POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

LYMPH3D: A new library to solve PDE problems with Discontinuous Galerkin methods on three-dimensional polytopic meshes

Tesi di Laurea Magistrale in
Mathematical Engineering - Ingegneria Matematica

Author: **Nicoletta De Giosa**

Student ID: 969084
Advisor: Prof. Paola F. Antonietti
Co-advisors: Prof. Ilario Mazzieri
Academic Year: 2022-2023

Abstract

The numerical approximation of partial differential equations (PDEs) posed on complicated geometries represents a challenging computational problem. Indeed, the use of mesh generators employing elements of standard shape, i.e. simplices or tensor product elements, can lead to very fine finite element meshes. Hence, the computational effort required to numerically approximate the underlying PDE problem may be prohibitively expensive. An alternative approach is to solve the PDE problem by using polytopic (polygonal or polyhedral) elements. The main advantage of choosing polytopic element shapes over classical simplicial/hexahedral elements is that the average number of elements needed to discretize complicated domains is substantially smaller and this allows to reduce the complexity of the given computational geometry. In this work of thesis we focus on discontinuous Galerkin methods on polytopic grids (PolyDG) to discretize differential problems. After recalling some numerical aspects on the polytopic meshes and some theoretical results on the PolyDG method, we develop a new library called LYMPH3D written in Fortran. The library can be used to solve PDE problems by using the PolyDG method in three-dimensions. We perform a convergence analysis on a simple geometry of a cube, employing first a tetrahedral and then a polyhedral mesh of the cube. The latter is obtained via agglomeration of a tetrahedral mesh. Finally, we demonstrate the capabilities of LYMPH3D considering the solution of a PDE problem on a challenging geometry, namely a human brain.

Keywords: Complicated geometries, Numerical approximation, Partial Differential Equations, Discontinuous Galerkin methods, Polytopic elements, Tetrahedral mesh, Computational complexity, three-dimensional mesh, New library, Fortran

Abstract in lingua italiana

L'approssimazione numerica delle equazioni alle derivate parziali (in breve EDP) poste su geometrie complicate rappresenta un problema di costo computazionale elevato. Infatti, l'uso di generatori di mesh che impiegano elementi di forme classiche, come ad esempio tetraedri o esaedri, può portare nella generazione di mesh agli elementi finiti che sono molto raffinate. Pertanto, lo sforzo computazionale richiesto per approssimare numericamente un problema di EDP può essere talmente costoso da essere proibitivo. Un approccio alternativo consiste nel risolvere un problema di EDP utilizzando elementi politopici (poligonali o poliedrici). Un vantaggio di scegliere elementi politopici rispetto ai classici elementi simpliciali/esaedrici è che il numero medio di elementi necessari per discretizzare domini complicati è sostanzialmente minore. Ciò permette di ridurre la complessità della geometria computazionale data. In questo lavoro di tesi ci focalizziamo sui metodi Discontinuous Galerkin su mesh politopiche (PolyDG, in breve) per discretizzare problemi differenziali. Dopo aver richiamato alcuni aspetti numerici sulle mesh politopiche ed alcuni risultati teorici sul metodo PolyDG, sviluppiamo una nuova libreria chiamata LYMPH3D scritta in Fortran. La libreria può essere utilizzata per risolvere un problema EDP utilizzando il metodo PolyDG in tre dimensioni. Analizziamo la convergenza della soluzione numerica su una semplice geometria di un cubo, utilizzando prima una mesh tetraedrica e poi poliedrica del cubo. Quest'ultima è ottenuta agglomerando la mesh tetraedrica. Infine dimostriamo le potenzialità di LYMPH3D considerando la soluzione di un problema di EDP su una geometria complicata, trattando il caso di un cervello umano.

Parole chiave: Geometria complicata, Approssimazione numerica, Equazioni alle derivate parziali, Metodi Discontinuous Galerkin, Elementi politopici, Mesh tetraedrica, Complessità computazionale, Mesh tridimensionale, Nuova libreria, Fortran

Contents

Introduction

Finite element methods (FEMs) are an indispensable computational tool for the accurate, efficient, and rigorous numerical approximation of continuum models arising in engineering, physics, biology, and many other disciplines.

However, a key underlying issue for all classes of finite element methods is the design of a suitable computational mesh upon which a generic PDE problem is discretized.

On one hand, the mesh should provide a good description of the given computational geometry with sufficient resolution for the computation of accurate numerical approximations to within desired accuracy constraints. On the other hand, if the mesh is too fine the computational time required to compute the solution is too high for practical applications. This issue is particularly pertinent when high order methods are employed since in this setting we would like to employ relatively coarse meshes, so that the polynomial degree may be suitably enriched.

Many engineering applications, such as fluid-structure interaction, or flow in fractured porous media, are characterized by a strong complexity of the physical domain. In the context of discretizing a differential model on a complex domain, whenever classical finite element methods are employed, the process of the mesh generation can potentially lead to very high computational costs.

Indeed, the classical finite element methods typically only support computational grids composed of standard element shapes; triangular or quadrilateral elements in two-dimensions (2D) and tetrahedral, hexahedral, prismatic or pyramidal elements in three-dimensions (3D). The use of these kind of elements necessitates the exploitation of very fine computational meshes when the geometry is complicated, if it includes for example a large number of small geometrical features or microstructures. In Figure 1, taken from [4], we can see an example of complicated geometry arising in biological applications. The image shows a finite element mesh of a porous scaffold employed for in vitro bone tissue growth, cf. [8, 9]. In such situations, for a given mesh generator, a large number of elements is required to produce even a ‘coarse’ mesh which adequately describes the underlying geometry. Thereby, the solution of the system of equations resulting, for example, from a

finite element discretization of the underlying PDE on a coarse mesh, may be impractical due to the large numbers of degrees of freedom involved.

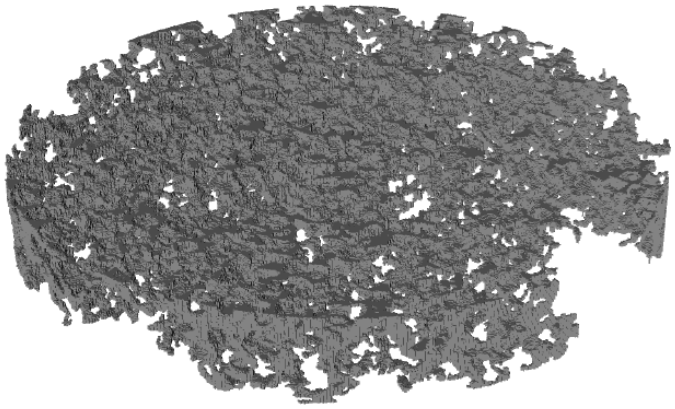


Figure 1: Example of a porous scaffold used for in vitro bone tissue growth, cf. [8, 9]. Figure taken from [4].

To overcome this problem in the last decade numerical methods that support computational meshes composed of polytopic elements (polygonal or polyhedral) have gained a lot of relevance. Indeed, one of the advantages of choosing polytopic element shapes over standard simplicial/hexahedral elements is that the average number of elements needed to discretize complicated domains is substantially smaller and this allows to reduce the complexity of the given computational geometry. This advantage becomes even more evident whenever the domain contains complex geometrical features; polytopic elements are naturally suited to applications in complicated/moving domains, for example, in solid mechanics, fluid-structure interaction, geophysical problems including earthquake engineering and flows in fractured porous media, and mathematical biology, cf. [40, 45].

Therefore, in the last few years intensive research has been undertaken on employing FEMs based on computational meshes consisting of polytopic elements and many methods have been presented in the literature. In the conforming setting, we mention the Composite Finite Element Method, see, e.g., [38, 39], the Mimetic Finite Difference Method, see, e.g., [15, 16, 30], the Polygonal Finite Element Method, see, e.g., [47], and the Extended Finite Element Method [37]. Moreover we mention the more recent Virtual Element Method [10] and the Hybrid High-Order method [1, 11–13, 22, 23, 32–35]. In the setting of non-conforming/discontinuous polygonal methods, we mention, for example, Composite Discontinuous Galerkin Finite Element methods [2, 3], Hybridizable Discontinuous

Galerkin methods [25–28], non-conforming Virtual Element methods [6, 21, 31], and Gradient Schemes [36].

In this thesis we implement and validate a new general-purpose library called LYMPH3D, written in Fortran, implementing discontinuous Galerkin methods on polytopic grids (PolyDG) in 3D, which is the natural extension of the classical discontinuous Galerkin methods on standard element shape grids to meshes composed of polytopic elements. Due to the fact that the discrete space is constructed based on employing piecewise discontinuous polynomials, DG methods are naturally suited to robustly support polytopic meshes. We will consider the three-dimensional case so we will deal with tetrahedra as standard elements and polyhedra as polytopes.

The general idea is to generate an initial mesh, based on employing standard mesh generators; then the elements of the initial mesh are suitably agglomerated, thus generating polyhedra. In this way we can obtain computational polyhedral grids with a smaller number of elements and therefore the total number of degress of freedom is substantially reduced.

In the following we provide a brief description of the contents of each of the next chapters. In Chapter 1, we recall some concepts about polytopic meshes introduced in [19], starting from the generalization of the standard shape-regularity property to polytopic elements and we recall some trace and inverse inequalities together with some polynomial approximation properties of the underlying discrete spaces. In Chapter 2, we introduce the notation and the key theoretical results needed to analyze PolyDG approximations together with some implementation aspects. In Chapter 3 we describe the library LYMPH3D following the implementation of PolyDG method in the three-dimensional case. In Chapter 4 we present some numerical examples and the convergence analysis relative to the test done and in Chapter 5 we do some final considerations and we see possible future developments. Finally, in Appendix A we report a selection of some of the most important functions of the library.

1 | Numerical aspects of polytopic meshes

We begin by introducing a general class of computational meshes consisting of polytopic elements, together with some technical assumptions. Then we report hp-version inverse estimates and approximation results that can be derived under these assumptions. These results are needed for the stability and convergence analysis of the PolyDG method.

We will use the following notation. For an open, bounded domain $\Omega \subset \mathbb{R}^d$, $d = 2, 3$, we denote by $H^s(\Omega)$ the standard Sobolev space of order s, for s real number, $s \geq 0$. For $s = 0$ we will write $L^2(\Omega)$ instead of $H^0(\Omega)$. The norm on $H^s(\Omega)$ is denoted by $||\cdot||_{H^s(\Omega)}$ and the seminorm by $|\cdot|_{H^s(\Omega)}$. Given a decomposition of the domain into a computational mesh $\mathscr{T}_h$, we denote by $H^s(\mathscr{T}_h)$ the standard *broken* Sobolev space, equipped with the broken norm $||\cdot||_{s,\mathscr{T}_h}$. Moreover, for $v \in H^1(\mathscr{T}_h)$, the broken gradient $\nabla_h v$ is defined by $(\nabla_h v)|_E = \nabla(v|_E)$, $E \in \mathscr{T}_h$. The symbols $\lesssim$ and $\gtrsim$ will signify that the inequalities hold up to multiplicative constants that are independent of the discretization parameters, but might depend on the physical parameters of the underlying problem. We refer to [7] and [20] for the details of the following theoretical contents about polytopic meshes. We introduce the subdivision of the computational domain Ω on $\mathbb{R}^d, d = 2, 3$, by means of polytopic meshes following the notation in [7]. The same notation will be employed throughout the all work.

Let $\mathscr{T}_h$ be a subdivision of the computational domain $\Omega \subset \mathbb{R}^d, d = 2, 3$, into disjoint open polygonal/polyhedral elements E. For each element we denote by |E| its measure, h_E its diameter and we set $h = \max_{E \in \mathscr{T}_h} h_E$.
We introduce the concept of mesh *interfaces*, which are defined as the intersection of the $(d-1)$-dimensional facets of two neighbouring elements. We need now to distinguish between the case when $d = 3$ and $d = 2$.

- For $d = 3$ each interface consists of a general polygon which we assume may be decomposed into a set of co-planar triangles. We refer to these $(d-1)$-dimensional simplices, whose union forms the interfaces of $\mathscr{T}_h$ as the *faces* of the computational

mesh. We denote the set of all the triangles by $\mathscr{F}_h$.

- For $d = 2$ the interfaces of $\mathscr{T}_h$ are simply piecewise linear line segments, i.e they consist of a set of $(d-1)$-dimensional simplices. The concepts of face and interface are in this case coincident; we will still call *faces* the line segments and denote by $\mathscr{F}_h$ the set of all faces.

Notice that $\mathscr{F}_h$ is always defined as a set of $(d-1)$-dimensional simplices. With this notation, we assume that the sub-tessellation of element interfaces into $(d-1)$-dimensional simplices is given. We point out that this assumption is not very restrictive; indeed, if the underlying mesh $\mathscr{T}_h$ stems from the agglomeration of a given simplicial mesh $\mathscr{T}_h^{fine}$, then the set of faces may be directly determined from the faces present in $\mathscr{T}_h^{fine}$ which form part of the interface of an agglomerated element $E \in \mathscr{T}_h$. We introduce a partition of the set $\mathscr{F}_h$ into two subsets

$$\mathscr{F}_h = \mathscr{F}_h^I \cup \mathscr{F}_h^B \,,$$

where $\mathscr{F}_h^I$ is the set of interior faces and $\mathscr{F}_h^B$ is the set of faces on the boundary of the domain $\partial\Omega$.

Moreover, if $\partial\Omega$ is split into the Dirichlet boundary $\partial\Omega_D$ and the Neumann boundary $\partial\Omega_N$, we will further decompose the set $\mathscr{F}_h^B = \mathscr{F}_h^D \cup \mathscr{F}_h^N$, where $\mathscr{F}_h^D$ and $\mathscr{F}_h^N$ are the boundary faces contained in Γ_D and Γ_N, respectively. In this definition it is implicit the assumption that the mesh $\mathscr{T}_h$ conforms to the partition of $\partial\Omega$.
Finally, given an element $E \in \mathscr{T}_h$, for any face $F \subset \partial E$, with $F \in \mathscr{F}_h$, we define $\mathbf{n}_F$ as the unit normal vector on F which points outwards from E.
We denote by S_E^F a d-dimensional simplex contained in E which shares with E a specific face $F \subset \partial E$, $F \in \mathscr{F}_h$. We need this notation to delineate the key assumptions that need to be satisfied by the polytopic mesh $\mathscr{T}_h$ in order to derive inverse inequalities and approximation results. To do this, we introduce the following definition.

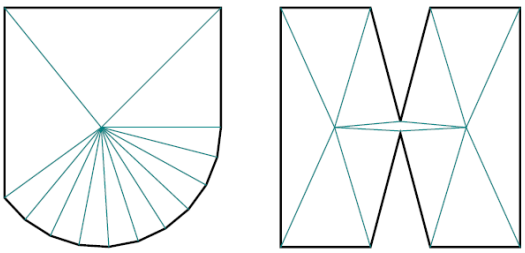
Definition 1.0.1. *A mesh $\mathscr{T}_h$ is said to be polytopic-regular if, for any $E \in \mathscr{T}_h$, there exists a set of non-overlapping (not necessarily shape-regular) d-dimensional simplices $\{S_E^F\}_{F\subset\partial E}$ contained in E, such that for all faces $F \subset \partial E$, the following condition holds*

$$h_E \lesssim \frac{d|S_E^F|}{|F|} \,,$$

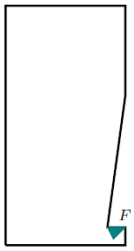
where the hidden constant is independent of the discretization parameters, the number of faces of the element, and the face measure.

In Figure 1.1a, taken from [7], we have two examples of polytopic-regular elements, while

Figure 1.1b shows an example of element that does not satifsy Definition 1.0.1, c.f. [7]. Moreover, in Figure 1.1a, there is an example where the union of the simplices $\{S_E^F\}_{F\subset\partial E}$ does not cover the whole element E. We note that this, in general might happen for elements that are polytopic-regular. We also remark that this definition does not require any restriction on either the number of faces per element or their relative measure.



(a) Figure taken from [7]. Two examples of polytopic-regular elements as in Definition 1.0.1. On the right there is an example of element that is not convered by the union of the simplices.



(b) Example of element that is not polytopic regular. Figure taken from [7].

Assumption 1.1. *We assume that the mesh $\mathscr{T}_h$ is polytopic-regular.*

Definition 1.1.1. *A covering $\mathscr{T}_\# = \{T_E\}$ related to the polytopic mesh $\mathscr{T}_h$ is a set of shape-regular d-dimensional simplices T_E, such that for each $E \in \mathscr{T}_h$, there exists a $T_E \in \mathscr{T}_\#$ such that $E \subsetneq T_E$.*

Assumption 1.2. *There exist a covering $\mathscr{T}_\#$ of $\mathscr{T}_h$ and a positive constant O_Ω, independent of the mesh parameters, such that*

$$\max_{E\in\mathscr{T}_h} card\{E' \in \mathscr{T}_h : E' \cap T_E \neq \emptyset, T_E \in \mathscr{T}_\# \ s.t.\ E \subset T_E\} \leq O_\Omega\,,$$

and $h_{T_E} \lesssim h_E$ for each pair $E \in \mathscr{T}_h$ and $T_E \in \mathscr{T}_\#$, with $E \subset T_E$.

Assumption 1.2 implies that when the computational mesh $\mathscr{T}_h$ is refined, the amount of overlap present in the covering $\mathscr{T}_\#$ remains bounded. We remark that Assumption 1.2 requires shape-regularity of the mesh covering $\mathscr{T}_\#$, but not shape-regularity of the computational mesh $\mathscr{T}_h$ itself.

1.0.1. Discrete spaces and trace operators

Let us consider a polytopic mesh partition $\mathscr{T}_h$ of the domain Ω and for each element $E \in \mathscr{T}_h$ we associate a local polynomial degree $p_E \geq 1$. We collect the p_E in the vector

$\mathbf{p} := \{p_E : E \in \mathscr{T}_h\}$. With this notation we introduce the following spaces.

$$\begin{aligned} V^{\mathbf{p}}(\mathscr{T}_h) &= \{v_h \in L^2(\Omega) : v|_E \in \mathbb{P}_{p_E}(E) \quad \forall E \in \mathscr{T}_h\}, \\ \mathbf{W}^{\mathbf{p}}(\mathscr{T}_h) &= \{\mathbf{w} \in [L^2(\Omega)]^d : \mathbf{w}|_E \in [\mathbb{P}_{p_E}(E)]^d \quad \forall E \in \mathscr{T}_h\} \\ \mathcal{W}^{\mathbf{p}}(\mathscr{T}_h) &= \{\mathbf{w} \in [L^2(\Omega)]^{d\times d}_{sym} : \mathbf{w}|_E \in [\mathbb{P}_{p_E}(E)]^{d\times d}_{sym} \quad \forall E \in \mathscr{T}_h\} \end{aligned} \tag{1.1}$$

We recall that $\mathbb{P}_p(E)$ denotes the space of polynomials of total degree p on E.
In order to efficiently deal with discontinuous functions, we now introduce average and jump operators on a face. Let $F \in \mathscr{F}_h^I$ be an interior face shared by the elements $E^{\pm}$. We define $\mathbf{n}^{\pm}$ to be the unit normal vectors on F pointing exterior to $E^{\pm}$, respectively. Then, for sufficiently regular scalar-valued, vector valued and tensor-valued functions q, $\mathbf{v}$, $\boldsymbol{\tau}$ respectively, we define the standard *average* $\{\cdot\}$ and *jump* $[\![\cdot]\!]$ operators on F as

$$\begin{aligned} \{q\} &= \frac{1}{2}(q^+ + q^-), \quad [\![q]\!] = q^+\mathbf{n}^+ + q^-\mathbf{n}^-, \\ \{\mathbf{v}\} &= \frac{1}{2}(\mathbf{v}^+ + \mathbf{v}^-), \quad [\![\mathbf{v}]\!] = \mathbf{v}^+ \cdot \mathbf{n}^+ + \mathbf{v}^- \cdot \mathbf{n}^-, \\ \{\boldsymbol{\tau}\} &= \frac{1}{2}(\boldsymbol{\tau}^+ + \boldsymbol{\tau}^-), \quad [\![\boldsymbol{\tau}]\!] = \boldsymbol{\tau}^+\mathbf{n}^+ + \boldsymbol{\tau}^-\mathbf{n}^-, \end{aligned} \tag{1.2}$$

where the subscript $\pm$ on q, $\mathbf{v}$,$\boldsymbol{\tau}$ denote the traces of the functions on F restricted to $E^{\pm}$, respectively.

On a boundary face $F \in \mathscr{F}_h^B$, we set analogously $\{q\} = q, [\![q]\!] = q\mathbf{n}, \{\mathbf{v}\} = \mathbf{v}, [\![\mathbf{v}]\!] = \mathbf{v} \cdot \mathbf{n}, \{\boldsymbol{\tau}\} = \boldsymbol{\tau}, [\![\boldsymbol{\tau}]\!] = \boldsymbol{\tau}\mathbf{n}$, where $\mathbf{n}$ is the outward normal vector on $\partial\Omega$. We remark two important identities:

$$\begin{aligned} [\![q\mathbf{v}]\!] &= [\![\mathbf{v}]\!]\{q\} + \{v\} \cdot [\![q]\!]\,, \\ \sum_{E\in\mathscr{T}_h} \int_{\partial E} q\mathbf{v} \cdot \mathbf{n}_E &= \int_{\mathscr{F}_h} \{\mathbf{v}\} \cdot [\![q]\!] + \int_{\mathscr{F}_h^I} [\![\mathbf{v}]\!]\{q\}\,, \end{aligned} \tag{1.3}$$

where we used the compact notation $\int_{\mathscr{F}_h} \cdot = \sum_{F\in\mathscr{F}_h} \int_F \cdot\,\cdot$

1.0.2. Trace inverse estimate on polytopic elements

Among the key tools employed to study the stability and error analysis of DG-methods we find the trace inverse estimates. They consist in using the norm of a polynomial on the element itself to bound the norm on the element's face/edge. Trace inverse estimates on polytopic elements are obtained under the polytopic-regular Assumption 1.1.

Lemma 1.3. *Let E be a polytope satisfying Assumption 1.1 and let $q \in \mathbb{P}_{p_E}(E)$. Then,*

we have

$$||q||^2_{L^2(\partial E)} \lesssim \frac{p_E^2}{h_E}||q||^2_{L^2(E)}\,, \tag{1.4}$$

where the hidden constant depends on the dimension d, but it is independent of the discretization parameters and the number of faces that the element possesses.

The proof of Lemma 1.3 can be found in [7].

1.0.3. Polynomial approximation over polytopic elements

Hp-interpolation estimates are another fundamental mathematical tool needed to study the a priori error analysis of PolyDG methods. Following [7], we report the results of [4, 20] where the standard estimates for simplices are extended to polytopic elements by considering appropriate coverings and submeshes consisting of d-dimensional simplices (where standard results can be applied). In [18] these results are further extended in order to be successfully applied also in the case when the number of edges/faces is unbounded. In order to state these results we need to define an appropriate extension operator.

Let $\mathscr{E} : H^m(E) \to H^m(\mathbb{R}^d)$, for any $E \in \mathscr{T}_h$ and $m \geq 0$, be the continuous extension operator introduced by Stein in [46] such that:

$$\mathscr{E}(q)|_\Omega = q, \qquad ||\mathscr{E}q||_{H^m(\mathbb{R}^d)} \lesssim ||q||_{H^m(\Omega)}, \qquad \forall q \in H^s(E).$$

Based on the existance of a suitable covering of the polytopic mesh (see Definition 1.1.1), we can state the following approximation result.

Lemma 1.4. *[4, 18, 20] Assume that Assumptions 1.1 and 1.2 are satisfied. Given $E \in \mathscr{T}_h$, let $T_E \in \mathscr{T}_\#$ be the corresponding simplex such that $E \subset T_E$ (see Definition 1.1.1). For $q \in L^2(\Omega)$, such that $\mathscr{E}q|_{T_E} \in H^{r_E}(T_E)$, for some $r_E \geq 0$, there exists a sequence of approximations $\Pi_E^{p_E} q \in \mathbb{P}_{p_E}(E)$, $p_E = 0, 1, 2, \ldots$, of q satisfying*

$$||q - \Pi_E^{p_E} q||_{H^m(E)} \lesssim \frac{h_E^{s_E - m}}{p_E^{r_E - m}}||\mathscr{E}q||_{H^{r_E}(T_E)}, \quad 0 \leq m \leq r_E. \tag{1.5}$$

Moreover, if $r_E \geq 1 + d/2$, then we have

$$||q - \Pi_E^{p_E} q||_{L^2(\partial E)} \lesssim \frac{h_E^{s_E - 1/2}}{p_E^{r_E - 1/2}}||\mathscr{E}q||_{H^{r_E}(T_E)}. \tag{1.6}$$

Here, $s_E = \min(p_E + 1, r_E)$ and the hidden constants depend on the shape-regularity of T_E, but are independent of q, h_E, p_E and the number of faces per element.

See [20] for a detailed proof of (1.5) and [18] for the proof of (1.6). We notice that the inequalities (1.4) and (1.6) hold not only on one of its edges/faces but on the whole boundary of E; this is important when we consider elements that contain an arbitrary number of faces in the error analysis.

2 | Discontinuous Galerkin methods on polytopic meshes

In this chapter, following [20] we introduce a generic second-order PDE with nonnegative characteristic form together with some notation. Next, we focus on a diffusion reaction problem, and we describe its PolyDG discretization. Finally, we discuss some implementation aspects. That will be necessary to understand the description of the library, central topic of Chapter 3.

2.1. PDEs with nonnegative characteristic form

Throughout this section, we introduce a linear second-order PDE problem with nonnegative characteristic form. Indeed, we stress that this class of equations includes a wide range of PDEs. Given Ω an open bounded, Lipschitz domain in $\mathbb{R}^d, d \geq 1$ with boundary $\partial\Omega$, consider the following PDE problem:

find $u : \Omega \to \mathbb{R}$ such that

$$-\nabla \cdot (a\nabla u) + \nabla \cdot (bu) + cu = f \text{ in } \Omega, \tag{2.1}$$

where $a = \{a_{ij}\}_{i,j=1}^d$, with $a_{ij} \in L^\infty(\Omega)$ and $a_{ij} = a_{ji}$ for $i, j = 1, \ldots, d$,
$\mathbf{b} = (b_1, \ldots, b_d)^\intercal \in [W^{1,\infty}(\Omega)]^d$, $c \in L^\infty(\Omega)$, $c \geq 0$ and $f \in L^2(\Omega)$.

Notice that $\in W^{m,p}(\Omega)$ is the Sobolev space defined as

$$W^{m,p}(\Omega) = \{u \in L^p(\Omega) \, : \, D^\alpha u \in L^p(\Omega) \text{ for } |\alpha| \leq m\}.$$

Problem (2.1) is referred to as an equation with nonnegative characteristic form on the set $\Omega \in \mathbb{R}^d$ if, at each $\mathbf{x} \in \bar{\Omega}$,

$$\sum_{i,j=1}^{d} a_{ij}(\mathbf{x})\xi_i\xi_j \geq 0 \tag{2.2}$$

for any vector $\boldsymbol{\xi} = (\xi_1, \ldots, \xi_d)$ in $\mathbb{R}^d$. To add suitable boundary conditions to the problem (2.1), we first subdivide the boundary $\partial\Omega$ of the computational domain Ω into appropriate subsets. To this end, let

$$\partial_0\Omega = \Big\{\mathbf{x} \in \partial\Omega : \sum_{i,j=1}^{d} a_{ij}(\mathbf{x})\, n_i n_j > 0\Big\},$$

where $\mathbf{n} = (n_1, \ldots, n_d)^\intercal$ denotes the unit outward normal vector to $\partial\Omega$.
On the 'hyperbolic' portion of the boundary $\partial\Omega\backslash\partial_0\Omega$ we define the inflow and outflow boundaries $\partial_-\Omega$ and $\partial_+\Omega$, respectively, by

$$\partial_-\Omega = \{\mathbf{x} \in \partial\Omega\backslash\partial_0\Omega : \mathbf{b}(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x}) < 0\},$$
$$\partial_+\Omega = \{\mathbf{x} \in \partial\Omega\backslash\partial_0\Omega : \mathbf{b}(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x}) \geq 0\}.$$

If $\partial_0\Omega$ is nonempty, we shall further divide it into two disjoint subsets $\partial\Omega_D$ and $\partial\Omega_N$, with $\partial\Omega_D$ nonempty and relatively open in $\partial\Omega$. From these definitions we have that $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N \cup \partial_-\Omega \cup \partial_+\Omega$. Assuming the (physically reasonable) hypothesis that $\mathbf{b}\cdot\mathbf{n} \geq 0$ on $\partial\Omega_N$, whenever $\partial\Omega_N$ is nonempty, we impose the following boundary conditions:

$$u = g_D \quad \text{on } \partial\Omega_D \cup \partial_-\Omega, \qquad \mathbf{n}\cdot(a\nabla u) = g_N \quad \text{on } \partial\Omega_N. \tag{2.3}$$

For an extension of this setting, allowing also for $\mathbf{b}\cdot\mathbf{n} < 0$ on $\partial\Omega_N$, we refer to [17].
The well-posedness of the boundary value problem (2.1), (2.3), in the case of homogeneous boundary conditions, has been studied in [41].

2.2. PolyDG discretization of diffusion reaction problems

We now consider the PolyDG discretization of a diffusion reaction PDE problem. Then we will see some convergence results related to this problem.

Given an open bounded Lipschitz domain Ω in $\mathbb{R}^d$, $d = 2, 3$, with boundary $\partial\Omega$, we consider the following PDE boundary-value problem subject to a Dirichlet boundary condition ($\partial\Omega_D = \partial\Omega$ and $\partial\Omega_N = \emptyset$):

find u such that

$$\begin{cases} -\nabla\cdot(a\nabla u) + cu = f & \text{in } \Omega, \\ u = g_D & \text{on } \partial\Omega. \end{cases} \tag{2.4}$$

Here, $f \in L^2(\Omega)$, $c \in L^\infty(\Omega)$, $a = \{a_{ij}\}_{i,j=1}^d$, with $a_{ij} \in L^\infty(\Omega)$ and $a_{ij} = a_{ji}$, for $i, j = 1, \ldots, d$, and, at each $\mathbf{x} \in \bar{\Omega}$,

$$\sum_{i,j=1}^{d} a_{ij}(\mathbf{x})\xi_i\xi_j \geq C_d|\xi|^2 > 0, \tag{2.5}$$

where C_d is a positive constant, for any vector $\xi = (\xi_1, \ldots, \xi_d)$ in $\mathbb{R}^d$.

The well-posedness of the boundary value problem (2.4), under the uniform ellipticity condition (2.5) can be deduced, based on employing the Lax-Milgram Theorem. See, for example, [14, 24].

Given the partition $\mathscr{T}_h$ of the domain Ω we recall the definition of the finite element space $V^{\mathbf{p}}(\mathscr{T}_h)$ that we introduced in Chapter 1:

$$V^{\mathbf{p}}(\mathscr{T}_h) = \{v \in L^2(\Omega) : v|_E \in \mathbb{P}_{p_E}(E) \quad \forall E \in \mathscr{T}_h\}.$$

By construction, the local elemental polynomial spaces employed within the definition of $V^{\mathbf{p}}(\mathscr{T}_h)$ are defined in the physical space, without the need to map from a given reference or canonical frame, as it typically the case for classical FEMs.

With this notation and with the boundary conditions imposed weakly we introduce the following DG formulation.

Find $u_h \in V^{\mathbf{p}}(\mathscr{T}_h)$ such that

$$\begin{aligned}\int_\Omega a\nabla_h u_h \cdot \nabla_h v_h \, d\mathbf{x} + \int_\Omega c\, u_h v_h \, d\mathbf{x} + \int_{\mathscr{F}_h} (-\{a\nabla_h u_h\} \cdot [\![v_h]\!] + \theta\{a\nabla_h v_h\} \cdot [\![u_h]\!])ds \\ + \int_{\mathscr{F}_h} \sigma[\![u_h]\!] \cdot [\![v_h]\!] \, ds = \int_\Omega f v_h \, d\mathbf{x} + \int_{\mathscr{F}_h^B} g_D(\theta a\nabla_h v_h \cdot \mathbf{n} + \sigma v_h) \, ds\end{aligned} \tag{2.6}$$

for all $v_h \in V^{\mathbf{p}}(\mathscr{T}_h)$, where $\nabla_h v_h$ denotes the broken gradient of v_h, $\theta \in \{-1, 0, 1\}$, $\sigma : \mathscr{F}_h \mapsto \mathbb{R}$ is referred to as the discontinuity-penalization function; the precise definition of σ depends on the local mesh size and the local polynomial degree. For the derivation of this formualation we need Definitions 1.2 and the identities (1.3) introduced in Chapter 1. For the details of this derivation see [20].

Notice that here we consider one popular family of schemes, referred to as interior penalty (IP) methods. The discrete formulation of these kind of methods sees the presence of two integral terms, one on the left and the other on the right hand side, multiplied by a

constant $\theta \in \{-1, 0, 1\}$:

$$\theta \int_{\mathscr{F}_h} \{a\nabla_h v_h\} \cdot [\![u_h]\!] ds, \qquad \theta \int_{\mathscr{F}_h^B} g_D \, a\nabla_h v_h \cdot \mathbf{n} \, ds \, .$$

Choosing $\theta = -1$ the method preserves the symmetry and the resulting formulation is called Symmetric Interior Penalty (SIP) method. Instead, setting $\theta = 1$ gives rise to the so-called Nonsymmetric Interior Penalty (NIP) method while $\theta = 0$ yelds the Incomplete Interior Penalty (IIP) method. Here we consider the Symmetric Interior Penalty PolyDG method, meaning choosing $\theta = -1$ in the previous discrete formulation (2.6). With this choice we write the following definitions.

- $B_d : V^{\mathbf{p}} \times V^{\mathbf{p}} \to \mathbb{R}$ is the bilinear form such that

$$\begin{aligned} B_d(w_h, v_h) = &\sum_{E \in \mathscr{T}_h} \int_E a\nabla w_h \cdot \nabla v_h \, d\mathbf{x} \\ &- \int_{\mathscr{F}_h} (\{a\nabla w_h\} \cdot [\![v_h]\!] + \{a\nabla v_h\} \cdot [\![w_h]\!] - \sigma [\![w_h]\!] \cdot [\![v_h]\!]) \, ds \, . \end{aligned} \tag{2.7}$$

- $B_r : V^{\mathbf{p}} \times V^{\mathbf{p}} \to \mathbb{R}$ is the bilinear form such that

$$B_r(w_h, v_h) = \sum_{E \in \mathscr{T}_h} \int_E c \, w_h v_h \, d\mathbf{x} \, . \tag{2.8}$$

- $B_h : V^{\mathbf{p}} \times V^{\mathbf{p}} \to \mathbb{R}$ is the bilinear form such that

$$B_h(w_h, v_h) = B_d(w_h, v_h) + B_r(w_h, v_h) \, . \tag{2.9}$$

- $F : V^{\mathbf{p}} \to \mathbb{R}$ is the linear functional such that

$$F(v_h) = \sum_{E \in \mathscr{T}_h} \int_E f v_h \, d\mathbf{x} - \int_{F_h^B} g_D (a\nabla v_h \cdot \mathbf{n} - \sigma v_h) \, ds \, . \tag{2.10}$$

Thereby, using these definitions we obtain the following (SIP) PolyDG discrete formulation.

Find $u_h \in V^{\mathbf{p}}(\mathscr{T}_h)$ such that

$$B_h(u_h, v_h) = F(v_h) \tag{2.11}$$

for all $v_h \in V^{\mathbf{p}}(\mathscr{T}_h)$. The well-posedness and stability properties of the above method

depend on the choice of the discontinuity-penalization σ. We expect that the choice of σ will be sensitive to the size of each face $F \in \mathscr{F}_h$, relative to the size of the element(s) which form F. In [20] we can find the convegence analysis for two cases based on employing different assumptions on the elements present in the computational mesh $\mathscr{T}_h$, assuming that the entries of a are constant on each element E, $E \in \mathscr{T}_h$, i.e., $a \in [V^{\mathbf{0}}(\mathscr{T}_h)]^{d\times d}_{sym}$. Then, the extension to general positive (semi)-definite diffusion tensors is treated in [19].

Here, for semplicity we report the results of the analysis of the problem in the case of a polytopic-regular computational mesh and choosing $c = 0$ and $a = I_d$ where I_d is the $d \times d$ identity matrix. Therefore, we refer to the bilinear form $B_h(\cdot,\cdot)$ as

$$B_h(w, v) = B_d(w, v) \quad \forall\, v, w \in V^{\mathbf{p}},$$

where $B_d(\cdot,\cdot)$ is defined as in (2.7) with $a = I_d$. We consider the linear functional $F(\cdot)$ is defined as in (2.10) with $a = I_d$.

The penalization function σ is face-wise defined as $\sigma : \mathscr{F}_h \to \mathbb{R}^+$ such that

$$\sigma = \alpha \begin{cases} \frac{p_E^2}{h_E} & \text{on } F \in \mathscr{F}_h^B \\ \frac{\max\{p_{E^+}^2, p_{E^-}^2\}}{\min\{h_{E^+}, h_{E^-}\}} & \text{on } F \in \mathscr{F}_h^I \end{cases}, \tag{2.12}$$

where α is a constant to be chosen large enough.

2.2.1. Well-Posedness of the PolyDG method and a priori error estimates

In this section, we report the stability results and the a priori estimates of the PolyDG method under the assumption of a polytopic-regular mesh $\mathscr{T}_h$.

We define the space $\mathscr{V} = H^1(\Omega) \oplus V^{\mathbf{p}}(\mathscr{T}_h)$ and we introduce the associated DG norm given by:

$$||v||^2_{DG} = \sum_{E\in\mathscr{T}_h} ||\nabla v||^2_{L^2(E)} + ||\sigma^{\frac{1}{2}}[\![v]\!]||^2_{L^2(\mathscr{F}_h)} \quad \forall v \in \mathscr{V},$$

where we used the notation $||\cdot||_{L^2(\mathscr{F}_h)} = \sum_{F\in\mathscr{F}_h} ||\cdot||_{L^2(F)}$.

We also introduce $\mathbf{\Pi}_{L^2} : [L^2(\Omega)]^d \to [V^{\mathbf{p}}(\mathscr{T}_h)]^d$ to denote the orhogonal L^2-projection onto the finite element space $[V^{\mathbf{p}}(\mathscr{T}_h)]^d$. With this notation, we define the suitable extensions

of the bilinear form $B_h(\cdot,\cdot)$:

$$\begin{aligned}\tilde{B}_h(w,v) = &\sum_{E\in\mathscr{T}_h}\int_E \nabla w\cdot\nabla v\,d\mathbf{x}\\ &-\int_{\mathscr{F}_h}\left(\{\mathbf{\Pi}_{L^2}(\nabla w)\}\cdot[\![v]\!]+\{\mathbf{\Pi}_{L^2}(\nabla v)\}\cdot[\![w]\!]-\sigma[\![w]\!]\cdot[\![v]\!]\right)ds,\end{aligned} \tag{2.13}$$

and of the linear functional $F(\cdot)$:

$$\tilde{F}(v) = \sum_{E\in\mathscr{T}_h}\int_E fv\,d\mathbf{x} - \int_{F_h^B} g_D(\mathbf{\Pi}_{L^2}(\nabla v)\cdot\mathbf{n}-\sigma v)\,ds, \tag{2.14}$$

for all $v, w \in \mathscr{V}$.

Then the PolyDG formulation may be rewritten in the following equivalent manner:

find $u_h \in V^{\mathbf{p}}(\mathscr{T}_h)$ such that

$$\tilde{B}_h(u_h, v_h) = \tilde{F}(v_h), \tag{2.15}$$

for all $v_h \in V^{\mathbf{p}}(\mathscr{T}_h)$.

For all $w, v \in V^{\mathbf{p}}(\mathscr{T}_h)$, we have $\tilde{B}_d(w,v) = B_d(w,v)$ and $\tilde{F}(v) = F(v)$, i.e., the two formulations give rise to the same PolyDG method. This formulation enables us to pursue the analysis without requiring $W^{1,\infty}$-norm approximation estimates, as we shall see below. Assuming that Assumption 1.1 holds and recalling the results of Section 1, we write below the coercivity and continuity bounds for the (extended) bilinear form $\tilde{B}_h(\cdot,\cdot)$ over $\mathscr{V}\times\mathscr{V}$.

Lemma 2.1. *Given that Assumption 1.1 holds and that the constant α appearing in the Definition 2.12 of the penalization function is chosen sufficiently large. Then, the bilinear form $\tilde{B}_h(\cdot,\cdot)$ is coercive and continuous over $\mathscr{V}\times\mathscr{V}$, i.e.,*

$$\tilde{B}_h(v,v) \gtrsim ||v||^2_{DG} \quad \textit{for all} \quad v\in\mathscr{V}, \tag{2.16}$$

and

$$\tilde{B}_h(w,v) \lesssim ||w||_{DG}||v||_{DG} \quad \textit{for all} \quad w,v\in\mathscr{V}. \tag{2.17}$$

Proof. The proof is based on writing the following identity:

$$\tilde{B}_h(v,v) = ||v||^2_{DG} - 2\int_{F_h}\{\mathbf{\Pi}_{L^2}(\nabla v)\}\cdot[\![v]\!]\,ds. \tag{2.18}$$

Now, bounding the second term on the right hand side of (2.18) using the trace inverse estimate stated in Lemma (1.3) and the stability of the L^2-projector $\mathbf{\Pi}_{L^2}$ in the L^2-norm, namely $||\mathbf{\Pi}_{L^2}\mathbf{v}||_{L^2(E)} \leq ||\mathbf{v}||_{L^2(E)}$, for $\mathbf{v} \in [\mathcal{V}]^d$, $E \in \mathscr{T}_h$, we obtain the coercivity and the continuity of the bilinear form $\tilde{B}_h(\cdot,\cdot)$. See [20] for the details. □

Hence, we report the following a priori error estimate assuming that Assumption 1.1 holds.

Theorem 2.2. *Let $\mathscr{T}_h = \{E\}$ be a subdivision of $\Omega \subset \mathbb{R}^d$, $d = 2, 3$ consisting of general polytopic elements satisfying Assumptions 1.1 and 1.2, with $\mathscr{T}_\# = \{T_E\}$ the associated covering of $\mathscr{T}_h$, cf. Definition 1.1.1. Let $u_h \in V^{\mathbf{p}}(\mathscr{T}_h)$, with $p_E \geq 1$ for all $E \in \mathscr{T}_h$ be the corresponding PolyDG solution defined by (2.11), where the discontinuity-penalization function σ is given by (2.12) with a penalty parameter α sufficiently large. If the analytical solution $u \in H^1(\Omega)$ of (2.1) satisfies $u|_E \in H^{r_E}(E), r_E > 3/2$, for each $E \in \mathscr{T}_h$, such that $\mathscr{E}u|_{T_E} \in H^{r_E}(T_E)$, where $T_E \in \mathscr{T}_\#$, with $E \subset T_E$, then*

$$||u - u_h||^2_{DG} \lesssim \sum_{E\in\mathscr{T}_h} \frac{h_E^{2(s_E-1)}}{p_E^{2(r_E-\frac{3}{2})}} ||\mathscr{E}u||^2_{H^{r_E}(T_E)}, \tag{2.19}$$

with $s_E = \min\{p_E + 1, r_E\}$ for all $E \in \mathscr{T}_h$. The hidden constants depend on the material parameters and the shape-regularity of $\mathscr{T}_\#$, but is independent of h_E, p_E and the number of element faces.

We refer to [20] for the proof of this result considering a more general definition of σ. To prove this result we need Lemma 1.4, in particular the esimate (1.6) to bound the integral term defined on the faces of the elements.

Now we report an a priori error estimate in the L^2-norm obtained by using a duality argument. In the following, we assume uniform orders, $p_E = p$ for all $E \in \mathscr{T}_h$, $p \geq 1$ and $h = \max_{E\in\mathscr{T}_h} h_E$. Moreover we assume that Ω is sufficiently regular so that u possesses the following regularity, $u \in H^r(\Omega)$ for some $r \geq 2$. Since we are using a duality argument, we also assume that Ω is sufficiently regular so that, for $g \in L^2(\Omega)$ the problem

$$-\nabla z = g \quad \text{in } \Omega, \quad z = 0 \quad \text{on } \partial\Omega \tag{2.20}$$

is well posed and its unique solution z satisfies the following elliptic regularity: $z \in H^2(\Omega)$ and $||z||_{H^2(\Omega)} \lesssim ||g||_{L^2(\Omega)}$. Then, the following holds.

Theorem 2.3. *Let $\mathscr{T}_h = \{E\}$ be a subdivision of $\Omega \subset \mathbb{R}^d$, $d = 2, 3$ consisting of general polytopic elements satisfying Assumptions 1.1 and 1.2 , with $\mathscr{T}_\# = \{T_E\}$ the associated covering of $\mathscr{T}_h$, cf. Definition 1.1.1. Let $u_h \in V^p(\mathscr{T}_h)$ be the corresponding PolyDG*

solution defined by (2.11), where the discontinuity-penalization function σ is given by (2.12) with with a penalty parameter α sufficiently large. Then,

$$||u - u_h||_{L^2(\Omega)} \lesssim \frac{h^s}{p^{r-1}}||u||_{H^s(\Omega)}, \tag{2.21}$$

where with $s = \min\{p+1, r\}$. The hidden constants depend on the material parameters and the shape-regularity of $\mathscr{T}_\#$, but is independent of h, p.

For $c \neq 0$ we have to make the assumption that there exists a positive constant γ_0 such that $c(x) \geq \gamma_0$ a.e. $x \in \Omega$. In this case we refer to the general bilinear form $B_h(\cdot,\cdot)$ as

$$B_h(w,v) = B_d(w,v) + B_r(w,v) \quad \forall\, v, w \in V^{\mathbf{p}},$$

where $B_d(\cdot,\cdot)$ is defined as in (2.7) with $a = I_d$. We introduce the following modified DG norm:

$$||v||^2_{DG} = ||v||^2_d + ||v||^2_r \quad \forall v \in \mathscr{V},$$

where the norm $||\cdot||_d$ is defined as

$$||v||^2_d = \sum_{E\in\mathscr{T}_h} ||\nabla v||^2_{L^2(E)} + ||\sigma^{\frac{1}{2}}[\![v]\!]||^2_{L^2(\mathscr{F}_h)} \quad \forall v \in \mathscr{V},$$

and the norm $||\cdot||_r$ is defined as

$$||v||^2_r = \sum_{E\in\mathscr{T}_h} ||c^{\frac{1}{2}}v||^2_{L^2(E)} \quad \forall v \in \mathscr{V}.$$

It is easy to see that $B_r(v,v) = ||v||^2_r$. The coercivity and the continuity of $B_d(\cdot,\cdot)$ on $\mathscr{V}\times\mathscr{V}$ with respect to the norm $||\cdot||_d$ follows from Lemma 2.1. The error estimates in norm DG (2.19) and in norm L^2 (2.21) holds also in this case, where $||\cdot||_{DG}$ is the modified DG norm. See [20] for the details.

2.3. Implementation aspects

2.3.1. Basis functions

In this section we recall the approach to construct the discrete space proposed in [20] and in [19]. This approach is based on first employing polynomial spaces over a chosen bounding box of each element $E \in \mathscr{T}_h$; then the element basis is simply constructed by restricting this space to E. More precisely, given an element $E \in \mathscr{T}_h$, we write B_E to denote its corresponding bounding box; selecting, for example, B_E to be the Cartesian bounding box, i.e., the sides of B_E are aligned with the Cartesian axes, then B_E can be easily constructed, such that $\bar{E} \subseteq \bar{B}_E$. In Figure 2.1 there is an example of a polygonal element E in $\mathbb{R}^2$ with its Cartesian bounding box B_E. On this Cartesian bounding box B_E we may define a standard polynomial space $\mathbb{P}_{p_E}(B_E)$ spanned by a set of basis functions $\phi_{i,E}$, $i, \ldots, N_{p_E} = \dim(\mathbb{P}_{p_E}(B_E))$.

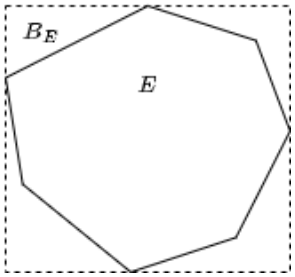


Figure 2.1: Cartesian bounding box for a polygon

Writing $B_E = \mathscr{I}_1 \times \mathscr{I}_2 \times \ldots \times \mathscr{I}_d$, where $\mathscr{I}_j, j = 1, \ldots, d$ and selecting $\hat{B} = (-1, 1)^d$ to be the reference hypercube, the bounding box B_E may be affinely mapped to $\hat{B}$, via the mapping $G_E : \hat{B} \to B_E$ such that

$$\mathbf{x} = G_E(\hat{\mathbf{x}}) = J_E\hat{\mathbf{x}} + \mathbf{c}, \tag{2.22}$$

where $J_E = \mathrm{diag}(h_1, \ldots, h_d)$, $\mathbf{c} = (m_1, \ldots, m_d)^\intercal$, and $\hat{\mathbf{x}}$ is a generic point in $\hat{B}$. We have that h_j is half of the length of the jth-side of B_E and m_j is the midpoint of $\mathscr{I}_j$, i.e respectively $h_j = (x_2^j - x_1^j)/2, j = 1, \ldots, d$ and $m_j = (x_2^j + x_1^j)/2, j = 1, \ldots, d$.

On $\hat{B}$ we may define employ tensor-product Legendre polynomials; to this end, writing $\{\hat{L}_i(\hat{x})\}_{i=0}^{\infty}$ to denote the family of $L^2(-1, 1)$-orhogonal Legendre polynomials, for example,

the space of polynomials $\mathbb{P}_{p_E}(\hat{B})$ of total degree p_E over $\hat{B}$ is given by

$$\mathbb{P}_{p_E}(\hat{B}) = span\{\hat{\phi}_{i,E}\}_{i=1}^{\dim(\mathbb{P}_{p_E}(\hat{B}))},$$

where

$$\hat{\phi}_{i,E}(\hat{\mathbf{x}}) = \hat{L}_{i_1}(\hat{x}_1)\hat{L}_{i_2}(\hat{x}_2)\dots\hat{L}_{i_d}(\hat{x}_d), \quad i_1+i_2+\dots+i_d \leq p_E, \quad i_E \geq 0,\, k=1,\dots,d\,,$$

and $\hat{\mathbf{x}} = (\hat{x}_1, \hat{x}_2, \dots, \hat{x}_d) \in [-1,1]$. Moreover we recall that

$$\hat{L}_i(x) = \frac{\mathscr{L}_i(x)}{||\mathscr{L}_i||_{L^2(-1,1)}}, \quad \text{with } \mathscr{L}_i(x) = \frac{1}{2^n i!}\frac{d}{dx}[(x^2-1)^n]\,.$$

Writing $L_i(x) = \hat{L}_i((x-m_j)/h_j)$, under the transformation (2.22), the space of polynomials $\mathbb{P}_{p_E}(B_E)$ of total degree p_E over B_E is given by

$$\mathbb{P}_{p_E}(B_E) = span\{\phi_{i,E}\}_{i=1}^{N_{p_E}},$$

where

$$\phi_{i,E}(\mathbf{x}) = L_{i_1}(x_1)L_{i_2}(x_2)\dots L_{i_d}(x_d), \quad i_1+i_2+\dots+i_d \leq p_E, \quad i_E \geq 0,\, k=1,\dots,d$$

and $\mathbf{x} = (x_1, x_2 \dots, x_d)$. Thereby, the polynomial basis over the general polytopic element E may be defined by simply restricting the support of $\{\phi_{i,E}\}, i = 1,\dots,N_{p_E}$ to E; i.e., the polynomial basis defined over E is given by $\{\phi_{i,E}|_E\}, i = 1,\dots,N_{p_E}$.

2.3.2. Quadrature rules

The design of efficient and accurate quadrature rules for general polytopes is a challenging task; while several approaches have been proposed within the literature, this still remains an open and active area of research. Below we report only one approach that is the one that has been implemented in the library LYMPH3D. Again we refer to [20] for the description of this approach.

Sub-Tessellation

The simplest approach is to simply construct a sub-tessellation of each polytopic element into standard element shapes, upon which standard quadrature rules may be employed. More precisely, given $E \in \mathscr{T}_h$, we first construct a non-overlapping sub-tessellation $E_{\mathscr{I}} = \{\tau_k\}$ consisting of standard element shapes, i.e. tetrahedra. Here, a general hybrid

sub-tessellation consisting of quadrilateral and triangular elements in $\mathbb{R}^2$, or tetrahedral, hexahedral, prismatic, and pyramidal elements in $\mathbb{R}^3$, may be constructed. On agglomerated meshes, the sub-tessellation will already be available; however, for reasons of efficiency, one may still wish to construct an alternative sub-tessellation that has a minimal number of elements. As an example, if we consider computing the PolyDG mass matrix, restricted to $E \in \mathscr{T}_h$, then we have that

$$\int_E wv\, d\mathbf{x} = \sum_{\tau_k \in E_{\mathscr{I}}} \int_{\tau_k} wv\, d\mathbf{x} \approx \sum_{\tau_k \in E_{\mathscr{I}}} \sum_{q=1}^{n_{q_3}} w(F_k(\boldsymbol{\xi}_q))v(F_k(\boldsymbol{\xi}_q))det(J_{F_k}(\boldsymbol{\xi}_q))w_q\,, \tag{2.23}$$

where $F_k : E_R \to \tau_k$ is the mapping from the reference element E_R to τ_k, with Jacobi matrix J_{F_k}, and $(\boldsymbol{\xi}_q, w_q)_{q=1}^{n_{q_3}}$ denotes the quadrature rule defined on E_R. We point out that while quadrature schemes based on employing a subtessellation of each polytopic element are straightforward to implement, they tend to be computationally expensive, in the sense that, depending on the cardinality of the sub-tessellation, the number of required function evaluations may be very large. This is particularly the case when the sub-tessellation employed is simply the background fine mesh $\mathscr{T}_h^{fine}$ used to construct a coarse agglomerated grid. For this reason more sophisticated quadrature free approaches have been proposed, see [5], however this is not considered in this work and will be the subject of future research.

2.3.3. Assembling of the algebraic linear system

We now cosider a second-order elliptic PDE problem with $a = I_d, \mathbf{b} = \mathbf{0}$ and $c \in \mathbb{R}$, subject to a Dirichlet boundary condition ($\partial\Omega_D$ and $\partial\Omega_N = \emptyset$), given by:
find $u \in H^1(\Omega)$ such that

$$\begin{cases} -\Delta u + cu = f \text{ in } \Omega\,, \\ \qquad\quad u = g_D \text{ on } \partial\Omega\,. \end{cases} \tag{2.24}$$

The PolyDG formulation of the problem is reported in (2.11) with $a = I_d$ in the definition of the bilinear form $B_d(\cdot,\cdot)$ and of the linear functional $F(\cdot)$.
For simplicity we consider $g_D = 0$. At the end of the section we will present the case of $g_D \neq 0$. By fixing a basis $\{\phi_i\}_{i=1}^{N_h}$, N_h denoting the dimension of the discrete space $V^{\mathbf{P}}(\mathscr{T}_h)$, (2.11) can be rewritten as:
find $\mathbf{u} \in \mathbb{R}^{N_h}$

$$(\mathbf{A} + c\,\mathbf{M})\mathbf{u} = \mathbf{f}, \tag{2.25}$$

where $\mathbf{u}$ contains the expansion coefficients of $u_h \in V^{\mathbf{P}}(\mathscr{T}_h)$,

- $\mathbf{f}$ is the right hand side vector given by

$$\mathbf{f}_i = \int_\Omega f\phi_i \, d\mathbf{x}, \quad i = 1, \dots, N_h. \tag{2.26}$$

- $\mathbf{M}$ is the mass matrix given by

$$\mathbf{M}_{i,j} = \int_\Omega \phi_i \, \phi_j \, d\mathbf{x}, \quad i, j = 1, \dots, N_h, \tag{2.27}$$

- $\mathbf{A}$ is the stiffness matrix given by

$$\mathbf{A} = \mathbf{V} - \mathbf{I}^\intercal - \mathbf{I} + \mathbf{S}\,, \tag{2.28}$$

  where

$$\mathbf{V}_{i,j} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \, d\mathbf{x}, \tag{2.29}$$

  and

$$\mathbf{I}_{i,j} = \int_{\mathscr{F}_h} [\![\phi_j]\!] \cdot \{\nabla_h \phi_i\} \, ds \quad \mathbf{S}_{i,j} = \int_{\mathscr{F}_h} \sigma [\![\phi_j]\!] \cdot [\![\phi_i]\!] \, ds, \tag{2.30}$$

  for any $i, j = 1, \dots, N_h$.

In Section 2.3.1 we introduced the polynomial basis over a general polytopic element E, $\{\phi_{i,E}|_E\}, i = 1, \dots, N_{p_E}$. For the sake of notation we omit the symbol specifying the support, meaning we just write $\phi_{i,E}$. We consider the case in which $p_E = p$ for all $E \in \mathscr{T}_h$ and $\Omega \subset \mathbb{R}^3$. We call N_p the number of degrees of freedom for any $E \in \mathscr{T}_h$. Therefore, if we choose $\mathbb{P}_p(B_E)$ as discrete space we have that $N_p = \dim(\mathbb{P}_p(B_E)) = (p+1)(p+2)(p+3)/6$. We call N_{poly} the number of polyhedra in the mesh $\mathscr{T}_h$. Moreover, we choose the penalization function according to (2.12) where h_E is the diameter of the bounding box B_E of the element E. With this notation we describe how we compute the entries in the local mass and element-based stiffness matrices. The local mass matrix is defined as the mass matrix restricted to $E \in \mathscr{T}_h$, i.e.,

$$\mathbf{M}^E_{i,j} = \int_E \phi_{i,E} \, \phi_{j,E} \, d\mathbf{x} \quad i, j = 1, \dots, N_p. \tag{2.31}$$

If we use the quadrature formulas that we introduced in (2.23) to approximate this inte-

gral, we obtain the following:

$$\mathbf{M}_{i,j}^{E} = \sum_{\tau_k \in E_{\mathscr{I}}} \int_{\tau_k} \phi_{i,E}\, \phi_{j,E}\, d\mathbf{x} \approx \sum_{\tau_k \in E_{\mathscr{I}}} \sum_{q=1}^{n_{q3}} \phi_{i,E}(F_k(\boldsymbol{\xi}_q))\, \phi_{j,E}(F_k(\boldsymbol{\xi}_q))\, det(J_{F_k}(\boldsymbol{\xi}_q))\, w_q, \tag{2.32}$$

where $E_{\mathscr{I}}$ is the tetrahedral sub-tessellation of $E \in \mathscr{T}_h$, $F_k : E_R \to \tau_k$ is the mapping from the reference tetrahedron E_R to the physical tetrahedron τ_k, with Jacobi matrix J_{F_k}, and $(\boldsymbol{\xi}_q, w_q)_{q=1}^{n_{q3}}$ denotes the quadrature rule defined on E_R.

The local element-based stiffness matrix is the component $\mathbf{V}_{i,j}$ of the stiffness matrix $\mathbf{A}_{i,j}$ restricted to $E \in \mathscr{T}_h$, i.e.,

$$\mathbf{V}_{i,j}^{E} = \int_E \nabla\phi_{j,E} \cdot \nabla\phi_{i,E}\, d\mathbf{x}. \tag{2.33}$$

In the three-dimensional case if we employ tensor-product Legendre polynomials we have that

$$\phi_{i,E}(\mathbf{x}) = L_{i_1}(x)L_{i_2}(y)L_{i_3}(z) \quad i_1 + i_2 + i_3 \leq p, \tag{2.34}$$

where L_i is obtained from $\hat{L}_i$ through the transformation G_E in (2.22).

If we compute the gradient of $\phi_{i,E}$ we obtain:

$$\nabla\phi_{i,E}(\mathbf{x}) = \begin{pmatrix} \frac{\mathrm{d}}{\mathrm{d}x} L_{i_1}(x)L_{i_2}(y)L_{i_3}(z) \\ L_{i_1}(x)\frac{\mathrm{d}}{\mathrm{d}y} L_{i_2}(y)L_{i_3}(z) \\ L_{i_1}(x)L_{i_2}(y)\frac{\mathrm{d}}{\mathrm{d}z} L_{i_3}(z) \end{pmatrix}. \tag{2.35}$$

The term $\mathbf{V}_{i,j}^{E}$ becomes:

$$\begin{aligned} \mathbf{V}_{i,j}^{E} = &\int_E \frac{\mathrm{d}}{\mathrm{d}x} L_{i_1}(x)L_{i_2}(y)L_{i_3}(z) \frac{\mathrm{d}}{\mathrm{d}x} L_{j_1}(x)L_{j_2}(y)L_{j_3}(z)\, d\mathbf{x} \\ &+ \int_E L_{i_1}(x)\frac{\mathrm{d}}{\mathrm{d}y} L_{i_2}(y)L_{i_3}(z)\, L_{j_1}(x)\frac{\mathrm{d}}{\mathrm{d}y} L_{j_2}(y)L_{j_3}(z)\, d\mathbf{x} \\ &+ \int_E L_{i_1}(x)L_{i_2}(y)\frac{\mathrm{d}}{\mathrm{d}z} L_{i_3}(z)\, L_{j_1}(x)L_{j_2}(y)\frac{\mathrm{d}}{\mathrm{d}z} L_{j_3}(z)\, d\mathbf{x}. \end{aligned}$$

Then, the approximation of this integral is performed by using the quadrature formulas in (2.23). Following the same steps as we did above for the local mass matrix, we obtain:

$$\begin{aligned}\mathbf{V}_{i,j}^E &= \sum_{\tau_k \in E_{\mathscr{F}}} \int_{\tau_k} \nabla\phi_{i,E} \cdot \nabla\phi_{j,E}\, d\mathbf{x} \\ &\approx \sum_{\tau_k \in E_{\mathscr{F}}} \sum_{q=1}^{n_{q_3}} \nabla\phi_{i,E}(F_k(\boldsymbol{\xi}_q)) \cdot \nabla\phi_{j,E}(F_k(\boldsymbol{\xi}_q))\, det(J_{F_k}(\boldsymbol{\xi}_q))\, w_q.\end{aligned} \tag{2.36}$$

Now we consider the terms on the interfaces. From the definition of the jump and average operators, cf. (1.2), recalling that each face $F \in \mathscr{F}_h^I$ is shared by the elements $E^{\pm}$, we expand the terms $\mathbf{S}_{i,j}$ and $\mathbf{I}_{i,j}$ as the sum of the integrals on the interior faces and on the boundary faces, i.e.,

$$\mathbf{S}_{i,j} = \mathbf{S}_{i,j}^I + \mathbf{S}_{i,j}^B, \qquad \mathbf{I}_{i,j} = \mathbf{I}_{i,j}^I + \mathbf{I}_{i,j}^B,$$

where

$$\begin{aligned}\mathbf{S}_{i,j}^I &= \sum_{F \in \mathscr{F}^I} \int_F \sigma[\![\phi_{j,E}]\!] \cdot [\![\phi_{i,E}]\!]\, ds = \sum_{F \in \mathscr{F}^I} \int_F \sigma(\phi_{j,E}^+ \mathbf{n}^+ + \phi_{j,E}^- \mathbf{n}^-) \cdot (\phi_{i,E}^+ \mathbf{n}^+ + \phi_{i,E}^- \mathbf{n}^-)\, ds \\ &= \sum_{F \in \mathscr{F}^I} \int_F \sigma\phi_{i,E}^+(\phi_{j,E}^+ - \phi_{j,E}^-)\, ds + \sum_{F \in \mathscr{F}^I} \int_F \sigma\phi_{i,E}^-(\phi_{j,E}^- - \phi_{j,E}^+)\, ds,\end{aligned}$$

$$\begin{aligned}\mathbf{I}_{i,j}^I &= \sum_{F \in \mathscr{F}_h^I} \int_F [\![\phi_{j,E}]\!] \cdot \{\nabla_h\phi_{i,E}\}\, ds = \sum_{F \in \mathscr{F}_h^I} \int_F \frac{1}{2}(\nabla_h\phi_{i,E}^+ + \nabla_h\phi_{i,E}^-) \cdot (\phi_{j,E}^+ \mathbf{n}^+ + \phi_{j,E}^- \mathbf{n}^-)\, ds \\ &= \sum_{F \in \mathscr{F}_h^I} \frac{1}{2} \int_F \nabla_h\phi_{i,E}^+ \cdot \mathbf{n}^+(\phi_{j,E}^+ - \phi_{j,E}^-)\, ds + \sum_{F \in \mathscr{F}_h^I} \frac{1}{2} \int_F \nabla_h\phi_{i,E}^- \cdot \mathbf{n}^-(\phi_{j,E}^- - \phi_{j,E}^+)\, ds,\end{aligned}$$

while on the boundary we have

$$\mathbf{S}_{i,j}^B = \sum_{F \in \mathscr{F}^B} \int_F \sigma[\![\phi_{j,E}]\!] \cdot [\![\phi_{i,E}]\!]\, ds = \sum_{F \in \mathscr{F}^B} \int_F \sigma\phi_{i,E}^+\phi_{j,E}^+\, ds,$$

and

$$\mathbf{I}_{i,j}^B = \sum_{F \in \mathscr{F}_h^B} \int_F \nabla_h\phi_{i,E}^+ \cdot \mathbf{n}^+\phi_{j,E}^+\, ds.$$

Now we define the local interface integrals $\mathbf{S}_{i,j}^F$ and $\mathbf{I}_{i,j}^F$ in the following way:

$$\mathbf{S}_{i,j}^F = \int_F \sigma[\![\phi_{j,E}]\!] \cdot [\![\phi_{i,E}]\!]\, ds, \quad \mathbf{I}_{i,j}^F = \int_F [\![\phi_{j,E}]\!] \cdot \{\nabla_h\phi_{i,E}\}\, ds. \tag{2.37}$$

We write the expansion of the term $\mathbf{S}_{i,j}^{F,I}$ on the interior faces:

$$\mathbf{S}_{i,j}^{F,I} = \int_F \sigma \phi_{i,E}^+ (\phi_{j,E}^+ - \phi_{j,E}^-)\, ds + \int_F \sigma \phi_{i,E}^- (\phi_{j,E}^- - \phi_{j,E}^+)\, ds = (\mathrm{I}) + (\mathrm{II}).$$

We see how to compute the integral (I) with the two-dimensional quadrature formulas. First, we divide the integral in two terms:

$$(\mathrm{I}) = \int_F \sigma \phi_{i,E}^+ (\phi_{j,E}^+ - \phi_{j,E}^-)\, ds = \mathbf{S}_{i,j}^{F,D} + \mathbf{S}_{i,j}^{F,N},$$

where,

$$\mathbf{S}_{i,j}^{F,D} = \int_F \sigma \phi_{i,E}^+ \phi_{j,E}^+\, ds \qquad \mathbf{S}_{i,j}^{F,N} = -\int_F \sigma \phi_{i,E}^+ \phi_{j,E}^-\, ds.$$

Now we consider the tetrahedral sub-tessellation $E_{\mathscr{I}} = \{\tau_k\}$ of the element E such that $F \subset \partial E$. We call f_l, $l = 1, \dots, 4$ the triangular faces of the tetrahedron τ_k. We define T_F as the set of all tetrahedra that belongs to the polyhedron E such that at least one of their faces f_l belongs to F:

$$T_F = \{\tau_k \in E_{\mathscr{I}} : \exists\, l \in 1, \dots, 4 : f_l \subset \partial \tau_k \cap F\}.$$

We define the maps $\psi_l : T_R \to E_R$, $l = 1, \dots, 4$, from the two-dimensional reference triangle T_R to the faces of the three-dimensional reference tetrahedron E_R. The approximation of the integral $\mathbf{S}_{i,j}^{F,D}$ is computed as:

$$\begin{aligned} \mathbf{S}_{i,j}^{F,D} &= \sum_{\tau_k \in T_F} \int_{f_l} \sigma \phi_{i,E}^+ \phi_{j,E}^+\, ds \\ &\approx \sum_{\tau_k \in T_F} \sum_{q=1}^{n_{q_2}} \sigma \phi_{i,E}^+ (F_k(\psi_l(\boldsymbol{\eta}_q)))\, \phi_{j,E}^+ (F_k(\psi_l(\boldsymbol{\eta}_q)))\, det(J_{F_k}(\psi_l(\boldsymbol{\eta}_q)))\, \hat{w}_q, \end{aligned} \tag{2.38}$$

where $F_k : E_R \to \tau_k$ is the mapping from the reference tetrahedron E_R to the physical tetrahedron τ_k, with Jacobi matrix J_{F_k}, and $(\boldsymbol{\eta}_q, \hat{w}_q)_{q=1}^{n_{q_2}}$ denotes the two-dimensional quadrature rule defined on T_R.

The same holds for the local term $\mathbf{S}_{i,j}^{F,B}$ on the boundary faces. In a similar way, the

approximation of the integral $\mathbf{S}_{i,j}^{F,N}$ is computed as:

$$\begin{aligned}\mathbf{S}_{i,j}^{F,N} &= -\sum_{\tau_k \in T_F} \int_{f_l} \sigma \phi_{i,E}^{+} \phi_{j,E}^{-}\, ds \\ &\approx -\sum_{\tau_k \in T_F} \sum_{q=1}^{n_{q_2}} \sigma \phi_{i,E}^{+}(F_k(\psi_l(\boldsymbol{\eta}_q)))\, \phi_{j,E}^{-}(F_k(\psi_l(\boldsymbol{\eta}_q)))\, det(J_{F_k}(\psi_l(\boldsymbol{\eta}_q)))\, \hat{w}_q,\end{aligned} \tag{2.39}$$

Notice that we considered the integral (I) in the sum because we are going to implement only this term. Indeed, suppose that E^+ and E^- are the two elements that share the face F. When we perform the loop on the elements to compute the integrals, we will assemble only term (I) when the current element is E^+. In this way the term (II) will be computed when the current element will be E^-. This allows us to consider the face F only once in the loop on the elements. The same reasoning holds for the term $\mathbf{I}_{i,j}^{F}$. We are going to implement only the first term of the sum written in the expansion written below. We write the expansion of the term $\mathbf{I}_{i,j}^{F,I}$ on the interior faces:

$$\mathbf{I}_{i,j}^{I} = \frac{1}{2}\int_F \nabla_h \phi_{i,E}^{+} \cdot \mathbf{n}^{+}(\phi_{j,E}^{+} - \phi_{j,E}^{-})\, ds + \frac{1}{2}\int_F \nabla_h \phi_{i,E}^{-} \cdot \mathbf{n}^{-}(\phi_{j,E}^{-} - \phi_{j,E}^{+})\, ds = \text{(I)} + \text{(II)},$$

and on the boundary faces:

$$\mathbf{I}_{i,j}^{B} = \int_F \nabla_h \phi_{i,E}^{+} \cdot \mathbf{n}^{+} \phi_{j,E}^{+}\, ds.$$

The computation of $\mathbf{I}_{i,j}^{F}$ follows the same steps of $\mathbf{S}_{i,j}^{F}$. The components of the local right hand side vector $\mathbf{f}^E$ defined as

$$\mathbf{f}_i^E = \int_E f \phi_{i,E}\, d\mathbf{x}$$

are computed with the quadrature formulas in the following way:

$$\mathbf{f}_i^E = \sum_{\tau_k \in E_{\mathscr{I}}} \int_{\tau_k} f \phi_{i,E}\, d\mathbf{x} \approx \sum_{\tau_k \in E_{\mathscr{I}}} \sum_{q=1}^{n_{q_3}} f(F_k(\boldsymbol{\xi}_q))\, \phi_{i,E}(F_k(\boldsymbol{\xi}_q))\, det(J_{F_k}(\boldsymbol{\xi}_q))\, w_q. \tag{2.40}$$

In the case of $g_D \neq 0$, Dirichlet boundary conditions can be enforced by penalization, i.e.,

$$\mathbf{f}_i = \sum_{E \in \mathscr{T}_h} \int_E f \phi_{i,E}\, d\mathbf{x} - \sum_{F \in \mathscr{F}_h} \int_F g_D \nabla_h \phi_{i,E}^{+} \cdot \mathbf{n}\, ds + \sum_{F \in \mathscr{F}_h} \int_F \sigma g_D \phi_{i,E}^{+}\, ds. \tag{2.41}$$

3 | Description of the LYMPH3D library

In this chapter we present LYMPH3D, the library that has been developed in this thesis.

LYMPH3D is a library written in Fortran. In LYMPH3D we can find the implementation of PolyDG method to discretize a PDE problem on a computational mesh made of polytopic elements. In this way this library it is perfectly suited to deal with problems featuring complicated geometries.

The code uses the open-source libraries, METIS for mesh agglomeration, MPI for message passing, and PETSc, for solving the linear system.

In this chapter we describe in detail the structure of the library, all the files and modules and what they are used for. Figure 3.1 shows the basic structure of the main file **Lymph3D**.

In Section 3.1 we describe the part of the library related to the mesh. Considering that we already have a file .mesh containing all the information about the mesh, we will see how we read the mesh and store the information needed to solve the equations.

In Section 3.2 we describe the part of the library related to the computation of the necessary tools to assemble the linear system, meaning the construction of the bounding box, the basis functions and the quadrature formulas.

In Section 3.3 we describe how we assemble the matrices and the term on the right and side and in Section 3.4 we will see how we actually solve the linear system.

In Section 3.5 we can find the description of the functions related to the post-processing part, the creation of the .vtk files necessary to visualize the solution and the computation of the errors in L^2-norm and in DG-norm.

In Section 3.6, we provide a brief user guide, and in Section 3.7 we will see how to generate the mesh in a simple case of a cube.

A selection of some of the most important functions that are used to perform the described steps is reported in Appendix A.

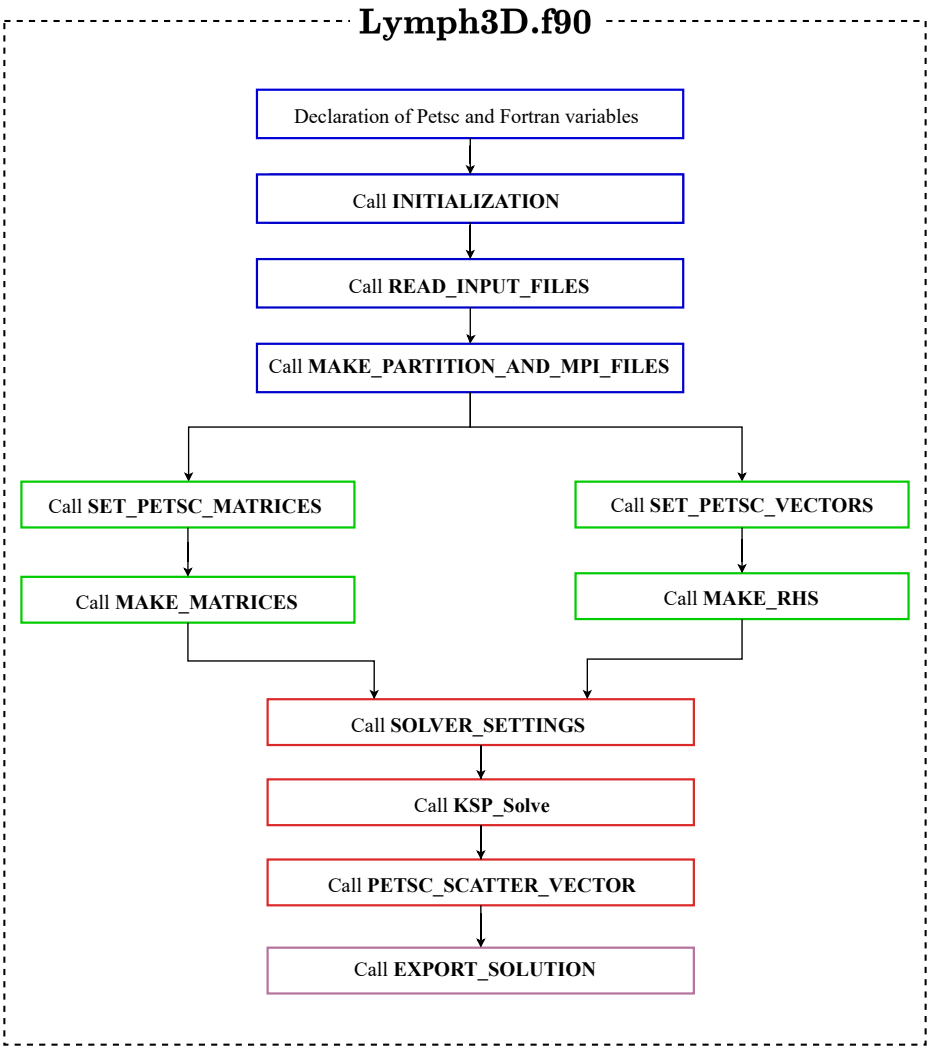


Figure 3.1: Basic structure of the main program **Lymph3D**. First, we have the declaration of the PETSc environment and the PETSc and Fortran variables. Next, we call the subroutine READ_INPUT_FILES to read the input file and the subroutine MAKE_PARTITION_AND_MPI_FILES to make the partition and to store the mesh information, then we define and assemble the PETSc matrices and vectors by calling SET_PETSC_MATRICES, MAKE_MATRICES, SET_PETSC_VECTORS and MAKE_RHS. We set the algebraic solver and finally solve the linear system by calling the PETSc function KSPSolve and then export the solution.

3.1. Reading input files and store mesh structure

The starting point of a finite element discretization consists in reading the mesh file and store the key information about the geometry of the problem. We provide here a schematic description of the modules and files related to these tasks that are present in the library. The main feature is given by the introduction of the structure **Polyhedron** storing the key properties of the polyhedral mesh. The significant changes can be seen in the Poly_mesh.f90 module, in the subroutine MAKE_PARTITION_AND_MPI_FILES.f90, and the implementation of the module Poly_geom.f90.

- Poly_setup_MPI.f90 where we can find the definition of the MPI variables as `mpi_id`, `mpi_np`, `mpi_ierr` and the subroutine INITIALIZATION.f90. This subroutine performs the initialization of the MPI and the Petsc MPI envinronments.
- The file Poly_mesh.f90 contains the definition of the **Mesh_Structure**, the **Element** structure and the **Polyhedron** structure.
    1. The struct **Element** is reported in Table 3.1.

Name field	type	Description
`el_type`	string	Type of the element (tetrahedra in our case)
`mat_prop`	integer	Id for heterogeneous materials
`num_vert`	integer	Number of vertices of the element (4 in our case)
`num_faces`	integer	Number of faces of the element (4 in our case)
`Degree`	integer	Local degree of the basis function for this element
`NDof_loc`	integer	Local number of degrees of fredom
`vert`	vector of integers of size(`num_vert`)	Indexes of the vertices of the element
`faces`	matrix of integers of size (`num_faces`,3)	Indexes of the vertices for every face of the element
`neigh_el`	matrix of integers of size (`num_faces`,4)	Properties of the neighbor elements
`normal`	matrix of real of size (`num_faces`,3)	Coordinates of the normal to each face
`area`	vector of real of size (`num_faces`)	Area of each face

Table 3.1: Fields of the structure Element

For example if we consider a tetrahedral mesh composed of only two tetrahedra, and we take the first tetrahedron we will have as Element structure fields the followings

- `el_type` $\rightarrow$ TETRA
- `mat_prop` $\rightarrow 1$
- `num_vert` $\rightarrow 4$
- `num_faces` $\rightarrow 4$
- `vert` $\rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$

- `faces` $\rightarrow \begin{bmatrix} 3 & 2 & 1 \\ 1 & 2 & 4 \\ 2 & 3 & 4 \\ 3 & 1 & 4 \end{bmatrix}$
- `normal` $\rightarrow \begin{bmatrix} 0 & -0.71 & 0.71 \\ -0.71 & 0.71 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$
- `area` $\rightarrow \begin{bmatrix} 0.35 \\ 0.35 \\ 0.25 \\ 0.25 \end{bmatrix}$
- `neigh_el` $\rightarrow$ The tetrahedra shares a face with just one neighbour within the same process and in this case, made by the same material; the other faces are boundary faces.

mpi-proc	mat_id	el_id	face_id
0	1	2	4
0	0	0	0
0	0	0	0
0	0	0	0

2. The struct **Polyhedron** contains the properties of a single polyhedron. In Table 3.2 we can see the description of its fields.

Name field	type	Description
num_tet_in_poly	integer	Number of tetrahedra contained in the polyhedron
tet_in_poly	vector of integers of size (num_tet_in_poly,1)	Global indexes of the tetrahedra contained in the polyhedron
b_box	matrix of real numbers of size (3,2)	Coordinates in the three different directions of the bounding box of the polyhedron
hk	real number	Diameter of the bounding box of the polyhedron
neigh_bbox	matrix of real numbers size (num_tet_in_poly,3,2)	Coordinates of the bounding box of a neighbouring polyhedron
neigh_hk	vector of real numbers of size (num_tet_in_poly)	Diameter of the bounding box of a neighboring polyhedron

Table 3.2: Fields of the structure Polyhedron

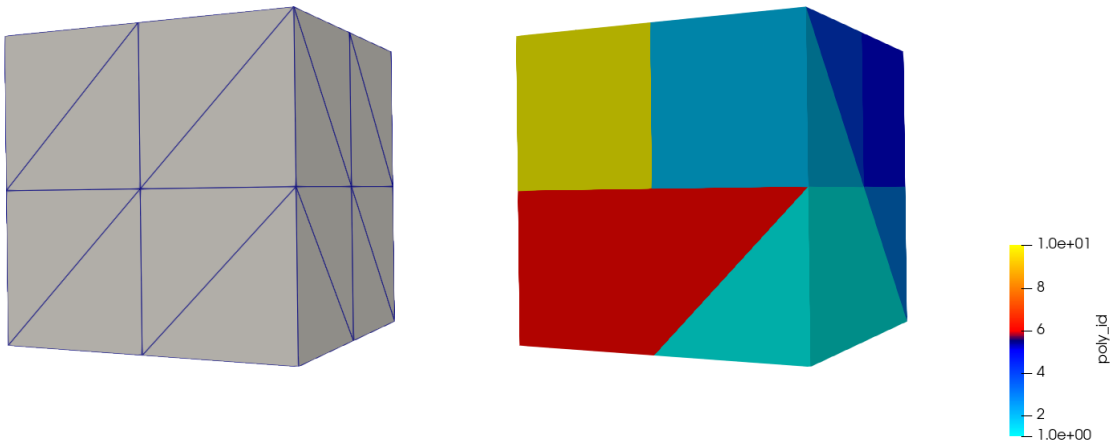
3. **Mesh_Structure** is a struct that contain the key information on the standard-shape elements of the mesh (tethraedra in our case), on the boundary faces (triangles) as the number of these kind of elements and their connettivity (ex. con_tet that contains the connettivity of a tethraedra). New additional fields were added to this structure to store the properties of the polyhedra. In the Table 3.3 below we report the fields of the **Mesh_Structure**, where we have highlighted in colour the new additional fields for the polyhedra.

Name field	type	Description
num_node	integer	Total number of vertices
num_tet	integer	Total number of tetrahedra
num_elem	integer	Total number of elements
num_poly	integer	Total number of polyhedra
con_tet	matrix of integers of size (num_tet,5)	Connettivity matrix of the tetrahedra
con_tria	matrix of integers of size (num_tria,4)	Connettivity matrix of the triangles
part_elem	vector of integers of size (num_tet)	For each tetrahedron it stores the id of the processor it belongs to after the partition
elem_in_poly	vector of integers of size (num_tet)	For each tetrahedron it stores the index of polyhedron it belongs to
elem_in_poly_loc	vector of integers of size (num_elem_loc)	For each local tetrahedron it stores the global index of polyhedron it belongs to
coord_x, coord_y, coord_z	vectors of size (num_nodes)	Coordinates of the vertices in direction x,y and z
num_elem_loc	integer	Local number of elements
num_node_loc	integer	Local number of vertices
num_poly_loc	integer	Local number of polyhedra
elem_loc2glo	vector of integers of size (num_elem_loc)	Local to global maps to go from the local enumeration to the global one for the elements
node_loc2glo	vector of integers of size (num_node_loc)	Local to global maps to go from the local enumeration to the global one for the nodes
poly_loc2glo	vector of integers of size (num_poly_loc)	Local to global maps to go from the local enumeration to the global one for the polyhedra
Elem_loc	vector of structures of type Element of length (num_elem_loc)	For each element it stores all the properties listed in Table 3.1
Poly	vector of structures of type Polyhedron of length (num_poly)	For each polyhedron it stores the properties listed in Table 3.2

Table 3.3: Fields of the structure Mesh_Structure

In Figure 3.2 we can see an example of a polyhedral mesh of a cube made of a number of polyhedra `num_poly`=10 (Figure 3.2b), and the respective tetrahedral subtassellation of the polyehdra with `num_tet`=48 (Figure 3.2a). In the Table 3.4 we report the first ten elements of the vector `elem_in_poly` for this particular mesh. The module Poly_mesh.f90 also contains the subroutines

1. allocate_Mesh_Structure to allocate the Mesh Structure
2. print_Dime_Mesh_Structure to print the Mesh Structure
3. print_Local_Mesh_Structure_VTK to create the file mesh_partition.vtk to visualize the mesh partitioned into different processors. In Figure 3.3 there is an example of output of this subroutine, where we have a tetrahedral mesh with 48 tetrahedra and the partition is done within 3 processors.



(a) Example of tetrahedral mesh with $N_{tet} = 48$.

(b) Example of agglomerated polyhedral mesh with $N_{poly} = 10$.

Figure 3.2: Polyhedral mesh (right) and relative tetrahedral sub-tessellation (left). The polyhedral mesh in Figure 3.2b is obtained via agglomeration of the tetrahedral mesh in Figure 3.2a.

Tetrahedra id	Polyhedra id
1	7
2	6
3	6
4	3
5	3
6	7
7	7
8	6
9	6
10	8

Table 3.4: The table refers to the polyhedral mesh in Figure 3.2b with $N_{poly} = 10$, obtained via agglomeration of the tetrahedral mesh with $N_{tet} = 48$ in 3.2a. The table shows how the agglomeration works. To each tetrahedron we associate an index from 1 to $N_{poly} = 10$ that refers to the polyhedron that contains that tetrahedron. The first coloumn contains the indices of the tetrahedra, while the second coloumn contains the indices of the associated polyhedron. We report here only the first 10 tetrahedra. The second coloumn represents the first 10 elements of the vector `elem_in_poly` for the mesh in Fig 3.2b.

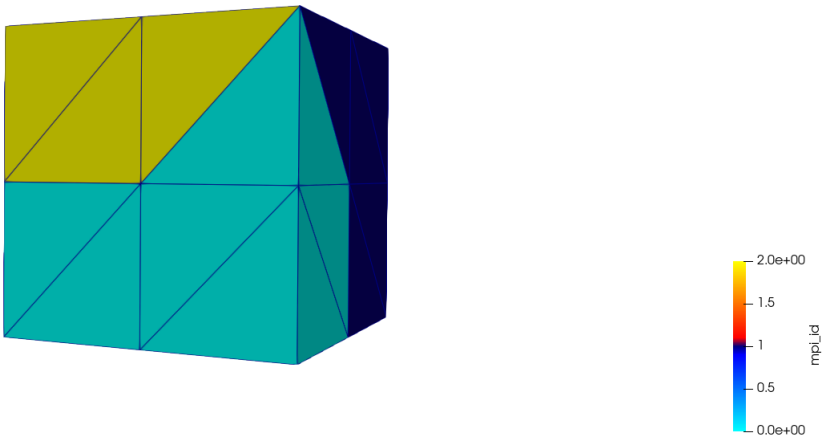


Figure 3.3: Example of a tetrahedral mesh partitioned into $n_p = 3$ processors

- The file Poly_global.f90 contains a module of the same name, which stores some

useful variables as `head_file` the name of the input file stored in the variable, `mate_file` the name of the file containing information on the materials and some variables related to the output and the measuring of the computational time. In this file we can also find other modules, in particular three of them `Poly_exit_codes`, `Poly_fail_codes`, `Poly_default_codes` are related to error and fail codes if something goes wrong reading the .input and the .mate file, and additional useful modules, the `qsort` algorithm for sorting elements of an array, the `local_search` module useful for parallel programs and the `find_poly` module. The `local_search` contains the subroutine GET_EL_LOC_FROM_EL_GLO that we need in order to find the local index of an element of the mesh. It takes as input the local to global map relative to a particular processor and the global index of the element and returns the local index of the element if the element is stored in that processor, zero otherwise. The new module `find_poly` contains the function FIND_TET_IN_POLY, needed to find all the tetrahedra contained in a single polyhedra. The function takes as input the vector `elem_in_poly`, the index of the polyhedron we are considering and it returns a vector that has the dimension of the number of tetrahedra contained in each polyhedron and it stores the indeces of these tetrahedra.

- The file Poly_readfile.f90 contains the subroutines that actually go through all the input files, reading the files line by line.
- Poly_geom.f90 contains the module `Poly_geom` where we can find the subroutine REFERENCE_MAP that perform the computation of the matrices of the reference map `Fk` from the reference simplex `(0,0,0)`, `(1,0,0)`,`(0,1,0)` `(0,0,1)` to to the physical tetrahedron, and the corresponding determinant `Jdet`, see [44]. This subroutine takes as input the coordinates of the vertices of the physical tetrahedron `x`,`y`,`z` and returns as output the followings.
  - `Fk` a matrix, size (3,4) containing the mappings from the reference tetrahedron to the physical tetrahedron
  - `Jinv` a matrix, size (3,3) containing the inverse of the jacobian of `Fk`
  - `Jdet` a scalar, determinant of the jacobian
- The READ_INPUT_FILES subroutine is called directly from the main file **Lymph3D** and it calls all the modules we saw above in order to set the header and the material files and to define and initialize the structure **Mesh_Structure**.
- The file MAKE_PARTITION_AND_MPI_FILES.F90 contains a subroutine with the same name that performs the partition of the mesh into the different proces-

sors and stores the local properties of the mesh into the correspondent fields of the **Mesh_Structure**. Moreover this subroutine creates output files in the folder **FILES_MPI** and writes the mesh infomation in the suitable output files. Each file contains the mesh information relative to a specific processor. For example if we use two processor, after calling this subroutine we will see in the folder **FILES_MPI** the files

1. mesh_000000.mpi, mesh_000001.mpi
   containing the mesh properties for the elements contained in processor 0 and processor 1 respectively
2. elem4proc.mpi
   containing two coloumns, the first coloumn is for the indexes of the tetrahedra and the second one for the id of the processor it belongs to
3. con_tet_000000.mpi,con_tet_000001.mpi
   containing the connettivity of the tetrahedra respectively for processor 0 and processor 1
4. con_tri_000000.mpi,con_tri_000001.mpi
   containing the connettivity of the triangles respectively for processor 0 and processor 1.

The subroutine MAKE_PARTITION_AND_MPI_FILES.f90 calls the following subroutines.

- MESH_PARTITIONING
  performs the partition of the mesh into different processors by using METIS and write the mpi file elem4proc.mpi.
- WRITE_PARTITION
  writes the partition in the mpi files storing the connettivity of tetrahedra and triangles.
- MESH_AGGLOMERATION
  generates the polyhedral mesh by agglomerating the tetrahedral mesh read from the .mesh file.
- MESH_CORRECTION
  ensures that each polyhedron contains at least one tetrahedron.
- CREATE_GLOBAL_POLY_MAP
  allocates and stores the field `elem_in_poly`.

- CREATE_LOCAL_MESH
  stores the local properties of the tetrahedral mesh into the correspondent fields of the **Mesh_Structure**.
- CREATE_VERT_LIST
  stores the coordinates of the vertices of the tetrahedra locally.
- CREATE_POLY_LIST
  stores the map `poly_loc2glo` to go from local polyhedron to the global one and initializes the other fields of the structure **Polyhedron**.
- CREATE_NORMAL_FACE
  performs the computation of the coordinates of the normal and the area to each face.
- CREATE_BBOX_EL
  performs the computation of the bounding box for each polyhedron. The key steps of this subroutine can be seen in Algorithm 3.2.
- CREATE_NEIGH_EL_TRIA
  stores the information of the neighbouring tetrahedra and polyhedra. Few changes were made to this subroutine to add the storage of the properties of the neighbouring polyhedra. These changes can be seen in Algorithm 3.3.
- WRITE_MESH_INFO writes the mesh information in suitable mpi files.

We focus on the subroutine MESH_AGGLOMERATION to see how we generate the polyhedral mesh by agglomerating the tetrahedral mesh. The agglomeration is performed based on employing the METIS library for graph partitioning, cf., for example, [42, 43]. METIS is a state-of-the-art graph partitioning algorithm. We note that for METIS to partition the fine mesh $\mathscr{T}_h$, the logical structure of $\mathscr{T}_h$ is first stored in the form of a graph, where each node represents an element domain of $\mathscr{T}_h$, and each link between two nodes represents a face shared by the two elements represented by the graph nodes. The resulting partition of $\mathscr{T}_h$ constructed by METIS is produced with the objective of minimizing the number of neighbours among each of the resulting partitions, or more precisely, the resulting polygonal elements. In Algorithm 3.1 we can find the lines of pseudocode needed do create the agglomerate mesh of polyhedra.

Algorithm 3.1 Generate an agglomerated mesh

1: Store number of polyhedra in variable `num_part`
2: Extract the connettivity matrix `con_tet` and the number of vertices of the tetrahedra `num_nodes`
3: Set options to have Fortran-style numbering and to force contigous partition
4: Call METIS_PartMeshDual function giving as input ncommon=4, `num_part`, `con_tet`, `num_elem_loc` and `num_nodes`
5: Store the output of METIS into the file 'elem_in_poly_loc.mpi'
6: Allocate and store `elem_in_poly_loc`

Note that if we want to solve the problem with the tetrahedral mesh we just have to set `num_parts`=`num_elem_loc`.

Notice that the mesh that we obtain as output of this agglomeration algorithm can present some issues. For instance, since we provide to the algorithm the number of polyhedra that we want `num_part`, we can have the problem of some empty polyhedra, meaning polyhedra that don't contain any tetrahedron. The role of the subroutine MESH_CORRECTION is to correct this problem if it presents. In this subroutine we read the agglomerated mesh and reduce the number of polyhedra until we have that each polyhedra contains at least one tetrahedron.

In Algorithm 3.2 there is a summary of the subroutine CREATE_BBOX_EL where we describe how we create the bounding box B_E for each $E \in \mathscr{T}_h$. Then, in Algorithm 3.3 we can see a schematic description of the few changes made to the subroutine CREATE_NEIGH_EL_TRIA. Indeed, as we will see later in Section 3.3, in order to assemble the stiffness matrix we need to access to the bounding box of the polyhedron we are considering, but also the bounding box of the neighboring polyhedron. When two neighbouring polyhedra belong to the same processor we can easily retrieve the bounding box of the neighbour. However, when the two neighboring polyhedra have neighboring tetrahedra in different processors, we need to save the coordinates of the bounding box of the neighboring polyhedra in a structure that is accessible by all processors.

Algorithm 3.2 Create Bounding Box

for $i := 1$, `num_poly_loc` **do**
  Allocate xx, yy, zz of size `num_tet_in_poly` * `num_vert`
  $t \leftarrow 1$
  for $ie = 1$, `num_tet_in_poly` **do**
    $xx(t) \leftarrow$ `coord_x`
    $yy(t) \leftarrow$ `coord_y`
    $zz(t) \leftarrow$ `coord_z`
    $t \leftarrow t + 1$
  end for
  `b_box(1,1)` $\leftarrow \min(xx)$; `b_box(1,2)` $\leftarrow \max(xx)$;
  `b_box(2,1)` $\leftarrow \min(yy)$; `b_box(2,2)` $\leftarrow \max(yy)$;
  `b_box(3,1)` $\leftarrow \min(zz)$; `b_box(3,2)` $\leftarrow \max(zz)$;
  `hk` $\leftarrow$ diameter of `b_box`
end for

Algorithm 3.3 Create Neighboring Bounding Box

...
Allocate containing the coordinates of the bounding boxes to send `x1_send_mpi`, `x2_send_mpi`, `y1_send_mpi`, `y2_send_mpi`, `z1_send_mpi`, `z2_send_mpi`
Allocate diameters to send `hk_send_mpi`
Send coordinartes of bounding boxes and diameters to all processors
for $i = 1$,`num_tria_loc` **do**
  ...
  if (Neighbouring tetrahedra sharing face i are in different processors) **then**
    ...
    if (The tetrahedra don't belong to the same polyhedron) **then**
      Recover the global index of the face `iface`
      `neigh_bbox(iface,1,1:2)` $\leftarrow$ [`x1_send_mpi` ; `x2_send_mpi`]
      `neigh_bbox(iface,2,1:2)` $\leftarrow$ [`y1_send_mpi` ; `y2_send_mpi`]
      `neigh_bbox(iface,3,1:2)` $\leftarrow$ [`z1_send_mpi` ; `z2_send_mpi`]
      `neigh_hk` $\leftarrow$ `hk_send_mpi`
    end if
    ...
  end if
end for
Deallocate coordinates and diameters sent

3.2. Basis functions and quadrature formulas

In this section we detail the module `basis_function` contained in basis_function.f90. Note that we call p the total degree of the basis functions and N_p the number of basis functions, denoted before as `Degree` and `NDof_loc`, respectively, see Table 3.1. The module `basis_function` contains the following subroutines:

- `blist`
  This subroutine returns the list of the degrees of monomials of the N_p basis functions up to a total degree p. The subroutine takes as input N_p and p and returns a matrix `blist` of size (`Np`,3).

- `quadrature`
  This subroutine computes quadrature nodes and weights over the simplex tetrahedron $(\boldsymbol{\xi}_q, w_q)_{q=1}^{n_{q_3}}$ and triangle $(\boldsymbol{\eta}_q, w_q)_{q=1}^{n_{q_2}}$. It computes also the maps ϕ_l, $l = 1, \ldots, 4$, from the two-dimensional reference triangle to the faces of the three-dimensional reference tetrahedron. These maps are stored in the matrix `node_maps`. We remark that we are using the quadrature formulas for the tetrahedra since we are considering the Sub-Tessellation method explained in Section 2.3.2. Below we report the old and the new notation for the quadrature nodes, both in three-dimensions and two-dimensions.
  - Number of 3D quadrature nodes: $n_{q_3} \rightarrow$ `nq3`
  - Number of 2D quadrature nodes: $n_{q_2} \rightarrow$ `nq2`
  - Weights of the 3D quadrature nodes: $w \rightarrow$ `wei3`
  - Weights of the 2D quadrature nodes: $\hat{w} \rightarrow$ `wei2`
  - Nodes of the 3D quadrature nodes: $\boldsymbol{\xi} \rightarrow$ `nod3`
  - Nodes of the 2D quadrature nodes: $\boldsymbol{\eta} \rightarrow$ `nod2`

- `LegendreP`
  Here we evaluate the scaled Legendre Polynomial $L_n(x)$ and its derivative $L'_n(x)$ in one dimension on the interval `int`, which corresponds to the edge of the bounding box B_E in one particular direction, at points `x`, of order given by `blist` considering that the total degree must be p. In order to compute the evaluations of the Legendre Polynomials we use the recursive formulas

$$(n+1)L_{n+1}(x) - (2n+1)xL_n(x) + nL_{n-1}(x) = 0,$$

$$L'_{n+1}(x) - xL'_n(x) = (n+1)L_n(x),$$

where $n = 1, \ldots, p-1$ and $L_o(x) = 1$. The subroutine takes as input `p`, `int`, `x` that is the vector of points we want to evaluate the polynomials in and `nq`, that is the length of this vector. It returns as output `P` and `Pder`.

- `basis`
  This subroutine evaluates the basis functions and their partial derivatives at the three-dimensional quadrature nodes `nod3` for every element by calling the previous subroutine `LegendreP`. The subroutine takes as input `Np`, `Fk`, `nod3`, the number of the quadrature nodes `nq3`, `b_box`, `blist` and it returns `phi` and `dphi` that correspond respectively to $\{\phi_{i,E}\}_{i=1}^{N_p}$ and $\{\nabla\phi_{i,E}\}_{i=1}^{N_p}$, see (2.34) and (2.35). We need `phi` and `dphi` in order to compute the volume integrals (2.27) and (2.29) using the quadrature formulas. See for example (2.32) where we use $\phi_{i,E}$ for the computation of the entries of the mass matrix.

- `basis_boundary`
  Here we can find the evaluation of the basis functions for every face f of the two neighbouring tratrahedra E^+ and E^-, that we call respectively `E1` and `E2`, at the two-dimensional quadrature nodes `nod2`. The subroutine takes as input `E1` and `E2`, `Np`, `Fk`, `nod2` , the number of the two-dimensional quadrature nodes `nq2`, the maps from the three-dimensional tetrahedron to the two dimensional faces of the tetrahedron `node_maps`, the bounding boxes `b_box1` and `b_box2` respectively of the polyhedron that contains the tetrahedron `E1` and of the polyhedron containing the neighbouring tetrahedron `E2`, `blist` and `e_E1`, the index of the face numbered from 1 to 4 according to the tetrahedron `E1`. The subroutine returns $\{\phi^+_{i,f}\}_{i=1}^{N_p}$ and $\{\phi^-_{i,f}\}_{i=1}^{N_p}$ both contained in `phi_b` and $\{\nabla\phi^+_{i,f}\}_{i=1}^{N_p}$, $\{\nabla\phi^+_{i,f}\}_{i=1}^{N_p}$ as `grad_b`. We need `phi_b` and `grad_b` in order to compute the integrals on the interface according to the quadrature formulas. See for example (2.38) and (2.39) where we use $\phi^+_{i,f}$ and $\phi^-_{i,f}$ for the computation of the entries of the matrix $\mathbf{S}$ on the internal faces.

3.3. Assembling of the algebraic linear system

The focus of this section is on the assembly part. Here we can find the subroutines that contain the assembly of the matrices and of the right hand side vector needed to solve the linear system. For now, this library can be used to solve diffusion reaction problems, therefore, beside the right hand side, we need to assemble the stiffness matrix $\mathbf{A}$ and the mass matrix $\mathbf{M}$. To solve the linear system we use an open source library called PETSc. PETSc, see `https://petsc.org/release/overview/`, the Portable, Extensible Toolkit

for Scientific Computation, is for the scalable (parallel) solution of scientific applications modeled by partial differential equations. Since PETSc requires the definition of its own matrices and vectors we have two files where we create and initialize the PETSc matrices and vectors.

- SET_PETSC_VECTORS.f90
  In this subroutine we create and initialize the two PETSc vectors, the right hand side and the solution.
- SET_PETSC_MATRICES.f90
  Here we can find a subroutine called SET_PETSC_MATRICES where we create and initialize the two PETSc matrices, the stiffness matrix and the mass matrix.

The real assembly part is performed with the following routines.

- assemble_local.f90
  In this module we assembly the local matrices and the right hand side vector. In particular, this module contains the following 5 subroutines.

  1. MAKE_STIFF_TET_LOC
     In this subroutine we assemble `V_loc`, the local term of the stiffness matrix approximating the integral on the tetrahedron by performing the loop on the three-dimensional quadrature nodes, see (2.36). It takes as input `Np`, `Jdet`, the weighths in three-dimensions `wei3`, `nq3`, the evaluations of the gradients of the basis functions at the quadrature nodes `dphi` and returns `V_loc`.
  2. MAKE_STIFF_FACE
     In this subroutine we assemble `I_loc`, `S_loc`, the terms of the stiffness matrix approximating the integrals on the faces of the tetrahedron and `IN_loc` and `SN_loc` on the faces of the neighbouring tetrahedron. See for example (2.38) and (2.39) for the details of the computation of the entries of the matrices `S_loc` and `SN_loc`. The subroutine takes as input θ, σ, `p`, `Np`, the neighbouring tetrahedra `E2`, `hk1` the diameter of the bounding box of the polyhedra containing the tetrahedra and `hk2` the diameter of the bounding box of the neighbouring polyhedra containing tetrahedra `E2`, the normal to the face `normal`, the area of the face `A`, the weights in two-dimensions `wei2`, `nq2` and the evaluations of the basis functions and their gradients at the quadrature nodes, `phi_b`, `grad_b`. It returns the matrices that we listed above.
  3. MAKE_RHS_TET
     Here we assemble `rhs_tet_loc`, the local right hand side term approximating

the integral on the tetrahedron, see (2.40). This subroutine takes as input `Np`, `Jdet`, the weighths in three-dimensions `wei3`, `nq3`, and the evaluations of the basis functions at the quadrature nodes `phi` and it returns the vector `rhs_tet_loc`.

4. MAKE_RHS_FACE
   In this subroutine we assemble `rhs_bd_loc`, the local right hand side term that approximates the integral on the faces of the tetrahedron that are boundary faces. Indeed, in the case of $g_D \neq 0$ we have also an integral term on the boundary faces in the definition of the right hand side vector, see (2.41). The subroutine takes as input θ, σ, `p`, `Np`, the neighbouring tetrahedra `E2`, the reference map `Fk`, `hk1` the diameter of the bounding box of the polyhedra containing the tetrahedra and `hk2` the diameter of the bounding box of the neighbouring polyhedra containing tetrahedra `E2`, the index of the face `e` the normal to the face `normal`, the area of the face `A`, the weights in two-dimensions `wei2`, the two-dimensional quadrature nodes `nod2`, `nq2`, the maps `node_maps` and `phi_b`,`grad_b`. It returns `rhs_bd_loc`.

5. MAKE_MASS_LOC
   In this subroutine we find the assembly of `M_loc` the local term of the mass matrix that approximates the integral on the tetrahedron, see (2.32). The subroutine takes as input `Np`, `Jdet`, the weighths `wei3`, `nq3`, `phi` and returns `M_loc`.

- MAKE_MATRICES.f90
  In this file there is a subroutine with the same name where we assemble the two PETSc matrices, the stifness matrix **A** and the mass matrix **M** by performing a loop on the elements. MAKE_MATRICES takes as input `Np`, the total number of degrees of freedom `Np(num_poly)`, the **Mesh_ Structure** and it returns the Petsc matrices `petsc_stiff`, `petsc_mass` assembled.

- MAKE_RHS.f90
  Here we can find a subroutine called MAKE_RHS where we assemble the term on the right hand side **f** by performing a loop on the elements. MAKE_RHS takes as input `Np`, the total number of degrees of freedom `Np(num_poly)`, the **Mesh_Structure** and it returns the vector `petsc_rhs` assembled.

3.4. Solving the linear system

We have assembled the Petsc matrices and vectors that we need to solve the linear system. Now we set the solver, direct or iterative, and optionally the preconditioner in the subroutine SOLVER_SETTINGS and finally we solve the linear system by calling the PETSc function KSPSolve. However, this can be done only after creating the object KSP. Directly from `https://petsc4py.readthedocs.io/en/stable/manual/ksp/`, the KSP object is the heart of PETSc, because it provides uniform and efficient access to all of the package's linear system solvers, including parallel and sequential, direct and iterative. KSP is intended for solving systems of the form:

$$\mathbf{Ax} = \mathbf{f}$$

where $\mathbf{A}$ denotes the matrix representation of a linear operator, $\mathbf{f}$ is the right hand side vector, and $\mathbf{x}$ is the solution vector. KSP uses the same calling sequence for both direct and iterative solution of a linear system.

To solve a linear system with KSP, we first create a solver context with the command KSPCreate, then we call the following routine to set the matrices associated with the linear system KSPSetOperators. For further details see manual of PETSc `https://petsc4py.readthedocs.io/en/stable/manual/`. Now that we have the solution as PETSc vector, we copy the solution into a vector. This task is performed by the subroutine PETSC_SCATTER_VECTOR. In addition, this routine collects the values of the solution from the different processors and stores them in the right locations of the final solution vector.

3.5. Post-processing

Once we solved the linear system, the program **Lymph3D** performs the post-processing by calling two subroutines contained in the module `post_processing`

- `export_solution`
  In this subroutine we evaluate the solution function at the vertices of the tetrahedra and then we write these values on a .vtk file in order to visualize them on a suitable software, as for example Paraview. This is done by calling the subroutines VTK_WRITE contained in the module MOD_VTK

- `errors`
  Here, starting from the known exact solution of the problem, if available, and the

approximated solution, we perform the computation of the errors in norm L^2 and DG.

3.6. User-Guide

In this section we provide a tutorial to use the library to solve a problem. We list here the steps to follow.

1. Build a mesh of tetrahedra of the desired geometry with a suitable software
2. Launch the file exoToMesh.m on Matlab that returns the file .mesh
3. Use this file .mesh to agglomerate the mesh with METIS
4. Change the file 'Poly.input' by changing the name of the mesh file
5. Modify the file test.f90 to modify the forcing term
6. Modify the file test.f90 to change α the coefficient in the definition of the penalty function or θ to change the method (to IIP or NIP)
7. Modify the file test1.mate to modify the total degree of basis function
8. Compile the program with the command `make` and then to run it with the command `mpirun -np 2 ../Lymph3D`. Here **Lymph3D** is the executable file and the number of processors is set to two.

3.7. Mesh Generation

In this thesis we used the software CUBIT to generate the mesh. From `https://cubit.sandia.gov/`, CUBIT is a full-featured software toolkit for robust generation of two-dimensional and three-dimensional finite element meshes and geometry preparation. Its main goal is to reduce the time required to generate meshes, particularly large meshes of complicated, interlocking assemblies.

We generate for example the mesh of a cube with $N_{tet} = 4496$ by implementing the lines of code on Cubit that we can see in Algorithm 3.4.

Algorithm 3.4 Generate tetrahedral mesh of cube

```
brick x 1
vol 1 scheme tetmesh
mesh vol 1 size 0.2
block 1 vol 1
block 2 surface 1 2 3 4 5
```

We have to export the file into format .e, then convert it into .txt, and finally we launch the file exToMesh.m in order to obtain the file .mesh that is going to be read by the proper functions in the library. In Figure 3.4 we can see the mesh that is generated with this lines of code. In order to visualize the mesh we created the file .vtk.

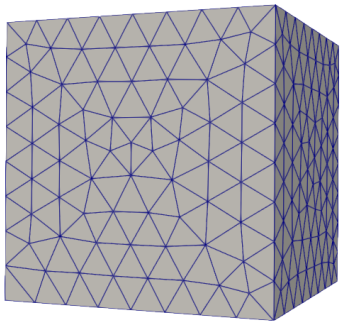


Figure 3.4: Tetrahedral mesh of a cube with $N_{tet} = 4496$.

In the Figure 3.5 we can see the mesh that is generated by METIS algorithm. In order to visualize the mesh we created the file .vtk. The different colours represent the polyhedra obtained with the agglomeration.

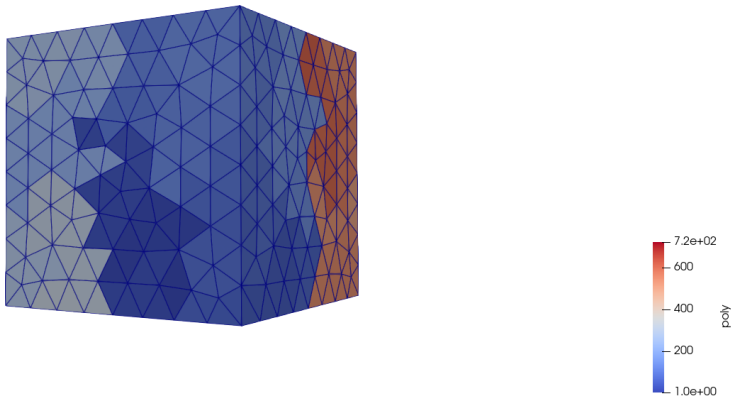


Figure 3.5: Example of the agglomerated mesh of a cube generated by METIS with 720 polyhedra giving as input the tetrahedral mesh in Figure 3.4.

4 | Numerical Tests

The aim of this chapter is to present some numerical results to test the convergence of the method and to test the performance of the LYMPH3D library.

4.1. Test case 1

Consider the following diffusion reaction problem introduced in Section 2.3.3.

Find u such that

$$\begin{cases} -\Delta u + cu = f \text{ in } \Omega\,, \\ \qquad\qquad u = g_D \text{ on } \partial\Omega\,. \end{cases} \tag{4.1}$$

where $\Omega = (0,1)^3$, $u_{ex}(x,y,z) = e^{xyz}$ and $g_D = u_{ex}$ on $\partial\Omega$.

First, we consider a pure diffusion equation, i.e., choosing $c = 0$ in problem (4.1). The forcing term is $f(x,y,z) = -e^{xyz}((xy)^2 + (xz)^2 + (yz)^2)$.

We solve this problem with the algorithm previously described on a succesively finer mesh, first using a tetrahedral mesh, and then a polyhedral mesh. As we already discussed in Chapter 3, the polyhedral grids are obtained by agglomeration starting from a tetrahedral mesh.

We solve the problem with the SIP method, meaning setting $\theta = -1$ in (2.6). The penalty discontinuity function is defined in (2.12) with penalty coefficient $\alpha = 10$, p is the polynomial approximation order assuming $p_E = p \geq 1\ \forall E \in \mathscr{T}_h$ and h_E is the diameter of the element E.

In Figure 4.1 we can see the numerical solution of the problem computed in the vertices of the tetrahedra obtained with the software Paraview.

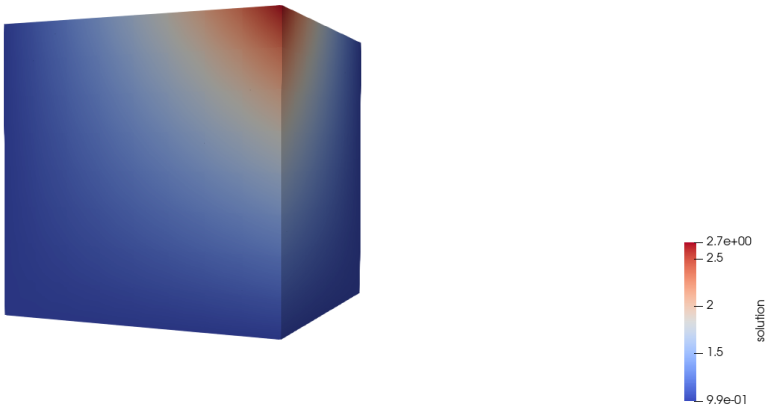


Figure 4.1: Test case 1. Numerical solution

In Figure 4.3 we investigate the convergence of the PolyDG on sequences of finer cube and tetrahedral meshes for different values of the polynomial degrees p between 1 and 3. We refine the mesh of the cube by varying the number of tetrahedra $N_{tet} \in \{48, 384, 1296, 3072\}$ and for each N_{tet} we solve the linear system.

In Figure 4.5 we investigate the convergence results of the PolyDG method applied to the same problem on a polyhedral mesh obtained by METIS starting from the tetrahedral meshes we used for the previous analysis. We consider in this case a varying number of polyhedra $N_{poly} \in \{10, 100, 400, 700\}$. For each N_{poly} we solve the linear system for $p = 1, 2, 3, 4$.

For each fixed p we plot the errors, measured in terms of both the $L^2(\Omega)$ norm and DG norm, versus the diameter of the elements, tetrehedra in the first case and polyhedra in the second one. In both cases we clearly observe that $||u - u_h||_{L^2(\Omega)}$ and $||u - u_h||_{DG}$ converge to zero at the optimal rates $\mathscr{O}(h^{p+1})$ and $\mathscr{O}(h^p)$, respectively, as the mesh size h tends to zero for each fixed p. The numerical results confirm the optimality of the PolyDG method for pure diffusion problems in accordance with the theoretical convergence results, see Theorem 2.2 and Theorem 2.3.

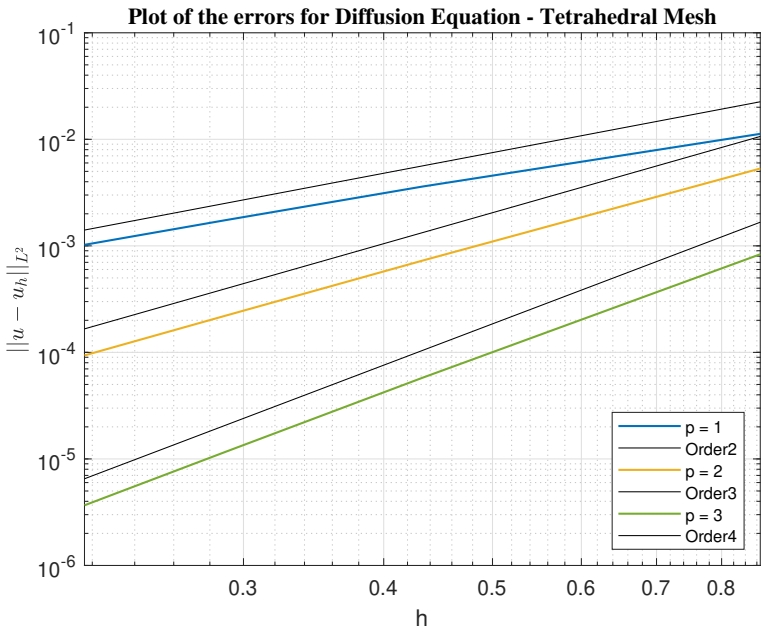


Figure 4.2: Test case 1. Computed errors in the L^2 norm for $p = 1, 2, 3$ (tetrahedral meshes)

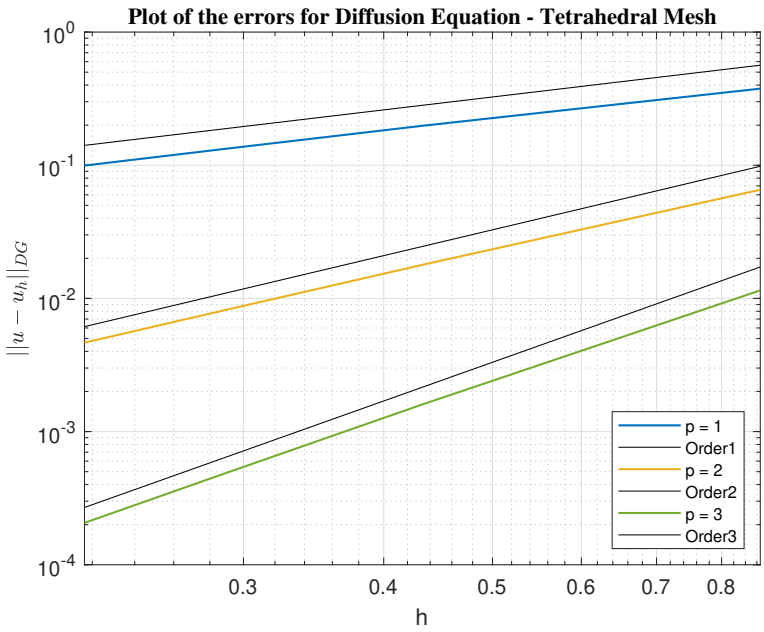


Figure 4.3: Test case 1. Computed errors in the DG norm for $p = 1, 2, 3$ (tetrahedral meshes)

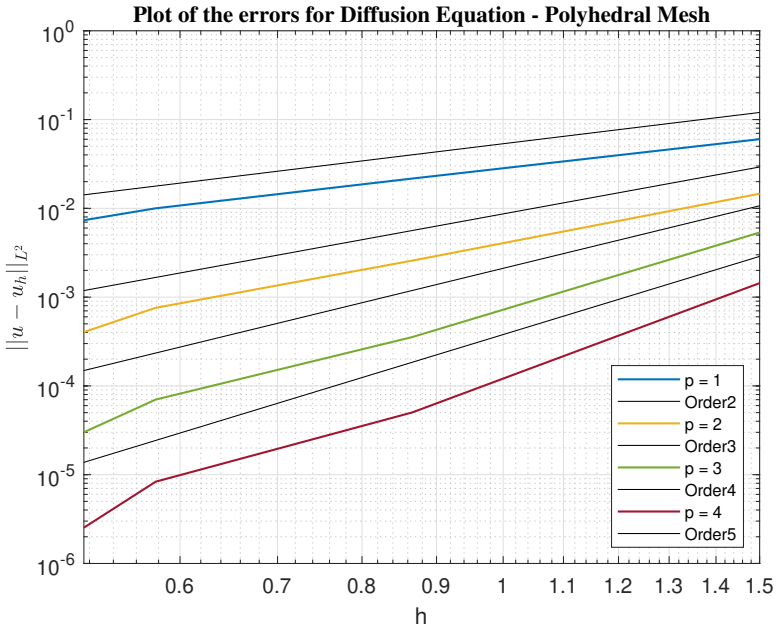


Figure 4.4: Test case 1. Computed errors in the L^2 norm for $p = 1, 2, 3, 4$ (polyhedral meshes)

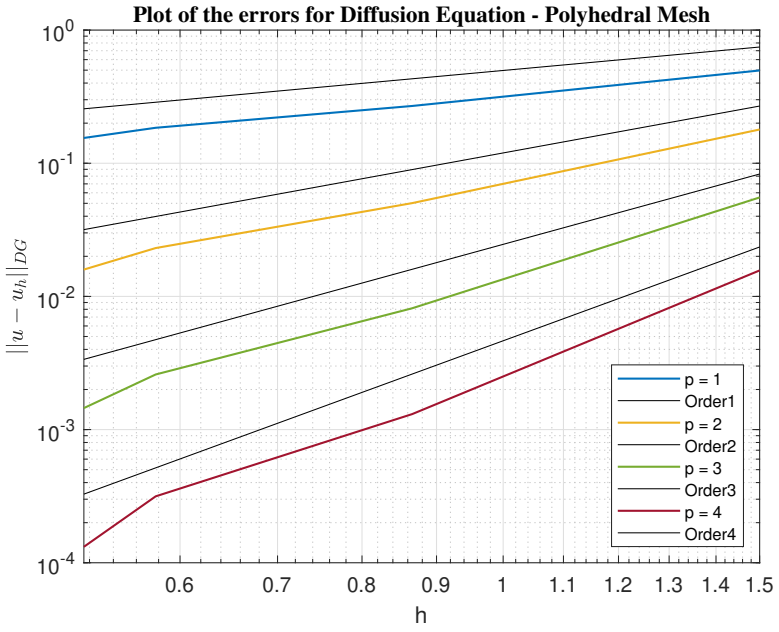


Figure 4.5: Test case 1. Computed errors in the DG norm for $p = 1, 2, 3, 4$ (polyhedral meshes)

4.2. Test case 2

We consider the diffusion reaction PDE problem (4.1) reported in Section 4.1 with $c = 0.5$, $\Omega = (0,1)^3$, $u_{ex}(x,y,z) = e^{xyz}$ and $g_D = u_{ex}$ on $\partial\Omega$. The forcing term, in this case is $f(x,y,z) = -e^{xyz}((xy)^2 + (xz)^2 + (yz)^2 - 0.5)$.

As in the previous example, we discretize the problem with the SIP method and we solve this problem with the algorithm previously described on a succesively finer mesh, using firstly a type of mesh composed of tetrahedra, and secondly a polyhedral mesh.

In Figure 4.6 we can see the numerical solution of the problem computed in the vertices of the tetrahedra obtained with the software Paraview. Notice that the solution is the same as before, which is coherent with our problem since we only changed the forcing term.

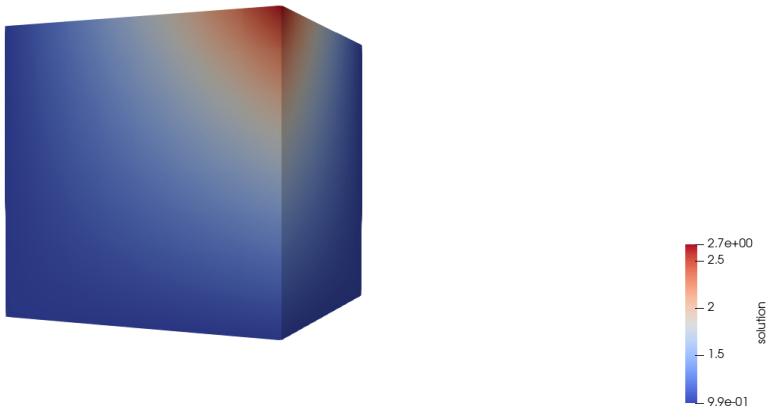


Figure 4.6: Numerical solution of the diffusion reaction equation

We consider the same refined meshes of the cube as before, composed of tetrahedra for the first case with the number of tetrahedra varying in $N_{tet} \in \{48, 384, 1296, 3072\}$ and polyhedra for the second case with the number of polyhedra varying in $N_{poly} \in \{10, 100, 400, 700\}$.

We plot the errors for $p = 1, 2, 3$, measured in terms of both the $L^2(\Omega)$ norm and DG norm versus the diameter of the elements, tetrehedra in Figure 4.8 and polyhedra in Figure 4.10. For the polyhedral mesh we report also the errors for $p = 4$. Again we observe that $||u - u_h||_{L^2(\Omega)}$ and $||u - u_h||_{DG}$ converge to zero at the optimal rates $\mathcal{O}(h^{p+1})$ and $\mathcal{O}(h^p)$ respectively, as the mesh size h tends to zero for each fixed p confirming the optimality of the PolyDG method for diffusion-reaction problems in accordance with the theoretical convergence results, see Theorem 2.2 and Theorem 2.3.

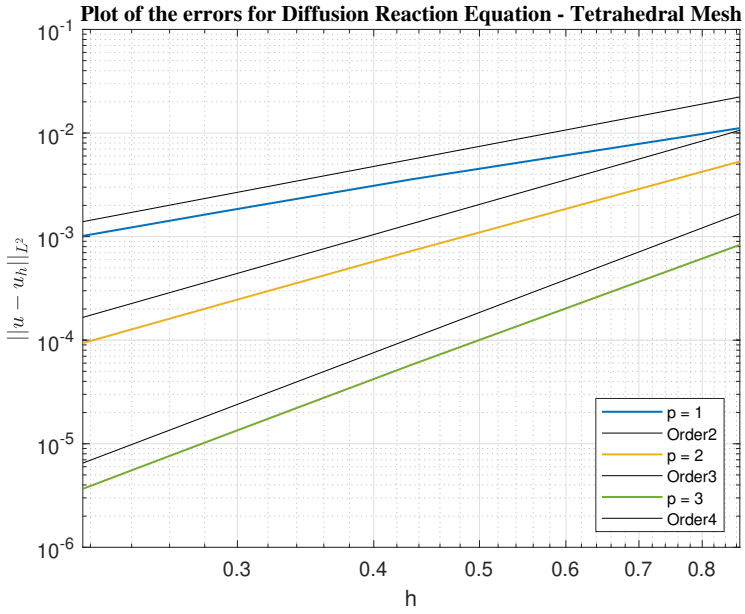


Figure 4.7: Test case 2. Computed errors in the L^2 norm for $p = 1, 2, 3$ (tetrahedral meshes)

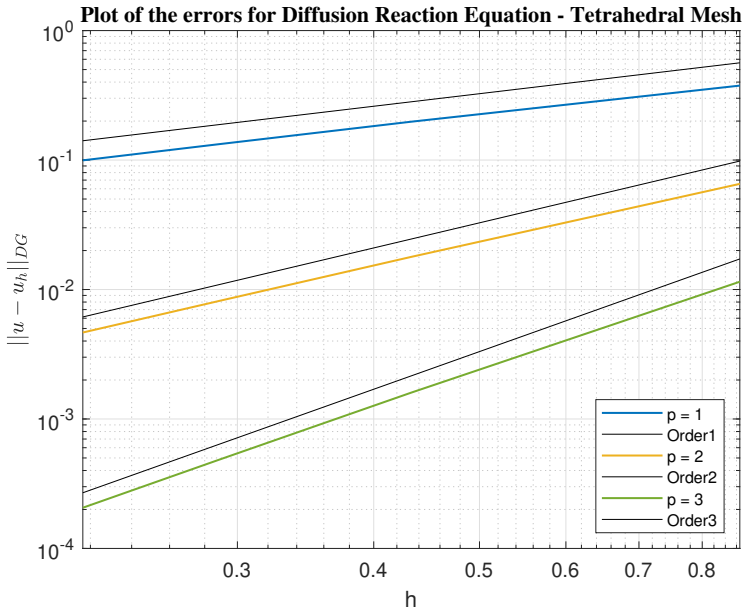


Figure 4.8: Test case 2. Computed errors in the DG norm for $p = 1, 2, 3$ (tetrahedral meshes)

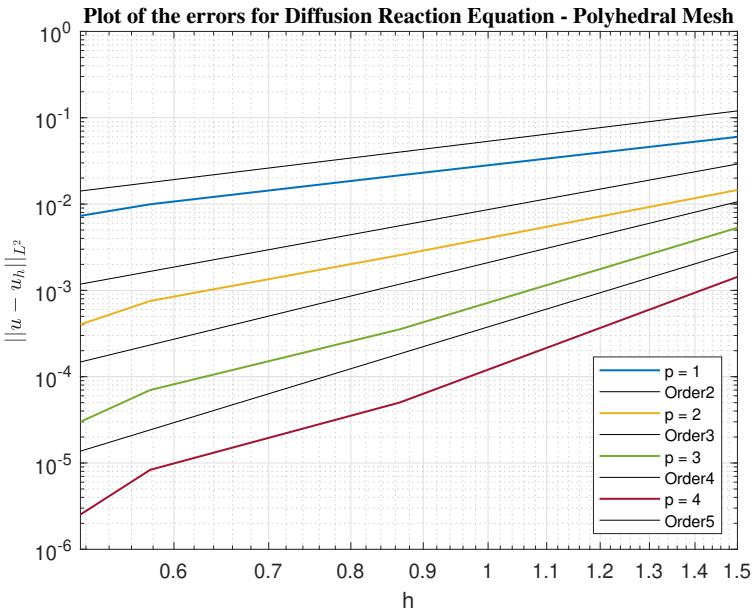


Figure 4.9: Test case 2. Computed errors in the L^2 norm for $p = 1, 2, 3, 4$ (polyhedral meshes)

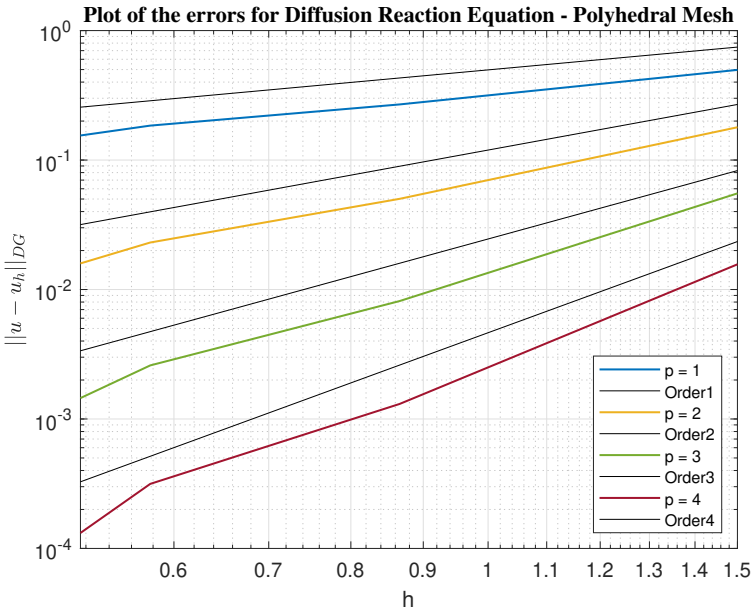


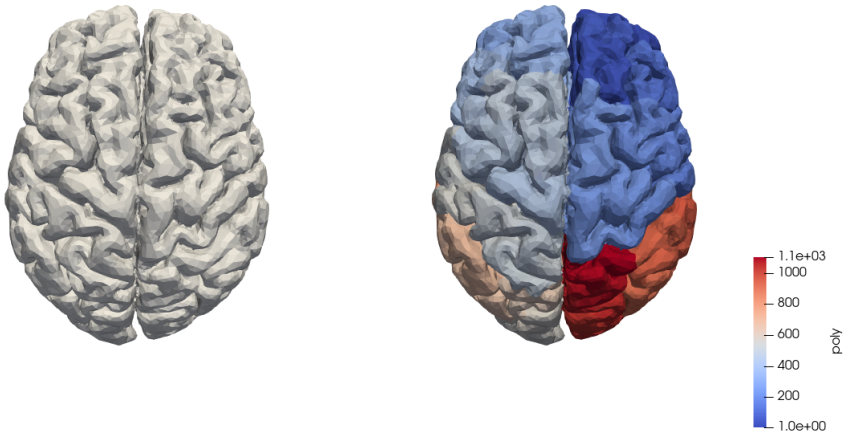
Figure 4.10: Test case 2. Computed errors in the DG norm for $p = 1, 2, 3, 4$ (polyhedral meshes)

4.3. Test case 3

We already mentioned that one of the advantages of choosing polytopic element shapes over standard simplicial/hexahedral elements is that the average number of elements needed to discretize complicated domains is smaller and this allows to reduce the complexity of the given computational geometry and that this advantage becomes even more evident whenever the domain contains complex geometrical features. In this section we present a numerical test in which we solve the PDE problem (4.1) in Section 4.1 with $c = 0$. We solve the diffusion problem with the SIP PolyDG method on the agglomerated mesh of a human brain with $f(x, y, z) = -e^{xyz}((xy)^2 + (xz)^2 + (yz)^2)$ and $g_D = e^{xyz}$.

The three-dimensional mesh of a human brain is one example of a very complicated geometry and the PolyDG method is perfectly suited to be employed in the context of brain modelling. In a recent work, [29] they discretize multiple networks poroelastic model of the human brain in space by using the PolyDG method.

If we consider the tetrahedral mesh in Figure 4.11a, we cannot solve the numerical problem on this kind of mesh because the number of degrees of freedom is too high for $p \geq 2$. For this reason we solve the numerical problem on the polyhedral mesh in Figure 4.11b and on the tetrahedral mesh in Figure 4.11a for $p = 1$. The polyhedral mesh is obtained with METIS by agglomerating the tetrahedral mesh in Figure 4.11a and by choosing a number of polyhedra $N_{poly} = 2000$. In Figure 4.12 we can see a plot of the numerical solution. We report in Table 4.1 the errors in the L^2 norm and in the DG norm for $p = 1$ solving the problem with the tetrahedral ($N_{tet} = 127824$) and polyhedral mesh ($N_{poly} = 2000$), respectively. We notice that solving the problem on the polyhedral mesh, we reduce the number of elements and the computed errors (both in L^2 and DG norm) are even lower than the errors obtained by solving the problem employing the tetrahedral mesh.



(a) Tetrahedral mesh of the brain with $N_{tet} = 127824$.

(b) Agglomerated mesh of the brain with $N_{poly} = 2000$.

Figure 4.11: Mesh of the human brain. In Figure 4.11a we have the tetrahderal mesh composed of $N_{tet} = 127824$. In Figure 4.11b we have the agglomerated polyhedral mesh with $N_{poly} = 2000$ obtained with METIS from the tetrahderal mesh in Figure 4.11a.

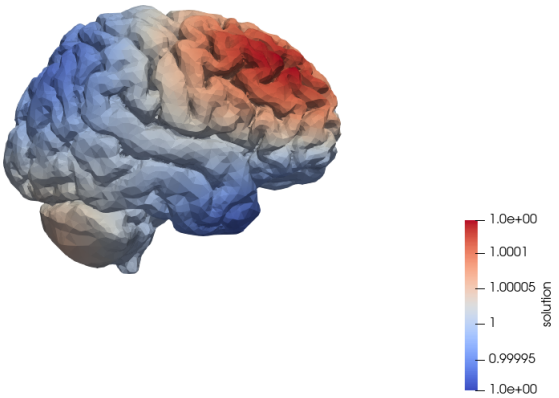


Figure 4.12: Test case 3. Numerical solution of the diffusion equation with $f(x, y, z) = -e^{xyz}((xy)^2 + (xz)^2 + (yz)^2)$ and $g_D = e^{xyz}$ solved on the agglomerated mesh of the brain with $N_{poly} = 2000$.

Element type	L^2 norm	DG norm	h
tetrahedra	$1.21\,10^{-6}$	$1.66\,10^{-3}$	$1.08\,10^{-2}$
polyhedra	$3.47\,10^{-8}$	$1.39\,10^{-5}$	$3.52\,10^{-2}$

Table 4.1: Test case 3. Computed errors in norm L^2 and DG for $p = 1$ solving the problem with the tetrahedral ($N_{tet} = 127824$) and polyhedral mesh ($N_{poly} = 2000$), respectively.

4.4. Test case 4

Now we investigate the numerical solution of a problem for which there is not a known analytical solution. We consider again the diffusion problem (4.1) with $c = 0$ and we choose $f(x, y, z) = e^{-(x^2+y^2+z^2)}$ and $g_D = 0$. We solve this problem with the SIP PolyDG method on the agglomerated mesh of the brain in Fig 4.11b with $N_{poly} = 2000$ and choosing $p = 1$. In Figure 4.13 we can see the numerical solution of the diffusion problem with this new function f.

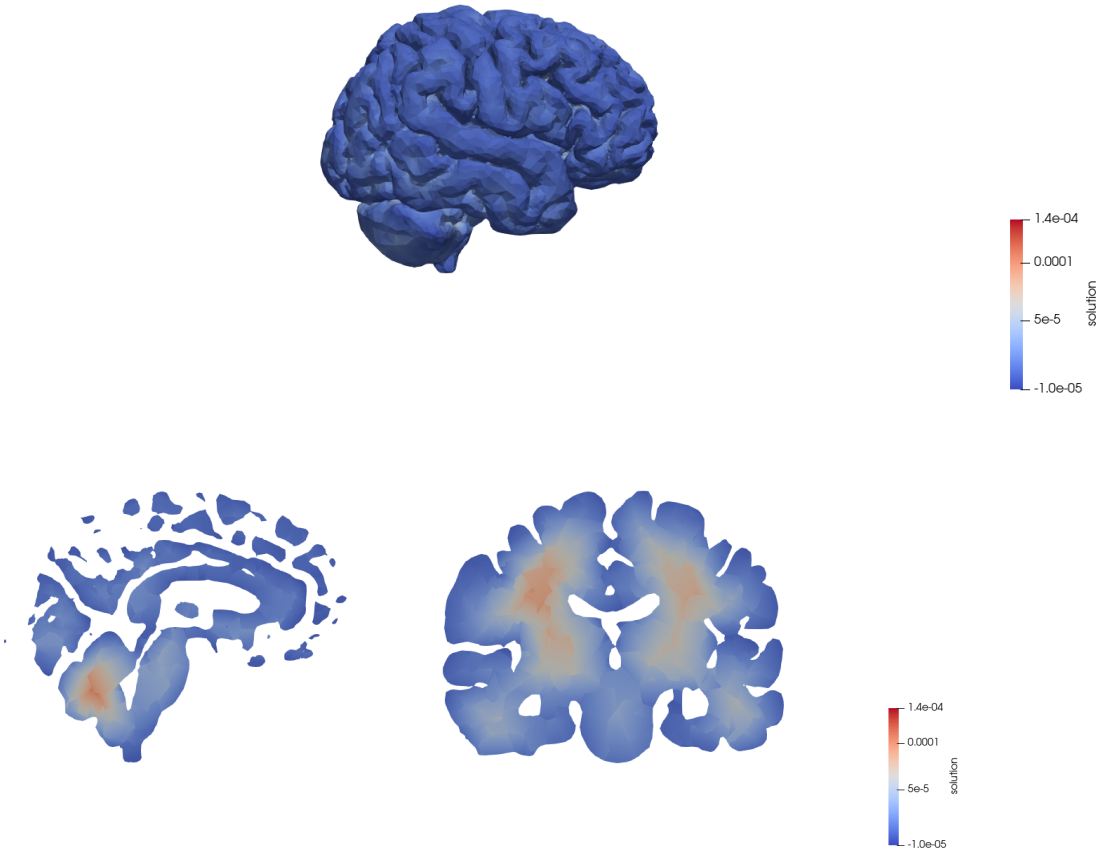


Figure 4.13: Test case 4. Numerical solution of the diffusion problem with $f(x, y, z) = e^{-(x^2+y^2+z^2)}$ and $g_D = 0$ solved on the agglomerated mesh of the brain with $N_{poly} = 2000$. In the first image there is the three-dimensional numerical solution of the brain. The other two images show the numerical solution on two different sections of the human brain.

5 | Conclusions and future developments

In this work of thesis we recalled the main theoretical results on the discontinuous Galerkin methods on polytopic mesh. We introduced the library LYMPH3D to solve diffusion-reaction problems with the PolyDG method in three-dimensions. We described in detail the main functions of the library and we presented two numerical tests of a pure diffusion equation and of a diffusion reaction equation on a simple mesh of a cube. These tests confirmed the known theoretical results on the discontinuous Galerkin methods on tetrahedral and polyhedral meshes. Therefore, we confirm the possibility to use this library to solve these kind of problems on agglomerated polyhedral meshes. Then we solved the diffusion equation on a complicated geometry of the human brain. There are many possible future developments of this work in order to make the library as general as possible. We recall that in the library we compute the integrals using the Sub-Tessellation method, see Section 2.3.2. This method always requires a tetrahedral mesh $\mathscr{T}_h^{fine}$. Here, this is not an issue since we create the polyhedral mesh via agglomeration of a tetrahedral mesh. Therefore, we already have at our disposal the mesh $\mathscr{T}_h^{fine}$. However, in order to compute the integrals to assemble the linear system we need to perform a loop on the tetrahedral elements, and this is computationally expensive. One possible improvement of this work is to introduce in the library the Quadrature Free algorithm (see [5]) to compute the integrals without the need of the tetrahedral mesh $\mathscr{T}_h^{fine}$. It has been shown that this integration approach leads to a considerable improvement in the computational performance compared to classical quadrature algorithms based on sub-tessellation, in both two and three-dimensions.

Moreover, for now the library can be used to solve diffusion equations on tetrahedral meshes and polyhedral meshes obtained as agglomeration of tetrahedral meshes. One idea could be to expand the library in order to read generic hybrid grids based on a convenient combination of hexahedral/tetrahedral/prysmatic/polytopic elements. These computational hybrid grids are easy to be generated. The idea is to generate an initial (hexahedral/tetrahedral in three dimensions) mesh, based on employing standard mesh

generators; then elements intersecting the irregularities in the geometry are suitably cut and/or agglomerated, thus generating polytopes, while keeping a regular structure elsewhere.

Another possible development of the library is to implement the algorithms to solve more complicated problems, first introducing also the transport term in a general elliptic PDE, then trying to solve a dynamic equation. The idea is that this library is the starting point in order to have an efficient open source library to solve heterogeneous differential problems.

Bibliography

[1] J. Aghili, D. A. Di Pietro, and B. Ruffini. An *hp*-Hybrid High-order method for variable diffusion on general meshes. *Computational Methods in Applied Mathematics*, 17(3):359–376, 2017.

[2] P. F. Antonietti, S. Giani, and P. Houston. *hp*-version composite discontinuous Galerkin methods for elliptic problems on complicated domains. *SIAM Journal on Scientific Computing*, 35(3):A1417–A1439, 2013.

[3] P. F. Antonietti, S. Giani, and P. Houston. Domain decomposition preconditioners for discontinuous Galerkin methods for elliptic problems on complicated domains. *Journal of Scientific Computing*, 60:203–227, 2014.

[4] P. F. Antonietti, A. Cangiani, J. Collis, Z. Dong, E. H. Georgoulis, S. Giani, and P. Houston. *Review of Discontinuous Galerkin Finite Element Methods for Partial Differential Equations on Complicated Domains*, pages 281–310. Springer International Publishing, Cham, 2016.

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

[6] P. F. Antonietti, G. Manzini, and M. Verani. The fully nonconforming virtual element method for biharmonic problems. *Mathematical Models and Methods in Applied Sciences*, 28(02):387–407, 2018.

[7] P. F. Antonietti, C. Facciolà, P. Houston, I. Mazzieri, G. Pennesi, and M. Verani. *High–order Discontinuous Galerkin Methods on Polyhedral Grids for Geophysical Applications: Seismic Wave Propagation and Fractured Reservoir Simulations*, pages 159–225. Springer International Publishing, Cham, 2021.

[8] E. Baas and J. H. Kuiper. A numerical model of heterogeneous surface strains in polymer scaffolds. *Journal of biomechanics*, 41(6):1374–1378, 2008.

[9] E. Baas, J. H. Kuiper, Y. Yang, M. A. Wood, and A. J. El Haj. In vitro bone growth

responds to local mechanical strain in three-dimensional polymer scaffolds. *Journal of biomechanics*, 43(4):733–739, 2010.

[10] L. Beirão da Veiga, F. Brezzi, A. Cangiani, G. Manzini, L. D. Marini, and A. Russo. Basic principles of virtual element methods. *Mathematical Models and Methods in Applied Sciences*, 23(01):199–214, 2013.

[11] L. Botti and D. A. Di Pietro. Assessment of hybrid high-order methods on curved meshes and comparison with discontinuous Galerkin methods. *Journal of Computational Physics*, 370:58–84, 2018.

[12] L. Botti, D. A. Di Pietro, and J. Droniou. A Hybrid High-Order method for the incompressible Navier–Stokes equations based on Temam's device. *Journal of Computational Physics*, 376:786–816, 2019.

[13] M. Botti, D. A. Di Pietro, and P. Sochala. A Hybrid High-Order method for nonlinear elasticity. *SIAM Journal on Numerical Analysis*, 55(6):2687–2717, 2017.

[14] S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*, volume 15 of *Texts in Applied Mathematics*. Springer New York, NY, 3 edition, 2008.

[15] F. Brezzi, A. Buffa, and K. Lipnikov. Mimetic finite differences for elliptic problems. *ESAIM: Mathematical Modelling and Numerical Analysis*, 43(2):277–295, 2009.

[16] A. Cangiani, G. Manzini, and A. Russo. Convergence analysis of the mimetic finite difference method for elliptic problems. *SIAM Journal on Numerical Analysis*, 47 (4):2612–2637, 2009.

[17] A. Cangiani, E. H. Georgoulis, and M. Jensen. Discontinuous Galerkin Methods for Mass Transfer through Semipermeable Membranes. *SIAM Journal on Numerical Analysis*, 51(5):2911–2934, 2013.

[18] A. Cangiani, Z. Dong, and E. H. Georgoulis. *hp*-Version space-time discontinuous Galerkin methods for parabolic problems on prismatic meshes, 2017.

[19] A. Cangiani, Z. Dong, E. H. Georgoulis, and P. Houston. *Hp-version discontinuous Galerkin methods on polytopic meshes*. SpringerBriefs in Mathematics,Springer International Publishing, 2017.

[20] A. Cangiani, Z. Dong, E. H. Georgoulis, and P. Houston. *Hp-Version Discontinuous Galerkin Methods on Polygonal and Polyhedral Meshes*. SpringerBriefs in Mathematics, 2017.

[21] A. Cangiani, G. Manzini, and O. J. Sutton. Conforming and nonconforming virtual

element methods for elliptic problems. *IMA Journal of Numerical Analysis*, 37(3): 1317–1354, 2017.

[22] F. Chave, D. A. Di Pietro, F. Marche, and F. Pigeonneau. A Hybrid High-Order Method for the Cahn–Hilliard problem in Mixed Form. *SIAM Journal on Numerical Analysis*, 54(3):1873–1898, 2016.

[23] F. Chave, D. A. Di Pietro, and L. Formaggia. A hybrid high-order method for Darcy flows in fractured porous media. *SIAM Journal on Scientific Computing*, 40 (2):A1063–A1094, 2018.

[24] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. Society for Industrial and Applied Mathematics, 2002.

[25] B. Cockburn, B. Dong, and J. Guzmán. A superconvergent LDG-hybridizable Galerkin method for second-order elliptic problems. *Mathematics of Computation*, 77(264):1887–1916, 2008.

[26] B. Cockburn, J. Gopalakrishnan, and R. Lazarov. Unified hybridization of discontinuous Galerkin, mixed, and continuous Galerkin methods for second order elliptic problems. *SIAM Journal on Numerical Analysis*, 47(2):1319–1365, 2009.

[27] B. Cockburn, J. Guzmán, and H. Wang. Superconvergent discontinuous Galerkin methods for second-order elliptic problems. *Mathematics of Computation*, 78(265): 1–24, 2009.

[28] B. Cockburn, J. Gopalakrishnan, and F.-J. Sayas. A projection-based error analysis of HDG methods. *Mathematics of Computation*, 79(271):1351–1367, 2010.

[29] M. Corti, P. F. Antonietti, L. Dede, and A. M. Quarteroni. Numerical Modelling of the Brain Poromechanics by High-Order Discontinuous Galerkin Methods. *M3AS*, 2023.

[30] B. Da Veiga, J. Droniou, and G. Manzini. A unified approach for handling convection terms in finite volumes and mimetic discretization methods for elliptic problems. *IMA journal of numerical analysis*, 31(4):1357–1401, 2011.

[31] B. A. de Dios, K. Lipnikov, and G. Manzini. The nonconforming virtual element method. *ESAIM: Mathematical Modelling and Numerical Analysis*, 50(3):879–904, 2016.

[32] D. Di Pietro and J. Droniou. A Hybrid High-Order method for Leray–Lions elliptic equations on general meshes. *Mathematics of Computation*, 86(307):2159–2191, 2017.

[33] D. A. Di Pietro and J. Droniou. W s, p-approximation properties of elliptic projectors on polynomial spaces, with application to the error analysis of a Hybrid High-Order discretisation of Leray–Lions problems. *Mathematical Models and Methods in Applied Sciences*, 27(05):879–908, 2017.

[34] D. A. Di Pietro and A. Ern. Hybrid high-order methods for variable-diffusion problems on general meshes. *Comptes Rendus Mathématique*, 353(1):31–34, 2015.

[35] D. A. Di Pietro and S. Krell. A Hybrid High-Order method for the steady incompressible Navier–Stokes problem. *Journal of Scientific Computing*, 74:1677–1705, 2018.

[36] J. Droniou, R. Eymard, and R. Herbin. Gradient schemes: generic tools for the numerical analysis of diffusion equations. *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique*, 50(3):749–781, 2016.

[37] T.-P. Fries and T. Belytschko. The extended/generalized finite element method: an overview of the method and its applications. *International journal for numerical methods in engineering*, 84(3):253–304, 2010.

[38] W. Hackbusch and S. A. Sauter. Composite finite elements for problems containing small geometric details: Part ii: Implementation and numerical results. *Computing and Visualization in Science*, 1(1):15–25, 1997.

[39] W. Hackbusch and S. A. Sauter. Composite finite elements for the approximation of PDEs on domains with complicated micro-structures. *Numerische Mathematik*, 75: 447–472, 1997.

[40] P. Houston and N. Sime. Numerical modelling of MPA-CVD reactors with the discontinuous Galerkin finite element method. *Journal of Physics D: Applied Physics*, 50(29):295202, 2017.

[41] P. Houston and E. Süli. Stabilised *hp*-Finite Element Approximation of Partial Differential Equations with Nonnegative Characteristic form. *Computing*, 66:99–119, 2001.

[42] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[43] G. Karypis and V. Kumar. METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, version 4.0. 2009.

[44] A. Quarteroni. *Modellistica Numerica per Problemi Differenziali*. Springer Milano, 2013.

[45] N. Sime. *Numerical modelling of chemical vapour deposition reactors*. PhD thesis, University of Nottingham, 2016.

[46] E. M. STEIN. *Singular Integrals and Differentiability Properties of Functions (PMS-30)*. Princeton University Press, 1970. ISBN 9780691080796.

[47] N. Sukumar and A. Tabarraei. Conforming polygonal finite elements. *International Journal for Numerical Methods in Engineering*, 61(12):2045–2066, 2004.

A | Appendix A

In Chapter 3, we have described how the library LYMPH3D works. Notice that in Section 3.1 we focused on the part of the library relative to the mesh. We have seen in detail the structures **Element**, **Polyhedron** and **Mesh_Structure** and how we perform the agglomeration of the tetrahedral mesh in the function MESH_AGGLOMERATION. Moreover, we have also seen how we compute the key properties of the polyhedra necessary to solve the problem, in the functions CREATE_POLY_LIST and CREATE_BBOX_EL (see Algorithm 3.2). In this Appendix we show the implementation in Fortran of some of the most important functions of the library. Below we can see the implementation of the following functions, in this order:

1. `Lymph3D` that is the main program
2. `basis` and `basis_boundary` from the module `basis_function`, see Section 3.2
3. `MAKE_MATRICES` briefly described in Section 3.3
4. `MAKE_RHS` briefly described in Section 3.3
5. `test` where we set the forcing term, the Dirichlet boundary conditions and the analytical solution, if available, to compute the errors.

Lymph3D.f90

```fortran
!    Author: Nicoletta De Giosa
!    This file is part of the library LYMPH3D
!
!> @brief Lymph3D (Discontinuous Galerkin methods on polyhedral meshes for PDE problems)

! Here starts the code Lymph3D

      program Lymph3D

#include<petsc/finclude/petscksp.h>

      use petscksp
      use petscmat
      use mpi
      use Poly_setup_mpi
      use test
      use Poly_global
      use Poly_data
      use Poly_mesh
      use post_processing

      implicit none

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     DEFINITION OF PETSC VARIABLES
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      Mat :: petsc_stiff,petsc_mass ! stiffness and mass matrices
      Vec :: petsc_sol !solution vector for the algebraic system
      Vec petsc_rhs ! Right hand side vector for the algebraic system

      KSP :: ksp
      PC :: pc

      integer (kind=4) :: i,j
      real(kind=8), dimension(:,:), ALLOCATABLE :: u
      real(kind=8), dimension(:), ALLOCATABLE :: u_loc
      integer(kind=4) :: total_dof,ndof,ndof_loc
      integer(kind=4) :: Np,p,Npoly
      integer(kind=4),dimension(:),allocatable :: petsc_num
      integer(kind=4),dimension(:),allocatable :: nnod_num
      real(kind=8), pointer:: sol_ptr(:)
      integer(kind=4) :: ip,iter
      real(kind=8) :: err_L2,err_L2_mpi
      real(kind=8) :: err_DG,err_DG_mpi
      real(kind=8) :: hmax,hmax_mpi

      logical :: IsPoly

      type(Data_Structure) :: PolyData
      type(Mesh_Structure) :: PolyMesh

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      allocate(mpi_stat(MPI_STATUS_SIZE))

      call INITIALIZATION()

      start = MPI_WTIME()

      if (mpi_id.eq.0) then
         write(*,'(A)')''
         write(*,'(A)')'<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<>'
         write(*,'(A)')'<                                                     >'
         write(*,'(A)')'<                        Lymph3D                      >'
         write(*,'(A)')'<                                                     >'
         write(*,'(A)')'<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<>'
         write(*,'(A)')''
      endif

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```fortran
!     READ INPUT FILES AND ALLOCATE VARIABLES
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      call READ_INPUT_FILES(PolyData,PolyMesh)

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     CHOOSE TO SOLVE WITH TETRAHEDRA
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      IsPoly=.false.
      Npoly=PolyMesh%num_elem;

!*!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     CHOOSE TO SOLVE WITH POLYHEDRA
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      Npoly=500;
      if (Npoly/=PolyMesh%num_elem) IsPoly=.true.

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     PARTITION OF THE GRID AND GENERATION OF LOCAL CONNECTIVITY
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      call MAKE_PARTITION_AND_MPI_FILES(PolyData,PolyMesh,Npoly)

      print *,'Done reading mesh'

      Np=PolyMesh%Elem_loc(1)%NDof_loc;
      p=PolyMesh%Elem_loc(1)%Degree

      total_dof=PolyMesh%num_poly*Np;
      ndof=PolyMesh%num_poly_loc*Np;

      allocate(petsc_num(total_dof))

      do i=1,total_dof
          petsc_num(i)=i-1
      end do

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     SET PETSC VECTORS AND MATRICES
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      print *,'SET PETSC MATRICES AND VECTORS'

      call SET_PETSC_MATRICES(total_dof,ndof,petsc_stiff,petsc_mass)

      call SET_PETSC_VECTORS(total_dof,ndof,petsc_rhs,petsc_sol)

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     ASSEMBLE MATRICES
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      print *,'ASSEMBLE PETSC MATRIX'

      call MAKE_MATRICES(PolyMesh,petsc_num,total_dof,Np,petsc_stiff,petsc_mass)

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     ASSEMBLE RIGHT HAND SIDE
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      print *,'ASSEMBLE RHS'

      call MAKE_RHS(PolyMesh,petsc_num,total_dof,Np, petsc_rhs)

      call MPI_BARRIER(MPI_COMM_WORLD, mpi_ierr)

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
      !     SETTING SOLVER
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      print *,'SETTING SOLVER'

      call SOLVER_SETTINGS(petsc_stiff,ksp,pc)

```

```fortran
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     DEALLOCATING PETSC MATRICES
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      PetscCallA(MatDestroy(petsc_stiff,mpi_ierr))
      PetscCallA(MatDestroy(petsc_mass,mpi_ierr))

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     CALLING SOLVER
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      print *,'Calling solver: '

      PetscCallA(KSPSolve(ksp, petsc_rhs, petsc_sol, mpi_ierr))

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     DESTROYING KSP SOLVER AND RIGH HAND SIDE
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      PetscCallA(KSPDestroy(ksp, mpi_ierr))
      PetscCallA(VecDestroy(petsc_rhs,mpi_ierr))

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     STORE LOCAL NUMERATION TO RECONSTRUCT THE SOLUTION
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      allocate(nnod_num(ndof))

      call CREATE_LOCAL_NODE_NUM(nnod_num,ndof)

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     SCATTER PETSC SOLUTION AND STORE IN A FORTRAN ARRAY
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      allocate(u_loc(ndof))

      if (IsPoly) then
            print *,'NO SCATTER SOLUTION'
            PetscCallA(VecGetArrayF90(petsc_sol,sol_ptr,mpi_ierr))
            u_loc(1:ndof) = sol_ptr
      else
            PRINT *,'SCATTER SOLUTION'
            call PETSC_SCATTER_VECTOR(ndof,total_dof,nnod_num,petsc_sol,u_loc)
      endif

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     DESTROY PETSC_SOL
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      PetscCallA(VecDestroy(petsc_sol,mpi_ierr))

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     RECONSTRUCT SOLUTION MATRIX FOR POST-PROCESSING
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      ndof_loc=PolyMesh%num_poly_loc
      allocate(u(Np,ndof_loc))
      u=RESHAPE(u_loc, (/Np, ndof_loc /))
      deallocate(u_loc)

      print *,'Done with the solution'

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     POST-PROCESSING: COMPUTING THE ERRORS
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      err_L2=0
      err_DG=0

      call ERRORS(PolyMesh, u,err_L2_mpi, err_DG_mpi)

      call MPI_BARRIER(MPI_COMM_WORLD,mpi_ierr)

      call MPI_REDUCE( err_L2_mpi, err_L2, 1, MPI_DOUBLE_PRECISION, MPI_SUM, &
                        0, MPI_COMM_WORLD, mpi_ierr)
```

```fortran

      call MPI_REDUCE( err_DG_mpi, err_DG, 1,MPI_DOUBLE_PRECISION , MPI_SUM, &
                        0, MPI_COMM_WORLD, mpi_ierr)

      if (mpi_id==0) then
            err_L2 = sqrt(err_L2);
            err_DG = sqrt(err_DG)
            print *,'Done with the errors!'
      endif
      call MPI_BARRIER(MPI_COMM_WORLD,mpi_ierr)

      if (mpi_id==0) then
            print *,'ERROR IN NORM L2: ',err_L2
            print *,'ERROR IN NORM DG: ',err_DG
      endif

      hmax_mpi=compute_hmax(PolyMesh)

      call MPI_REDUCE( hmax_mpi, hmax, 1,MPI_DOUBLE_PRECISION , MPI_MAX, &
                        0, MPI_COMM_WORLD, mpi_ierr)

      call MPI_BARRIER(MPI_COMM_WORLD,mpi_ierr)

      if(mpi_id==0) print *,hmax

      if (mpi_id==0) call WRITE_ERRORS(p,err_DG,err_L2,hmax)

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     POST-PROCESSING:EXPORTING THE SOLUTION
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      call EXPORT_SOLUTION(PolyMesh,u,mpi_id)

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!    END SETUP
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      finish = MPI_WTIME()
      call calc_time(time_hour, time_min, time_sec, int(finish-start))

      if (mpi_id.eq.0) then
         write(*,'(A)')
         write(*,'(A)')'-------------------------------------------------------'
         write(*,'(A,I2,A,I2,A,I2,A)') &
                   'Set-up time = ',time_hour,' h ' ,time_min,' m ' ,time_sec,' s'
         write(*,'(A)')'-------------------------------------------------------'
         write(*,'(A)')
      endif

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
!     FINALIZE MPI AND PETSC
!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      call PetscFinalize(mpi_ierr)
      call MPI_FINALIZE(mpi_ierr)

      end program Lymph3D
```

basis_function.f90

```
371     subroutine basis(phi, dphi, b_box ,Np, blist, Fk, nod3 , nq3)
372     implicit none
373
374     !> This function evaluates the basis functions at the 3D quadrature nodes
375     !  for every polyhedral element.
376
377         integer(kind=4) :: nq3,Np
378         real(kind=8), dimension(3,2) :: b_box
379         real(kind=8),dimension(3,4) :: Fk
380         integer (kind=4), dimension(Np,3) :: blist
381         real(kind=8),dimension(4,nq3) :: nod3
382         real(kind=8), dimension(Np,nq3) :: phi
383         real(kind=8), dimension(3,Np,nq3) :: dphi
384         real(kind=8),dimension(nq3) :: x_p,y_p,z_p
385         real(kind=8),dimension(2) :: intx,inty,intz
386         real(kind=8),dimension(3,nq3) :: pt
387         real(kind=8), dimension(nq3) :: valx,valy,valz
388         real(kind=8), dimension(nq3) :: dvalx,dvaly,dvalz
389
390         integer(kind=4)::j,q,f,l
391
392         do j=1,Np
393             do q=1,nq3
394                 phi(j,q)=0.0
395                 do l=1,3
396                     dphi(l,j,q) = 0.0
397                 end do
398             end do
399         end do
400
401         do j=1,3
402             do q=1,nq3
403                 pt(j,q)=0.0
404                 do l=1,4
405                     pt(j,q) = pt(j,q)+Fk(j,l)*nod3(l,q)
406                 end do
407             end do
408         end do
409
410         do q=1,nq3
411             x_p(q)=pt(1,q)
412             y_p(q)=pt(2,q)
413             z_p(q)=pt(3,q)
414         end do
415
416         do j=1,2
417             intx(j)=b_box(1,j)
418             inty(j)=b_box(2,j)
419             intz(j)=b_box(3,j)
420         end do
421         do f = 1,Np
422             do q=1,nq3
423                 valx=0.0
424                 dvalx(q)=0.0
425                 valy=0.0
426                 dvaly(q)=0.0
427                 valz=0.0
428                 dvalz(q)=0.0
429             end do
430
431             call LegendreP(valx, dvalx, x_p, blist(f,1), intx, nq3)
432             call LegendreP(valy, dvaly, y_p, blist(f,2), inty, nq3)
433             call LegendreP(valz, dvalz, z_p, blist(f,3), intz, nq3)
434             do q=1,nq3
435                 phi(f,q) = valx(q)*valy(q)*valz(q)
436                 dphi(1,f,q) = dvalx(q)*valy(q)*valz(q)
437                 dphi(2,f,q) = valx(q)*dvaly(q)*valz(q)
438                 dphi(3,f,q) = valx(q)*valy(q)*dvalz(q)
439             end do
440         end do
441
442     end subroutine basis
443
444
445     subroutine basis_boundary(phi_b,grad_b,e_E1,E2,b_box1,b_box2,blist,Np, &
446                               Fk, node_maps, nod2, nq2)
447
448         !> This function evaluates for every face the basis functions both of the
449         !  two sharing tratrahedra E1 and E2 at the 2D quadrature nodes.
450
```

```
        integer(kind=4) :: nq2,Np
        real(kind=8), dimension(3,2) :: b_box1
        real(kind=8), dimension(3,2) :: b_box2
        integer(kind=4),dimension(Np,3) :: blist
        real(kind=8), dimension(3,4) :: Fk
        real(kind=8), dimension(4,nq2) :: nod2
        real(kind=8), dimension(4,4,4) :: node_maps
        real(kind=8), dimension(Np,nq2,2) :: phi_b
        real(kind=8), dimension(3,Np,nq2,2) :: grad_b
        real(kind=8), dimension(3,nq2) :: pt
        real(kind=8), dimension(3,4) :: temp

        real(kind=8),dimension(nq2) :: x_p,y_p,z_p
        real(kind=8),dimension(2) :: intx,inty,intz

        real(kind=8), dimension(nq2) :: valx,valy,valz
        real(kind=8), dimension(nq2) :: dvalx,dvaly,dvalz
        integer(kind=4) :: j,q,l,f
        integer(kind=4) :: e_E1,E2

        do j=1,Np
            do q=1,nq2
                phi_b(j,q,1)=0.0
                phi_b(j,q,2)=0.0
                do l=1,3
                    grad_b(l,j,q,1) = 0.0
                    grad_b(l,j,q,2) = 0.0
                end do
            end do
        end do

        do j=1,3
            do q=1,4
                temp(j,q)=0.0
                do l=1,4
                    temp(j,q) = temp(j,q)+Fk(j,l)*node_maps(l,q,e_E1)
                end do
            end do
        end do

        do j=1,3
            do q=1,nq2
                pt(j,q)=0.0
                do l=1,4
                    pt(j,q)=pt(j,q)+temp(j,l)*nod2(l,q)
                enddo
            enddo
        enddo

        do q=1,nq2
            x_p(q)=pt(1,q)
            y_p(q)=pt(2,q)
            z_p(q)=pt(3,q)
        end do

        do j=1,2
            intx(j)=b_box1(1,j)
            inty(j)=b_box1(2,j)
            intz(j)=b_box1(3,j)
        end do

        do f = 1,Np

            do q=1,nq2
                valx=0.0
                dvalx(q)=0.0
                valy=0.0
                dvaly(q)=0.0
                valz=0.0
                dvalz(q)=0.0
            end do

            call LegendreP(valx, dvalx, x_p, blist(f,1), intx, nq2)
            call LegendreP(valy, dvaly, y_p, blist(f,2), inty, nq2)
            call LegendreP(valz, dvalz, z_p, blist(f,3), intz, nq2)

            do q=1,nq2
                phi_b(f,q,1) = valx(q)*valy(q)*valz(q)
                grad_b(1,f,q,1) = dvalx(q)*valy(q)*valz(q)
                grad_b(2,f,q,1) = valx(q)*dvaly(q)*valz(q)
```

```
                grad_b(3,f,q,1) = valx(q)*valy(q)*dvalz(q)
            end do
        end do

        if (E2 .ne. 0) then
            do j=1,2
                intx(j)=b_box2(1,j)
                inty(j)=b_box2(2,j)
                intz(j)=b_box2(3,j)
            end do
            do f = 1,Np
                do q=1,nq2
                    valx=0.0
                    dvalx(q)=0.0
                    valy=0.0
                    dvaly(q)=0.0
                    valz=0.0
                    dvalz(q)=0.0
                end do
                call LegendreP(valx, dvalx, x_p, blist(f,1), intx, nq2)
                call LegendreP(valy, dvaly, y_p, blist(f,2), inty, nq2)
                call LegendreP(valz, dvalz, z_p, blist(f,3), intz, nq2)
                do q=1,nq2
                    phi_b(f,q,2) = valx(q)*valy(q)*valz(q)
                    grad_b(1,f,q,2) = dvalx(q)*valy(q)*valz(q)
                    grad_b(2,f,q,2) = valx(q)*dvaly(q)*valz(q)
                    grad_b(3,f,q,2) = valx(q)*valy(q)*dvalz(q)
                end do

            end do
        end if

    end subroutine basis_boundary
```

MAKE_MATRICES.f90

```
!    Author: Nicoletta De Giosa
!    This file is part of the library LYMPH
!
!> @brief MAKE_MATRICES (Assemble of the petsc stiffness and mass matrices)

subroutine MAKE_MATRICES(PolyMesh,local_petsc_num,Np,total_dof, &
                         petsc_stiff,petsc_mass)
#include<petsc/finclude/petscksp.h>

    use petscksp
    use Poly_mesh
    use mpi
    use Poly_setup_mpi
    use test
    use basis_function
    use assemble_local
    use local_search

    implicit none

    Mat :: petsc_stiff,petsc_mass
    PetscScalar :: val(1),val1(1),val2(1),val3(1),val4(1)
    PetscInt :: irow(1), jcol(1),irow2(1),jcol2(1)

    type(Mesh_Structure) :: PolyMesh

    integer(kind=4) :: nq3,nq2,Np,N,total_dof
    integer(kind=4) :: ie_loc,ie_glob
    integer(kind=4) :: ivert,id_node
    integer(kind=4) :: ipoly_loc,ipoly_glob,ipoly2_loc,ipoly2_glob
    integer(kind=4) :: n_tet_in_poly,iface_poly
    integer(kind=4) :: beg,beg2
    integer(kind=4) :: q,i,j,k,l
    integer(kind=4) :: e,E1,E2
    integer(kind=4), dimension(total_dof) :: local_petsc_num
    integer(kind=4), dimension(:,:), allocatable ::blist
    real(kind=8), dimension(4,4,4) :: node_maps
    real(kind=8), dimension(2,3,4) :: node_maps_inv
    real(kind=8), dimension(:,:), allocatable :: nod3,nod2
    real(kind=8), dimension(:), allocatable :: wei3,wei2
    real(kind=8), dimension(:,:), allocatable :: phi
    real(kind=8), dimension(:,:,:), allocatable :: dphi
    real(kind=8), dimension(:,:,:), allocatable :: phi_b
    real(kind=8), dimension(:,:,:,:), allocatable :: grad_b
    real(kind=8), dimension(3,4) :: Fk
    real(kind=8), dimension(4) :: x,y,z
    real(kind=8), dimension(3,3) :: Jinv
    real(kind=8) :: Jdet

    real(kind=8),dimension(Np,Np) :: V_loc,M_loc
    real(kind=8),dimension(Np,Np) :: S_loc,I_loc,IN_loc,SN_loc

    real(kind=8),dimension(Np,Np) :: temp1
    real(kind=8),dimension(3) :: temp2
    real(kind=8),dimension(Np) :: temp3,temp4

    real(kind=8),dimension(3) :: nn
    real(kind=8) :: theta,sigma,c
    integer(kind=4),dimension(4) :: face_flag

    ! Set the properties
    call set_properties(sigma,theta,c)

    N=PolyMesh%Elem_loc(1)%Degree;

    ! Compute the quadrature nodes in 3D and 2D
    call quadrature(nod2,wei2,nod3,wei3,node_maps,node_maps_inv,N,nq3,nq2)

    allocate (blist(Np,3))
    call basis_list(blist,N,Np)

    allocate(phi(Np,nq3))
    allocate(dphi(3,Np,nq3))
    allocate(phi_b(Np,nq2,2))
    allocate(grad_b(3,Np,nq2,2))

```

```
    ! Begin loop on the tetrahedra
    do ie_loc = 1, PolyMesh%num_elem_loc

        ! Initialize volume matrices

        do i=1,Np
            do  j=1,Np
                V_loc(i,j)=0.0
            enddo
        enddo

        do i=1,Np
            do  j=1,Np
                M_loc(i,j)=0.0
            enddo
        enddo

        ! Compute coordinates of the tetrahedron

        do ivert = 1, PolyMesh%Elem_loc(ie_loc)%num_vert

            call FIND_POS_LOC_NODE(PolyMesh%node_loc2glo,PolyMesh%num_node_loc, &
                                   PolyMesh%Elem_loc(ie_loc)%vert(ivert),id_node)

            x(ivert)=PolyMesh%coord_x(id_node)
            y(ivert)=PolyMesh%coord_y(id_node)
            z(ivert)=PolyMesh%coord_z(id_node)

        enddo

        ! Compute reference map for the tetrahedron
        call jacobians(x , y, z, Fk , Jinv , Jdet)

        ie_glob=PolyMesh%elem_loc2glo(ie_loc)

        ! Find the polyhedron that contains the tetrahedron
        ipoly_glob=PolyMesh%elem_in_poly(ie_glob)

        call GET_EL_LOC_FROM_EL_GLO(PolyMesh%poly_loc2glo, &
                                    PolyMesh%num_poly_loc, &
                                    ipoly_glob,ipoly_loc)

        ! Evaluate basis functions on the edges of the Bounding Box of the polyhedron
        call basis(phi, dphi, PolyMesh%Poly(ipoly_loc)%b_box, Np, blist, &
                  Fk, nod3 , nq3)

        ! Compute the local volume stiffness matrix V_loc
        call MAKE_STIFF_TET_LOC(Np,Jdet,wei3,nq3,dphi,V_loc)

        ! Compute the local volume mass matrix M_loc
        call MAKE_MASS_LOC(Np,Jdet,wei3,nq3,phi,M_loc)

        ! Insert the values of V_loc in the entries of the global stiffness matrix
        beg=(ipoly_glob-1)*Np+1
        do i=1,Np
            do j=1,Np
                val(1)  = V_loc(i,j)
                irow(1) = local_petsc_num(beg+i-1) !(in_petsc)
                jcol(1) = local_petsc_num(beg+j-1) !(ip_petsc)
                if (val(1) .ne. 0) then
                    PetscCall(MatSetValues(petsc_stiff, 1, irow, 1, jcol, val, ADD_VALUES, mpi_ierr))
                endif
            enddo
        enddo

        ! Insert the values of M_loc in the entries of the global mass matrix
        do i=1,Np
            do j=1,Np
                val(1)  = M_loc(i,j)
                irow(1) = local_petsc_num(beg+i-1) !(in_petsc)
                jcol(1) = local_petsc_num(beg+j-1) !(ip_petsc)
                if (val(1) .ne. 0) then
                    PetscCall(MatSetValues(petsc_mass, 1, irow, 1, jcol, val, ADD_VALUES, mpi_ierr))
                endif
            enddo
        enddo

        E1=ie_loc

        ! Begin loop on the faces of the tetrahedron E1
```

```
157
158         do e=1,PolyMesh%Elem_loc(ie_loc)%num_faces
159
160             face_flag(e)=0
161
162             ! Initialize face matrices
163
164             do i=1,Np
165                 do j=1,Np
166                     I_loc(i,j)=0.0
167                     S_loc(i,j)=0.0
168                     IN_loc(i,j)=0.0
169                     SN_loc(i,j)=0.0
170                 enddo
171             enddo
172
173             ! Find neighbouring tetrahedron E2
174             E2=PolyMesh%Elem_loc(ie_loc)%neigh_el(e,2)
175
176             ! If E2 is not on the boundary, find the polyhedron in which
177             ! it is contained
178             if (E2 /=0) then
179                 ipoly2_glob=PolyMesh%elem_in_poly(E2)
180                 call GET_EL_LOC_FROM_EL_GLO(PolyMesh%poly_loc2glo, &
181                             PolyMesh%num_poly_loc, &
182                             ipoly2_glob,ipoly2_loc)
183             endif
184
185             ! If E2 is not on the boundary, check if the two neighbouring
186             ! tetrahedra belong to the same polyhedron
187             if  (E2 /=0) then
188                 if (ipoly_glob == ipoly2_glob) then
189                     face_flag(e)=1
190                 endif
191             end if
192
193             if ( face_flag(e) ==0 ) then
194                 nn(1)=PolyMesh%Elem_loc(ie_loc)%normal(e,1)
195                 nn(2)=PolyMesh%Elem_loc(ie_loc)%normal(e,2)
196                 nn(3)=PolyMesh%Elem_loc(ie_loc)%normal(e,3)
197
198                 ! If the two polyhedra belong to the same processor compute
199                 ! the basis functions on the faces and the local matrices
200                 ! on the faces
201                 ! If not, retrieve bbox of neighbouring element from
202                 ! neigh_bbox and compute the basis functions on the faces
203                 ! and the local matrices on the faces
204
205                 if (ipoly2_loc==0) then
206
207                     n_tet_in_poly=PolyMesh%Poly(ipoly_loc)%num_tet_in_poly
208
209                     do j=1,n_tet_in_poly
210                         if (PolyMesh%Poly(ipoly_loc)%tet_in_poly(j)==ie_glob) then
211                             iface_poly=PolyMesh%Elem_loc(ie_loc)%num_faces*(j-1)+e
212                         endif
213                     enddo
214
215                     call basis_boundary(phi_b,grad_b,e,E2, PolyMesh%Poly(ipoly_loc)%b_box,&
216                                         PolyMesh%Poly(ipoly_loc)%neigh_bbox(iface_poly,:,:),&
217                                         blist, Np, Fk, node_maps, nod2, nq2)
218
219                     call MAKE_STIFF_FACE(theta,sigma,N,Np,E2,PolyMesh%Poly(ipoly_loc)%hk, &
220                                         PolyMesh%Poly(ipoly_loc)%neigh_hk(iface_poly), nn, &
221                                         PolyMesh%Elem_loc(ie_loc)%area(e),wei2,nq2,phi_b,&
222                                         grad_b,S_loc,I_loc,IN_loc,SN_loc)
223                 else
224                     call basis_boundary(phi_b,grad_b,e,E2,PolyMesh%Poly(ipoly_loc)%b_box,&
225                                             PolyMesh%Poly(ipoly2_loc)%b_box,blist, Np, &
226                                             Fk, node_maps, nod2, nq2)
227
228                     call MAKE_STIFF_FACE(theta,sigma,N,Np,E2,PolyMesh%Poly(ipoly_loc)%hk, &
229                                         PolyMesh%Poly(ipoly2_loc)%hk, nn, &
230                                         PolyMesh%Elem_loc(ie_loc)%area(e),wei2,nq2,phi_b,&
231                                         grad_b,S_loc,I_loc,IN_loc,SN_loc)
232
233                 endif
234             endif
235
236             ! Insert the values of S_loc in the entries
```

```
            ! of the global stiffness matrix
            do i=1,Np
                do j=1,Np
                    val(1) = S_loc(i,j)
                    irow(1) = local_petsc_num(beg+i-1) !(in_petsc)
                    jcol(1) = local_petsc_num(beg+j-1) !(ip_petsc)
                    if (val(1) .ne. 0.0) then
                        PetscCall(MatSetValues(petsc_stiff, 1, irow, 1, jcol, val, ADD_VALUES, mpi_ierr))
                    endif
                enddo
            enddo

            ! Insert the values of I_loc,SN_loc and IN_loc in the entries
            ! of the global stiffness matrix
            if (E2==0) then
                do i=1,Np
                    do j=1,Np
                        val(1) = I_loc(i,j)
                        irow(1) = local_petsc_num(beg+i-1) !(in_petsc)
                        jcol(1) = local_petsc_num(beg+j-1) !(ip_petsc)
                        if (val(1) .ne. 0.0) then
                            PetscCall(MatSetValues(petsc_stiff, 1, irow, 1, jcol, val, ADD_VALUES, mpi_ierr))
                        endif
                        if (val(1) .ne. 0.0) then
                            PetscCall(MatSetValues(petsc_stiff, 1, jcol, 1, irow, val, ADD_VALUES, mpi_ierr))
                        endif
                    enddo
                enddo
            else
                beg2 = (ipoly2_glob-1)*Np+1
                do i=1,Np
                    do j=1,Np
                        val1(1) = I_loc(i,j)
                        val2(1) = IN_loc(i,j)
                        val3(1) = SN_loc(i,j)
                        irow(1) = local_petsc_num(beg+i-1) !(in_petsc)
                        jcol(1) = local_petsc_num(beg+j-1) !(ip_petsc)
                        irow2(1) = local_petsc_num(beg2+i-1) !(in_petscc)
                        jcol2(1) = local_petsc_num(beg2+j-1) !(ip_petsc)

                        if (val1(1) .ne. 0) then
                            PetscCall(MatSetValues(petsc_stiff, 1, irow, 1, jcol, val1, ADD_VALUES, mpi_ierr))
                        endif
                        if (val1(1) .ne. 0) then
                            PetscCall(MatSetValues(petsc_stiff, 1, jcol, 1, irow, val1, ADD_VALUES, mpi_ierr))
                        endif
                        if (val2(1) .ne. 0) then
                            PetscCall(MatSetValues(petsc_stiff, 1, irow, 1, jcol2, val2, ADD_VALUES, mpi_ierr))
                        endif
                        if (val2(1) .ne. 0) then
                            PetscCall(MatSetValues(petsc_stiff, 1, jcol2, 1, irow, val2, ADD_VALUES, mpi_ierr))
                        endif
                        if (val3(1) .ne. 0) then
                            PetscCall(MatSetValues(petsc_stiff, 1, irow, 1, jcol2, val3, ADD_VALUES, mpi_ierr))
                        endif
                    enddo
                enddo

            endif
        enddo
    enddo

    call MPI_BARRIER(MPI_COMM_WORLD, mpi_ierr)

    ! Assembly of petsc stiffness matrix and mass petsc matrix
    PetscCall(MatAssemblyBegin(petsc_stiff, MAT_FINAL_ASSEMBLY, mpi_ierr))
    PetscCall(MatAssemblyEnd(petsc_stiff, MAT_FINAL_ASSEMBLY, mpi_ierr))

    PetscCall(MatAssemblyBegin(petsc_mass, MAT_FINAL_ASSEMBLY, mpi_ierr))
    PetscCall(MatAssemblyEnd(petsc_mass, MAT_FINAL_ASSEMBLY, mpi_ierr))

    ! Compute the total matrix (A+cM)
    PetscCall(MatAXPY(petsc_stiff,c,petsc_mass, DIFFERENT_NONZERO_PATTERN,mpi_ierr))

    deallocate(phi)
    deallocate(dphi,phi_b)
    deallocate(grad_b)

end subroutine MAKE_MATRICES
```

MAKE_RHS.f90

```
!    Author: Nicoletta De Giosa
!    This file is part of the library LYMPH
!
!> @brief MAKE_RHS (Assemble of the petsc right hand side)

subroutine MAKE_RHS(PolyMesh,local_petsc_num,Np,total_dof,petsc_rhs)

#include<petsc/finclude/petscksp.h>

    use petscksp
    use Poly_mesh
    use mpi
    use Poly_setup_mpi
    use test
    use basis_function
    use assemble_local
    use local_search

    implicit none

    Vec petsc_rhs
    PetscScalar :: val(1)
    PetscInt :: irow(1)

    type(Mesh_Structure) :: PolyMesh

    integer(kind=4) :: nq3,nq2,Np,N,total_dof
    integer(kind=4) :: ie_loc,ie_glob,ivert,id_node,ipoly_glob,ipoly_loc,ipoly2_glob,ipoly2_loc
    integer(kind=4) :: n_tet_in_poly,iface_poly
    integer(kind=4) :: e,E1,E2
    integer(kind=4) :: q,i,j,k,t,l
    integer(kind=4) :: beg,beg1
    integer(kind=4), dimension(4) :: face_flag
    integer(kind=4),dimension(total_dof) :: local_petsc_num
    integer(kind=4),dimension(:,:), allocatable::blist
    real(kind=8),dimension(4,4,4) :: node_maps
    real(kind=8),dimension(2,3,4) :: node_maps_inv
    real(kind=8),dimension(:,:), allocatable :: nod3,nod2
    real(kind=8),dimension(:), allocatable :: wei3,wei2
    real(kind=8), dimension(:,:), allocatable :: phi
    real(kind=8), dimension(:,:,:), allocatable :: dphi
    real(kind=8), dimension(:,:,:), allocatable :: phi_b
    real(kind=8), dimension(:,:,:,:), allocatable :: grad_b
    real(kind=8), dimension(3,4) :: Fk
    real(kind=8), dimension(4) :: x,y,z
    real(kind=8),dimension(3,3) :: Jinv
    real(kind=8),dimension(3) :: temp2
    real(kind=8),dimension(Np) :: temp3
    real(kind=8),dimension(3) :: nn
    real(kind=8),dimension(Np) :: rhs_tet_loc
    real(kind=8),dimension(Np) :: rhs_face_bd_loc
    real(kind=8) :: Jdet
    real(kind=8) :: theta,sigma,c

    ! Set the properties
    call set_properties(sigma,theta,c)

    N=PolyMesh%Elem_loc(1)%Degree;

    ! Compute the quadrature nodes in 3D and 2D
    call quadrature(nod2,wei2,nod3,wei3,node_maps,node_maps_inv,N,nq3,nq2)

    allocate (blist(Np,3))
    call basis_list(blist,N,Np)

    allocate(phi(Np,nq3))
    allocate(dphi(3,Np,nq3))
    allocate(phi_b(Np,nq2,2))
    allocate(grad_b(3,Np,nq2,2))

    ! Begin loop on the tetrahedra

    do ie_loc = 1, PolyMesh%num_elem_loc

        ! Initialize rhs term on the volume

```

```
        do i=1,Np
            rhs_tet_loc(i)=0.0;
        enddo

        ! Compute coordinates of the tetrahedron

        do ivert = 1, PolyMesh%Elem_loc(ie_loc)%num_vert

            call FIND_POS_LOC_NODE(PolyMesh%node_loc2glo,PolyMesh%num_node_loc, &
                                PolyMesh%Elem_loc(ie_loc)%vert(ivert),id_node)

            x(ivert)=PolyMesh%coord_x(id_node)
            y(ivert)=PolyMesh%coord_y(id_node)
            z(ivert)=PolyMesh%coord_z(id_node)

        enddo

        ! Compute reference map for the tetrahedron
        call jacobians(x , y, z, Fk , Jinv , Jdet)

        ie_glob=PolyMesh%elem_loc2glo(ie_loc)

        ! Find the polyhedron that contains the tetrahedron
        ipoly_glob=PolyMesh%elem_in_poly(ie_glob)

        call GET_EL_LOC_FROM_EL_GLO(PolyMesh%poly_loc2glo, &
                            PolyMesh%num_poly_loc, &
                            ipoly_glob,ipoly_loc)

        ! Evaluate basis functions on the edges of the
        !Bounding Box of the polyhedron
        call basis(phi, dphi, PolyMesh%Poly(ipoly_loc)%b_box,&
                   Np, blist, Fk, nod3 , nq3)

        ! Compute the local volume rhs term rhs_tet_loc
        call MAKE_RHS_TET(Np,Fk,Jdet,nod3,wei3,nq3,phi,rhs_tet_loc)

        ! Insert the values of rhs_tet_loc in the entries of the
        ! global right hand side vector petsc_rhs

        beg=(ipoly_glob-1)*Np+1
        do i=1,Np
            val(1)  = rhs_tet_loc(i)
            irow(1) = local_petsc_num(beg+i-1) !(in_petsc)
            if (val(1) .ne. 0.0) then
                PetscCall(VecSetValues(petsc_rhs, 1, irow, val, ADD_VALUES, mpi_ierr))
            endif
        enddo

        E1=ie_loc

        ! Begin loop on the faces of the tetrahedron E1

        do e=1,PolyMesh%Elem_loc(ie_loc)%num_faces
            face_flag(e)=0;

            ! Initialize face right hand side term

            do i=1,Np
                rhs_face_bd_loc(i)=0.0;
            enddo

            ! Find neighbouring tetrahedron E2
            E2=PolyMesh%Elem_loc(ie_loc)%neigh_el(e,2)

            ! If E2 is not on the boundary, find the
            ! polyhedron in which it is contained
            if (E2 /=0) then
                ipoly2_glob=PolyMesh%elem_in_poly(E2);
                call GET_EL_LOC_FROM_EL_GLO(PolyMesh%poly_loc2glo, &
                            PolyMesh%num_poly_loc, &
                            ipoly2_glob,ipoly2_loc)
            endif

            ! If E2 is not on the boundary, check if the two neighbouring
            ! tetrahedra belong to the same polyhedron
            if  (E2 /=0) then
                if (ipoly_glob == ipoly2_glob) then
                    face_flag(e)=1;
                endif
```

```
157             end if
158
159             if ( face_flag(e) == 0 ) then
160
161                 nn(1)=PolyMesh%Elem_loc(ie_loc)%normal(e,1)
162                 nn(2)=PolyMesh%Elem_loc(ie_loc)%normal(e,2)
163                 nn(3)=PolyMesh%Elem_loc(ie_loc)%normal(e,3)
164
165                 ! If the two polyhedra belong to the same processor
166                 ! compute the basis functions on the faces
167                 ! and the local right hand side term on the faces
168                 ! If not, retrieve bbox of neighbouring element from
169                 ! neigh_bbox and compute the basis functions on the
170                 ! faces and the local right hand side term on the faces
171
172                 if (ipoly2_loc == 0) then
173
174                     n_tet_in_poly=PolyMesh%Poly(ipoly_loc)%num_tet_in_poly
175
176                     do j=1,n_tet_in_poly
177                         if (PolyMesh%Poly(ipoly_loc)%tet_in_poly(j)==ie_glob) then
178                             iface_poly=PolyMesh%Elem_loc(ie_loc)%num_faces*(j-1)+e
179                         endif
180                     enddo
181
182                     call basis_boundary(phi_b,grad_b,e, E2, &
183                                         PolyMesh%Poly(ipoly_loc)%b_box,&
184                                         PolyMesh%Poly(ipoly_loc)%neigh_bbox(iface_poly,:,:),&
185                                         blist,Np, Fk, node_maps, nod2, nq2)
186
187                     call MAKE_RHS_FACE(theta,sigma,N,Np,e,E2,&
188                                         PolyMesh%Poly(ipoly_loc)%hk,&
189                                         PolyMesh%Poly(ipoly_loc)%neigh_hk(iface_poly),&
190                                         nn,PolyMesh%Elem_loc(ie_loc)%area(e),Fk,&
191                                         nod2,wei2,nq2,node_maps,phi_b,grad_b,rhs_face_bd_loc)
192
193                 else
194                     call basis_boundary(phi_b,grad_b,e, E2, &
195                                         PolyMesh%Poly(ipoly_loc)%b_box,&
196                                         PolyMesh%Poly(ipoly2_loc)%b_box, &
197                                         blist, Np, Fk, node_maps, nod2, nq2)
198
199                     call MAKE_RHS_FACE(theta,sigma,N,Np,e,E2,&
200                                         PolyMesh%Poly(ipoly_loc)%hk,&
201                                         PolyMesh%Poly(ipoly2_loc)%hk,&
202                                         nn,PolyMesh%Elem_loc(ie_loc)%area(e),&
203                                         Fk,nod2,wei2,nq2,node_maps,phi_b,grad_b,rhs_face_bd_loc)
204                 endif
205
206                 ! If the neighbouring tetrahedra is on the boundary,
207                 ! insert the values of rhs_face_bd_loc in the entries
208                 ! of the global rhs vector
209                 if (E2==0) then
210                     do i=1,Np
211                         val(1)  = rhs_face_bd_loc(i)
212                         irow(1) = local_petsc_num(beg+i-1) !(in_petsc)
213                         if (val(1) .ne. 0.0) then
214                             PetscCall(VecSetValues(petsc_rhs, 1, irow, val, ADD_VALUES, mpi_ierr))
215                         endif
216                     enddo
217                 endif
218
219             endif
220         enddo
221     enddo
222
223     call MPI_BARRIER(MPI_COMM_WORLD, mpi_ierr)
224
225     ! Aseembly petsc rhs
226     PetscCall(VecAssemblyBegin(petsc_rhs,mpi_ierr))
227     PetscCall(VecAssemblyEnd(petsc_rhs,mpi_ierr))
228
229     end subroutine MAKE_RHS
```

test.f90

```
module test
    implicit none
    contains

    ! definition of the forcing term f

    function f(p)result(r)
        implicit none
            real(kind=8) r
            real(kind=8),dimension(3) :: p
            real(kind=8) :: c,sigma,theta

            call set_properties(sigma,theta,c)

            r = -EXP(p(1)*p(2)*p(3))*((p(1)*p(2))**2+(p(1)*p(3))**2+ &
                (p(2)*p(3))**2-c)
            !r = -EXP(p(1)*p(2)*p(3))*((p(1)*p(2))**2+(p(1)*p(3))**2+ &
            !    (p(2)*p(3))**2)
            !r = EXP( - (p(1)*p(1))**2 - (p(2)*p(2))**2 - (p(3)*p(3))**2)

    end function f

    ! Definition of the Dirichlet boundary condition
    function gd(p)result(r)
    implicit none
        real(kind=8) r
        real(kind=8),dimension(3) :: p
        r = EXP(p(1)*p(2)*p(3))
        !r=0*EXP(p(1)*p(2)*p(3));
    end function gd

    ! Definition of the analytical solution, if available
    ! (to compute the errors)
    function uex(p)result(r)
        real(kind=8) r
        real(kind=8),dimension(3) :: p
        r = EXP(p(1)*p(2)*p(3))
    end function uex


    ! Definition of the gradient of the analytical solution, if available
    ! (to compute the errors)

    function uex_grad(p)result(r)
        real(kind=8),dimension(3) :: r
        real(kind=8),dimension(3) :: p
        real(kind=8) :: u
        u=uex(p)
        r(1) = u * p(2) * p(3)
        r(2) = u * p(1) * p(3)
        r(3) = u * p(1) * p(2)

    end function uex_grad

    ! Set the properties of the numerical method
    subroutine set_properties(sigma,theta,c)
        implicit none
        real(kind=8) :: sigma,theta
        real(kind=8) :: c

        sigma = 10 ! penalty coefficient
        theta = -1 ! IP method
        c=0.5; ! reacgtion coefficient

    end subroutine set_properties

end module test
```

List of Figures

List of Tables