alpha^- \|\delta_h\|_{H^1_0(\Omega)} \leq \tilde{C} \Big(\mathcal{F}_h + \|u_I - u_\pi\|_{H^1_0(\Omega)} + \|u_\pi - u\|_{H^1_0(\Omega)} \Big)$$

Therefore, we finally obtain:

$$\begin{aligned}\|u-u_h\|_{H^1_0(\Omega)} &\leq \|u-u_I\|_{H^1_0(\Omega)} + \|\delta_h\|_{H^1_0(\Omega)} \text{ (triangle inequality)}\\ &\leq \left(\frac{\tilde{C}}{\alpha^-}+1\right)\left(\mathcal{F}_h + \|u_I-u_\pi\|_{H^1_0(\Omega)} + \|u_\pi - u\|_{H^1_0(\Omega)}\right).\end{aligned}$$

Consequently, we have proved statement 2 with $C = \frac{\tilde{C}}{\alpha^-}$, and the theorem is showed.

□

Remark 1.7 A few remarks on the theorem:

- The integer $k \geq 1$ will correspond to the order of accuracy of the method.
- The stability property (hypothesis 2.c) is crucial to show the existence and uniqueness of the solution since without it, we cannot show that a_h has the required properties to apply Lax-Milgram theorem.

Consequently, for every integer $k \geq 1$, our aim is now to build the subspace V_h, the symmetric bilinear form a_h and the element f_h so that they approximate well the original problem and they verify the properties of the theorem.

1.2.3 Choice of the degrees of freedom

For this whole part, we consider a certain domain decomposition T_h for some h, and K any element of T_h. In 2D, K can consequently be any simple polygon, while in 3D, K can be any polyhedron. Moreover, for any geometrical domain D, let n_e^D be the number of edges of D, n_v^D be its number of vertices, and n_f^D be its number of faces. Note that for a 1D domain, the faces, the edges and the vertices represent the same geometrical elements. As a convention, we will thus set $n_e^D = 0$ in this case, that is only if D is 1-dimensional. We also note that for a 2D domain, the faces and the edges represent the same geometrical elements as well. Furthermore, in 2D, the number of edges of D is equal to the number of vertices of D, even if here edges and vertices are two distinct geometrical entities. In 3D, however, the number of vertices and of edges is not the same.

Let us also fix an integer $k \geq 1$. Finally, for any element $K \in T_h$, we will denote E_K the set of edges of K with respect to the domain decomposition T_h, and F_K its set of faces. Note that as a consequence, $F_K = \{K\}$ if the dimension d is 1 or 2, and $E_K = \{K\}$ if $d = 1$.

We now define, for every face f_K of K, the set $\mathbb{B}_k(\partial f_K)$ as the set of continuous functions on ∂f_K that are *polygonal* on each edge of ∂f_K. That is:

$$\mathbb{B}_k(\partial f_K) := \Big\{ v \in C^0(\partial f_K) : \forall e \in E_{\partial f_K}, v|_e \in \mathbb{P}_k(e) \Big\}.$$

Proposition 1.8 *The space $\mathbb{B}_k(\partial f_K)$ is a linear space. Moreover, its dimension is equal to $n_v^{f_K} + n_e^{f_K}(k-1)$.*

Proof. First, let us prove that it is a linear space. Let $v, w \in \mathbb{B}_k(\partial f_K)$ and λ in $\mathbb{R}$. Then, since v and w are continuous on ∂f_K, then $\lambda v + w$ is also continuous on ∂f_K. Moreover, for all $e \in E_{\partial f_K}$, $v|_e, w|_e \in \mathbb{P}_k(e)$. Consequently, $(\lambda v + w)\,|_e = \lambda v|_e + w|_e \in \mathbb{P}_k(\partial f_K)$.
Now, let us compute the dimension of $\mathbb{B}_k(\partial f_K)$. We know that a polynomial of degree at most k is uniquely determined by its values in $k+1$ points:

- its values at the vertices, which gives us $n_v^{f_K}$ conditions;
- its values at $k-1$ additional points on each edge of f_K, if $\dim(f_K) > 1$, which gives us $n_e^{f_K}(k-1)$ additional conditions.

Therefore, the dimension of $\mathbb{B}_k(\partial f_K)$ is equal to $n_v^{f_K} + n_e^{f_K}(k-1)$. □

Remark 1.9 Since f_K is either a polygon (if $d = 1$ or 2) or a segment (if $d = 1$), then in any case, $n_e^{f_K} = n_f^{f_K}$, so that

$$\dim\left(\mathbb{B}_k\left(\partial f_K\right)\right) = n_v^{f_K} + n_e^{f_K}(k-1) = k n_v^{f_K} = k n_e^{f_K}.$$

We have chosen to state the proposition without using this fact to make clear the choice of degrees of freedom that will follow in this report.

Now, recall that we want to build a space V_h such that for all $K \in T_h$, we have $\mathbb{P}_k(K) \subset V_h|_K$. Moreover, we remark that every function v belongs to $\mathbb{P}_k(K)$ if and only if $\Delta v \in \mathbb{P}_{k-2}(K)$. This is the reason why we decide to introduce the following local spaces $V^{K,k}$. To define it, we separate the case $d = 1$ or 2 to the case $d = 3$.

<u>Case 1:</u> $d = 1$ or 2.
Let us define:

$$V^{K,k} := \Big\{ v \in H^1(K) : v|_{\partial K} \in \mathbb{B}_k(\partial K), \Delta v \in \mathbb{P}_{k-2}(K) \Big\}.$$

We recall that $\mathbb{P}_{-1} = \{0\}$. Note that if in $V^{K,k}$ we were asking $\Delta v|_K$ to belong to a lower-degree polynomial space, we would have $\mathbb{P}_k(K) \not\subset V^{K,k}$, which is bad since we want the opposite. Instead, if in $V^{K,k}$ we were asking $\Delta v|_K$ to belong to a higher-degree polynomial space, then we would still have in $V^{K,k}$ all the polynomials of degree at most k as well. However, it adds some useless complexity to the finite space, i.e. we would need to determine more degrees of freedom without obtaining a better method at the end.

Proposition 1.10 *The dimension $N^{K,k}$ of the space $V^{K,k}$ is equal to*

$$N^{K,k} = n_v^K + (k-1)$$

if $d = 1$, or if $d = 2$, it is equal to

$$N^{K,k} = n_v^K + n_e^K(k-1) + \frac{k(k-1)}{2}.$$

To have a first insight on this, let us first look at some particular cases. If $k = 1$, $V^{K,1}$ is the space of harmonic functions (since for every function v in $V^{K,1}, \Delta v|_K = 0$), that are linear on the edges of K. Consequently, these functions are completely determined by their values on the n_v^K vertices of K, and $N^{K,1} = n_v^K$.

If $k = 2$, $V^{K,2}$ is the space of functions that have a constant Laplacian and that are linear or quadratic on the boundary of K. Moreover, for every constant c and every boundary function $h \in \mathbb{B}_2(\partial K)$, by Lax-Milgram theorem, there is a unique function $v \in H^1(K)$ such that $\Delta v = c$ in K, and $v = h$ on ∂K. The differential problem has to be interpreted in a weak way. Consequently,

$$N^{K,2} = \dim\left(\mathbb{B}_2(\partial K)\right) + 1 = n_v^K + n_e^K + 1.$$

We give now the proof of the general case:

Proof. For every $q \in \mathbb{P}_{k-2}(K)$, and for every $h \in \mathbb{B}_k(\partial K)$, by Lax-Milgram theorem, there is a unique function $v \in H^1(K)$ such that $\Delta v = q$ in K and $v = h$ on ∂K, weakly. Consequently,

$$\begin{aligned}\dim\left(V^{K,k}\right) := N^{K,k} &= \dim\left(\mathbb{B}_k\left(\partial K\right)\right) + \dim\left(\mathbb{P}_{k-2}(K)\right) \\ &= n_v^K + n_e^K(k-1) + \binom{k-2+d}{k-2}.\end{aligned}$$

So if $d = 1$, $N^{K,k} = n_v^K + n_e^K(k-1) + (k-1) = n_v^K + (k-1)$, since $n_e^K = 0$, and if $d = 2$, we obtain $N^{K,k} = n_v^K + n_e^K(k-1) + \frac{k(k-1)}{2}$. □

Case 2: $d = 3$.
We need intermediate steps before defining $V^{K,k}$, so let us first define the following: for all $f_k \in F_K$,

$$V^{f_K,k} := \left\{v \in H^1(f_K) : v|_{\partial f_K} \in \mathbb{B}_k(f_K), \Delta v \in \mathbb{P}_{k-2}(f_K)\right\}.$$

Note that this is the exact same definition as the one of $V^{K,k}$ for $d = 1, 2$, except that this time, we define the set of functions that are H^1 on the faces of K, and not directly on the element K itself. With the exact same proof as before, we consequently know that the dimension of $V^{f_K,k}$ is equal to $n_v^{f_K} + n_e^{f_K}(k-1) + \frac{k(k-1)}{2}$.

We now enlarge this space on functions that are continuous on ∂K entirely. Since $\partial K = \cup_{f_K \in F_K} f_K$, This gives us the following definition:

$$V^{\partial K,k} := \left\{v \in C^0(\partial K) : v|_{f_K} \in V^{f_K,k}, \forall f_K \in F_K\right\}.$$

To compute the dimension of this space, we have to be careful not to take multiple times the same edge into account, since each edge is shared by two faces. That is, the dimension is not the sum over the faces f_K of the dimension of $V^{f_K,k}$. Instead, if we come back from the proof of Proposition 1.8, we get:

$$\begin{aligned}\dim\left(V^{\partial K,k}\right) &= n_v^K + n_e^K(k-1) + \sum_{f_K \in F_K}\left(\dim\left(\mathbb{P}_{k-2}\left(f_K\right)\right)\right) \\ &= n_v^K + n_e^K(k-1) + n_f^K\frac{k(k-1)}{2}.\end{aligned}$$

We are now finally able to define the space $V^{K,k}$, similarly as in the 1- and 2-dimensional cases:

$$V^{K,k} = \left\{v \in H^1(K) : v|_{\partial K} \in V^{\partial K,k}, \Delta v \in \mathbb{P}_{k-2}(K)\right\}.$$

To sum up briefly, $V^{K,k}$ is a finite dimensional subspace of $H^1(K)$ composed of functions that are polynomials of degree at most k on each edge of K, and whose gradient is a polynomial of degree at most $k-2$ in the element K itself and on each face of K.

Proposition 1.11 *The dimension $N^{K,k}$ of the space $V^{K,k}$ is equal to*

$$N^{K,k} = n_v^K + n_e^K(k-1) + n_f^K \frac{k(k-1)}{2} + \frac{k(k^2-1)}{6}.$$

Proof. Indeed,

$$\begin{aligned} N^{K,k} &= \dim\left(V^{\partial K,k}\right) + \dim\left(\mathbb{P}_{k-2}(K)\right) \\ &= n_v^K + n_e^K(k-1) + n_f^K \frac{k(k-1)}{2} + \binom{k-2+d}{k-2} \\ &= n_v^K + n_e^K(k-1) + n_f^K \frac{k(k-1)}{2} + \frac{k(k^2-1)}{6}. \end{aligned}$$

□

The dimension of the local space $V^{K,k}$ correspond to the number of degrees of freedom we have to choose to have an easy way to express any local function belonging to this space. VEM is based on the following choice. For all $v_h \in V^{K,k}$, v_h is uniquely defined if we know the following values:

- the values of v_h at each vertex of K. Let $\mathcal{V}^{K,k}$ be the set of such values;
- if $d > 1$, the values of v_h at $k-1$ intermediate points on each edge of K. Let $\mathcal{E}^{K,k}$ be the set of such values;
- if $d = 3$, the moments up to order $k-2$ of v_h on each face $f_K \in F_K$, that is $\frac{1}{|f_K|}\int_{f_K} m(\mathbf{x}) v_h(\mathbf{x})\, \mathrm{d}\mathbf{x}$, for all $m \in \mathcal{M}_{k-2}(f_K)$, where

  $$\mathcal{M}_{k-2}(D) = \left\{ \left(\frac{\mathbf{x}-\mathbf{x}_D}{h_D}\right)^{\mathbf{s}} : |\mathbf{s}| \leq k-2 \right\},$$

  with $\mathbf{x}_D$ being the barycenter of the geometrical element D, h_D its diameter, $|\mathbf{s}| = \sum_{i=1}^{d} s_i$ with $s_i \in \mathbb{N}_0$, and $\mathbf{x}^{\mathbf{s}} = \prod_{i=1}^{d} x_i^{s_i}$, x_i being the i-th coordinate of $\mathbf{x}$. Let $\mathcal{F}^{K,k}$ be the set of such values.
- the moments up to order $k-2$ of v_h in K, that is $\frac{1}{|K|}\int_K m(\mathbf{x}) v_h(\mathbf{x})\, \mathrm{d}\mathbf{x}$, for all $m \in \mathcal{M}_{k-2}(K)$, where $\mathcal{M}_{k-2}(K)$ is defined in the previous point. Let $\mathcal{P}^{K,k}$ be the set of such values.

Note that for $k = 2$, the only moment in $\mathcal{M}_{k-2}(D)$ is the mean value of v_h in D, for every geometrical domain D. Moreover, in 1D, the degrees of freedom are only $\mathcal{V}^{K,k} \cup \mathcal{P}^{K,k}$, and in 2D, the degrees of freedom are only $\mathcal{V}^{K,k} \cup \mathcal{E}^{K,k} \cup \mathcal{P}^{K,k}$. We easily verify the following proposition:

Proposition 1.12 *The total number of chosen degrees of freedom is equal to $N^{K,k}$.*

Proof. Indeed, we first see straightforward that $\mathcal{V}^{K,k}$ contains n_v^K degrees of freedom, and $\mathcal{E}^{K,k}$ contains $n_e^K(k-1)$ degrees of freedom. Then, we have to distinguish the cases $d=1$, $d=2$ and $d=3$.

<u>Case 1:</u> $d=1$.
The cardinality of $\mathcal{P}^{K,k}$ is equal to the cardinality of $\mathcal{M}_{k-2}(K)$, which is the number of multi-indices $\mathbf{s}$ such that $|\mathbf{s}| \leq k-2$ is equal to the dimension of $\mathbb{P}_{k-2}(K)$, that is $k-1$. Consequently,

$$\#\left(\mathcal{V}^{K,k} \cup \mathcal{P}^{K,k}\right) = n_v^K + (k-1) = \dim\left(V^{K,k}\right).$$

<u>Case 2:</u> $d=2$.
In this case, we do not consider $\mathcal{F}^{K,k}$, but we do consider $\mathcal{P}^{K,k}$. Moreover, the number of multi-indices $\mathbf{s}$ such that $|\mathbf{s}| \leq k-2$ is also equal to the dimension of $\mathbb{P}_{k-2}(K)$, but with K that is now in 2 dimensions. So that $\#(\mathcal{M}_{k-2}(K)) = \frac{k(k-1)}{2}$. Consequently,

$$\#\left(\mathcal{V}^{K,k} \cup \mathcal{E}^{K,k} \cup \mathcal{P}^{K,k}\right) = n_v^K + n_e^K(k-1) + \frac{k(k-1)}{2} = \dim\left(V^{K,k}\right).$$

<u>Case 3:</u> $d=3$.
In this case, the cardinal of $\mathcal{P}^{K,k}$ is also equal to the number of multi-indices $\mathbf{s}$ such that $|\mathbf{s}| \leq k-2$, that is to the dimension of $\mathbb{P}_{k-2}(K)$. But this time, K is in 3 dimensions, so $\dim\left(\mathbb{P}_{k-2}(K)\right) = \frac{k(k^2-1)}{6}$. Moreover, we also have to consider the set $\mathcal{F}^{K,k}$ of degrees of freedom. This set has cardinality $n_f^K \cdot \#\left(\mathcal{M}_{k-2}(f_K)\right) = n_f^K \frac{k(k-1)}{2}$, since f_K is a 2-dimensional geometrical object. Consequently, we obtain:

$$\begin{aligned}
&\#\left(\mathcal{V}^{K,k} \cup \mathcal{E}^{K,k} \cup \mathcal{F}^{K,k} \cup \mathcal{P}^{K,k}\right) \\
&= n_v^K + n_e^K(k-1) + n_f^K\frac{k(k-1)}{2} + \frac{k(k^2-1)}{6} \\
&= \dim\left(V^{K,k}\right).
\end{aligned}$$

□

Remark 1.13 On one hand, the degrees of freedom in $\mathcal{V}^{K,k}$ and in $\mathcal{E}^{K,k}$ uniquely determine a polynomial of degree at most k on each edge of K. That is, v_h is uniquely determined on the edges of K by those two sets of degrees of freedom. On the other hand, $\mathcal{F}^{K,k}$ and $\mathcal{P}^{K,k}$ determine the L^2-projection of v_h onto $\mathbb{P}_{k-2}(f_K)$ for all $f_K \in F_K$, and onto $\mathbb{P}_{k-2}(K)$, respectively.

Let us show now this important proposition:

Proposition 1.14 *The degrees of freedom of $\mathcal{V}^{K,k} \cup \mathcal{E}^{K,k} \cup \mathcal{F}^{K,k} \cup \mathcal{P}^{K,k}$ are unisolvent for $V^{K,k}$. This means that for any values of the degrees of freedom, there exist a unique v_h in $V^{K,k}$ defined from them.*

Proof. In remark 1.13, we have seen that from the degrees of freedom, we can define a v_h in $V^{K,k}$. It thus remains to prove that the map that sends the degrees of freedom onto a function of $V^{K,k}$ is injective.

Let us call P_{k-2}^D the projection operator, in the $L^2(D)$-norm, onto the space $\mathbb{P}_{k-2}(D)$, for any space D. We thus want to show that any function $v_h \in V^{K,k}$ such that:

1. $v_h = 0$ on $\cup_{e \in E_K} \bar{e}$,
2. $P^{f_K}_{k-2} v_h = 0$ in f_K, for all $f_K \in F_K$,
3. $P^{K}_{k-2} v_h = 0$ in K,

is identically zero in K. In particular, condition 1. means that $v_h = 0$ on ∂f_K, for all $f_K \in F_K$.

To do so, we first show that $\Delta v_h|_{f_K} = 0$ in f_K, for all $f_K \in K$. So that together with 1., we have the following problem for all $f_K \in F_K$: find $w_h \in V^{f_K,k}$ such that $\Delta w_h = 0$ in f_K and $w_h = 0$ on $\partial f_K \subset E_K$. We can easily see that the function $w_h \equiv 0$ is solution of this problem. Moreover, by Strang theorem, the problem admits a unique solution. Consequently, the only solution is the identically zero solution in $V^{f_K,K}$. This would thus show that $v_h|_{f_K}$ is zero in $V^{f_K,K}$, for all $f_K \in F_K$. Therefore, $v_h|_{\partial K} \equiv 0$.

Then, if we show that $\Delta v_h = 0$ in the whole element K, then together with the preceding result, we would have the following problem: find v_h in $V^{K,k}$ such that $\Delta v_h = 0$ in K and $v_h = 0$ on ∂K. Again by Strang theorem, the problem admits a unique solution, and the identically zero function in $V^{K,k}$ is this solution. Consequently, v_h is identically zero in K, and thus the degrees of freedom chosen above are unisolvent for $V^{K,k}$.

It thus remains to show the two points just mentioned. So let $f_K \in F_K$ and let $v_h \in V^{K,k}$ such that v_h verifies 1., 2. and 3. In the following, we will write v_h instead of $v_h|_{f_K}$ in order to lighten the proof. We will first solve the following auxiliary problem: $\forall q \in \mathbb{P}_{k-2}(f_K)$, find $w \in H^1_0(f_K)$ such that

$$a^{f_K}(w, v) = \int_{f_K} qv \, \mathrm{d}f_K, \text{ for all } v \in H^1_0(f_K), \tag{12}$$

which can also be written as $-\Delta w = q$ in f_K, $w = 0$ on ∂f_K but it is meant in a weak form. Since we know by Lax-Milgram theorem that there exists a unique solution w to this problem, we will write this solution as $w = -\Delta^{-1}_{L^2(f_K)}(q)$. Indeed, since we look at functions in $H^1_0(f_K)$, then the bilinear form a^{f_K} is continuous and coercive, and the proof is similar as the one done on the bilinear form a (see equation 4). Now, let us define the map $R : \mathbb{P}_{k-2}(f_K) \to \mathbb{P}_{k-2}(f_K)$ such that:

$$R(q) := P^{f_K}_{k-2}\left(-\Delta^{-1}_{L^2(f_K)}(q)\right) = P^{f_K}_{k-2}(w).$$

Let us see that R is injective. Indeed, for every $q \in \mathbb{P}_{k-2}(f_K)$,

$$\begin{aligned}
\int_{f_K} qR(q) \, \mathrm{d}f_K &= \int_{f_K} qP^{f_K}_{k-2}\left(-\Delta^{-1}_{L^2(f_K)}(q)\right) \mathrm{d}f_K \\
&= \int_{f_K} qP^{f_K}_{k-2}(w) \, \mathrm{d}f_K \qquad \left(\text{by definition of } \Delta^{-1}_{L^2(f_K)}\right) \\
&= \int_{f_K} qw \, \mathrm{d}f_K \quad \left(\text{since } q \in \mathbb{P}_{k-2}(f_K) \text{ and definition of } P^{f_K}_{k-2}\right) \\
&= a^{f_K}(w, w). \qquad (\text{equation 12 with } v = w)
\end{aligned}$$

So since $w \in H^1_0(f_K)$, we have:

$$R(q) = 0 \Leftrightarrow a^{f_K}(w, w) = 0 \Leftrightarrow w = 0 \ (a^{f_K} \text{ coercive bilinear form}) \ \Leftrightarrow q = 0.$$

The implication left-to-right of the last implication is obvious since any identically zero function has a (weak) Laplacian equal to 0. The implication right-to-left comes from Lax-Milgram theorem and from the fact that the identically zero function in f_K is zero on the boundary of f_K and has a Laplacian equal to zero in f_K. Consequently, the map R is injective. Moreover, since $v_h = 0$ on ∂f_K, then $v_h \in H_0^1(f_K)$, and since $\Delta v_h \in \mathbb{P}_{k-2}(f_K)$, then:

$$P_{k-2}^{f_K}(v_h) = P_{k-2}^{f_K}\left(-\Delta_{L^2(f_K)}^{-1}\left(-\Delta v_h\right)\right) = R\left(-\Delta v_h\right).$$

The first equality comes from the definition of $\Delta_{L^2(f_K)}^{-1}$, while the second one comes from the definition of R. Finally, we get

$$P_{k-2}^{f_K}(v_h) = 0 \Rightarrow R\left(-\Delta v_h\right) = 0 \Rightarrow -\Delta v_h = 0.$$

Therefore, we have just proved that for all $f_K \in F_K$, then $\Delta v_h|_{f_K} = 0$. Consequently, following what we have said before, $v_h|_{\partial K} \equiv 0$.

We can now repeat the exact same proof with v_h instead of $v_h|_{f_K}$, a^K instead of a^{f_K}, and with K instead of f_K, to show that we get $\Delta v_h = 0$ in K, knowing that $v_h = 0$ in ∂K and assuming hypothesis 3. introduced at the beginning of the proof. Therefore, v_h is identically zero in K, and thus the map that sends the degrees of freedom onto a function of $V^{K,k}$ is injective. Consequently, the degrees of freedom are unisolvent for $V^{K,k}$. $\square$

Remark 1.15 We emphasis the fact that the Laplace operator present in the definition of $V^{K,k}$ is here the most natural choice, but it could be replaced by any other second-order elliptic operator. The only mandatory and required properties on $V^{K,k}$ are that:

- $\dim\left(V^{K,k}\right) = N^{K,k}$,
- $\mathbb{P}_k(K) \subset V^{K,k}$,
- $V^{K,k}$ is made of functions that are in polynomials of degree at most k on each edge,
- the degrees of freedom $\mathcal{V}^{K,k} \cup \mathcal{E}^{K,k} \cup \mathcal{F}^{K,k} \cup \mathcal{P}^{K,k}$ are unisolvent for $V^{K,k}$.

1.2.4 Construction of the discretized space V_h

Now, we have all the tools to build the discretized space V_h on the whole space Ω, that verifies all conditions of Theorem 1.5, as desired. So given a space decomposition T_h into simple polygons, and for every integer $k \geq 1$, we define $V_h := \bigcup_{K \in T_h} V^{K,k}$, with the right homogeneous Dirichlet boundary conditions imposed. More precisely, we define V_h independently for the case 1/2D or 3D:

Case 1: $d = 1$ or 2.

$$V_h := \bigcup_{K \in T_h} V^{K,k} = \left\{v \in H_0^1(\Omega) : v|_{\partial K} \in \mathbb{B}_k(\partial K) \text{ and } \Delta v|_K \in \mathbb{P}_{k-2}(K), \forall K \in T_h\right\}.$$

Case 2: $d = 3$.

$$V_h := \bigcup_{K \in T_h} V^{K,k} = \left\{v \in H_0^1(\Omega) : v|_{\partial K} \in V^{\partial K,k} \text{ and } \Delta v|_K \in \mathbb{P}_{k-2}(K), \forall K \in T_h\right\}.$$

Proposition 1.16 *The dimension N^{tot} of V_h is equal to:*

- $N^{tot} = N_v + N_p(k-1)$ *if* $d = 1$*;*
- $N^{tot} = N_v + N_e(k-1) + N_p\frac{k(k-1)}{2}$ *if* $d = 2$*;*
- $N^{tot} = N_v + N_e(k-1) + N_f\frac{k(k-1)}{2} + N_p\frac{k(k^2-1)}{6}$ *if* $d = 3$*,*

where N_v is the total number of internal vertices, N_e is the total number of internal edges, N_f is the total number of internal faces, and N_p is the total number of elements in T_h.

Proof. This is deduced in a straightforward manner from Propositions 1.10 and 1.11, keeping in mind that we impose homogeneous Dirichlet boundary conditions. This is why we only consider *internal* vertices, edges and faces. □

Then, the global degrees of freedom in V_h are chosen in the same way as the local degrees of freedom. For all $v_h \in V_h$, v_h is uniquely determined if we know the following values:

- the values of v_h at each internal vertex of T_h. Let $\mathcal{V}$ be the set of such values;
- if $d > 1$, the values of v_h at $k-1$ intermediate points on each internal edge of T_h. Let $\mathcal{E}$ be the set of such values;
- if $d = 3$, the moments up to order $k-2$ of v_h on each internal face f of T_h, that is $\frac{1}{|f|}\int_f m(\mathbf{x})v_h(\mathbf{x})\,\mathrm{d}\mathbf{x}$, for all $m \in \mathcal{M}_{k-2}(f)$, where as before,

$$\mathcal{M}_{k-2}(D) = \left\{ \left(\frac{\mathbf{x}-\mathbf{x}_D}{h_D}\right)^{\mathbf{s}} : |\mathbf{s}| \leq k-2 \right\},$$

  with $\mathbf{x}_D$ being the barycenter of the geometrical element D, h_D its diameter, $|\mathbf{s}| = \sum_{i=1}^d s_i$ with $s_i \in \mathbb{N}_0$, and $\mathbf{x}^{\mathbf{s}} = \prod_{i=1}^d x_i^{s_i}$, x_i being the i-th coordinate of $\mathbf{x}$. Let $\mathcal{F}$ be the set of such values.

- the moments up to order $k-2$ of v_h in every element $K \in T_h$, that is $\frac{1}{|K|}\int_K m(\mathbf{x})v_h(\mathbf{x})\,\mathrm{d}\mathbf{x}$, for all $m \in \mathcal{M}_{k-2}(K)$, where $\mathcal{M}_{k-2}(K)$ is defined in the previous point. Let $\mathcal{P}$ be the set of such values.

Finally, the homogeneous Dirichlet boundary conditions impose $v_h = 0$ on the vertices, edges and faces belonging to $\partial\Omega$. In the exact same way as for Proposition 1.12, it is straightforward to verify that the total number of chosen degrees of freedom (that is $\#\mathcal{P} + \#\mathcal{V} + \#\mathcal{E} + \#\mathcal{F}$) is indeed equal to the dimension N^{tot} of V_h. Moreover, Proposition 1.14 implies that the degrees of freedom $\mathcal{P} \cup \mathcal{V} \cup \mathcal{E} \cup \mathcal{F}$ are unisolvent for V_h.

Consequently, since our aim is to be able to apply Theorem 1.5, it remains to build the symmetric bilinear form a_h and the right-hand side element f_h so that they approximate well the original problem and verify the properties needed.

1.2.5 Projection operator Π^∇

To be able to build the symmetric bilinear form a_h, especially in 3 dimensions, we need to introduce the projection operator $\Pi^\nabla : V^{K,k} \to \mathbb{P}_k(K)$, for all integer $k \geq 1$ and for all element $K \in T_h$. To be more precise, we should write this operator as $\Pi^\nabla_{K,k}$, but for a sake of simplicity, we will stick to the notation Π^∇. As before, we write n_v^K the number of vertices of K. The operator is defined through the following orthogonality condition: for all $v \in V^{K,k}$,

$$\int_K \nabla p \cdot \nabla \left(\Pi^\nabla v - v\right) \mathrm{d}K = 0, \text{ for all } p \in \mathbb{P}_k(K), \tag{13}$$
$$\begin{cases} \frac{1}{n_v^K} \sum_{i=1}^{n_v^K} v(v_i) = \frac{1}{n_v^K} \sum_{i=1}^{n_v^K} \Pi^\nabla v(v_i) \text{ if } k = 1; \\ \text{or} \\ \frac{1}{|K|} \int_K v \,\mathrm{d}K = \frac{1}{|K|} \int_K \Pi^\nabla v \,\mathrm{d}K \text{ if } k \geq 2, \end{cases}$$

where $\{v_i\}_{i=1}^{n_v^K}$ is the set of vertices of K.

This projection operator is defined so that for every $v_h \in V^{K,k}$, we can compute $\Pi^\nabla v_h$ using only the degrees of freedom of v_h, defined in the previous section. The proof of this fact can be found in [4].

Note that the projection operator Π corresponds to the a^K-orthogonal projection on $\mathbb{P}_k(K)$. Moreover, if we take $v = q \in \mathbb{P}_k(K) \subset V^{K,k}$, then $\Pi^\nabla q = q$, so in particular,

$$\begin{aligned} \|\nabla q\|^2_{L^2(K)} &= \int_K (\nabla q)^2 \,\mathrm{d}K \\ &\overset{(13)}{=} \int_K \nabla q \cdot \nabla \left(\Pi^\nabla q\right) \mathrm{d}K \\ &\overset{(13)}{=} \int_K \nabla \left(\Pi^\nabla q\right) \cdot \nabla \left(\Pi^\nabla q\right) \mathrm{d}K \\ &= \|\nabla \left(\Pi^\nabla q\right)\|^2_{L^2}. \end{aligned}$$

Consequently, $\Pi^\nabla q = q + C$, where C is a constant, for all $q \in \mathbb{P}_k(K)$. But thanks to the second equation of (13), this constant is equal to 0, so that $\Pi^\nabla q = q$. Let us see in the next section why this relation is useful.

1.2.6 Construction of the symmetric bilinear form a_h

Let us recall that for all $K \in T_h$, for all $v, w \in V = H_0^1(\Omega)$, $a^K(w, v) = \int_K \nabla w \cdot \nabla v \,\mathrm{d}K$ and $a(w, v) = \sum_{K \in T_h} a^K(w, v)$. Now, for any element K of T_h, if we fix an integer $k \geq 1$, then for all $v \in V^{K,k}$ and for all $p \in \mathbb{P}_k(K)$,

$$a^K(p, v) = \int_K \nabla p \cdot \nabla v \,\mathrm{d}K = -\int_K \Delta p v \,\mathrm{d}K + \int_{\partial K} \frac{\partial p}{\partial n} v \,\mathrm{d}s, \tag{14}$$

thanks to an integration by parts. Since $\Delta p \in \mathbb{P}_{k-2}(K)$, Δp can be written with respect to the basis of $\mathbb{P}_{k-2}(K)$ made by $\mathcal{M}_{k-2}(K)$, so that it is a linear combination of moments. Then, thanks to the chosen degrees of freedom, the first integral of the right hand side of equation (14) can be exactly computed.

Moreover, for every $e \in \partial K$, $\frac{\partial p}{\partial n} \in \mathbb{P}_{k-1}(e)$, and if $d = 1$ or $d = 2$, $v \in \mathbb{P}_k(e)$. So $\frac{\partial p}{\partial n} v \in \mathbb{P}_{2k-1}(e)$. In the case $d = 1$, ∂K is composed of 2 points, i.e. vertices and edges are equal. So since we know v_h at each internal vertex of T_h, then the second integral of the right hand side of equation (14) can be exactly computed. In the case $d = 2$, ∂K is composed of edges, and we know v on k points on each edge $e \in \partial K$ thanks to the chosen degrees of freedom. So if we choose an appropriate quadrature formula (Gauss-Legendre-Lobatto is fine), we can also compute exactly the second integral of the right hand side of equation (14). A good choice of degrees of freedom on each edge of T_h would thus be the $k-1$ internal Gauss-Legendre-Lobatto points, plus the vertices.

Finally, if $d = 3$, ∂K is composed of faces. Unlike the case $d = 1$ or 2, on any face, v does not necessarily belong to $\mathbb{P}_k(e)$. Consequently, to be able to compute exactly $\int_{\partial K} \frac{\partial p}{\partial n} v \, \mathrm{d}s$, we need to slightly modify the definition of $V^{K,k}$ for $d = 3$. Recall that $V^{K,k}$ has been defined from $V^{f_K,k}$ for all $f_K \in F_K$, so that we now make the following choice:

$$V^{f_K,k} := \Bigg\{ v \in H^1(f_K) : v|_{\partial f_K} \in \mathbb{B}_k(\partial f_K), \Delta v \in \mathbb{P}_k(f_K), \\ \int_{f_K} v(\mathbf{x}) m(\mathbf{x}) \, \mathrm{d}\mathbf{x} = \int_{f_K} \Pi^\nabla v(\mathbf{x}) m(\mathbf{x}) \mathrm{d}\mathbf{x}, \forall m \in \mathcal{M}_{k-1,k}(f_K) \Bigg\},$$

where we recall again that

$$\mathcal{M}_{k-1,k}(f_K) = \left\{ \left(\frac{\mathbf{x} - \mathbf{x}_{f_K}}{h_{f_K}} \right)^{\mathbf{s}} : |\mathbf{s}| = k-1, k \right\},$$

with $\mathbf{x}_{f_K}$ being the barycenter of the face f_K, h_{f_K} its diameter, $|\mathbf{s}| = \sum_{i=1}^d s_i$ with $s_i \in \mathbb{N}_0$, and $\mathbf{x}^{\mathbf{s}} = \prod_{i=1}^d x_i^{s_i}$, x_i being the i-th coordinate of $\mathbf{x}$. Note that here, the projection operator Π^∇ projects functions belonging to $V^{f_K,k}$ into $\mathbb{P}_k(f_K)$, for any $f_K \in F_K$. Unlike the previous definition, the Laplacian is not required to be in $\mathbb{P}_{k-2}$ but in $\mathbb{P}_k$, and extra conditions on the moments of the function are required.

Now we have to check that the dimension of $V^{f_K,k}$ is still the same as before. To know the dimension of $V^{f_K,k}$, we first look more specifically into the last two conditions. $\Delta v \in \mathbb{P}_k(f_K)$ gives us $\dim(\mathbb{P}_k(f_K))$ conditions. Moreover, we know that the moments on f_K up to some order $\tilde{k}$ form a basis of $\mathbb{P}_{\tilde{k}}(f_K)$. Consequently, imposing the last condition of the definition of $V^{f_K,k}$ for all $m \in \mathcal{M}_{k-1,k}(f_K)$ and imposing $\Delta v \in \mathbb{P}_k(f_K)$ is equivalent to being left with $\dim(\mathbb{P}_{k-2}(f_K))$ degrees of freedom. Therefore,

$$\dim\left(V^{f_K,k}\right) = \dim\left(\mathbb{B}_k\left(\partial f_K\right)\right) + \dim\left(\mathbb{P}_{k-2}(f_K)\right) = n_v^{f_K} + n_e^{f_K}(k-1) + \frac{k(k-1)}{2}.$$

For a more rigorous proof of this, and to prove the existence of functions in $V^{f_K,k}$, please refer to [1]. Note moreover that $V^{K,k}$ built from $V^{f_K,k}$ can still be described with the same degrees of freedom as previously, and it still contains all the polynomials of degree k.

Thanks to this new definition which will always be used from now on, one can prove that in the 3D case, the second integral of the right hand side of equation (14) can be exactly computed from the chosen degrees of freedom. Indeed, since $\frac{\partial p}{\partial n} \in \mathbb{P}_{k-1}(f)$, for all face $f \in \partial K$, and since $\mathcal{M}_{k-1}(f)$ is a basis for $\mathbb{P}_{k-1}(f)$, then $\frac{\partial p}{\partial n}$ can be decomposed into a linear combination of moments of order at most $k-1$. Then, thanks to the last property of $V^{f,k}$, we know that for all $m \in \mathcal{M}_{k-1}(f)$, $\int_f v(\mathbf{x})m(\mathbf{x})\,\mathrm{d}\mathbf{x} = \int_f \Pi^\nabla v(\mathbf{x})m(\mathbf{x})\mathrm{d}\mathbf{x}$. Any considered moment has degree at most $k-1$, and $\Pi^\nabla v$ is by definition a polynomial of degree at most k. So $\int_{\partial K} \frac{\partial p}{\partial n} v\,\mathrm{d}s$ can be written as a linear combination of integrals of polynomials of degree at most $2k-1$. Since we know from section 1.2.5 that we can exactly compute $\Pi^\nabla v$ using only the degrees of freedom of v, then if we choose an appropriate quadrature formula (as Gauss-Legendre-Lobatto), we can compute exactly the second integral of the right hand side of equation (14).

Consequently, with the chosen degrees of freedom and for $d = 1, 2$ or 3, it is possible to compute exactly the value of $a^K(p,v)$ for any $K \in T_h$, $p \in \mathbb{P}_k(K)$ and $v \in V^{K,k}$. This result will be used to build a computable symmetric bilinear form a_h that satisfies the conditions of Theorem 1.5.

Let $k \geq 1$ and $K \in T_h$. Now, we want to build a_h such that it verifies the conditions of Theorem 1.5, in particular the k-consistency and the stability properties that we recall here:

- (k-consistency) for all $p \in \mathbb{P}_k(K)$, for all $v_h \in V_h|_K$, the relation $a_h^K(p, v_h) = a^K(p, v_h)$ is verified;
- (stability) there exist $\alpha^-, \alpha^+ \geq 0$ two constants independent of h and of K such that for all $v_h \in V_h|_K$,
$$\alpha^- a^K(v_h, v_h) \leq a_h^K(v_h, v_h) \leq \alpha^+ a^K(v_h, v_h).$$

In section 1.2.5, we have seen that for all $q \in \mathbb{P}_k(K)$, $\Pi^\nabla q = q$, so that a natural choice to satisfy the k-consistency would be to take for all $u, v \in V^{K,k}$, $a_h^K(u,v) = a^K(\Pi^\nabla u, \Pi^\nabla v)$. However, in this case, stability cannot always be verified, so that an other term needs to be added, as explained in the following theorem.

Theorem 1.17 *Let S^K be a symmetric bilinear form such that there exist two constants $c_0, c_1 > 0$ independent from K and h that verify*

$$c_0 a^K(v,v) \leq S^K(v,v) \leq c_1 a^K(v,v), \forall v \in V^{K,k} \text{ with } \Pi^\nabla v = 0. \tag{15}$$

If for all $u, v \in V^{K,k}$, a_h^K is defined by

$$a_h^K(u,v) := a^K(\Pi^\nabla u, \Pi^\nabla v) + S^K(u - \Pi^\nabla u, v - \Pi^\nabla v),$$

then a_h^K is a bilinear form that satisfies both the k-consistency and the stability properties.

Remark 1.18 Note that the definition of a_h^K in the previous theorem is inspired by the Pythagoras theorem on a^K, applied as follows. Since Π^∇ is the a^K-orthogonal projection of $V^{K,k}$ on $\mathbb{P}_k(K)$, we have for all $u, v \in V^{K,k}$,

$$a^K(u,v) = a^K(\Pi^\nabla u, \Pi^\nabla v) + a^K(u - \Pi^\nabla u, v - \Pi^\nabla v).$$

Proof. First, a_h^K is a symmetric bilinear form since a^K and S^K are, and since Π^∇ is linear too. Then, for all $p \in \mathbb{P}_k(K)$, we know that $p = \Pi^\nabla p$, so $S^K(p - \Pi^\nabla p, v - \Pi^\nabla v) = 0$, for all $v \in V^{K,k}$. Consequently, by definition of a_h^K and of Π^∇,

$$a_h^K(p,v) = a^K(\Pi^\nabla p, \Pi^\nabla v) = a^K(p, \Pi^\nabla v) = a^K(p,v),$$

that is the k-consistency property is verified.

Finally, for all $v \in V^{K,k}$, $\Pi^\nabla\left(\Pi^\nabla v - v\right) = \Pi^\nabla v - \Pi^\nabla v = 0$ by linearity of a^K and since $\Pi^\nabla v \in \mathbb{P}_k(K)$. So thanks to the definition of a_h^K and to relation (15),

$$\begin{aligned} a_h^K(v,v) &\leq a^K(\Pi^\nabla v, \Pi^\nabla v) + c_1 a^K(v - \Pi^\nabla v, v - \Pi^\nabla v) \\ &\leq \max\{1, c_1\}\big(a^K(\Pi^\nabla v, \Pi^\nabla v) + a^K(v - \Pi^\nabla v, v - \Pi^\nabla v)\big). \end{aligned}$$

Very similarly, for all $v \in V^{K,k}$,

$$\begin{aligned} a_h^K(v,v) &\leq a^K(\Pi^\nabla v, \Pi^\nabla v) + c_0 a^K(v - \Pi^\nabla v, v - \Pi^\nabla v) \\ &\geq \min\{1, c_0\}\big(a^K(\Pi^\nabla v, \Pi^\nabla v) + a^K(v - \Pi^\nabla v, v - \Pi^\nabla v)\big). \end{aligned}$$

Consequently, a_h^K also verifies the stability property. □

It remains to find a suitable symmetric positive definite bilinear form S^K. In order to verify relation (15), we need S^K to scale like a^K on $\mathrm{Ker}(\Pi^\nabla)$. We recall that $N^{K,k} = \dim(V^{K,k})$. Let $\{\phi_i\}_{i=1}^{N^{K,k}}$ be the Lagrangian basis of $V^{K,k}$, that is $\mathrm{dof}_i(\phi_i) = \delta_{ij}, \forall i,j = 1,\ldots,N^{K,k}$, where $\mathrm{dof}_i : V^{K,k} \to \mathbb{R}$ maps each function of $V^{K,k}$ to its i-th degree of freedom. Before defining S^K, we need two lemmas.

Lemma 1.19 *All the degrees of freedom scale as* 1*, that is they are invariant under rescaling of the elements of the mesh.*

Proof. Let $\hat{K}$ be an element of T_h and consider the change of variable $\mathbf{x} = h\hat{\mathbf{x}}$ that maps the element $\hat{K}$ onto an element K. For each basis function $\hat{\phi}_i$, $i = 1,\ldots,N^{\hat{K},k}$, we define ϕ_i on K as $\phi_i(\mathbf{x}) = \phi_i(h\hat{\mathbf{x}}) := \hat{\phi}_i(\hat{\mathbf{x}}) = \hat{\phi}_i\left(\frac{\mathbf{x}}{h}\right)$, and $N^{K,k} = N^{\hat{K},k}$. If $\hat{\phi}_i$ is a basis function corresponding to a vertex or an edge-point degree of freedom, then $\mathrm{dof}_i(\phi_i) = \mathrm{dof}_i(\hat{\phi}_i) = 1$ since the change of variable maps vertices of $\hat{K}$ to vertices of K and edge-points of $\hat{K}$ to edge-points of K. If instead, $\hat{\phi}_i$ is a basis function corresponding to a moment degree of freedom, then taking $\hat{D} = \hat{f}$ any face of $\hat{K}$ (if $d = 3$) and $m = 2$, or $\hat{D} = \hat{K}$ and $m = 3$, and D the image of $\hat{D}$ by the change of variables, then: for all $|\mathbf{s}| \leq k - 2$,

$$\begin{aligned} 1 &= \frac{1}{|\hat{D}|}\int_{\hat{D}} \hat{\phi}_i(\hat{\mathbf{x}}) \left(\frac{\hat{\mathbf{x}} - \hat{\mathbf{x}}_{\hat{D}}}{h_{\hat{D}}}\right)^{\mathbf{s}} \mathrm{d}\hat{\mathbf{x}} \\ &= \frac{h^m}{|D|}\int_D \phi_i(\mathbf{x}) \left(\frac{h\left(\mathbf{x} - \mathbf{x}_D\right)}{h h_D}\right)^{\mathbf{s}} \frac{1}{h^m}\,\mathrm{d}\mathbf{x} \\ &= \frac{1}{|D|}\int_D \phi_i(\mathbf{x}) \left(\frac{\mathbf{x} - \mathbf{x}_D}{h_D}\right)^{\mathbf{s}} \mathrm{d}\mathbf{x}. \end{aligned}$$

□

Lemma 1.20 *Given $h > 0$ the size of a mesh decomposition T_h, K an element of T_h and $k \geq 1$ an integer, then $h^{2-d}a^K(\phi_i,\phi_i)$ scales like 1 for all $i = 1,\ldots,N^{K,k}$. That is, $h^{2-d}a^K(\phi_i,\phi_i)$ is independent from K and h, or equivalently, it is invariant under rescaling the elements of the mesh.*

Proof. Let $\hat{K}$ be an element of $T_{\hat{H}}$ and consider the change of variables $\mathbf{x} = h\hat{\mathbf{x}}$ that maps the element $\hat{K}$ onto an element K. More generally, this change of variables maps any element of the mesh $T_{\hat{H}}$ onto an element of the mesh T_H, where $H = h\hat{H}$. As in the previous lemma, for each basis function $\hat{\phi}_i$, $i = 1, \ldots, N^{\hat{K},k}$, we define ϕ_i on K as $\phi_i(\mathbf{x}) = \phi_i(h\hat{\mathbf{x}}) := \hat{\phi}_i(\hat{\mathbf{x}}) = \hat{\phi}_i\left(\frac{\mathbf{x}}{h}\right)$, and $N^{K,k} = N^{\hat{K},k}$. Then for all $i = 1, \ldots, N^{K,k}$,

$$\begin{aligned}
\hat{H}^{2-d} a^{\hat{K}}(\hat{\phi_i, \phi_i}) &= \hat{H}^{2-d} \int_{\hat{K}} |\nabla \hat{\phi}_i|^2 \, \mathrm{d}\hat{K} \\
&= \frac{H^{2-d}}{h^{2-d}} \int_K |\nabla \phi_i h|^2 \frac{1}{h^d} \, \mathrm{d}K \\
&= H^{2-d} \int_K |\nabla \phi_i|^2 \, \mathrm{d}K.
\end{aligned}$$

□

Thanks to the two previous lemma, we can now choose correctly the symmetric bilinear form S^K introduced in Theorem 1.17.

Proposition 1.21 *$S^K : V^{K,k} \times V^{K,k} \to \mathbb{R}$ defined by $S^K(u, v) = h^{d-2} \sum_{r=1}^{N^{K,k}} \mathrm{dof}_r(u)\mathrm{dof}_r(v)$, for all $u, v \in V^{K,k}$, is a symmetric bilinear form that verifies condition (15) of Theorem 1.17.*

Proof. It is straightforward to see that S^K is symmetric. Moreover, since the functions dof_i are linear for all $i = 1, \ldots, N^{K,k}$, then S^K is bilinear. It remains to prove condition (15). Let $v \in V^{K,k} \cap \mathrm{Ker}(\Pi^\nabla)$. Since $\{\phi_i\}_{i=1}^{N^{K,k}}$ is a basis of $V^{K,k}$, then there exist $v_1, \ldots, v_{N^{K,k}} \in \mathbb{R}$ such that $v = \sum_{i=1}^{N^{K,k}} v_i \phi_i$. Consequently,

$$S^K(v, v) = h^{d-2} \sum_{r=1}^{N^{K,k}} \left(\mathrm{dof}_r(v)\right)^2 = h^{d-2} \sum_{r=1}^{N^{K,k}} v_r^2.$$

Moreover,

$$\begin{aligned}
a^K(v, v) &= \sum_{r=1}^{N^{K,k}} v_r^2 a^K(\phi_r, \phi_r) \\
&\leq \max_r \left\{ a^K(\phi_r, \phi_r) \right\} \sum_{r=1}^{N^{K,k}} v_r^2 \\
&= \max_r \left\{ h^{2-d} a^K(\phi_r, \phi_r) \right\} h^{d-2} \sum_{r=1}^{N^{K,k}} v_r^2; \\
a^K(v, v) &= \sum_{r=1}^{N^{K,k}} v_r^2 a^K(\phi_r, \phi_r) \\
&\geq \min_r \left\{ a^K(\phi_r, \phi_r) \right\} \sum_{r=1}^{N^{K,k}} v_r^2 \\
&= \min_r \left\{ h^{2-d} a^K(\phi_r, \phi_r) \right\} h^{d-2} \sum_{r=1}^{N^{K,k}} v_r^2.
\end{aligned}$$

Since we know from 1.20 that $h^{2-d}a^K(\phi_r, \phi_r)$ is independent from h and K for all $r = 1, \dots, N^{K,k}$, then $\frac{1}{c_1} := \min_r \left\{h^{2-d}a^K(\phi_r, \phi_r)\right\}$ and $\frac{1}{c_0} := \max_r \left\{h^{2-d}a^K(\phi_r, \phi_r)\right\}$ are two constants independent from h, K and r. Moreover, $h^{d-2}\sum_{r=1}^{N^{K,k}} v_r^2 = S^K(v, v)$, so that:

$$c_0 a^K(v, v) \leq S^K(v, v) \leq c_1 a^K(v, v),$$

and thus S^K verifies property (15). $\square$

To sum up, we have been able to define a_h^K by

$$a_h^K(u, v) := a^K(\Pi^\nabla u, \Pi^\nabla v) + S^K(u - \Pi^\nabla u, v - \Pi^\nabla v),$$

for all $u, v \in V^{K,k}$, so that it verifies all the conditions of Theorem 1.5. The last remaining part to apply this theorem is to build the right-hand side element f_h.

1.2.7 Construction of the right-hand side f_h

In this part is presented the choice made to approximate F from the weak formulation of the differential problem (3). Since F is a functional on the Hilbert space $V = H_0^1(\Omega)$ (the considered norm in V is the semi-norm of $H^1(\Omega)$), we know by Riesz representation theorem that there is a unique element $x_f \in V$ such that for all $v \in V$, $F(v) = \int_\Omega \nabla v \cdot \nabla x_f \, \mathrm{d}\Omega$. We separate the case $k = 1$ to the case $k \geq 2$ for technical reasons that will become obvious in the following.

Case 1: $k \geq 2$.
f_h is defined as the $L^2(K)$-projection of x_f onto the space $\mathbb{P}_{k-2}(K)$, on each $K \in T_h$. Let us write it $f_h := P_{k-2}^K x_f$, for all $K \in T_h$. In this way, we have: for all $v_h \in V_h$,

$$\langle f_h, v_h \rangle = \sum_{K \in T_h} \int_K f_h v_h \, \mathrm{d}K := \sum_{K \in T_h} \int_K \left(P_{k-2}^K x_f\right) v_h \, \mathrm{d}K \tag{16}$$

Since for all $K \in T_h$, $P_{k-2}^K x_f \in \mathbb{P}_{k-2}(K)$ and $\mathcal{M}_{k-2}(K)$ is a basis of $\mathbb{P}_{k-2}(K)$, then $P_{k-2}^K x_f$ can be written as a linear combination of the elements of $\mathcal{M}_{k-2}(K)$. Plugging it into equation (16) gives us a linear combination of moments of v_h of order at most $k - 2$ in every element K of T_h. These values are known since they exactly corresponds to some of the degrees of freedom of v_h. Consequently, this is a quantity that can be computed directly from the degrees of freedom.

Case 2: $k = 1$.
f_h is a piecewise constant function defined as follows: for all $v_h \in V_h$,

$$\langle f_h, v_h \rangle := \sum_{K \in T_h} \int_K P_0^K x_f \, \bar{v_h} \, \mathrm{d}K = \sum_{K \in T_h} P_0^K x_f \, \bar{v_h},$$

where $\bar{v_h} = \frac{1}{n_v^K} \sum_{i=1}^{n_v^K} v_h(V_i)$ with $\{V_i\}_{i=1}^{n_v^K}$ the set of vertices of K. Since the value of v_h at the vertices of every element are amongst the degrees of freedom of v_h, then this is a computable quantity from the degrees of freedom only.

Therefore, we have now in hand all the tools to apply Theorem 1.5. In the next section, we will state results about approximation and interpolation errors, to make more precise the estimation of the H^1-error of the solution present in the theorem.

1.2.8 Approximation and projection errors

Only results without proofs will be presented in this section. However, references where to find the proofs will be given. Moreover, on chapter 4 of this report will be reported the numerical errors found on some particular cases, together with some convergence numerical analysis.

Theorem 1.22 (Projection error) *Assume that there exists $\gamma > 0$ such that for all h, each element K in T_h is a union of a finite number of star-shaped domains with respect to any point of a ball of radius greater then γh_K. Then there exists a constant $C = C(k, \gamma)$ such that for every s with $1 \leq s \leq k+1$ and for every $w \in H^s(K)$, there is $w_\pi \in \mathbb{P}_k(K)$ such that*

$$\|w - w_\pi\|_{L^2(K)} + h_K |w - w_\pi|_{H^1(K)} \leq C h_K^s |w|_{H^s(K)}.$$

Proof. The proof can be found in [6] by S.C. Brenner and R.L. Scott. □

Theorem 1.23 (Interpolation error) *Assume that there exists $\gamma > 0$ such that for all h, each element K in T_h is a union of a finite number of star-shaped domains with respect to any point of a ball of radius greater then γh_K. Then there exists a constant $C = C(k, \gamma)$ such that for every s with $2 \leq s \leq k+1$, for every h, for all $K \in T_h$ and for all $w \in H^s(K)$, there exists $w_I \in V^{K,k}$ such that*

$$\|w - w_I\|_{L^2(K)} + h_K |w - w_I|_{H^1(K)} \leq C h_K^s |w|_{H^s(K)}.$$

Proof. The proof can also be found in [6] by S.C. Brenner and R.L. Scott. □

Theorem 1.24 (Approximation of the right-hand side) *If $k = 1$, for all $v_h \in V_h$, there exists a constant C such that*

$$\langle f_h, v_h \rangle - F(v_h) \leq C h \left(\sum_{K \in T_h} |f|_{H^1(K)}^2 \right)^{\frac{1}{2}} |v_h|_{H^1(K)}.$$

Now, if $k \geq 2$, for all $v_h \in V_h$, there exists a constant C such that

$$\langle f_h, v_h \rangle - F(v_h) \leq C h^k \left(\sum_{K \in T_h} |f|_{H^{k-1}(K)}^2 \right)^{\frac{1}{2}} |v_h|_{H^1(K)}.$$

Proof. The proof can be found in [3] by L. Beirão da Veiga, F. Brezzi, A. Cangiani, G. Manzini, L.D. Marini and A.Russo. □

2 Algorithms

In this chapter a theoretical description of the algorithms used is given, to determine some specific properties of the mesh elements. Indeed, one of the principal characteristics of VEM is the possibility to consider meshes with any type of polygons/polyhedrons, even possibly concave.

Since it has not been possible to find one single article speaking about all of the encountered problems, the principal algorithms used are presented here. Most of them have been found online (see [5][9][10]), but we have modified some to adapt them to the problem.

2.1 Polygons

In our implementation, a polygon is defined as an *ordered* set of points. It is necessary to compute:

- its area,
- an oriented normal vector,
- its barycenter.

The implementation of each of those characteristics is different if the polygon belongs to a 2D or to a 3D space.

2.1.1 Area in 2D

Let n be the number of vertices of a given polygon, and (x_i, y_i) the coordinates of the i-th vertex, with $1 \leq i \leq n$. To compute the area A of a polygon, we have used the following Gauss-Green formula [5]:

$$A = \frac{1}{2}\left|\sum_{i=1}^{n}(x_i y_{i+1} - x_{i+1} y_i)\right|,$$

where the following convention is used: $x_{n+1} = x_1, y_{n+1} = y_1$.

2.1.2 Area in 3D

For the 3D case, we have used a similar algorithm. Let $(O; \mathbf{e_x}, \mathbf{e_y}, \mathbf{e_z})$ be an orthonormal Cartesian coordinate system of the space, and let $V = \{V_i : 1 \leq i \leq n\}$ be the set of vertices of a given n-vertex polygon. For $i = 1, \dots, n$, let (v_{1i}, v_{2i}, v_{3i}) be the the coordinates of V_i. We also consider $\mathbf{n}$ a normal vector to the polygon, with coordinates (n_1, n_2, n_3). The algorithm is as follows:

1. Project the polygon on one of the three principal plane $(O; \mathbf{e_x}, \mathbf{e_y})$, $(O; \mathbf{e_x}, \mathbf{e_z})$ or $(O; \mathbf{e_y}, \mathbf{e_z})$ in the following way: set to 0 the component v_{ji} of each vertex i such that $j = \arg\max_{1\leq k\leq 3} \|n_k\|$. In this way, the polygon is equivalent to a polygon embedded in a 2D space instead of a 3D one, and setting to 0 such component of all the vertices avoids the case of a polygon that is already parallel to one of the three principal planes. That is, we avoid the case of a degenerated polygon by choosing in a reasonable way on which plane we project it.

2. Compute the area of the projected polygon with Gauss-Green formula (see paragraph 2.1.1).

3. Divide the area of the projected polygon by the following scale factor:

$$\frac{\max_{1\leq i\leq 3}(n_i)}{\|n\|}.$$

The results gives the area of the original polygon embedded in a 3D space.

2.1.3 Oriented normal vector

Since in our definition of a polygon, the vertices are ordered, the orientation of the normal is obtained thanks to the right-hand rule. Such computation is non elementary for concave polygons, it would be easier to only compute the direction of the normal vector.

Using again n to be the number of vertices of a given polygon, let now (x_i, y_i, z_i) be the coordinates of vertex V_i for $i = 1, \dots, n$. Using Newell algorithm (see [9]), in the case of an embedded polygon in a 3D space, the oriented normal vector of coordinates (n_x, n_y, n_z) is obtained as follows:

$$\begin{cases} n_x = \sum_{i=0}^{n}(y_i - y_{i+1})(z_i + z_{i+1}) \\ n_y = \sum_{i=0}^{n}(z_i - z_{i+1})(x_i + x_{i+1}) \\ n_z = \sum_{i=0}^{n}(x_i - x_{i+1})(y_i + y_{i+1}). \end{cases}$$

In the 2D case, the algorithm is similar except that we do not need to compute n_x and n_y since they will be set to 0.

2.1.4 Barycentre

Given a polygon with n vertices, embedded in a 2D space, let C be its barycenter. With the same notations as in paragraph 2.1.3, we compute the coordinates (c_x, c_y) of the barycenter as follows [5]:

$$\begin{cases} c_x = \frac{1}{6A}\sum_{i=0}^{n}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i) \\ c_y = \frac{1}{6A}\sum_{i=0}^{n}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i), \end{cases}$$

where A is the signed area of the polygon, given by

$$A = \frac{1}{2}\sum_{i=1}^{n}(x_i y_{i+1} - x_{i+1} y_i).$$

The case of a polygon embedded in a 3D space is similar:

1. If the polygon is embedded in one of the planes parallel to $(O; \mathbf{e_x}, \mathbf{e_y})$, $(O; \mathbf{e_x}, \mathbf{e_z})$ or $(O; \mathbf{e_y}, \mathbf{e_z})$, then the computation is similar to the 2D case.

2. Otherwise, we project the polygon on one of the 3 planes $(O; \mathbf{e_x}, \mathbf{e_y})$, $(O; \mathbf{e_x}, \mathbf{e_z})$ or $(O; \mathbf{e_y}, \mathbf{e_z})$ as in step 1 of the algorithm described in 2.1.2, and then we proceed as in the 2D case to determine 2 out of the 3 coordinates. To determine the last coordinate, it is projected on one of the two other planes and the procedure is the same.

2.2 Polyhedron

In our implementation, a polyhedron is defined as a *non-ordered* set of polygons. Unlike the implementation of polygons, no algorithm to compute the exact barycenter of polyhedrons has been implemented. Indeed, such value is not necessary for this project, and its implementation would just lose in a useless manner some computational time. It is instead necessary to determine the following characteristics of any polyhedron:

- its volume,
- if the normal vectors of the faces are all coherently oriented,
- if the coherently oriented normal vectors are all external or internal to the polyhedron.

2.2.1 Volume

To compute the volume of a polyhedron Ω with exterior normal vector $\mathbf{n} = (n_x, n_y, n_z)$, we use the divergence theorem

$$\int_\Omega \nabla \cdot F \,\mathrm{d}\Omega = \int_{\partial\Omega} F \cdot \mathbf{n}\,\mathrm{d}s,$$

with $F(x, y, z) = x$ for all $(x, y, z) \in \Omega$. So that:

$$\int_\Omega \mathrm{d}\Omega = \int_{\partial\Omega} \mathbf{e_x} \cdot \mathbf{n}\,\mathrm{d}s = \int_{\partial\Omega} x n_x \,\mathrm{d}s.$$

If k is the number of faces of the polyhedron, and if $\{A_i : 1 \leq i \leq k\}$ is the set of its faces, then the second integral can be decomposed into integrals over each A_i:

$$\int_{\partial\Omega} x n_x \,\mathrm{d}s = \sum_{i=1}^{k} \int_{A_i} x n_x \,\mathrm{d}s.$$

Now, for each $1 \leq i \leq k$, let $|A_i|$ be the area of the polygon A_i, and x_i be the x-coordinate of the barycenter of A_i. Then, the volume V of the given polyhedron is:

$$V = \int_\Omega \mathrm{d}\Omega = \sum_{i=1}^{k} \int_{A_i} x n_x \,\mathrm{d}s = \sum_{i=1}^{k} |A_i| x_i n_x.$$

Since the area, the barycenter and the normal vector to every face (polygon) of the polyhedron are known (see section 2.1), then this formula is fast to implement.

2.2.2 Coherent orientation of the normal vectors of the faces

Given a polyhedron, we have seen in paragraph 2.1.3 that it is possible to compute oriented normal vectors of all of its faces. However, we now want the normal vectors of all of its faces to be coherently oriented, that is we want all of them to be either internal or external to the polyhedron, but not part of them internal and the rest external. To do so, the algorithm used is the following:

1. Randomly order the faces of the polyhedron.

2. For every face, check if it has a common edge with any other face coming after it in the ordering.

    - If two faces do have a common edge, and if the points defining the common edge appear in an opposite order for both representations of this common edge, then the orientation of the normal vectors is coherent for these two faces (both normal vectors are either external or internal to the polyhedron).
    - Otherwise, if two faces do have a common edge but the points appear in the same order for both representations of this common edge, then the order of the vertices in the second polyhedron face's edge has to be inverted to get a coherent orientation. Then, modify the order of the faces in order to have the second face right after the original one in the ordering.
    - If a face does not have any common edge with any face coming after it in the ordering, continue without any further operation.

2.2.3 Exterior normal vectors to the polyhedron

Once we have obtained a coherent orientation of the normal vectors of the faces of the polyhedron, it is necessary to make sure they are all external to the polyhedron. And if it is not the case, we have to inverse all the orientations. This step is necessary to correctly compute and take into account the boundary terms in numerical methods.

The algorithm that has been implemented is a 3D version of the *Ray casting algorithm* (see [10]). In 2D, this algorithm determines if a point is external or internal to a polygon. To obtain this information, we count the number of intersections between the boundary of the polygon and the segment from an external point to the point we are interested in. If this number is even, the point is external to the polygon. If it is odd, then the point is internal to the polygon. This method can be extrapolated to the 3D case, but it requires high computational costs.

There is a problem both in the 2D and in the 3D cases. In 2D, this is when an intersection happens at a vertex instead of at an edge (here, we intend it as without its extremities that are the vertices of the polygon/polyhedron); in 3D, this is when the intersection happens at a vertex or at an edge, instead of at a face (intended as an open set). In this case, to obtain the desired result, it is necessary to modify the external point that we consider (the 3D generalization of this affirmation is detailed in the algorithm behind).

Since all the faces are coherently oriented, it is sufficient to correctly orientate only one of the faces' normal vector. If we had to modify the chosen face's normal vector, then all the other ones, corresponding to the other faces, need to be inverted too. Here is the algorithm used:

1. Consider one face of the given polyhedron, and a point randomly chosen on it.
2. Consider the normal vector of this face starting from this point, and extend it to a half-line.
3. If such extension intersects a vertex or an edge of the polyhedron, then randomly select an other point on the face and repeat the first steps.

4. Count the number of intersections of this half-line with the faces of the polyhedron (without counting the initial chosen point). This is the most expensive step, from a computational point of view.

5. If the number is even, then the normal vector is external to the polyhedron; if it is odd, then the normal vector is internal to the polyhedron, and the orientation of all faces of the polyhedron has to be inverted.

3 Implementation

This part will explain our implementation of linear VEM (i.e. $k = 1$) for the resolution of Laplace problem with Dirichlet boundary conditions in 2D and 3D, on respectively polygonal or polyhedral meshes. `C++` has been used, making the most of the fact that it is an object oriented programming language. Moreover, a Makefile has been written to automate compilation, and a Doxygen full documentation of the implementation can also be automatically generated. To visualize the results, we have however preferred to create some Python scripts, taking advantage of Mayavi library for 3D plots. We have tried to adopt a coding style as coherent and easily readable as possible, to make the comprehension easier.

3.1 Global view on the code

The code is based on 21 classes and a main file. Making the most of *template programming* techniques, declaration and implementation of each class are both in a header file. Memory layout has been done with the most recent `C++` techniques in order to avoid memory leaks. In particular, where pointers are necessary, they have been implemented as smart (shared or weak) pointers. While writing the code, our aim has always been to create a code as general and reusable as possible. The code is essentially and ideally divided into two main parts:

- **The mesh:** 7 classes serve to its implementation, where any 2D or 3D mesh composed by non-necessarily convex elements are modelled. The flexibility brought by VEM with respect to the geometry makes this part quite complex and computationally not negligible. Computing some properties of the polygons/polyhedrons (as the elements volume, external normal, etc.) is significantly more complex than for standard grids or triangular meshes, but it is essential for the problem resolution.

- **Problem solving using VEM:** in this part, we have first tried to create a code that solves Laplace problem on any mesh regardless of the type of solver and of the imposed boundary conditions. Based from this first working part, we have implemented the Virtual Element Method solver and Dirichlet boundary conditions, the only solver and boundary conditions considered in this project. The idea has been to implement a code that is as general as possible, so that we could modify the method (for example to use the Finite Element Mehod instead of VEM) by only writing a new class corresponding to a new solver, without touching the part corresponding to the problem or the boundary conditions. In the same way, we could easily add some classes to consider other equations or other boundary conditions.

Apart from those two main parts, there is a *main* file where the appropriate solver with the appropriate problem parameters are ran. The parameters needed to solve the problem are specified in a *datafile* so that we do not need to recompile the whole code for any small modification. The datafile need the following informations:

- the name of two files, one containing the coordinates of the vertices of the considered mesh (variable `inputPoint`), the other one containing the connections between those vertices that represent the elements or the edges of the mesh (variable `inputConnection`);

- the type of mesh to know if we consider a 2D or a 3D domain (variable `meshType`);

- the connection file type to know if the faces of the elements are given in the connection file (variable `fileType`);
- the expression of the force term function, of the boundary condition function and, if known, the expression of the exact solution to be able to compute the error. A specific parser has been implemented and used to be able to interpret those lines;
- the type of coordinates used in the functions expressions, it can be either cartesian if `x,y,z` are used, or spherical if `r,phi,theta` are used;
- the type of boundary condition of the problem and the type of solver. In this version of the code, only `dirichlet` and `VEM` can be specified, but the code can be easily extended;
- the name of the output files;
- the variable `errorAction` that can be either `append` or `override` whether we want to create a new file to write the error in it, or we want to append this file;
- the variable `real` that specifies the type used to solve the problem, and it can be either `double` or `long double`.

As already said, we have also added a separate part written in Python in order to have a graphical visualization of the results. Python has been chosen thanks to its simplicity compared to `C++` for the implementation of graphical interfaces, and thanks to the presence of the library *Mayavi*, well adapted to 3D visualization. This has been implemented to be complementary to the rest of the code, in order to furnish a fast way to graphically verify the solutions.

3.2 Mesh

We have chosen to write from scratch the code to create a mesh because of the particularity of VEM to handle more general elements compared to the classical Finite Element Method.

A mesh in 3D has been thought as an ensemble of polyhedrons. Each polyhedron is composed of a set of polygons that are themselves defined as an ordered set of points. The class `Mesh` is based on the 3 classes that represent polyhedrons, polygons and points. Those three classes are tightly interconnected: each polyhedron contains a vector of pointers to the points corresponding to its vertices, and an other vector of pointers to the polygons corresponding to its faces. This makes the constructor of such objects more difficult to write, but the difficulties have been surmounted thanks to the *variadic template* techniques and to a method fully dedicated to the creation of instances. In this situation, we have also chosen to use *shared pointers* when pointers are needed to make the memory management easier to handle. It has been decided not to use *unique pointers* since inside the program, the same pointer can be used by different instances of different classes.

A mesh in 2D has been implemented in a similar way, but avoiding the presence of polyhedrons. However, a class Edge has not been necessary, thus it has not been implemented in order to keep a structure as simple as possible.

Two template parameters, *embedded* and *real* are present in every class. They guarantee some flexibility with respect to the chosen space (2D or 3D) and to the precision of the data (double or long double types can be used).

3.2.1 Point and MeshPoint

The classes `Point` and `MeshPoint` both represent a point in the 2D or 3D space. However, as their names suggest, `Point` represents a generic point while `MeshPoint` represent a point belonging to a mesh. In each instance of `MeshPoint` the polygons and polyhedrons that have this point as vertex are also stored. No copy constructor is implemented in `MeshPoint`, copying any such instance is forbidden. `MeshPoint` also keeps into memory the ID of the point corresponding to the given mesh, and the information about whether a point is on the boundary of the considered geometry or not.

Class `Point` is also used to represent a (geometrical) vector. Indeed, a point is only represented by an array of numbers corresponding to its coordinates. This is also sufficient to represent a vector. This is why class Point also contains methods to compute the norm of a vector or a cross product.

The presence or the absence of the copy constructor is however the main difference between both classes: in class Point, it is very useful to compute faster everything we need; but in `MeshPoint` it would even be harmful. Indeed, it would be difficult to keep the tight interconnection between classes used to create meshes if one of their component (as a vertex) were copied.

Here is reported the implementation of class `Point` and its main methods:

Listing 1: File `Point.h`

```
template <long embedded,typename real=double>
class Point {

protected:
  array<real,embedded> coordinates;

public:
  // CONSTRUCTORS
  Point(const array<real,embedded>& inputArray);
  Point(const Point<embedded,real>& inputPoint);
  Point(const MeshPoint<embedded,real>& inputPoint);

  // Constructor with variadic template
  template <typename... Args>
  Point(Args... arguments);

  // STANDARD METHODS
  long maxIndex() const;         //!< Maximum index of the point
  long maxAbsIndex() const;      //!< Maximum index of the point with absolute
// value
  real norm() const;  //!< L2 norm of the vector
  real normL1() const; //!< L1 norm of the vector

  real& operator[](long index); //!< Get an element by reference

  template <long embedded2,typename real2>
  friend Point<embedded2,real2> cross(const Point<embedded2,real2>& point1,
const Point<embedded2,real2>& point2);
};
```

Here is reported the implementation of class `MeshPoint` and its main methods:

Listing 2: File `MeshPoint.h`

```
template <long embedded,typename real=double>
class MeshPoint: public Point<embedded,real> {
protected:
  // PROPERTIES
  long pointID; //!< ID of the MeshPoint
  real value;           //!< Value in the point after the resolution of the
//problem
  bool isBoundary;      //<! Tells if the MeshPoint is on the boundary

  vector<weak_ptr<Polygon<embedded, real>>> polygonVector;
  vector<weak_ptr<Polyhedron<embedded,real>>> polyhedronVector;

  // CONSTRUCTORS
  // Constructor with variadic template
  template <typename... Args>
  MeshPoint(Args... arguments);

  // STANDARD METHODS
  void addPolygon(weak_ptr<Polygon<embedded,real>> inputPolygon);
//!< It inserts a new Polygon in polygon vector
  void addPolyhedron(weak_ptr<Polyhedron<embedded,real>> inputPolyhedron);
//!< It insert a new Polyhedron in polyhedron vector
```

3.2.2 Polygon and Polyhedron

The two classes `Polygon` and `Polyhedron` represent a generic polygon or polyhedron, not necessarily convex. Both classes are to be understood as mesh components; but we have not implemented two different classes of both as for Point since it has not been necessary. The absence of copy constructor is then required as for `MeshPoint`, for the same reasons (see paragraph 3.2.1). For both classes `Polygon` and `Polyhedron`, the only way to create a new instance is by using methods `make_shared_Polygon` or `make_shared_Polyhedron`, that return a shared pointer to the new instance and that take care of handling connexions between them (since they are components of a given mesh). The use of *variadic template* techniques guarantee some flexibility and readability of the code. For example, a `Polygon` can be initialized either from an `std::vector` of `MeshPoints`, or from an arbitrary number of `MeshPoints`.

`Polygon` is modelled as an ordered set of `Points`, in 2D or 3D. The order is necessary in order to be able to determine the sense of a normal vector, and in order to know if a polygon is convex or not (see paragraph 2.1). During the instantiation, the area, the normal vector and the barycenter of the polygon are computed. `Polyhedron` instead, is a non ordered set of `Polygons` (see paragraph 2.2). During the instantiation are computed the volume, the external normal and the barycenter, and the normal vectors of the faces are coherently oriented (see paragraph 2.2 for the algorithms used).

Here is reported the implementation of class `Polygon` and its main methods:

Listing 3: File `Polygon.h`

```
template <long embedded, typename real=double>
class Polygon: public std::enable_shared_from_this<Polygon<embedded,real>> {
protected:
  // PROPERTIES
  // Vector of ordered vertexes
  vector<shared_ptr<MeshPoint<embedded,real>>> pointVector;

  bool isBoundary;      // Tells if the Polygon is on the boundary
  real area;    // \return the area of the Polygon
  Vector<3, real> normal;       // return the oriented normal to the Polygon
  Point<embedded, real> centroid;       // return the centroid of the Polygon
  vector<weak_ptr<Polyhedron<embedded,real>>> polyhedronVector;

public:
  // CONSTRUCTORS
  Polygon(const privateStruct &, const vector<shared_ptr<
MeshPoint<embedded,real>>>& vertexVector);
  // Constructor with variadic template.
  template <typename... Args>
  Polygon(const privateStruct&,Args... arguments);

  template <typename... Args>
  static shared_ptr<Polygon<embedded,real>> make_shared_Polygon
(Args... arguments);

public:
  // STANDARD METHODS
  // Add a new vertex to the Polygon
  void addPoint(const shared_ptr<MeshPoint<embedded,real>>& p1);
  // Add a new Polyhedron having this as face
  void addPolyhedron(weak_ptr<Polyhedron<embedded,real>>
polyhedron);

  Vector<embedded,real> computeCentroid();
  real computeArea();
  real getDiameter();
  real hTriangle(); // Compute the maximum distance between 2 vertexes.

  shared_ptr<MeshPoint<embedded,real>> isPointAVertex(Point
<embedded,real>& point);
  bool isPointInside(Point<embedded,real>& point);
  array<shared_ptr<MeshPoint<embedded,real>>,2>
```

```
isPointOnBoundary(Point<embedded,real>& point);

  Vector<3,real> computeNormal();
  void switchPointsOrder(); // Invert the orientation of the Polygon
```

Here is reported the implementation of class `Polyhedron` and its main methods:

Listing 4: File `Polyhedron.h`

```
template <long embedded, typename real=double>
class Polyhedron: public enable_shared_from_this<Polyhedron
<embedded,real>> {

protected:
  // PROPERTIES
  // Stores the faces of the Polyhedron
  vector<shared_ptr<Polygon<embedded,real>>> polygonVector;

  bool isBoundary;      // Tells if the Polygon is on the boundary
  real volume;  // return the volume of the Polyhedron
  Point<embedded,real> centroid;        // Not the real centroid, only the mean
// of the vertexes. It returns the centroid of the Polyhedron

  // Stores the vertexes of the Polyhedron
  vector<shared_ptr<MeshPoint<embedded,real>>> pointVector;

public:
  // CONSTRUCTORS
  Polyhedron(const privateStruct&,const vector<shared_ptr
<Polygon<embedded,real>>>& inputPolygonVector);
  // Constructor with variadic template. DO NOT USE.
  template <typename... Args>
  Polyhedron(const privateStruct&,Args... arguments)

  template <typename ...Args>
  static shared_ptr<Polyhedron<embedded, real>>
make_shared_Polyhedron;

  // STANDARD METHODS
  // Add a new Polygon face to the Polyhedron
  void addPolygon(shared_ptr<Polygon<embedded,real>>&
inputPolygon);
  // Add a new vertex to the Polyhedron
  void addPoint(shared_ptr<MeshPoint<embedded,real>>&
inputPoint);
  Point<embedded,real> computeCentroid();       // return the centroid
  real computeVolume(); // return the volume of the Polyhedron
  real getDiameter();           // return the maximum distance between 2 Points
  // Makes the normal to each face pointing towards the external of the Polyhedron
  void fixExternalNormal();
  // Makes the normal of each face pointing in the same direction (either inward or
// outward)
  void fixFacesOrientation();
  // Maximum distance between vertexes. Necessary for the mesh.
  real hTriangle();
  void linkPoints();     // Makes all vertexes pointing to this
  void linkPolygons();  // Makes all Polygons pointing to this

  void switchFacesOrientation();         // Invert the orientation of all faces
  // If a new face is added, also his vertexes are added to the pointVector
  void updatePointVector();
```

3.2.3 Mesh, Mesh2D, Mesh3D

`Mesh` is an abstract class that represents a very general mesh (a solid in a 3D space, a surface in a 2D or in a 3D space, etc.). In each instance of this class are memorized two vectors: one vector of `Points` and one of `Polygons` or `Polyhedrons` depending on the chosen space. Basic methods are declared to read different types of files and build the mesh from them; they will suitably be implemented on the derived classes. Two other dedicated virtual methods are also declared in this class to keep track of the boundary elements and to eventually set other mesh properties. A constructor has also been defined and will be automatically called from the constructors of the derived classes.

`Mesh2D` is a derived class of `Mesh`, used to model 2D meshes in a 2D space. For the lifting of the boundary elements, we look for edges present only in a unique polygon of the mesh. To optimize the computational time of research of boundary elements, we have introduced a vector of point tuples called `pairVector`. This allows us to avoid the introduction of a class representing the edges, that would just have made the code uselessly more complex. Once created the mesh, `pairVector` is analysed to find non duplicated edges.

`Mesh3D` is similar to `Mesh2D` but to model 3D meshes in a 3D space. Because of the higher complexity of the mesh, it has been decided to also save the pointers to the faces of the polygon into a vector. This has allowed us to obtain the boundary elements with a similar algorithm to the 2D case, but comparing the faces instead of the edges. In this case, and to make it faster, the algorithm to compare the faces is directly called at the instantiation of every new face.

Here is reported the implementation of class `Mesh` and its main methods:

Listing 5: File `Mesh.h`

```
template <long embedded,typename baseElement,OpenEnum isOpen=
OPEN, typename real=double>
class Mesh {
protected:

    // Vector of Polygon or Polyhedron
    vector<shared_ptr<baseElement>> elementVector;
    // Vector of the vertexes of each element
    vector<shared_ptr<MeshPoint<embedded,real>>> pointVector;
```

```
public:
    long numberOfElements;
    long numberOfPoints;

    virtual real hTriangle();    // paramether h of the Mesh

    // Method that calls the functions that read the input file
    void initialize(string pointFile,string connection,
MeshType meshType=ANYTHING3D);

    // It obtains the pointVector from a file
    virtual void setPointVector(string file);
    // It obtains the elementVector from the connections
    virtual void setElementVector(string connections, MeshType
meshType);

    // Virtual method to keep into account the boundary
// elements
    virtual void setBoundaryElements()=0;
    //  Virtual method used to set other things, like pointIDs
    virtual void setRemainingThings()=0;

    // methods to set the elementVector
    virtual void setTetrahedronMesh(string connection);
    virtual void setTriangleMesh(string connection);
    virtual void setAnything3DMesh(string connection);
    virtual void setAnything2DMesh(string connection);
    virtual void setFileType1Mesh(string connection);
    virtual void setFileType2Mesh(string connection);

    virtual void sort(); //!< Sort the pointVector based on pointID
```

Here is reported the implementation of class `Mesh2D` and its main methods:

Listing 6: File `Mesh2D.h`

```
template <typename real=double>
class Mesh2D: public Mesh<2,Polygon<2,real>,OPEN,real> {
private:
    // This is to obtain internal and external points
    vector<pair<long,long>> pairVector;

public:
    long numberOfBoundaryPoints;

    // Constructor with input file
    Mesh2D(string pointFile,string connectionFile,MeshType
meshType=ANYTHING2D);

    template <typename... Args >
    shared_ptr<Polygon<2,real>> newPolygon(Args... arguments);

    // Mesh of ANYTHING2D type
    virtual void setAnything2DMesh(string connection);

    virtual void setBoundaryElements();
    virtual void setRemainingThings();
```

Here is reported the implementation of class `Mesh3D` and its main methods:

Listing 7: File `Mesh3D.h`

```
template <typename real=double>
class Mesh3D : public Mesh<3, Polyhedron<3,real>, OPEN, real> {
protected:
    // Vector of all the Polyhedron faces
    vector<shared_ptr<Polygon<3,real>>> polygonVector;

public:
    long numberOfPolygons;
    long numberOfBoundaryPoints;

    // Constructor with input file
    Mesh3D(string pointFile,string connectionFile,MeshType
meshType=ANYTHING3D);

    // STANDARD METHODS
    // Method to create a Polygon after having read it.
    template <typename... Args >
    shared_ptr<Polygon<3,real>> newFace(Args... arguments);

    virtual void setBoundaryElements();
    virtual void setRemainingThings();

    // Mesh of TETRAHEDRON type
    virtual void setTetrahedronMesh(string connection);
    // Mesh of ANYTHING3D type
    virtual void setAnything3DMesh(string connection);
```

3.3 Laplace equation and VEM solver

For the part of the code that solves a numerical problem, we have tried to write a code as modular as possible, to be able to change solver easily or to consider different boundary conditions (even if in this project, only VEM and Dirichlet boundary conditions have been implemented). We have deeply use *template programming* techniques to guarantee this modularity. Indeed, classes that model the numerical problem are all template classes, for which two parameters are needed: one for the type of solver, `SolverType`, and one for the boundary conditions, `BoundaryConditionType`. A last class has been implemented to compute the error. To manage matrices, vectors and linear systems, the open-source Eigen library has been used.

3.3.1 Problem and Laplace classes

The abstract class `Problem` models a very generic numerical problem, that can be solved through a linear system made of a stiffness matrix and of a forcing term. The aim of class `Problem` is to define a common interface for all types of problems, containing the definition of all the necessary methods and the implementation of the most generic ones. In particular, the assembling of the stiffness matrix and of the forcing term are left to the derived classes while a default linear system solver is implemented in the parent class (BiCGSTAB) that can be overwritten. In `Problem` are also implemented methods to compute the error and to write the output on a file, since this is common to all the solvers.

`Laplace` is a class that inherits from `Problem` in order to model Laplace problems, leaving the full choice of the type of solver and of appropriate boundary conditions, as discussed earlier. In this class are implemented the methods to assemble the linear system and the global stiffness matrix, even if it leaves the computation of the local stiffness matrix to the classes dedicated to the solver and to the boundary conditions.

Here is reported the implementation of class `Problem` and its main methods:

Listing 8: File `Problem.h`

```
template <long embedded,typename MeshType,typename real=double>
class Problem {

protected:
    const MeshType& mesh;       //!< Mesh on which the problem is based

public:
    SparseMatrix<real> stiffnessMatrix;
    VectorX<real> knownTerm;
    VectorX<real> solution;

public:
    // General constructor for the Problem
    Problem(const MeshType& inputMesh)

    // Virtual method to compute the stiffness matrix
    virtual void computeStiffnessMatrix()=0;
    // Virtual method to compute the known term
    virtual void computeKnownTerm()=0;

    virtual void computeSolution();
    virtual void operator()();  // Method to execute the method.
    //FreeFem style.

    virtual void displayError(muParserInterface<embedded,real>&
realSolutionFunction,string outputError="error.err",string errorAction="append");
//!<  It displays the error after the computation and prints it to a file
        virtual void write(string outputPoints="points.point",string
outputConnections="connections.conn",string outputSolution="solution.sol");
//!< Full output to given files
        virtual void writeSolution(string outputSolution="solution.sol");
//!< Output the solution to a file
```

Here is reported the implementation of class `Laplace` and its main methods:

Listing 9: File `Laplace.h`

```
template <long embedded,typename MeshType,typename SolverType,
typename BoundaryConditionType,typename real=double>
class Laplace: public Problem<embedded,MeshType,real> {
private:
    real diffusionCoeff;
public:
    // Vector used to fast build the stifnessMatrix.
    vector<Triplet<real>> tripletList;
    long numberOfElements;
    BoundaryConditionType boundaryCondition;
    SolverType solver;

    // CONSTRUCTORS
        /* Standard constructor
         *
         * \param inputMesh Mesh on which the problem is defined
         * \param inputForceTerm muParserInterface containing the expression of
the ForceTerm
         * \param inputBoundaryFunction muParserInterface containing the
expression of the boundary conditions
         */
    Laplace(const MeshType& inputMesh,muParserInterface<embedded,real>&
inputForceTerm,muParserInterface<embedded,real>& inputBoundaryFunction,real
inputDiffusionCoeff=1);

    // General method. It invokes the methods of the Solver.
    void computeStiffnessMatrix();
    // General method. It invokes the methods of the Solver and BoundaryCondition
    void computeKnownTerm();
```

3.3.2 Monomials and MonomialsPolygon

First, we recall some VEM theory for a better comprehension. Given a mesh element, we call monomial m_α of degree $|\alpha|$ the following quantity:

$$m_\alpha := \left(\frac{\mathbf{x}-\mathbf{x_k}}{h_k}\right)^\alpha,$$

where $\mathbf{x_k}$ is the barycenter of the element, and h_k its diameter. For VEM solver of degree 1, it is necessary to evaluate the monomials at the vertices of the elements, and to compute their gradients on the elements. We note that for VEM of order 1, neither $\mathbf{x_k}$ nor h_k have to be exactly computed as far as the same values are used to compute all the monomials of a same element. Consequently, it has not been necessary to implement the computation of the exact barycenter of an element, operation computationally very expensive. All of this has been implemented in class `Monomials` for the case in which a 2D (resp. 3D) element is embedded in a 2D (resp. 3D) space.

Class `MonomialsPolygon`, that inherits from `Monomials`, however, is used to compute the same quantities but in the case where a 2D element (a polygon) is embedded in a 3D space. The main difference comes from the fact that in the case of a 2D (resp. 3D) element in a 2D (resp. 3D) space, the gradient is a vector whose components are all equal, while for a 2D element in a 3D space the gradient has to be projected on the plane of the polygon we are considering.

Here is reported the implementation of class `Monomials` and its main methods:

Listing 10: File `Monomials.h`

```
template <long embedded,typename baseElement,typename real=
double>
class Monomials {
public:
    const shared_ptr<baseElement>& element; //!< Element is a pointer
// on a Polygon or Polyhedron
    real diameter;  //!< Diameter of the element
    Point<embedded,real> centroid;  //!< Centroid of the element
    // It's the gradient of the monomial. It's 1/diameter
    real gradient;

    // CONSTRUCTOR
    Monomials(const shared_ptr<baseElement>& figure);

    // Function to evaluate the monomial in a point
    real evaluate (const Point<embedded,real>& p, long i);
}
```

Here is reported the implementation of class `MonomialsPolygon` and its main methods:

Listing 11: File `MonomialsPolygon.h`

```
template <typename real>
class MonomialsPolygon: public Monomials<3, Polygon<3,real>,
real> {
public:
// these are virtual indexes used to project the Polygon on an appropriate plane
    long indexX;
    long indexY;
    long indexZ;
    Vector<3,real> gradientX;
    Vector<3,real> gradientY;

    MonomialsPolygon(const shared_ptr<Polygon<3,real>>& figure);
    // Function to evaluate the monomial in a point
    real evaluate (const Point<3,real>& p, long i);
```

3.3.3 Solver, SolverVEM, SolverVEM2D and SolverVEM3D

Class `Solver` and its inherited classes contain the implementation of the numerical solvers. Concerning the VEM solver, we refer to chapter 1 for the description of the method, but we will give in this paragraph a description of its implementation.

`Solver` is an abstract class that represents a generic solver. Only one method is present in this class, `computeLocalK`, whose aim is to compute the local stiffness matrix. It is consequently a class that can be extended to any numerical method that need the computation of a local stiffness matrix on every element.

To implement the VEM solver, it has not been possible to create a unique class for the 2D and the 3D cases. As a consequence, a generic abstract class `SolverVEM` has been created and directly inherits from `Solver`, while `SolverVEM2D` and `SolverVEM3D` inherit from `SolverVEM`. In `SolverVEM` are implemented the methods that are common to the 2D and the 3D cases, while in the lower-level classes are implemented the few specific methods left.

The VEM solver follows the procedure explained in the article [4], where matrices G, B, D, $\Pi^{\Delta}_{\star}, \Pi^{\Delta}$ need to be computed in order to then be able to get the local matrix:

$$\mathbf{K^h_E} = (\mathbf{\Pi^{\Delta}_{\star}})^{\mathbf{T}}\tilde{\mathbf{G}}(\mathbf{\Pi^{\Delta}_{\star}}) + (\mathbf{I} - \mathbf{\Pi^{\Delta}})^{\mathbf{T}}(\mathbf{I} - \mathbf{\Pi^{\Delta}}).$$

Classes `Monomials` and `MonomialsPolygon` have also been used to evaluate the monomials, characteristic of the virtual elements method. Computing B and the forcing term has been left to the lower-level classes while the rest has been implemented in `SolverVEM`. The 3D case has required a more complex evaluation of boundary terms. Computing the interpolating VEM polynomial on every face has also been necessary, and this operation considerably increases the computational time.

An important matrix property of VEM is the identity $G = BD$, which is very useful to test the code. A control has thus been implemented in order to tell the user if this identity is not verified. We have indeed tested the code thanks to this identity, also on very particular geometries, such as the one showed in Figure **??**.

Here is reported the implementation of class `Solver` and its main methods:

Listing 12: File `Solver.h`

```
template <long embedded,typename baseElement,typename
MatrixType,typename real>
class Solver {
protected:
    muParserInterface<embedded,real>& forceTerm; //!< ForceTerm function to use,
// in the form of a muParserInterface expression

public:
    Solver(muParserInterface<embedded,real>& inputForceTerm); //!< Very generic
// constructor

    // Main virtual method. To be implemented in subclasses
    virtual MatrixType computeLocalK(const shared_ptr
<baseElement>& element)=0;
};
```

Here is reported the implementation of class `SolverVEM` and its main methods:

Listing 13: File `SolverVEM.h`

```
template <long embedded,long elementDimension,typename
baseElement, typename MonomialType,typename real=double>
class SolverVEM: public Solver<embedded,baseElement,
Matrix<real,Dynamic,Dynamic>,real> {

protected:
    virtual Matrix<real,elementDimension+1,elementDimension
+1> computeG(MonomialType& monomial);

    virtual Matrix<real,elementDimension+1,Dynamic>
computeB(const shared_ptr<baseElement>& polyhedron,
MonomialType& monomial)=0;

    virtual Matrix<real,Dynamic,elementDimension+1>
computeD(MonomialType& monomial);

    virtual Matrix<real,elementDimension+1,Dynamic>
computePIStar(
Matrix<real,elementDimension+1,elementDimension+1>&G,
Matrix<real,elementDimension+1,Dynamic>&B);

    virtual Matrix<real,Dynamic,Dynamic> computePI(
Matrix<real,elementDimension+1,Dynamic>&PIStar,
Matrix<real,Dynamic,elementDimension+1>&D);

public:
    /** Basic constructor. A lot of parameters are given as template.
        *
        * \param inputForceTerm muParserInterface containing the expression
        * of the ForceTerm
        */
    SolverVEM(muParserInterface<embedded,real>& inputForceTerm);

    // Compute the local stiffness matrix
    virtual Matrix<real,Dynamic,Dynamic> computeLocalK(const
shared_ptr<baseElement>& element);
    //!< Virtual method to compute the known term.
    virtual real computeKnownTerm(const shared_ptr<baseElement>&
element, const shared_ptr<MeshPoint<embedded,real>>& point)=0;
};
```

Here is reported the implementation of class SolverVEM2D and its main methods:

Listing 14: File SolverVEM2D.h

```
template <typename real=double>
class SolverVEM2D: public SolverVEM<2,2,Polygon<2,real>,
Monomial2D<real>,real> {
public:
    virtual Matrix<real,3,Dynamic> computeB(const shared_ptr
<Polygon<2,real>>& polygon,Monomial2D<real>& monomial);

public:
     SolverVEM2D(muParserInterface<2,real>& inputForceTerm);

     virtual real computeKnownTerm(const shared_ptr<Polygon<2,
real>>& element,
const shared_ptr<MeshPoint<2,real>>& point);
};
```

Here is reported the implementation of class SolverVEM3D and its main methods:

Listing 15: File SolverVEM3D.h

```
template <typename real=double>
class SolverVEM3D: public SolverVEM<3,3,Polyhedron<3,real>,
Monomial3D<real>,real> {
public:
    virtual Matrix<real,3,3> computeGPolygon(MonomialsPolygon
<real>& monomial);
    virtual Matrix<real,4,Dynamic> computeB(const shared_ptr
<Polyhedron<3,real>>&
polyhedron,Monomial3D<real>& monomial);

    virtual Matrix<real,3,Dynamic> computeBPolygon(const
shared_ptr<Polygon<3,real>>&
polyhedron,MonomialsPolygon<real>& monomial);

    virtual Matrix<real,Dynamic,3> computeDPolygon(
MonomialsPolygon<real>& monomial);

    virtual Matrix<real,3,Dynamic> computePIStarPolygon(Matrix
<real,3,3>&G, Matrix<real,3,Dynamic>&B);

public:
    SolverVEM3D(muParserInterface<3,real>& inputForceTerm);

    virtual real computeKnownTerm(const shared_ptr<Polyhedron
<3, real>>& element, const shared_ptr<MeshPoint<3,real>>&
point);
};
```

3.3.4 BoundaryCondition, Dirichlet

Class `BoundaryCondition` is an abstract class that represents any type of boundary conditions and handle them. Class `Dirichlet` inherits from `BoundaryCondition` and takes only care of Dirichlet boundary conditions, as its name suggests. To impose this type of condition, the upper-left block of the stiffness matrix (whose rows correspond to boundary elements) needs to be diagonal. Moreover, the same rows corresponding to the boundary elements in the forcing term need to be correctly set.

Those two classes are based on the three following main methods:

- Process the local stiffness matrix before assembling the global stiffness matrix. In the Dirichlet case, if an element is on the boundary, its local stiffness matrix is not assembled with the other ones in order to keep the upper-left block of the global stiffness matrix diagonal.
- Process the global stiffness matrix after its assembling. In the Dirichlet case, we make sure that the upper-left block is diagonal.
- Process the forcing term after it has been created, that is we set correctly the rows corresponding to the boundary elements.

Here is reported the implementation of class `BoundaryCondition` and its main methods:

Listing 16: File `BoundaryCondition.h`

```
template <long embedded,typename MeshType,typename MeshElement,
typename real=double>
class BoundaryCondition {
protected:
    const MeshType& mesh;
    muParserInterface<embedded,real>& boundaryFunction;

    BoundaryCondition(const MeshType& inputMesh,muParserInterface<embedded,real>&
inputBoundaryFunction);

public:
    // Decide wheter the computed Kloc will be added to the global matrix.
    // It depends on the boundary condition.
    virtual void addToTripletList(Matrix<real,Dynamic,
Dynamic>& Kloc, MeshElement& element,vector<Triplet<real>>&
tripletList)=0;

    // Changes the stiffnessMatrix to take into account the boundary condition.
    virtual void assignBoundaryConditionOnStiffnessMatrix
(vector<Triplet<real>>& tripletList)=0;

    // Changes the known term to take into account the boundary condition
    virtual void assignBoundaryConditionOnKnownTerm
(VectorX<real>& knownTerm)=0;
};
```

Here is reported the implementation of class `Dirichlet` and its main methods:

Listing 17: File `Dirichlet.h`

```
template <long embedded,typename MeshType,typename MeshElement,
typename real>
class Dirichlet: public BoundaryCondition<embedded,MeshType,
MeshElement,real> {
    long numberOfPoints;
    long numberOfBoundaryPoints;

public:
    // Standard constructor
    Dirichlet(const MeshType& mesh,muParserInterface<embedded,real>&
boundaryFunction);

    // Decide wheter the computed Kloc will be added to the global matrix.
    virtual void addToTripletList(Matrix<real,Dynamic,Dynamic>&
Kloc, MeshElement& element,vector<Triplet<real>>& tripletList);

    // Changes the stiffnessMatrix to take into account the boundary condition.
    virtual void assignBoundaryConditionOnStiffnessMatrix
(vector<Triplet<real>>& tripletList);

    // Changes the known term to take into account the boundary condition
    virtual void assignBoundaryConditionOnKnownTerm(VectorX
<real>& knownTerm);
};

template <typename real=double>
using Dirichlet3D=Dirichlet<3,Mesh3D<real>,Polyhedron<3,real>,
real>;

template <typename real=double>
using Dirichlet2D=Dirichlet<2,Mesh2D<real>,Polygon<2,real>,
real>;
```

3.3.5 Error

`Error` is a class reserved for error computation. The error is computed in two different ways:

- l^∞ norm: the maximum between the exact and the numerical solution in the nodes;
- H^1 discrete norm: computed as $u^T K u$, where u is the difference between the real and the numerical solutions on the nodes, and K is the stiffness matrix.

Here is reported the implementation of class `Error` and its main methods:

Listing 18: File `Error.h`

```
template <long embedded,typename real=double>
class Error {
protected:
    const VectorX<real>& solution;
    VectorX<real> realSolution;
    VectorX<real> difference;
    const vector<shared_ptr<MeshPoint<embedded,real>>>&
pointVector;
    muParserInterface<embedded,real>& realSolutionFunction;

    SparseMatrix<real>& stiffnessMatrix;

    // This computes the exact solution, from realSolutionFunction
    void computeRealSolution();

public:
    Error(const VectorX<real>& inputSolution,const vector<shared_ptr<MeshPoint
<embedded,real>>>& inputPointVector,muParserInterface<embedded,real>&
inputRealSolutionFunction,SparseMatrix<real>& inputStiffnessMatrix);

    // L infinite norm of the error
    real LInfinity();

    // H1 discrete norm of the error
    real H1Discrete();

    void displayError();   //!< Print the computer error
};
```

4 Results

4.1 Visualization of the results

For the visualization of the results, we have decided to use the Python programming language instead of `C++` thanks to a great choice of libraries. This part is divided into two parts, depending on the used libraries:

- With the Matplotlib library, we have created 2 scripts for the 3D visualization of either a polyhedron or an entire mesh. The main goal of this script is mainly to test the code; it has not been optimized and thus it might be very slow for large meshes. It is however efficient for smaller mesh and allows us to have a quick visual feedback on the results and their coherence.

- With the Mayavi library, we have create 3 scripts to visualize the solution. `plotPoints3D` only shows the vertices of the polyhedrons that are part of the mesh, together with the solution at these points. `plot2D` and `plot3D`, instead, show the solution on the whole mesh. In the 3D case, Mayavi visualization tools can be used to see the solution in different part of the mesh, such as on a section of the mesh for instance.

In the following Figures 1 to 5, we show some of the visual results that have been obtained. Figure 1 shows a particular case of polyhedron (in 3D) that our implementation can handle, and its visualization with Matplotlib. The script used to generate such image is `plotSinglePolyhedron`.

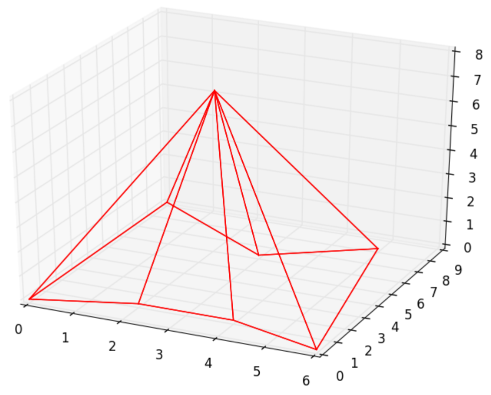

Figure 1: Plot with Matplotlib. A specific mesh element that VEM can handle.

Then Figure 2 shows a particular case of a cubic mesh in 3D. This image has been generated using the script `plotMesh`.

In Figure 3, we can see the solution using linear VEM of the following differential problem:

$$\begin{cases} -\Delta u = \frac{\pi^2}{2}\cos\left(\frac{\pi x}{2}\right)\cos\left(\frac{\pi y}{2}\right) \text{ in } \Omega = (-1,1)^2; \\ u = 0 \text{ on } \partial\Omega, \end{cases}$$

whose exact solution is $u(x, y) = \cos\left(\frac{\pi x}{2}\right)\cos\left(\frac{\pi y}{2}\right)$, for all $(x, y) \in \Omega = (-1, 1)^2$. A rectangular mesh has been used to obtain this result.

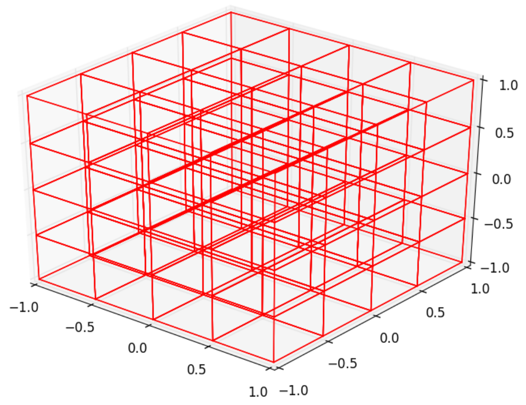

Figure 2: Plot with Matplotlib. Cubic mesh.

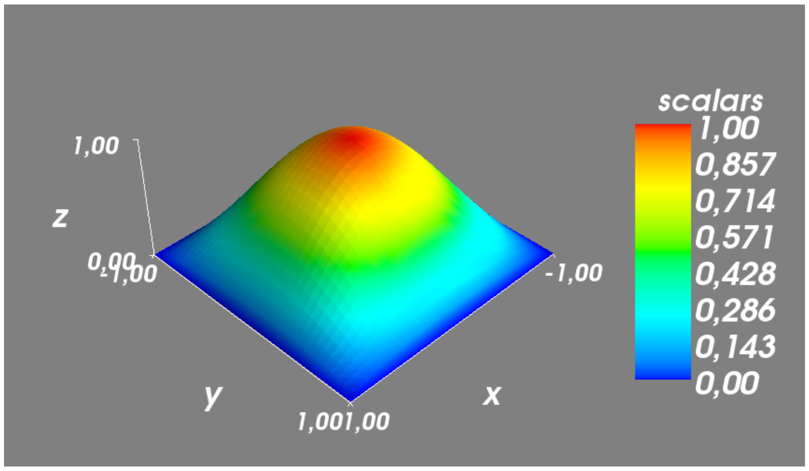

Figure 3: Plot with Mayavi. VEM solution on a 2D grid mesh.

Figures 4 and 5 show the solution using linear VEM of the following differential problem:

$$\begin{cases} -\Delta u = 6 \text{ in } \Omega = (0, 1) \times (0, 2\pi) \times (0, \pi). \\ u = 0 \text{ on } \partial\Omega, \end{cases}$$

where everything is expressed in spherical coordinates (r, ϕ, θ), including Laplace operator. The exact solution is $u(r, \phi, \theta) = 1 - r^2$, for all $(r, \phi, \theta) \in (0, 1) \times (0, 2\pi) \times (0, \pi)$. Figure 4 shows only the solution on a section of the sphere while in Figure 5, we can see the results on each vertex of the mesh. The size of the vertices and their color both correspond to the value of the solution on them.

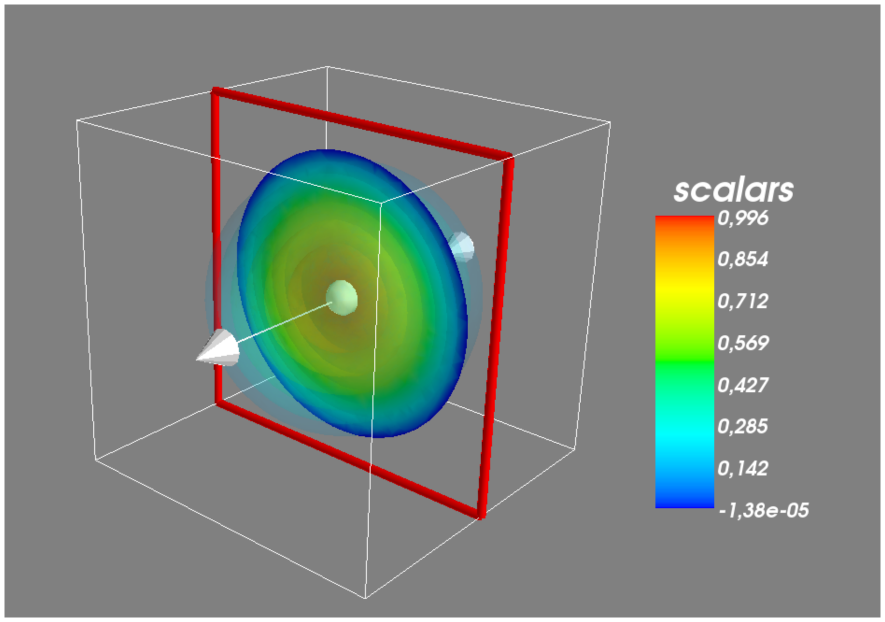

Figure 4: Plot with Mayavi. VEM solution on a section of a 3D sphere, with a mesh made of tetrahedra. The section can be dynamically chosen.

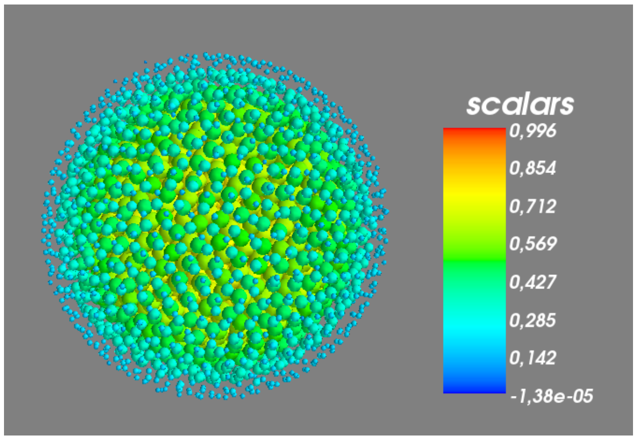

Figure 5: Plot with Mayavi. VEM solution on every vertex of a 3D sphere, with a mesh made of tetrahedra.

4.2 Numerical results

In this section, we will analyse numerically the convergence under h-refinement (that is when the mesh is refined) of the absolute a priori H^1-error of the solution, i.e. $\|u_h - u_{\text{ex}}\|_{H^1(\Omega)}$. u_h is the solution found with the linear Virtual Element Method while u_{ex} is the exact solution. Some particular differential problem will be analysed and presented behind.

Let Ω be the full sphere centered in $(0, 0, 0)$ and of radius 1. Consider a mesh decomposition and let h be the maximum diameter of the elements of this mesh. The considered meshes can be found in the folder `Mesh/Sphere3D/`. We will analyse the 3D behaviour of the error when we solve the following problem, expressed in spherical coordinates: find u such that

$$\begin{cases} -\Delta u = 6 \text{ in } \Omega = (0,1)^2; \\ u = 0 \text{ on } \partial\Omega. \end{cases}$$

The exact solution of this problem is $u(r,\phi,\theta) = 1 - r^2$, for all $(r,\phi,\theta) \in \Omega$.

In Figure 6 is plotted the absolute a priori error in $H^1(\Omega)$-norm. We can clearly see that asymptotically, the error behaves as h, which is what one could expect after reading Theorem 1.5 and section 1.2.8.

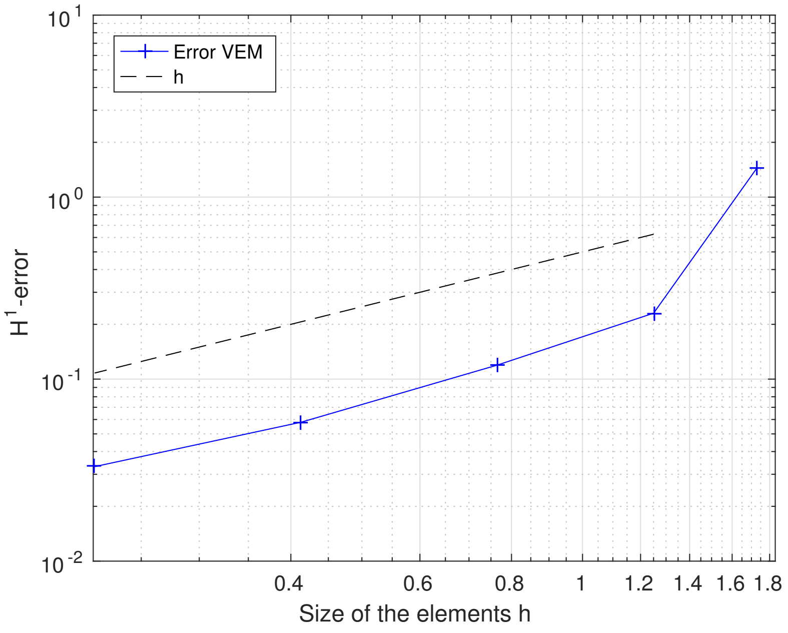

Figure 6: Convergence of the H^1-error under h-refinement for Laplace 3D problem on the sphere centered in $(0,0,0)$ and of radius 1, with homogeneous Dirichlet boundary conditions.

This behaviour is again confirmed when computing the slope of the log-log scale line. Indeed, we suppose that $\text{err} := \|u_h - u_{\text{ex}}\|_{H^1(\Omega)} = Ch^p$, where C is a constant depending only on the solution and on k, the degree of the underlying polynomials, which is here 1. Our aim is to find p, so that we compute

$$\frac{\log\left(\frac{\text{err}(h_2)}{\text{err}(h_1)}\right)}{\log\left(\frac{h_2}{h_1}\right)} = p, \tag{17}$$

where $\text{err}(h_1)$ and $\text{err}(h_2)$ are the absolute a priori errors using a mesh size equal to h_1 and h_2, respectively. The values of the p found thanks to the last 4 points of Figure 6 are reported in the following Table 1 and confirm our hypothesis.

h_1	h_2	p
1.2502	0.7646	1.3371
0.7646	0.4134	1.1726
0.4134	0.2160	0.8667

Table 1: Sum up of the convergence rates under h-refinement, on Laplace 2D problem.

Furthermore, we have remarked that in the case of a very regular mesh composed of always equal simple convex polygons, we obtain a quadratic convergence, both in 2D and in 3D. Here are presented two examples of such phenomenon, one in 2D and one in 3D. In the following, we will only consider a decomposition of the space into identical elements, so that we can define h as the size of the elements. More precisely, we only consider squared or cubic elements, respectively for a 2D or a 3D problem, so that h is the length of each side of each element. Moreover, we will consider the geometry $\Omega = (0,1)^d$, where $d = 2$ in the 2D case, and $d = 3$ in the 3D case. Then $h = \frac{1}{n_e}$, where n_e is the number of elements used on each direction. We have chosen to consider $n_e = 1, 2, 4, 8, 16, 32, 64$ and 128 elements, that is h ranges from 1 to $\frac{1}{128}$. n_e is chosen as a sequence of powers of 2 in order to have embedded spaces when the number of elements is increased.

First, we will analyse the 2D behaviour of the error when we solve the following problem: find u such that

$$\begin{cases} -\Delta u = 98\pi^2 \sin(7\pi x)\sin(7\pi y) \text{ in } \Omega = (0,1)^2; \\ u = 0 \text{ on } \partial\Omega. \end{cases}$$

The exact solution of this problem is $u(x,y) = \sin(7\pi x)\sin(7\pi y)$, for all $(x,y) \in \Omega$. This problem has been chosen so that the error between the numerical and the exact solution is large, since the solution oscillates a lot. In this way, the behaviour of the error can more easily be observed, since there is no risk to attain the machine epsilon too early.

In Figure 7 is plotted the absolute a priori error in $H^1(\Omega)$-norm. We can clearly see that asymptotically, the error behaves as h^2, which is one order of magnitude more than what one could expect from Theorem 1.5. It does not mean that the theorem is wrong, it just mean that the estimation is pessimistic is some cases such as this one, when the mesh is composed of regular and simple polygonal elements.

This behaviour is again confirmed when computing the slope of the log-log scale line. The values of the p found thanks to the last 4 points of Figure 7 are reported in the following Table 2 and confirm our hypothesis. Finally, the non asymptotic behaviour is more chaotic since the approximation is very raw with very few elements. Let us add that the same behaviour has been observed when taking $u_{\text{ex}}(x,y) = \exp(7xy)$ on $(0,1)^2$, so that it is not solution-specific.

Now, we analyse the 3D behaviour of the error when we solve the following problem: find u such that

$$\begin{cases} -\Delta u = 127\pi^2 \sin(7\pi x)\sin(7\pi y)\sin(7\pi z) \text{ in } \Omega = (0,1)^3; \\ u = 0 \text{ on } \partial\Omega. \end{cases}$$

The exact solution of this problem is $u(x,y,z) = \sin(7\pi x)\sin(7\pi y)\sin(7\pi z)$, $\forall (x,y,z) \in \Omega$.

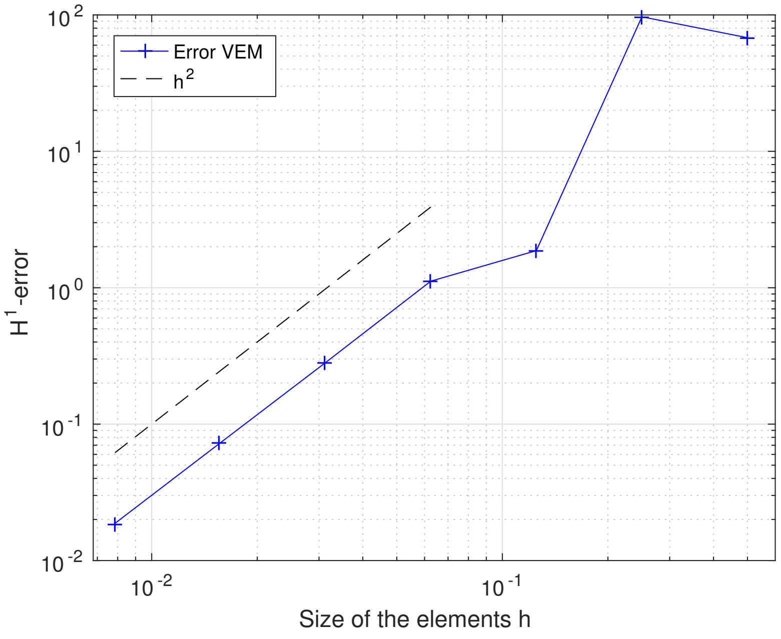

Figure 7: Convergence of the H^1-error under h-refinement for Laplace 2D problem on $(0,1)^2$ with homogeneous Dirichlet boundary conditions.

h_1	h_2	p
16^{-1}	32^{-1}	1.9884
32^{-1}	64^{-1}	1.9612
64^{-1}	128^{-1}	1.9651

Table 2: Sum up of the convergence rates under h-refinement, on Laplace 2D problem, with a very regular rectangular mesh.

Again and for the same reason as before, this problem has been chosen so that the error between the numerical and the exact solution is large, since the solution oscillates a lot.

In Figure 8 is plotted the absolute a priori error in $H^1(\Omega)$-norm. We can see again that asymptotically, the error behaves as h^2, which is again one order of convergence more than what one could expect from Theorem 1.5. To see it in a clearer way, we should refine the mesh even more, but a lot of RAM is needed to solve the problem (> 32GB). Indeed, with $h = 128$, we already take into account $128^3 > 2$ million degrees of freedom.

Similarly as in the 2D case, we can algebraically compute the slope of the log-log scale line to confirm our hypothesis, thanks to the same formula (17). The values of p found thanks to the last 3 points of Figure 8 are reported in the following Table 3 and indeed confirm the behaviour.

Let us add that the same behaviour has been observed when taking $u_{\text{ex}}(x, y, z) = \exp(7xyz)$ on $(0,1)^3$, so that it is not solution-specific. Note moreover that this h^2-component of the

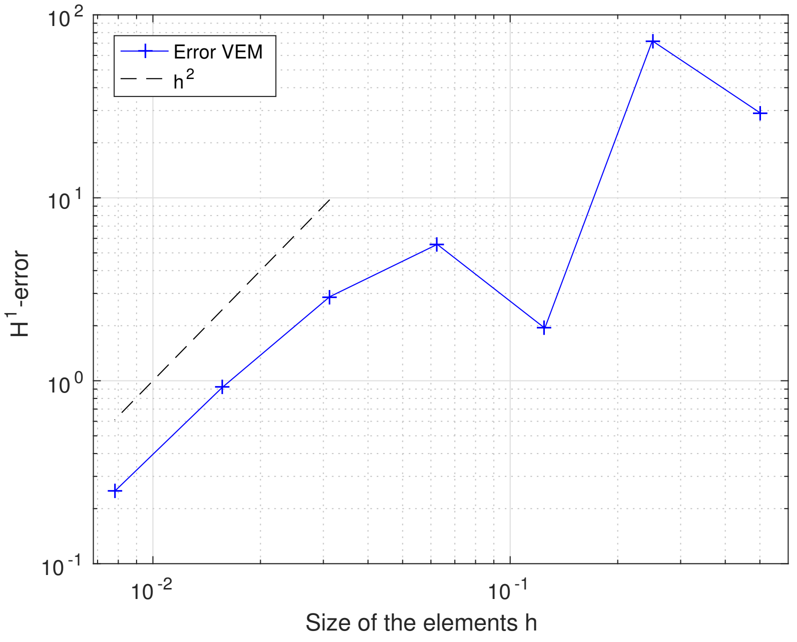

Figure 8: Convergence of the H^1-error under h-refinement for Laplace 3D problem on $(0,1)^3$ with homogeneous Dirichlet boundary conditions.

h_1	h_2	p
32^{-1}	64^{-1}	1.7497
64^{-1}	128^{-1}	1.9293

Table 3: Sum up of the convergence rates under h-refinement, on Laplace 3D problem.

error is a typical characteristic of finite element-type methods: the classical finite element methods [8], the spectral element methods [7], the isogeometric analysis [2].

5 Conclusion

To sum up, the Virtual Element Method has been introduced together with convergence theorems that proves its good behaviour. A big advantage of this method with respect to the classical Finite Element Method is that VEM does not only have polynomial basis functions, and it requires to compute the values of those non-polynomial functions only on well-chosen degrees of freedom. Moreover, we came up with an implementation of VEM for linear basis polynomials, in 2D and 3D, that can be easily reused to be generalized to different dimensions or for a higher degree of polynomials thanks to the use of the full power of `C++` such as template metaprogramming and class inheritance. Finally, we have verified the well behaviour of the method and its implementation on different problems in 2D and in 3D. And we have performed a numerical convergence analysis of the absolute a priori error under h-refinement: the error is an $O(h^2)$ under mesh refinement.

As a future work, this project could be even more generalized. In particular, the implementation could be done for boundary conditions other than Dirichlet ones, for an operator more general than Laplace, and for some underlying polynomial degree k higher than one. If this last point is done, it would then be interesting to study the behaviour of the error under what is generally called p-refinement, that is when the degree of the underlying polynomials is increased.

References

[1] B. Ahmed, A. Alsaedi, F. Brezzi, L.D. Marini, and A. Russo. Equivalent projectors for virtual element methods. *Computational Mathematics and Applications*, 66:376–391, 2013.

[2] Y. Bazilevs, L. Beirao da Veiga, J.A. Cottrell, T.J.R. Hughes, and G. Sangalli. Isogeometric analysis: Approximation, stability and error estimates for h-refined meshes. *Mathematical Methods and Models in Applied Sciences*, (16):1031–1090, 2006.

[3] 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(1):199–214, 2013.

[4] 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(8):1541–1573, 2014.

[5] Paul Bourke. Calculating the area and centroid of a polygon. *Available from http://paulbourke. net/geometry/polygonmesh*, 1988.

[6] S.C. Brenner and R.L. Scott. The mathematical theory of finite element methods. *Texts in Applied Mathematics, Springer-Verlag*, 15:1541–1573, 2008.

[7] C. Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Zang. *Spectral Methods. Fundamentals in Single Domains.* Springer Verlag, Berlin Heidelberg New York, 2006.

[8] A. Quarteroni. *Numerical Models for Differential Problems*, volume 8 of *MS&A, Modeling, Simulation & Applications.* Springer Milan, 2014.

[9] Daniel Sunday. Fast polygon area and newell normal computation. *journal of graphics tools*, 7(2):9–13, 2002.

[10] Wikipedia. Point in polygon — wikipedia, the free encyclopedia, 2016. [Online; accessed 11-February-2017].