E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Integration of the meshing tool GMSH with Matlab/Octave for the resolution of 3D Boundary Value Problems with simplicial finite elements

Bachelor's Degree in
Industrial Engineering

Bachelor's Thesis in Industrial Engineering

Alejandro Duque Salazar
Director: Víctor Domínguez Báguena
Pamplona, June 2023

upna

Escuela Técnica Superior de Ingeniería Industrial, Informática y de Telecomunicación

Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Bachelor's thesis in Industrial Engineering

Integration of the meshing tool GMSH with Matlab/Octave for the resolution of 3D Boundary Value Problems with simplicial finite elements

Alejandro Duque Salazar

Director: Víctor Domínguez Báguena

Pamplona, June 2023

Abstract

This work presents an efficient and documented 3D Finite Element Method (FEM) code with tetrahedral elements up to fourth degree in Matlab/Octave. The code is designed to solve boundary problems with Dirichlet, Neumann, and Robin conditions for linear elliptic partial differential equations. We also provide tools for integrating with the meshing software GMSH. We demonstrate the flexibility and extensibility of the code with several numerical experiments. These examples are also presented for tutorial purposes. Furthermore, we provide a basic background on the finite element method, mainly focusing on the ideas necessary for its implementation to supplement this project's coding nature.

Resumen

En este trabajo se presente un código eficiente y documentado del Método de Elementos Finitos (MEF) en tres dimensiones, con elementos tetrahédricos de hasta grado cuatro cuyo uso se extiende para Octave y Matlab. Este código se ha diseñado para resolver problemas de contorno con condiciones Dirichlet, Neumann y Robin para ecuaciones en derivadas parciales lineares y elípticas. También desarrollamos una herramienta para integrar este código con el programa informático GMSH. Demostramos la flexibilidad y extensibilidad del código a través de varios ejemplos numéricos. Estos ejemplos también servirán a modo de tutorial para comprender el funcionamiento del código. Además, se proporciona una introducción del MEF enfocándonos en las ideas necesarias para su implementación.

Agradecimientos

Quiero expresar mi sincero agradecimiento a todos aquellos que me han apoyado de alguna forma a lo largo de mi trayecto en este grado y han contribuido a la culminación de este Trabajo de Fin de Grado.

En primer lugar, deseo agradecer a mi tutor, Víctor, quien con su excentricidad hemos pasado momentos realmente divertidos durante la realización de este proyecto. Su invaluable ayuda ha sido fundamental en su ejecución y siempre ha estado disponible para cualquier cosa, incluso por muchas horas viendo códigos sin sentido, en las que llegaba un punto en el que el sentido común ya se iba de vacaciones.

También debo agradecer a mi familia, en particular a Ángela y Franqui, con quienes he compartido mi estancia en España y han estado siempre para mí. Valeria, Jean-Philippe y mi mamá, quienes me han brindado un apoyo inmenso, y no nos olvidemos de Edith.

A Ana e Íñigo, con los que he podido compartir un rato en el laboratorio de Matemáticas Aplicadas. Ambos tienen una gran capacidad de trabajo que les llevará lejos.

A mis amigos: Byron, Bryan (o mejor conocidos como Brion y Brian), Elvis, Guiland, Michele y Sergio. Gracias por ser como son y comprenderme a pesar de mis defectos y fortalezas.

Por último, quiero expresar mi aprecio a los profesores que me han impartido clases durante estos años y a quienes siempre recordaré con cariño: Víctor, apareces una vez más en esta lista, pero es que eres un personaje, tu sentido del humor es único, me encanta. Blanca, gracias por ayudarme y confiar en mí, tu alma de poeta y de espíritu libre ha creado frases míticas como: “hay que escuchar a las ecuaciones”. Una gran frase que recordaré con muchas risas y que espero que no me pase, porque eso significaría que he perdido la poca cordura que me queda. Eugenio, por tu gran pasión y dedicación a la educación. A cada uno de ustedes, solo tengo palabras de elogio.

Contents

Nomenclature

α	Robin coefficient
$\boldsymbol{\lambda}$	Barycentric coordinates in a $n-$simplex
$\boldsymbol{\beta}$	Advection vector
$\boldsymbol{A}^K_{\boldsymbol{\beta},x_i}$	Term of the stiffness matrix associated to coordinates x_i
$\boldsymbol{A}_{\boldsymbol{\beta}}$	Advection matrix
$\boldsymbol{A}^K_{\boldsymbol{\beta}}$	Local advection matrix
$\boldsymbol{b}_f$	Source term
$\boldsymbol{b}^K_f$	Local source term
$\boldsymbol{i}^A$	Vector of global indices of the nodes inside A
$\boldsymbol{i}^K$	Vector of global indices of the nodes inside K
$\boldsymbol{J}_K$	Jacobian matrix of F_K
$\boldsymbol{M}_c$	Mass Matrix
$\boldsymbol{M}^K_c$	Local mass Matrix
$\boldsymbol{R}_\alpha$	Boundary mass matrix
$\boldsymbol{R}^K_\alpha$	Local boundary mass matrix
$\boldsymbol{S}^K_{\underline{\underline{\boldsymbol{\kappa}}},x_ix_j}$	Term of the stiffness matrix associated to coordinates x_i and x_j
$\boldsymbol{S}^K_{\underline{\underline{\boldsymbol{\kappa}}}}$	Local stiffness matrix
$\boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}}}$	Stiffness matrix
$\boldsymbol{t}_g$	Local traction vector
$\boldsymbol{v}^K_\ell$	Node with local index ℓ inside the tetrahedron K
$\boldsymbol{v}_j$	Node in the mesh with global index j
Γ	Boundary of Ω
λ, G	Lamé parameters
$\mathbb{P}_m$	Set of all polynomial with degree less than or equal to m
$\mathcal{T}_h$	Set of tetrahedra
Ω	Domain of study
$\underline{\underline{\boldsymbol{\kappa}}}$	Diffusion matrix
$\underline{\underline{\boldsymbol{\sigma}}}$	Stress tensor
$\underline{\underline{\boldsymbol{\varepsilon}}}$	Strain tensor

φ_j	Basis function of P_h^m associated to node $\boldsymbol{v}_j$
$\widehat{\boldsymbol{n}}$	Unit vector normal to Γ
$\widehat{A}$	Reference triangle
$\widehat{K}$	Reference tetrahedron
$\widehat{N}_\ell$	Lagrange basis inside the reference element associated to node $\widehat{\boldsymbol{v}}_\ell$
A	Triangular face of a tetrahedron
c	Reaction function
f	Source function
F_A	Surface parameterization of A with $\widehat{A}$
F_K	Affine mapping between $\widehat{K}$ and K
g_N	Neumann data
K	Tetrahedron
N_ℓ^K	Lagrange basis inside the tetrahedron K associated to node $\boldsymbol{v}_\ell^K$
P_h^m	Set of continuous piecewise polynomials in Ω
u_D	Dirichlet data
`dofA`	Degrees of freedom inside the face A
`dofK`	Degrees of freedom inside the element K
`nttrh`	Number of tetrahedrons in the mesh

Chapter 1

Introduction

The study of differential equations holds immense importance in the fields of engineering and sciences as a whole. However, except for specific cases with theoretical interest, the majority of these equations cannot be solved analytically. As a result, numerical methods become necessary to approximate their solutions.

In order to properly define a differential problem, it is crucial to establish appropriate conditions. In engineering and sciences, it is common to encounter problems where boundary conditions are needed for solving the differential equations. By incorporating appropriate boundary conditions, numerical methods can provide valuable approximations to solve differential equations, enabling engineers and scientists to gain insights into complex physical phenomena that cannot be tackled analytically. Moreover, one of the significant advantages of using numerical methods in engineering is that it allows predictions of system behavior without additional investments. This makes numerical methods a cost-effective and reliable approach to make accurate predictions with a certain degree of accuracy.

Usual methods used to approximate these problems include Finite Differences, Finite Element Method, Spectral Methods, Finite Volume Method and adaptations of them. Each method has its own advantages and disadvantages, and the appropriate choice depends highly on the application at hand. Among these methods, the Finite Element Method has emerged as one of the most versatile approaches due to its capability to model problems involving complex geometries accurately and with ease.

In the field of engineering, there are numerous specialized software packages available that facilitate the computation required to solve specific problems. Initially, these software packages were only capable of solving a limited range of problems. However, with the significant increase in computational power over the last few decades, it has become more common to have software programs that are capable of solving a wide variety of different problems. The availability of such software has significantly reduced the computational burden on engineers, allowing them to focus more on the interpretation of results and the optimization of designs.

One drawback of these software packages is their initial cost. They often come with licensing fees that may be affordable for large institutions, but can pose a financial challenge for smaller ones. However, once acquired, these software packages result in a useful tool for modelling physical problems. Additionally, with sufficient experience, users can better determine the most suitable approach to solving a specific problem.

For instance, problems exhibiting symmetry can be simplified by using one-dimensional elements. While this may seem like an oversimplification, it can be adequate for simple geometries, and the computational cost is significantly reduced as the problem can be easily resolved. On the other hand, if more detailed information is required, two-dimensional elements can be employed. Although this increases the computational cost, it allows for modeling geometry

details that were not accounted in the previous case. Thus, the increased computational cost is accompanied by a more accurate solution.

However, for geometries with greater complexity, which are typically the most common cases (such as automotive components, biomedical devices, aerospace components, and others), three-dimensional elements are necessary to obtain an accurate solution. This inevitably leads to a significant increase in computational cost, but it allows for the modeling of problems that were not possible with one or two-dimensional elements. Among the commonly used elements for such problems are tetrahedral elements, as they can approximate complex geometries really well.

Even though numerical methods provide solutions, it is still crucial for users to have an understanding of the advantages and limitations of these methods, to assess whether the obtained solution is satisfactory. By combining expertise in numerical methods and knowledge of the physics involved, engineers and scientists can ensure that the numerical solution is meaningful and aligns with the expected behavior of the system of study.

For this reason, the objective of this project is to develop suitable code for solving boundary value problems using the Finite Element Method. The aim is to gain a deep understanding of the underlying principles of this method and provide code that is freely accessible for anyone to use without requiring a substantial investment and without the need of computers with high specifications. To aid us in the meshing process, we have chosen to utilize the software GMSH.

GMSH is an open-source finite element meshing software designed for numerical simulations. It offers both pre-processing and post-processing capabilities, making it suitable for a wide range of problem-solving tasks. The software provides a user-friendly graphical user interface (GUI) for creating and editing mesh geometries. Additionally, it has its own scripting language that enables the automation of various processes. This software supports a diverse range of mesh elements in multiple dimensions, including line elements for one-dimensional geometries, quadrangular and triangular elements for two-dimensional geometries, and tetrahedral and hexahedral elements for three-dimensional geometries. Furthermore, GMSH allows for the export of meshes to popular simulation software such as OpenFOAM and ANSYS.

One limitation of GMSH is that it is not extensively documented compared to other software tools. Due to its specialized nature, the availability of online tutorials and resources for learning GMSH is more limited. This can pose challenges for new users seeking assistance in utilizing the software. However, there are some resources available, such as the tutorial provided in [14] and the documentation section within GMSH itself [8], which offer step-by-step instructions using the GUI and the scripting language for geometry generation and finite element problems resolution.

Mastery of the scripting language is essential for those seeking to utilize the software to its full capacity. However, the generation of simple geometries is doable from the above references, which provides enough tools for their construction. Furthermore, for geometries with varying degrees of complexity, there is always the use of available geometries online that can be downloaded, or alternatively, one can employ FreeCAD to construct them. This software employs the OpenCASCADE geometry kernel, enabling the creation and modification of complex geometries through modules or sketches. Both GMSH and FreeCAD utilize the same geometry kernel, facilitating the export of files between the two software programs. For example, geometries created in FreeCAD can be exported in the `.step` format.

Considering the challenges associated with the less user-friendly scripting language and the lack of comprehensive tutorials in the use of solving problems using this software, we have decided to work with a programming software that is more commonly used. Therefore, for

this project, we have chosen to work with Matlab, and to ensure accessibility, we have also considered the use of Octave, an open-source alternative to Matlab. In our implementation, we will focus on tetrahedral elements with equispaced nodes, as they offer a straightforward approach to grasp the fundamental concepts of the Finite Element Method.

In the second chapter, we will provide an introduction to the fundamental concepts of the Finite Element Method. This will include an overview of the basis functions utilized for approximating the solution of the problem, as well as the formulation of the variational problem and its discretization and finally the assembly of the linear system. Furthermore, we will show some examples on how to extend these ideas to solve a larger variety of problems, showcasing the versatility of the method.

On to the third chapter, we will move on to the finite element package `FEM3D_GMSH` that has been developed specifically for exporting mesh files from GMSH and how to use them to solve FEM problems in Matlab or Octave. In this chapter, we will give a throughout explanation of the code, providing interested users with a comprehensive understanding of its inner workings and functionality.

Lastly, the final chapter will feature a collection of numerical examples that highlight the practical application of the developed package in solving *real* problems. Through these examples, we will showcase how the package can be effectively employed to obtain accurate and insightful solutions.

Chapter 2

Finite Element Method

2.1 The Boundary Value Problem

Let Ω be a compact polygonal domain in $\mathbb{R}^3$, see for example Figure 2.1, denote by Γ its boundary which is split into three disjoints parts (one or two of them could be void): $\Gamma = \Gamma_D \cup \Gamma_N \cup \Gamma_R$.

In Ω we consider the following boundary value problem: for given functions $f : \Omega \to \mathbb{R}$, $u_D : \Gamma_D \to \mathbb{R}$ (Dirichlet data), $g_N : \Gamma_N \to \mathbb{R}$ (Neumann data) and $g_R : \Gamma_R \to \mathbb{R}$ (Robin data), find $u : \Omega \to \mathbb{R}$ so that

$$\begin{cases} -\nabla \cdot \left(\underline{\underline{\boldsymbol{\kappa}}} \nabla u\right) + \boldsymbol{\beta} \cdot \nabla u + cu &= f, & \text{in } \Omega, \\ u &= u_D, & \text{on } \Gamma_D, \\ \left(\underline{\underline{\boldsymbol{\kappa}}} \nabla u\right) \cdot \widehat{\boldsymbol{n}} &= g_N, & \text{on } \Gamma_N, \\ \left(\underline{\underline{\boldsymbol{\kappa}}} \nabla u\right) \cdot \widehat{\boldsymbol{n}} + \alpha u &= g_R, & \text{on } \Gamma_R. \end{cases} \tag{2.1}$$

Here $c : \Omega \to \mathbb{R}$ is a reaction function, $\alpha : \Omega \to \mathbb{R}$ is the Robin coefficient, $\boldsymbol{\beta} : \Omega \to \mathbb{R}^3$ is a velocity field and $\underline{\underline{\boldsymbol{\kappa}}} : \Omega \to \mathbb{R}^{3\times 3}$ is the diffusion term, a symmetric positive-definite matrix function. The unit outward normal vector to Γ is denoted with $\widehat{\boldsymbol{n}}$. On the other hand, "∇" is the gradient operator and "$\nabla\cdot$" the divergence of a vector function.

The partial differential equation in (2.1) is an elliptic linear differential equation [9] known as the advection-diffusion-reaction equation in stationary state. It has applications in various branches of science and engineering, such as electrostatics, fluid mechanics, and solid material conservation of heat.

For example, in the special case $\underline{\underline{\boldsymbol{\kappa}}}$ is the identity matrix we obtain the Poisson Equation $-\Delta u = f$ where $\Delta = \nabla \cdot (\nabla)$ is the Laplacian operator. In electrostatics, this equation models the electric potential resulting from the charge density and the electric conductivity of the medium. In fluid mechanics, the homogeneous Poisson equation ($f = 0$), also known as the Laplace Equation [25], describes an incompressible and irrotational fluid in terms of a potential function u that satisfies the equation $-\Delta u = 0$.

The conservation of energy principle in an anisotropic solid material gives rise to the equation $\frac{\partial u}{\partial t} - \nabla \cdot (\underline{\underline{\boldsymbol{\kappa}}} \nabla u) = f$, where u represents temperature. Here, $\underline{\underline{\boldsymbol{\kappa}}}$ is a symmetric positive definite tensor of thermal conductivities, a property which measures the material ability to conduct heat along different directions. In many practical applications, it is common for systems to reach a permanent state rapidly, making it more interesting to study the final behavior rather than considering the entire transient term. In the case of a homogeneous and isotropic material, the thermal conductivity remains constant, resulting in the Poisson equation [27]. Although it is often assumed that heat generation is negligible, there are cases where non-zero heat generation terms must be considered. Examples of such cases include nuclear fuels, solids with

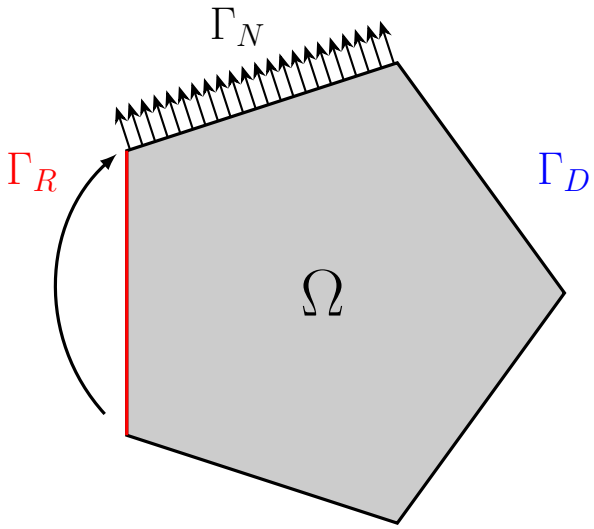


Figure 2.1: 2D equivalent domain representation of the problem (2.1) and its boundaries.

electrical currents passing through them, and materials subjected to radiation. In the latter scenario, the absorbed portion can be modeled as internal heat generation. For instance, in [1], the authors incorporated this effect when modeling the solid layers of photovoltaic panels.

The transport equation $\frac{\partial u}{\partial t} - \nabla \cdot (\underline{\underline{\boldsymbol{\kappa}}} \nabla u) + \boldsymbol{\beta} \cdot \nabla u = 0$ is another example of an elliptic linear differential equation that models the concentration of a substance along the domain Ω as it is advected by a fluid with velocity $\boldsymbol{\beta}$ and diffused due to concentration gradients [23]. From a mathematical perspective, this velocity vector can be any function that is sufficiently regular in Ω. However, in practice, this expression is valid only for incompressible fluids, that is, fluids whose velocity field satisfies $\nabla \cdot \boldsymbol{\beta} = 0$.

The expression $-\nabla \cdot (\underline{\underline{\boldsymbol{\kappa}}} \nabla u) + \omega^2 u = 0$, is usually called the Helmholtz equation, here ω^2 is the frequency of harmonic solutions to the problem. This equation models various phenomena, including acoustics wave scattering and electromagnetism fields [16, 17, 20]. Also in some cases, a reaction term may appear in chemical and biological processes [11].

2.2 Finite elements

We introduce in this section the finite elements space we will consider in this work.

2.2.1 The mesh

The domain discretization is a crucial step in the finite element method. This procedure involves splitting the original domain into smaller disjoints subdomains called elements:

$$\overline{\Omega} = \bigcup_{K_j \in \mathcal{T}_h} K_j, \quad K_i \cap K_j = \emptyset \text{ if } i \neq j.$$

For our purposes, we will use tetrahedra (3-simplices) for the discretization process. Some authors refer to this as the "tetrahedralization of the domain". The set of all tetrahedra in the resulting mesh is denoted by $\mathcal{T}_h = \{K_j\}_{j=1}^{\texttt{nttrh}}$, where `nttrh` is the number of tetrahedral subdomains. In the mesh, the tetrahedra are assumed to be non-degenerate, that is, they have non-zero volume. Additionally, any pair of tetrahedra must share at most a common vertex, an entire face, or an entire edge; therefore, we are considering conformal meshes.

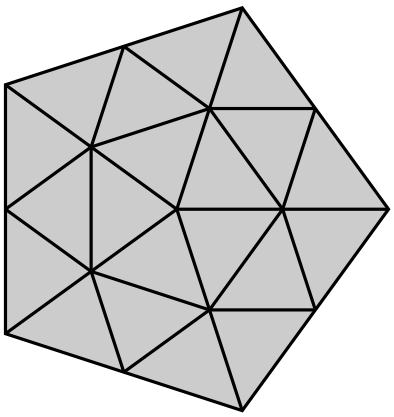


Figure 2.2: Conformal mesh of the domain.

If an element has one face lying on the boundary Γ, that face is entirely contained in Γ_D, Γ_N or Γ_R. Although not necessary, it is often followed that an element cannot have two or more faces on the boundary. We depict in Figure 2.2 a (2D-equivalent) example of such a mesh.

The finite element method we implement in this project involves a *natural* approximation by a polygonal domain, say Ω_h, defined by the underlying mesh. Although the ideas presented here work for this case (assuming the elements are small enough to approximate accurately the domain), the case of non-polygonal and smooth domains Ω can also be considered using this method, in this case, an additional error occurs due to the approximation of the curved domain (see for example [13]).

2.2.2 Finite Element Space

Let P_h^m be the space of piecewise polynomials, continuous in Ω:

$$P_h^m = \{v \in \mathcal{C}(\overline{\Omega}) : v|_K \in \mathbb{P}_m, \forall K \in \mathcal{T}_h\}, \tag{2.2}$$

where $v|_K$ is the function restricted to the element K and $\mathbb{P}_m$ is the set of all polynomials of degree less than or equal to m. In this context, K is referred to as a $\mathbb{P}_m$ element. Note that the continuity of v inside each tetrahedron K is already satisfied by the polynomial nature of $v|_K$. However, to ensure the overall continuity of v in $\overline{\Omega}$, it is necessary to impose that for any two elements K_1 and K_2 sharing a common edge, face, or vertex, the polynomials $v|_{K_1}$ and $v|_{K_2}$ coincide on that entity.

Any polynomial function in K can be written as a sum of monomials of degree at most m. Such a polynomial can be expressed as

$$v|_K(\boldsymbol{x}) = v|_K(x,y,z) = \sum_{\substack{i,j,k\geq 0 \\ i+j+k\leq m}} a_{ijk}^K x^i y^j z^k,$$

where a_{ijk}^K are coefficients that can be determined, among other and different ways, by giving the polynomial values at certain points $\{\boldsymbol{v}_i^K\}$ inside K (usually called an interpolation problem). These points are called the nodes of the element, and to specify the polynomial uniquely we need

$$\texttt{dofK} = \binom{3+m}{3} = \tfrac{(m+3)(m+2)(m+1)}{6}$$

of them [6]. For example, the degree one polynomial is given by

$$v|_K = a^K_{000} + a^K_{100}x + a^K_{010}y + a^K_{001}z,$$

which contains $\binom{3+1}{3} = 4$ coefficients.

The continuity requirement can be satisfied by enforcing the value of v for each face A, in which case $v|_A$ is just a bivariate polynomial. This polynomial can be fully defined by appropriately locating

$$\texttt{dofA} = \binom{2+m}{2} = \frac{(m+2)(m+1)}{2}$$

nodes on A out of the `dofK` nodes in the tetrahedron.

Instead of using the monomial polynomial basis, we can use a different basis based on the Lagrange functions associated with the node distribution inside an element $\{\boldsymbol{v}^K_k\}_{k=1}^{\texttt{dofK}}$. The Lagrange basis functions are mth degree polynomials satisfying the Lagrange condition

$$N^K_\ell(\boldsymbol{v}^K_k) = \begin{cases} 1, & \ell = k \\ 0, & \ell \neq k \end{cases} \qquad \forall k, \ell \in \{1, \ldots, \texttt{dofK}\}\,. \tag{2.3}$$

This basis function allows us to express the polynomial $p^K_m(\boldsymbol{x})$ as

$$p^K_m(\boldsymbol{x}) = \sum_{\ell=1}^{\texttt{dofK}} p^K_m(\boldsymbol{v}^K_\ell) N^K_\ell(\boldsymbol{x}), \tag{2.4}$$

i.e., using this basis, the coefficients are just the values of the polynomial at the nodes. These coefficients are referred to as the degrees of freedom of the element K.

Both the monomial polynomial basis and the Lagrange basis represent the same polynomial, as a polynomial of degree m passing through the `dofK` nodes is unique (provided they are appropriately placed). However, the Lagrange basis provides a significant advantage in that the degrees of freedom are associated with the nodal values, resulting in a more straightforward construction and operation (evaluation, differentiation, integration, etc) of the polynomial.

Once this distribution of the nodes has been fixed, we can define the set of all, say `nNodes`, nodes in the mesh as the union of all nodes of each element $K \in \mathcal{T}_h$,

$$\{\boldsymbol{v}_j\}_{j=1}^{\texttt{nNodes}} = \bigcup_{K \in \mathcal{T}_h} \boldsymbol{v}^K_j.$$

In other words, a finite element is determined by its values at `nNodes` points in Ω, points which are strongly dependent of the mesh. Each node $\boldsymbol{v}_j$ has an associated index j that enables us to distinguish the nodes in the mesh, this index is known as the global index of $\boldsymbol{v}_j$.

We use these indexed nodes to construct a basis $\varphi_1, \ldots, \varphi_{\texttt{nNodes}} = \{\varphi_j\}_{j=1}^{\texttt{nNodes}}$ of P^m_h using the Lagrange basis. Here, each member φ_j is a global shape function associated with a node $\boldsymbol{v}_j$ in the mesh. The basis functions φ_j can be constructed as a piecewise polynomial function defined as

$$\varphi_j(\boldsymbol{x}) = \begin{cases} N^{K_1}_{j_1}(\boldsymbol{x}), & \boldsymbol{x} \in K_1, \\ N^{K_2}_{j_2}(\boldsymbol{x}), & \boldsymbol{x} \in K_2, \\ \qquad\vdots & \\ N^{K_n}_{j_n}(\boldsymbol{x}), & \boldsymbol{x} \in K_n, \\ 0, & \text{otherwise}, \end{cases} \tag{2.5}$$

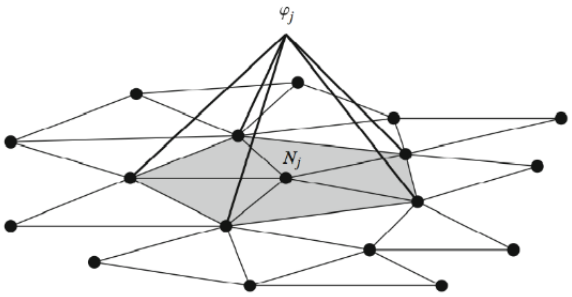


Figure 2.3: Example of the 2D analog shape function [15].

where $K_1, \ldots, K_n$ are the elements containing the node with global index j and $N_{j_i}^{K_i}$ the associated Lagrange basis of the node in K_i. An example of such a function for $\mathbb{P}_1$ elements in a triangular mesh is shown in Figure 2.3.

Example. Consider $m = 1$, linear elements, so that

$$P_h^1 = \{v \in \mathcal{C}(\overline{\Omega}) \mid v|_K \in \mathbb{P}_1, \ \forall K \in \mathcal{T}_h\}.$$

A natural choice would be taken on each K, its vertices as the nodes $\{\boldsymbol{v}_\ell^K\}_{\ell=1}^4$. Indeed, a polynomial of degree 1 is uniquely determined by its values at four non-coplanar points. Conversely, if the values of the finite element are specified at this set of points, we can obviously construct a piecewise linear polynomial.

Any function $v \in P_h^1$ is also continuous, since for any pair of adjacent elements K_1 and K_2 sharing a (triangular) face A, and denoting $v_1 = v|_{K_1}$ and $v_2 = v|_{K_2}$, then $v_1|_A = v_2|_A$ (which means that v is continuous across face A) since both functions are polynomials of two-variables and degree 1 which coincide at three non-colinear points, namely, the vertices of the face A.

Hence, there is bijection between linear finite elements and its values at this set of nodes. That is, for determining a function in P_h^1 it suffices to provide its values at the vertices of the tetrahedra in the mesh and vice versa. □.

We will show how this construction can be extended to finite elements of higher degree. To accomplish this, we will introduce the concept of barycentric coordinates. These coordinates will be used to define the distribution of nodes, initially on the reference element, and subsequently on arbitrary elements.

2.2.3 Barycentric Coordinates

The barycentric coordinates in a $n-$simplex (line, triangle, tetrahedron, ...) $T \in \mathbb{R}^n$ are defined as the set of coordinates $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_{n+1})$ such that for any point $\boldsymbol{x} \in \mathbb{R}^n$ they satisfy

$$\begin{cases} \sum_{i=1}^{n+1} \lambda_i &= 1, \\ \boldsymbol{x} &= \sum_{i=1}^{n+1} \lambda_i \boldsymbol{x}_i, \end{cases}$$

where $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{n+1} \in \mathbb{R}^n$ are the vertices of the simplex. With this definition we can define the relative position of $\boldsymbol{x}$ with respect to the vertices of T. In matrix notation, these expressions are equivalent to

$$\begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_{n+1} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_{n+1} \end{bmatrix}. \tag{2.6}$$

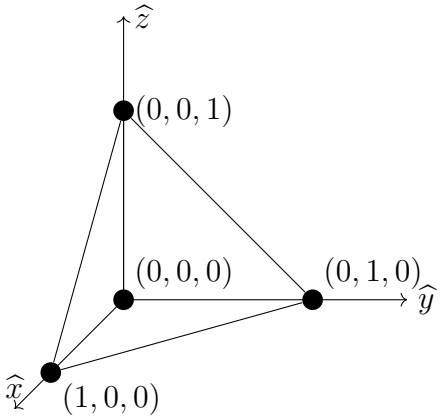


Figure 2.4: Reference tetrahedral element.

In particular, a 2-simplex is any non-degenerate triangle A with vertices $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$. The previous expression becomes

$$\begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ \boldsymbol{x}_1 & \boldsymbol{x}_2 & \boldsymbol{x}_3 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix},$$

and for a 3-simplex (a tetrahedron) K with vertices $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4$ it becomes

$$\begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \boldsymbol{x}_1 & \boldsymbol{x}_2 & \boldsymbol{x}_3 & \boldsymbol{x}_4 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{bmatrix}.$$

2.2.4 Reference Element

In this section we will define the polynomial interpolation problem on what we will refer to as the **reference element**, depicted in Figure 2.4, and defined as

$$\widehat{K} = \{\widehat{\boldsymbol{x}} = (\widehat{x}, \widehat{y}, \widehat{z}) \mid 0 \leq 1 - \widehat{x} - \widehat{y} - \widehat{z} \leq 1,\ 0 \leq \widehat{x} \leq 1,\ 0 \leq \widehat{y} \leq 1,\ 0 \leq \widehat{z} \leq 1\}.$$

The vertices of this element are given and indexed as follows:

$$\begin{aligned} \widehat{\boldsymbol{x}}_1 &= (0,0,0), \\ \widehat{\boldsymbol{x}}_2 &= (1,0,0), \\ \widehat{\boldsymbol{x}}_3 &= (0,1,0), \\ \widehat{\boldsymbol{x}}_4 &= (0,0,1). \end{aligned}$$

Let us denote the barycentric coordinates for any point $\widehat{\boldsymbol{x}}$ in the reference element by $\widehat{\boldsymbol{\lambda}} = \begin{bmatrix} \widehat{\lambda}_1 & \widehat{\lambda}_2 & \widehat{\lambda}_3 & \widehat{\lambda}_4 \end{bmatrix}^\top$. Certainly $\widehat{\boldsymbol{\lambda}} = \widehat{\boldsymbol{\lambda}}(\widehat{\boldsymbol{x}})$ (that is, the barycentric coordinates depend on the point $\widehat{\boldsymbol{x}}$), but if the context makes it clear which specific point or points we are referring to, we will omit any explicit reference to the point in our notation.

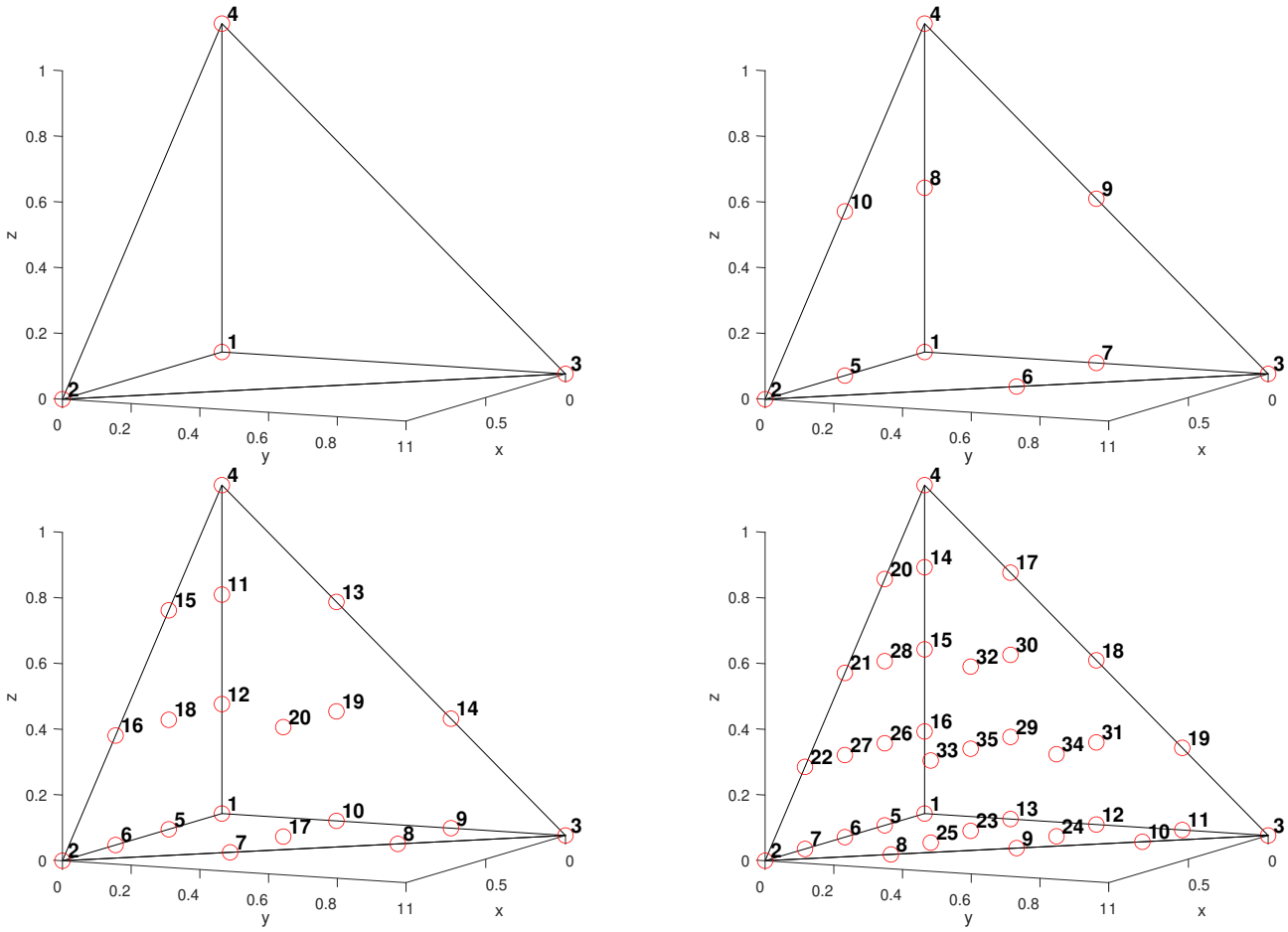


Figure 2.5: Nodes numbering on the $\mathbb{P}_1$ (top-left), $\mathbb{P}_2$ (top-right), $\mathbb{P}_3$ (bottom-left) and $\mathbb{P}_4$ (bottom-right) reference elements.

Since in this case $\widehat{\boldsymbol{\lambda}}$ are determined by solving the linear system

$$\begin{bmatrix} 1 \\ \widehat{x} \\ \widehat{y} \\ \widehat{z} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \widehat{\lambda}_1 \\ \widehat{\lambda}_2 \\ \widehat{\lambda}_3 \\ \widehat{\lambda}_4 \end{bmatrix}$$

we conclude that

$$\begin{aligned} \widehat{\lambda}_1 &= 1 - \widehat{x} - \widehat{y} - \widehat{z}, \\ \widehat{\lambda}_2 &= \widehat{x}, \\ \widehat{\lambda}_3 &= \widehat{y}, \\ \widehat{\lambda}_4 &= \widehat{z}. \end{aligned} \tag{2.7}$$

Clearly, barycentric coordinates are themselves polynomials of degree 1 in these variables that equals one at one vertex and zero at the others.

By examining the expressions for the barycentric coordinates, we can establish a useful connection with the boundary of $\widehat{K}$, which enables us to classify points as either inside or outside the reference tetrahedron. Specifically, if at least one barycentric coordinates is negative, then the point is outside $\widehat{K}$, while a point lying inside $\widehat{K}$ will have non-negative barycentric coordinates.

Now, let us define the set of nodes for the interpolation problem on $\mathbb{P}_m$ for any integer m as:

$$\{\widehat{\boldsymbol{v}}_i\} = \left\{ \left(\frac{i}{m}, \frac{j}{m}, \frac{k}{m} \right) \mid \text{with } i,j,k \text{ non-negative integers satisfying } i+j+k \leq m \right\}. \quad (2.8)$$

Here, for the sake of simplicity we omit any reference to m in the notation for the nodes. The number of nodes can be easily verified to be $\binom{3+m}{3}$ which coincides with `dofK`, the dimension of the space $\mathbb{P}_m$, i.e., the number of coefficients of any three-variable polynomial of degree m. Therefore, it can be reasonably expected that the values of a polynomial at these nodes uniquely determine the polynomial.

We can index every node with a different label inside this element (local index). The convention followed is the one adopted in GMSH [8]. The basic idea of the indexing is the following (see figure 2.5 to check the indexing for $m = 1,\ 2,\ 3$ and 4):

- The nodes at the vertices are indexed according to the order assumed earlier.
- The nodes at the interior of the edges are indexed in the following order: first, the nodes along the edge from vertex $\widehat{\boldsymbol{x}}_1$ to vertex $\widehat{\boldsymbol{x}}_2$; then, the nodes along the edge from vertex $\widehat{\boldsymbol{x}}_2$ to vertex $\widehat{\boldsymbol{x}}_3$; next, the nodes along the edge from vertex $\widehat{\boldsymbol{x}}_3$ to vertex $\widehat{\boldsymbol{x}}_1$. The remaining nodes are indexed starting from the fourth vertex to all other vertices, starting from the edge shared with vertex $\widehat{\boldsymbol{x}}_1$, followed by the edge shared with vertex $\widehat{\boldsymbol{x}}_2$, and ending with the remaining edge.
- The nodes inside the faces of the tetrahedron are indexed such that the normal vector points outwards, following the right-hand rule.
- The indexing of the internal nodes follows the convention of lower-order tetrahedral elements. For the finite element space $\mathbb{P}_4$, there is only one node in the interior of $\widehat{K}$ and its index is trivial. For higher-order elements, the internal nodes are distributed and indexed following the idea of the $\mathbb{P}_{m-4}$ reference element.

To show the well posedness of the interpolation problem, that is, that a polynomial of degree m is uniquely determined by its value at this set of points, it suffices to construct the associated Lagrange basis $\{\widehat{N}_i\}_{i=1}^{\texttt{dofK}}$. That is, mth degree polynomials that vanish at all nodes except for one. The use of barycentric coordinates greatly facilitates this purpose. For $\mathbb{P}_1$ it is almost trivial: the associated Lagrange basis is

$$\widehat{N}_\ell(\widehat{\boldsymbol{x}}) = \widehat{\lambda}_\ell(\boldsymbol{x}), \quad \ell = 1,2,3,4,$$

the barycentric coordinates themselves since in this case the nodes $\{\widehat{\boldsymbol{v}}_n\}_{n=1}^4$ are just the vertices of $\widehat{K}$ and therefore

$$\widehat{N}_\ell(\widehat{\boldsymbol{v}}_n) = \begin{cases} 1, & n = \ell \\ 0, & n \neq \ell \end{cases} \qquad \forall \ell, n \in \{1,2,3,4\}.$$

Consider now the case of higher degree polynomials, i.e. $m \geq 2$. Let $\widehat{\boldsymbol{v}}_i$ be the ith node with barycentric coordinates given by

$$\left(\frac{i_1}{m}, \frac{i_2}{m}, \frac{i_3}{m}, \frac{i_4}{m} \right)$$

(recall that i_1, i_2, i_3, i_4 are non-negative integers whose precise values are given by the nodes defined above) satisfying $i_1 + i_2 + i_3 + i_4 = m$. The Lagrange function associated to this node is given by

$$\widehat{N}_i(\widehat{\boldsymbol{x}}) = \prod_{j=1}^{4} \underbrace{\prod_{\ell=0}^{i_j-1} \frac{\widehat{\lambda}_j(\widehat{\boldsymbol{x}}) - \frac{\ell}{m}}{\widehat{\lambda}_j(\widehat{\boldsymbol{v}}_i) - \frac{\ell}{m}}}_{p_j(\widehat{\boldsymbol{x}})}. \tag{2.9}$$

This is an extension to the one-dimensional Lagrange interpolation [26]. We see easily that:

1. $\widehat{N}_i(\widehat{\boldsymbol{x}})$ is the product of four polynomials $p_j(\widehat{\boldsymbol{x}})$, $j = 1, \ldots, 4$ of degree i_1, i_2, i_3 and i_4 respectively, which proves that $\widehat{N}_i \in \mathbb{P}_m$.
2. Due to the way the nodes where constructed, for every other node $\widehat{\boldsymbol{v}}_\ell$ with $\ell \neq i$ there has to be at least one barycentric coordinate, say the kth one, such that

$$\widehat{\lambda}_k(\widehat{\boldsymbol{v}}_\ell) = \frac{\ell_k}{m} < \frac{i_k}{m} = \widehat{\lambda}_k(\widehat{\boldsymbol{v}}_i),$$

   which implies that $p_k(\widehat{\boldsymbol{v}}_\ell) = 0$ and so $\widehat{N}_i(\widehat{\boldsymbol{x}})$ vanishes at $\widehat{\boldsymbol{v}}_\ell$.
3. If $\ell = i$, clearly $p_j(\widehat{\boldsymbol{v}}_\ell) = 1$, since all the factors in the above product are one at this point. Hence, $\widehat{N}_i(\widehat{\boldsymbol{v}}_i) = 1$.

It is worth noting that the polynomial $\widehat{N}_i(\widehat{\boldsymbol{x}})$ can be conveniently represented using the barycentric coordinates, thus allowing us to use $\widehat{N}_i(\widehat{\boldsymbol{\lambda}}(\widehat{\boldsymbol{x}}))$ instead. Therefore, we can use these functions to construct any arbitrary polynomial $p(\widehat{\boldsymbol{x}}) \in \mathbb{P}_m$ in the reference element as

$$p_m^{\widehat{K}}(\widehat{\boldsymbol{x}}) = \sum_{i=1}^{\texttt{dofK}} p_m^{\widehat{K}}(\widehat{\boldsymbol{v}}_i) \widehat{N}_i(\widehat{\boldsymbol{\lambda}}(\widehat{\boldsymbol{x}})). \tag{2.10}$$

The barycentric coordinates of the nodes for the mth ($m = 1, 2, 3, 4$) reference element and their associated Lagrange basis are given below. The tables have to read as follows: the panel in the middle displays, in the ith row the barycentric coordinates of the ith node. The right panel express the function $\widehat{N}_i$ using the barycentric coordinate system:

- $\mathbb{P}_1$ element:

i	$\widehat{\lambda}_1$	$\widehat{\lambda}_2$	$\widehat{\lambda}_3$	$\widehat{\lambda}_4$	$\widehat{N}_i$
1	1	0	0	0	$\widehat{\lambda}_1$
2	0	1	0	0	$\widehat{\lambda}_2$
3	0	0	1	0	$\widehat{\lambda}_3$
4	0	0	0	1	$\widehat{\lambda}_4$

- $\mathbb{P}_2$ element:

i	$\widehat{\lambda}_1$	$\widehat{\lambda}_2$	$\widehat{\lambda}_3$	$\widehat{\lambda}_4$	$\widehat{N}_i$
1	1	0	0	0	$2\widehat{\lambda}_1\left(\widehat{\lambda}_1 - \frac{1}{2}\right)$
2	0	1	0	0	$2\widehat{\lambda}_2\left(\widehat{\lambda}_2 - \frac{1}{2}\right)$

3	0	0	1	0	$2\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{2}\right)$
4	0	0	0	1	$2\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{2}\right)$
5	$\frac{1}{2}$	$\frac{1}{2}$	0	0	$4\widehat{\lambda}_1\widehat{\lambda}_2$
6	0	$\frac{1}{2}$	$\frac{1}{2}$	0	$4\widehat{\lambda}_2\widehat{\lambda}_3$
7	$\frac{1}{2}$	0	$\frac{1}{2}$	0	$4\widehat{\lambda}_1\widehat{\lambda}_3$
8	$\frac{1}{2}$	0	0	$\frac{1}{2}$	$4\widehat{\lambda}_1\widehat{\lambda}_4$
9	0	0	$\frac{1}{2}$	$\frac{1}{2}$	$4\widehat{\lambda}_3\widehat{\lambda}_4$
10	0	$\frac{1}{2}$	0	$\frac{1}{2}$	$4\widehat{\lambda}_2\widehat{\lambda}_4$

- $\mathbb{P}_3$ element:

i	$\widehat{\lambda}_1$	$\widehat{\lambda}_2$	$\widehat{\lambda}_3$	$\widehat{\lambda}_4$	$\widehat{N}_i$
1	1	0	0	0	$\frac{9}{2}\widehat{\lambda}_1\left(\widehat{\lambda}_1-\frac{1}{3}\right)\left(\widehat{\lambda}_1-\frac{2}{3}\right)$
2	0	1	0	0	$\frac{9}{2}\widehat{\lambda}_2\left(\widehat{\lambda}_2-\frac{1}{3}\right)\left(\widehat{\lambda}_2-\frac{2}{3}\right)$
3	0	0	1	0	$\frac{9}{2}\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{3}\right)\left(\widehat{\lambda}_3-\frac{2}{3}\right)$
4	0	0	0	1	$\frac{9}{2}\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{3}\right)\left(\widehat{\lambda}_4-\frac{2}{3}\right)$
5	$\frac{2}{3}$	$\frac{1}{3}$	0	0	$\frac{27}{2}\widehat{\lambda}_1\widehat{\lambda}_2\left(\widehat{\lambda}_1-\frac{1}{3}\right)$
6	$\frac{1}{3}$	$\frac{2}{3}$	0	0	$\frac{27}{2}\widehat{\lambda}_1\widehat{\lambda}_2\left(\widehat{\lambda}_2-\frac{1}{3}\right)$
7	0	$\frac{2}{3}$	$\frac{1}{3}$	0	$\frac{27}{2}\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_2-\frac{1}{3}\right)$
8	0	$\frac{1}{3}$	$\frac{2}{3}$	0	$\frac{27}{2}\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{3}\right)$
9	$\frac{1}{3}$	0	$\frac{2}{3}$	0	$\frac{27}{2}\widehat{\lambda}_1\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{3}\right)$
10	$\frac{2}{3}$	0	$\frac{1}{3}$	0	$\frac{27}{2}\widehat{\lambda}_1\widehat{\lambda}_3\left(\widehat{\lambda}_1-\frac{1}{3}\right)$
11	$\frac{1}{3}$	0	0	$\frac{2}{3}$	$\frac{27}{2}\widehat{\lambda}_1\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{3}\right)$
12	$\frac{2}{3}$	0	0	$\frac{1}{3}$	$\frac{27}{2}\widehat{\lambda}_1\widehat{\lambda}_4\left(\widehat{\lambda}_1-\frac{1}{3}\right)$
13	0	0	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{27}{2}\widehat{\lambda}_3\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{3}\right)$
14	0	0	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{27}{2}\widehat{\lambda}_3\widehat{\lambda}_4\left(\widehat{\lambda}_3-\frac{1}{3}\right)$
15	0	$\frac{1}{3}$	0	$\frac{2}{3}$	$\frac{27}{2}\widehat{\lambda}_2\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{3}\right)$
16	0	$\frac{2}{3}$	0	$\frac{1}{3}$	$\frac{27}{2}\widehat{\lambda}_2\widehat{\lambda}_4\left(\widehat{\lambda}_2-\frac{1}{3}\right)$
17	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0	$27\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_3$
18	$\frac{1}{3}$	$\frac{1}{3}$	0	$\frac{1}{3}$	$27\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_4$
19	$\frac{1}{3}$	0	$\frac{1}{3}$	$\frac{1}{3}$	$27\widehat{\lambda}_1\widehat{\lambda}_3\widehat{\lambda}_4$
20	0	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	$27\widehat{\lambda}_2\widehat{\lambda}_3\widehat{\lambda}_4$

- $\mathbb{P}_4$ element:

i	$\widehat{\lambda}_1$	$\widehat{\lambda}_2$	$\widehat{\lambda}_3$	$\widehat{\lambda}_4$	$\widehat{N}_i$
1	1	0	0	0	$\frac{32}{3}\widehat{\lambda}_1\left(\widehat{\lambda}_1-\frac{1}{2}\right)\left(\widehat{\lambda}_1-\frac{1}{4}\right)\left(\widehat{\lambda}_1-\frac{3}{4}\right)$
2	0	1	0	0	$\frac{32}{3}\widehat{\lambda}_2\left(\widehat{\lambda}_2-\frac{1}{2}\right)\left(\widehat{\lambda}_2-\frac{1}{4}\right)\left(\widehat{\lambda}_2-\frac{3}{4}\right)$
3	0	0	1	0	$\frac{32}{3}\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{2}\right)\left(\widehat{\lambda}_3-\frac{1}{4}\right)\left(\widehat{\lambda}_3-\frac{3}{4}\right)$

4	0	0	0	1	$\frac{32}{3}\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{2}\right)\left(\widehat{\lambda}_4-\frac{1}{4}\right)\left(\widehat{\lambda}_4-\frac{3}{4}\right)$
5	$\frac{3}{4}$	$\frac{1}{4}$	0	0	$\frac{128}{3}\widehat{\lambda}_1\widehat{\lambda}_2\left(\widehat{\lambda}_1-\frac{1}{2}\right)\left(\widehat{\lambda}_1-\frac{1}{4}\right)$
6	$\frac{2}{4}$	$\frac{2}{4}$	0	0	$64\widehat{\lambda}_1\widehat{\lambda}_2\left(\widehat{\lambda}_1-\frac{1}{4}\right)\left(\widehat{\lambda}_2-\frac{1}{4}\right)$
7	$\frac{1}{4}$	$\frac{3}{4}$	0	0	$\frac{128}{3}\widehat{\lambda}_1\widehat{\lambda}_2\left(\widehat{\lambda}_2-\frac{1}{2}\right)\left(\widehat{\lambda}_2-\frac{1}{4}\right)$
8	0	$\frac{3}{4}$	$\frac{1}{4}$	0	$\frac{128}{3}\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_2-\frac{1}{2}\right)\left(\widehat{\lambda}_2-\frac{1}{4}\right)$
9	0	$\frac{2}{4}$	$\frac{2}{4}$	0	$64\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_2-\frac{1}{4}\right)\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
10	0	$\frac{1}{4}$	$\frac{3}{4}$	0	$\frac{128}{3}\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{2}\right)\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
11	$\frac{1}{4}$	0	$\frac{3}{4}$	0	$\frac{128}{3}\widehat{\lambda}_1\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{2}\right)\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
12	$\frac{2}{4}$	0	$\frac{2}{4}$	0	$64\widehat{\lambda}_1\widehat{\lambda}_3\left(\widehat{\lambda}_1-\frac{1}{4}\right)\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
13	$\frac{3}{4}$	0	$\frac{1}{4}$	0	$\frac{128}{3}\widehat{\lambda}_1\widehat{\lambda}_3\left(\widehat{\lambda}_1-\frac{1}{2}\right)\left(\widehat{\lambda}_1-\frac{1}{4}\right)$
14	$\frac{1}{4}$	0	0	$\frac{3}{4}$	$\frac{128}{3}\widehat{\lambda}_1\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{2}\right)\left(\widehat{\lambda}_4-\frac{1}{4}\right)$
15	$\frac{2}{4}$	0	0	$\frac{2}{4}$	$64\widehat{\lambda}_1\widehat{\lambda}_4\left(\widehat{\lambda}_1-\frac{1}{4}\right)\left(\widehat{\lambda}_4-\frac{1}{4}\right)$
16	$\frac{3}{4}$	0	0	$\frac{1}{4}$	$\frac{128}{3}\widehat{\lambda}_1\widehat{\lambda}_4\left(\widehat{\lambda}_1-\frac{1}{2}\right)\left(\widehat{\lambda}_1-\frac{1}{4}\right)$
17	0	0	$\frac{1}{4}$	$\frac{3}{4}$	$\frac{128}{3}\widehat{\lambda}_3\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{2}\right)\left(\widehat{\lambda}_4-\frac{1}{4}\right)$
18	0	0	$\frac{2}{4}$	$\frac{2}{4}$	$64\widehat{\lambda}_4\widehat{\lambda}_4\left(\widehat{\lambda}_3-\frac{1}{4}\right)\left(\widehat{\lambda}_4-\frac{1}{4}\right)$
19	0	0	$\frac{3}{4}$	$\frac{1}{4}$	$\frac{128}{3}\widehat{\lambda}_3\widehat{\lambda}_4\left(\widehat{\lambda}_3-\frac{1}{2}\right)\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
20	0	$\frac{1}{4}$	0	$\frac{3}{4}$	$\frac{128}{3}\widehat{\lambda}_2\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{2}\right)\left(\widehat{\lambda}_4-\frac{1}{4}\right)$
21	0	$\frac{2}{4}$	0	$\frac{2}{4}$	$64\widehat{\lambda}_2\widehat{\lambda}_4\left(\widehat{\lambda}_2-\frac{1}{4}\right)\left(\widehat{\lambda}_4-\frac{1}{4}\right)$
22	0	$\frac{3}{4}$	0	$\frac{1}{4}$	$\frac{128}{3}\widehat{\lambda}_2\widehat{\lambda}_4\left(\widehat{\lambda}_2-\frac{1}{2}\right)\left(\widehat{\lambda}_2-\frac{1}{4}\right)$
23	$\frac{2}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	0	$128\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_1-\frac{1}{4}\right)$
24	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{2}{4}$	0	$128\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
25	$\frac{1}{4}$	$\frac{2}{4}$	$\frac{1}{4}$	0	$128\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_2-\frac{1}{4}\right)$
26	$\frac{2}{4}$	$\frac{1}{4}$	0	$\frac{1}{4}$	$128\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_4\left(\widehat{\lambda}_1-\frac{1}{4}\right)$
27	$\frac{1}{4}$	$\frac{2}{4}$	0	$\frac{1}{4}$	$128\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_4\left(\widehat{\lambda}_2-\frac{1}{4}\right)$
28	$\frac{1}{4}$	$\frac{1}{4}$	0	$\frac{2}{4}$	$128\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{4}\right)$
29	$\frac{2}{4}$	0	$\frac{1}{4}$	$\frac{1}{4}$	$128\widehat{\lambda}_1\widehat{\lambda}_3\widehat{\lambda}_4\left(\widehat{\lambda}_1-\frac{1}{4}\right)$
30	$\frac{1}{4}$	0	$\frac{1}{4}$	$\frac{2}{4}$	$128\widehat{\lambda}_1\widehat{\lambda}_3\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{4}\right)$
31	$\frac{1}{4}$	0	$\frac{2}{4}$	$\frac{1}{4}$	$128\widehat{\lambda}_1\widehat{\lambda}_3\widehat{\lambda}_4\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
32	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{2}{4}$	$128\widehat{\lambda}_2\widehat{\lambda}_3\widehat{\lambda}_4\left(\widehat{\lambda}_4-\frac{1}{4}\right)$
33	0	$\frac{2}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$128\widehat{\lambda}_2\widehat{\lambda}_3\widehat{\lambda}_4\left(\widehat{\lambda}_2-\frac{1}{4}\right)$
34	0	$\frac{1}{4}$	$\frac{2}{4}$	$\frac{1}{4}$	$128\widehat{\lambda}_2\widehat{\lambda}_3\widehat{\lambda}_4\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
35	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$256\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_3\widehat{\lambda}_4$

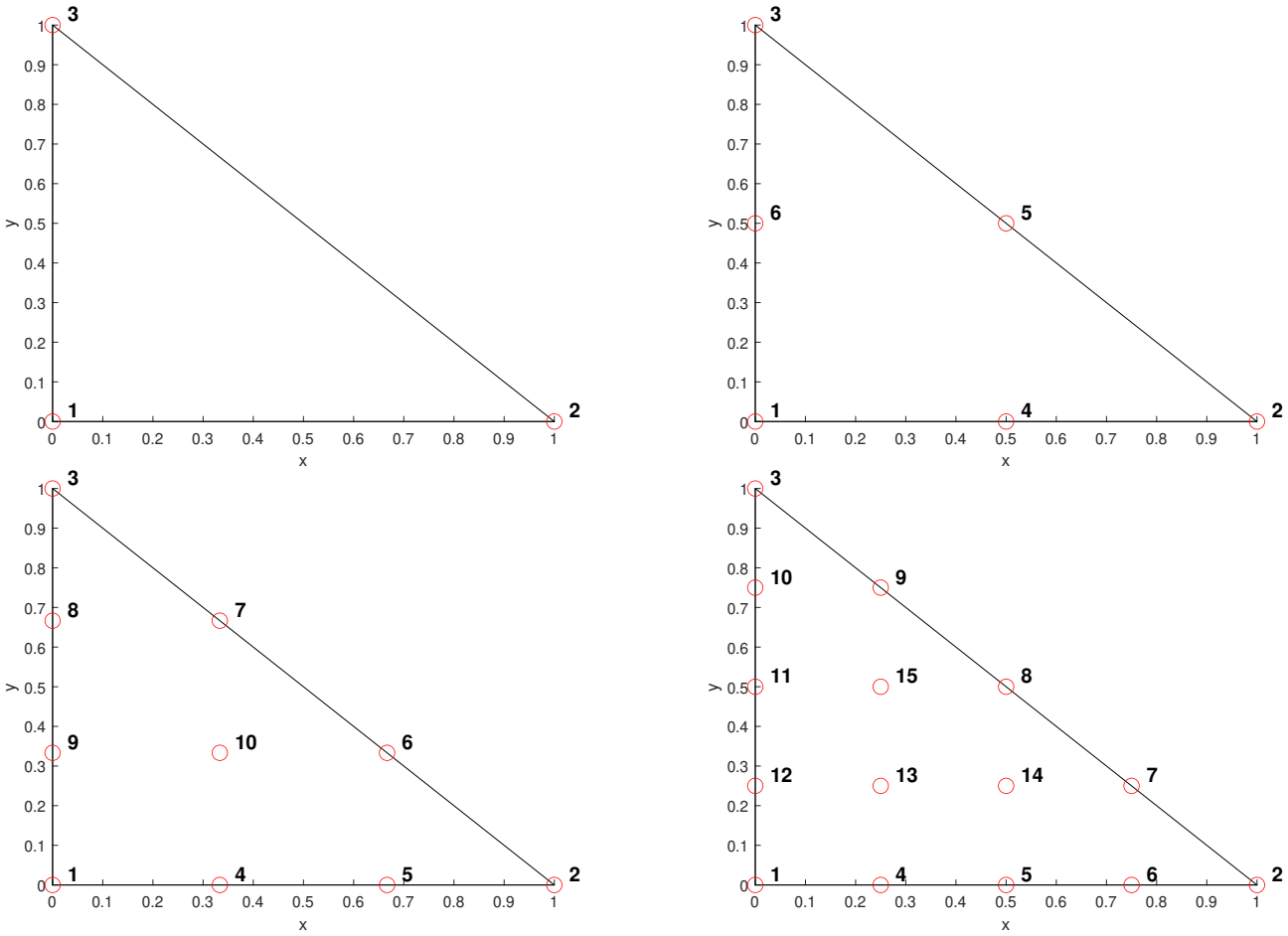


Figure 2.6: Nodes numbering on the $\mathbb{P}_1$ (top-left), $\mathbb{P}_2$ (top-right), $\mathbb{P}_3$ (bottom-left) and $\mathbb{P}_4$ (bottom-right) reference boundary elements.

Faces

It is necessary to perform a similar analysis to the interpolation problem on the faces of each tetrahedron. In this case will give only a simple sketch of this construction.

Let the reference triangle be given by

$$\widehat{A} = \{\widehat{\boldsymbol{x}} = (\widehat{x}, \widehat{y}) | \, 0 \leq 1 - \widehat{x} - \widehat{y} \leq 1, 0 \leq \widehat{x} \leq 1, 0 \leq \widehat{y} \leq 1\}$$

which is placed in the $\widehat{x} - \widehat{y}$ plane and has vertices

$$\begin{aligned} \widehat{\boldsymbol{x}}_1 &= (0, 0), \\ \widehat{\boldsymbol{x}}_2 &= (1, 0), \\ \widehat{\boldsymbol{x}}_3 &= (0, 1). \end{aligned}$$

The corresponding barycentric coordinates are then given by

$$\begin{aligned} \widehat{\lambda}_1 &= 1 - \widehat{x} - \widehat{y}, \\ \widehat{\lambda}_2 &= \widehat{x}, \\ \widehat{\lambda}_3 &= \widehat{y}. \end{aligned}$$

Any mth degree bivariate polynomial $\mathbb{P}_m$ is uniquely determined by its value at the set of nodes given, in barycentric coordinates, by

$$\{\widehat{\boldsymbol{v}}_i\} = \left(\frac{i_1}{m}, \frac{i_2}{m}, \frac{i_3}{m} \right), \quad 0 \leq i_1, \ i_2, \ i_3, \quad i_1 + i_2 + i_3 = m.$$

The ordering for the nodes of the reference triangle of order m are depicted in Figure 2.6, where they are located at the vertices, edges, and interior of the triangle. The indexing follows the GMSH convention:

- Firstly, the vertices are indexed using the same order as above.
- Next, the nodes inside each edge of the triangle are indexed, starting from the edge lying on the $\widehat{x}$ axis and following a counterclockwise indexing.
- Finally, the interior nodes are defined recursively, considering them as lower-order elements. For $\mathbb{P}_m$ elements, the interior nodes are indexed following the rules for $\mathbb{P}_{m-3}$ elements.

The associated bivariate polynomial Lagrange basis is given by

$$\widehat{N}_i(\widehat{\boldsymbol{x}}) = \prod_{j=1}^{3} \prod_{\ell=0}^{i_j-1} \frac{\widehat{\lambda}_j(\widehat{\boldsymbol{x}}) - \frac{\ell}{m}}{\widehat{\lambda}_j(\widehat{\boldsymbol{v}}_i) - \frac{\ell}{m}}. \tag{2.11}$$

We list below the barycentric coordinates of the nodes for the mth ($m = 1, 2, 3$) reference element and their associated Lagrange basis:

- $\mathbb{P}_1$ element:

i	$\widehat{\lambda}_1$	$\widehat{\lambda}_2$	$\widehat{\lambda}_3$	$\widehat{N}_i$
1	1	0	0	$\widehat{\lambda}_1$
2	0	1	0	$\widehat{\lambda}_2$
3	0	0	1	$\widehat{\lambda}_3$

- $\mathbb{P}_2$ element:

i	$\widehat{\lambda}_1$	$\widehat{\lambda}_2$	$\widehat{\lambda}_3$	$\widehat{N}_i$
1	1	0	0	$2\widehat{\lambda}_1\left(\widehat{\lambda}_1 - \frac{1}{2}\right)$
2	0	1	0	$2\widehat{\lambda}_2\left(\widehat{\lambda}_2 - \frac{1}{2}\right)$
3	0	0	1	$2\widehat{\lambda}_3\left(\widehat{\lambda}_3 - \frac{1}{2}\right)$
4	$\frac{1}{2}$	$\frac{1}{2}$	0	$4\widehat{\lambda}_1\widehat{\lambda}_2$
5	0	$\frac{1}{2}$	$\frac{1}{2}$	$4\widehat{\lambda}_2\widehat{\lambda}_3$
6	$\frac{1}{2}$	0	$\frac{1}{2}$	$4\widehat{\lambda}_1\widehat{\lambda}_3$

- $\mathbb{P}_3$ element:

i	$\widehat{\lambda}_1$	$\widehat{\lambda}_2$	$\widehat{\lambda}_3$	$\widehat{N}_i$
1	1	0	0	$\frac{9}{2}\widehat{\lambda}_1\left(\widehat{\lambda}_1 - \frac{1}{3}\right)\left(\widehat{\lambda}_1 - \frac{2}{3}\right)$
2	0	1	0	$\frac{9}{2}\widehat{\lambda}_2\left(\widehat{\lambda}_2 - \frac{1}{3}\right)\left(\widehat{\lambda}_2 - \frac{2}{3}\right)$
3	0	0	1	$\frac{9}{2}\widehat{\lambda}_3\left(\widehat{\lambda}_3 - \frac{1}{3}\right)\left(\widehat{\lambda}_3 - \frac{2}{3}\right)$
4	$\frac{2}{3}$	$\frac{1}{3}$	0	$\frac{27}{2}\widehat{\lambda}_1\widehat{\lambda}_2\left(\widehat{\lambda}_1 - \frac{1}{3}\right)$
5	$\frac{1}{3}$	$\frac{2}{3}$	0	$\frac{27}{2}\widehat{\lambda}_1\widehat{\lambda}_2\left(\widehat{\lambda}_2 - \frac{1}{3}\right)$

6	0	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{27}{2}\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_2-\frac{1}{3}\right)$
7	0	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{27}{2}\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{3}\right)$
8	$\frac{1}{3}$	0	$\frac{2}{3}$	$\frac{27}{2}\widehat{\lambda}_1\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{3}\right)$
9	$\frac{2}{3}$	0	$\frac{1}{3}$	$\frac{27}{2}\widehat{\lambda}_1\widehat{\lambda}_3\left(\widehat{\lambda}_1-\frac{1}{3}\right)$
10	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	$27\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_3$

- $\mathbb{P}_4$ element:

i	$\widehat{\lambda}_1$	$\widehat{\lambda}_2$	$\widehat{\lambda}_3$	$\widehat{N}_i$
1	1	0	0	$\frac{32}{3}\widehat{\lambda}_1\left(\widehat{\lambda}_1-\frac{1}{2}\right)\left(\widehat{\lambda}_1-\frac{1}{4}\right)\left(\widehat{\lambda}_1-\frac{3}{4}\right)$
2	0	1	0	$\frac{32}{3}\widehat{\lambda}_2\left(\widehat{\lambda}_2-\frac{1}{2}\right)\left(\widehat{\lambda}_2-\frac{1}{4}\right)\left(\widehat{\lambda}_2-\frac{3}{4}\right)$
3	0	0	1	$\frac{32}{3}\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{2}\right)\left(\widehat{\lambda}_3-\frac{1}{4}\right)\left(\widehat{\lambda}_3-\frac{3}{4}\right)$
4	$\frac{3}{4}$	$\frac{1}{4}$	0	$\frac{128}{3}\widehat{\lambda}_1\widehat{\lambda}_2\left(\widehat{\lambda}_1-\frac{1}{2}\right)\left(\widehat{\lambda}_1-\frac{1}{4}\right)$
5	$\frac{2}{4}$	$\frac{2}{4}$	0	$64\widehat{\lambda}_1\widehat{\lambda}_2\left(\widehat{\lambda}_1-\frac{1}{4}\right)\left(\widehat{\lambda}_2-\frac{1}{4}\right)$
6	$\frac{1}{4}$	$\frac{3}{4}$	0	$\frac{128}{3}\widehat{\lambda}_1\widehat{\lambda}_2\left(\widehat{\lambda}_2-\frac{1}{2}\right)\left(\widehat{\lambda}_2-\frac{1}{4}\right)$
7	0	$\frac{3}{4}$	$\frac{1}{4}$	$\frac{128}{3}\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_2-\frac{1}{2}\right)\left(\widehat{\lambda}_2-\frac{1}{4}\right)$
8	0	$\frac{2}{4}$	$\frac{2}{4}$	$64\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_2-\frac{1}{4}\right)\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
9	0	$\frac{1}{4}$	$\frac{3}{4}$	$\frac{128}{3}\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{2}\right)\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
10	$\frac{1}{4}$	0	$\frac{3}{4}$	$\frac{128}{3}\widehat{\lambda}_1\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{2}\right)\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
11	$\frac{2}{4}$	0	$\frac{2}{4}$	$64\widehat{\lambda}_1\widehat{\lambda}_3\left(\widehat{\lambda}_1-\frac{1}{4}\right)\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
12	$\frac{3}{4}$	0	$\frac{1}{4}$	$\frac{128}{3}\widehat{\lambda}_1\widehat{\lambda}_3\left(\widehat{\lambda}_1-\frac{1}{2}\right)\left(\widehat{\lambda}_1-\frac{1}{4}\right)$
13	$\frac{2}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$128\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_1-\frac{1}{4}\right)$
14	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{2}{4}$	$128\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_3-\frac{1}{4}\right)$
15	$\frac{1}{4}$	$\frac{2}{4}$	$\frac{1}{4}$	$128\widehat{\lambda}_1\widehat{\lambda}_2\widehat{\lambda}_3\left(\widehat{\lambda}_2-\frac{1}{4}\right)$

2.2.5 Arbitrary elements: tetrahedra and faces

Let K be a tetrahedron of vertices $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4\}$. The affine transformation $F_K : \widehat{K} \to K$ defined as

$$\begin{aligned} F_K(\widehat{\boldsymbol{x}}) &= \begin{bmatrix} x_2-x_1 & x_3-x_1 & x_4-x_1 \\ y_2-y_1 & y_3-y_1 & y_4-y_1 \\ z_2-z_1 & z_3-z_1 & z_4-z_1 \end{bmatrix} \begin{bmatrix} \widehat{x} \\ \widehat{y} \\ \widehat{z} \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{x}_2-\boldsymbol{x}_1 & \boldsymbol{x}_3-\boldsymbol{x}_1 & \boldsymbol{x}_4-\boldsymbol{x}_1 \end{bmatrix} \widehat{\boldsymbol{x}} + \boldsymbol{x}_1 \\ &= B_K\widehat{\boldsymbol{x}} + \boldsymbol{x}_1, \end{aligned} \tag{2.12}$$

with $\boldsymbol{x}_i = (x_i, y_i, z_i)$ maps the reference tetrahedron $\widehat{K}$ into K. It is immediate that

$$F_K(\widehat{\boldsymbol{x}}_i) = \boldsymbol{x}_i,$$

i.e., F_K maps the vertices of $\widehat{K}$ to the vertices of K. The inverse mapping

$$\widehat{\boldsymbol{x}} = F_K^{-1}(\boldsymbol{x}) = B_K^{-1}(\boldsymbol{x} - \boldsymbol{x}_1)$$

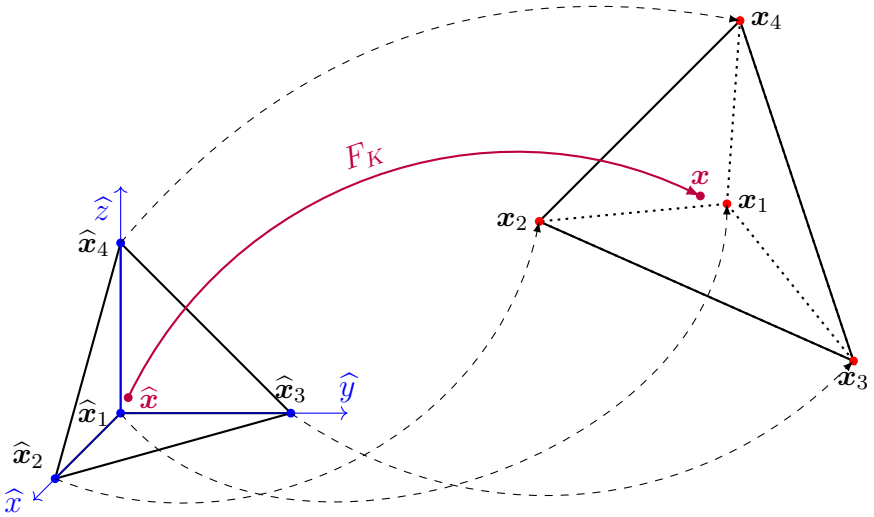


Figure 2.7: Mapping from $\widehat{K}$ to K.

exists provided that B_K is invertible. Notice that this is always the case as it only depends on the geometry of K which is assumed to be non-degenerate for proper meshes. Therefore there is a one-to-one correspondence between points in $\widehat{K}$ and K as shown in Figure 2.7.

Let us consider a point $\boldsymbol{x}$ inside K satisfying $\boldsymbol{x} = F_K(\widehat{\boldsymbol{x}})$ with barycentric coordinates given by

$$\boldsymbol{\lambda}(\boldsymbol{x}) = \lambda_1\boldsymbol{x}_1 + \lambda_2\boldsymbol{x}_2 + \lambda_3\boldsymbol{x}_3 + \lambda_4\boldsymbol{x}_4.$$

Expanding F_K yields

$$\begin{aligned}\boldsymbol{x} &= (\boldsymbol{x}_2 - \boldsymbol{x}_1)\widehat{x} + (\boldsymbol{x}_3 - \boldsymbol{x}_1)\widehat{y} + (\boldsymbol{x}_4 - \boldsymbol{x}_1)\widehat{z} + \boldsymbol{x}_1 \\ &= (1 - \widehat{x} - \widehat{y} - \widehat{z})\boldsymbol{x}_1 + \widehat{x}\boldsymbol{x}_2 + \widehat{y}\boldsymbol{x}_3 + \widehat{z}\boldsymbol{x}_4 \\ &= \widehat{\lambda}_1\boldsymbol{x}_1 + \widehat{\lambda}_2\boldsymbol{x}_2 + \widehat{\lambda}_3\boldsymbol{x}_3 + \widehat{\lambda}_4\boldsymbol{x}_4.\end{aligned}$$

Comparing the above expressions term by term reveals that the barycentric coordinates remain invariant under this transformation, that is, $\lambda_i(\boldsymbol{x}) = \widehat{\lambda}_i(\widehat{\boldsymbol{x}})$ for $i = 1, 2, 3, 4$ for any point $\boldsymbol{x}$ in K. This property is crucial, as it indicates the nodes between the two elements are linked by their barycentric coordinates. This enables us to compute the coordinates of $\boldsymbol{x}$ of a point from $\widehat{\boldsymbol{\lambda}}(\widehat{\boldsymbol{x}})$ using

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix} \begin{bmatrix} \widehat{\lambda}_1 \\ \widehat{\lambda}_2 \\ \widehat{\lambda}_3 \\ \widehat{\lambda}_4 \end{bmatrix}$$

or equivalently

$$\boldsymbol{x} = \begin{bmatrix}\boldsymbol{x}_1 & \boldsymbol{x}_2 & \boldsymbol{x}_3 & \boldsymbol{x}_4\end{bmatrix} \widehat{\boldsymbol{\lambda}}. \tag{2.13}$$

Notice also that if $p_m^K(\boldsymbol{x})$ is the mth degree polynomial in $\boldsymbol{x} = (x, y, z)$ so is $(p_m^{\widehat{K}} \circ F_K)(\widehat{\boldsymbol{x}})$ in $\widehat{\boldsymbol{x}} = (\widehat{x}, \widehat{y}, \widehat{z})$. That is, F_K maps bijectively polynomials of degree m onto polynomials of degree m. Hence, for solving a polynomial interpolation problem in K we can move to the reference

element, solve the problem there, and go back to K using the affine mapping. Indeed, we can define $\boldsymbol{v}_i^K$ to be the ith node in K satisfying

$$\boldsymbol{v}_i^K = F_K(\widehat{\boldsymbol{v}}_i), \quad i = 1, 2, \ldots, \texttt{dofK} \tag{2.14}$$

where $\{\widehat{\boldsymbol{v}}_i\}$ is the set of nodes in $\widehat{K}$ for the interpolation problem in $\mathbb{P}_m$. Only the vertices of the reference element $\widehat{K}$ need to be specified for this mapping. The rest of the nodes follows from them, however there are multiple ways to map the vertices. It is customary to select a specific mapping that ensures a positive determinant for the matrix B_K. With this we have defined the nodes in an arbitrary element in the mesh, they are just the nodes obtained from applying F_K to every tetrahedron in the mesh. Then the Lagrange basis for the polynomials $p_m^K(\boldsymbol{x})$ defined in (2.4) satisfies

$$N_i^K(\boldsymbol{x}) = \widehat{N}_i \circ F_K^{-1}(\boldsymbol{x}),$$

with $\widehat{N}_i$ the Lagrange function in $\widehat{K}$. Equivalently $\widehat{N}_i = N_i^K \circ F_K(\widehat{\boldsymbol{x}})$. Therefore the explicit computation of N_i^K is not needed, instead we can use the polynomial representation over the reference tetrahedron given in (2.10).

We can do the same for a triangular face. Let A be the triangle with vertices $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3\}$, a face of a tetrahedron $K \in \mathcal{T}_h$. In this case the function $F_A : \widehat{A} \to A$ defined as

$$F_A(\widehat{\boldsymbol{x}}) = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \\ z_2 - z_1 & z_3 - z_1 \end{bmatrix} \begin{bmatrix} \widehat{x} \\ \widehat{y} \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \tag{2.15}$$

parameterizes the face using the reference triangle $\widehat{A}$. A similar argument shows that the barycentric coordinates of a point in a triangle A remain invariant and $N_i^K|_A(\boldsymbol{x}) = \widehat{N}_i \circ F_A^{-1}(\boldsymbol{x})$.

Here an additional imposition has to be done, recall that the boundary conditions of the differential problem assumes the unit normal points outwards, therefore when performing the parameterization, the vertices should be mapped to A in such a way that the normal vector of this oriented surface also points outwards. With this mapping defined, any operation over a face of a tetrahedron can be done using the reference triangle. This prevents the inconvenience of computing the two variables polynomial from the Lagrange basis in K when restricted to a face A.

2.2.6 Finite element space revisited

We are ready to introduce the set of nodes for mth degree finite element space P_h^m on a regular tetrahedral mesh. For arbitrary m recall that we have defined the set of `nNodes` nodes given by

$$\{\boldsymbol{v}_j\}_{j=1}^{\texttt{nNodes}} = \left\{\boldsymbol{v}_i^K : K \in \mathcal{T}_h, \quad i = 1, \ldots, \texttt{dofK}\right\} \tag{2.16}$$

where $\{\boldsymbol{v}_i^K\}$ are the nodes of the mth degree interpolation problem in K (see (2.14)). Then, for $m = 1$ this is made up of the set of vertices of tetrahedra. For $m = 2$, we have the vertices and midpoints of the edges. For $m = 3$, we have vertices, two equispaced points on each edge, and the barycenters of each face. In the case of $m = 4$, we have vertices, three equispaced points on each edge, three inner points on each face, and the barycenter of the tetrahedra.

Given a node $\boldsymbol{v}_i$, let $K_1, \ldots, K_n$ be the elements to which this node belongs, then the basis

function associated to this node is

$$\varphi_i(\boldsymbol{x}) = \begin{cases} N_{i_1}^{K_1}(\boldsymbol{x}), & \boldsymbol{x} \in K_1, \\ N_{i_2}^{K_2}(\boldsymbol{x}), & \boldsymbol{x} \in K_2, \\ \vdots & \\ N_{i_n}^{K_n}(\boldsymbol{x}), & \boldsymbol{x} \in K_n, \\ 0, & \text{otherwise.} \end{cases}$$

Here i_n is the local index of node $\boldsymbol{v}_i$ in the element K_n, i.e., $F_{K_n}(\widehat{\boldsymbol{v}}_n) = \boldsymbol{v}_i$. Clearly φ_i is piecewise polynomial. Furthermore, it is continuous: if A is a common face for K_{i_r} and K_{i_s} the distribution of the interpolating nodes in A for these elements coincide which ensures $N_{i_r}^{K_r}|_A = N_{i_s}^{K_s}|_A$. Hence, we can conclude that

$$P_h^m \ni v = \sum_{j=1}^{\texttt{nNodes}} v_j \varphi_j(\boldsymbol{x}), \quad v_j = v(\boldsymbol{v}_j).$$

Notice that the evaluation of the finite element function can be done as follows: Given a point $\boldsymbol{x}$, find K, the element this point belongs to. Find its barycentric coordinates with respect to this tetrahedron ad compute

$$v(\boldsymbol{x}) = \sum_{j=1}^{\texttt{dofK}} v_j^K \widehat{N}_j(\widehat{\boldsymbol{\lambda}}(\boldsymbol{x})).$$

2.3 Finite Element Method

2.3.1 Variational Formulation

In the following, we present the variational formulation of the boundary problem for the differential problem (2.1):

$$\begin{cases} -\nabla \cdot \left(\underline{\underline{\boldsymbol{\kappa}}} \nabla u\right) + \boldsymbol{\beta} \cdot \nabla u + cu &= f, & \text{in } \Omega, \\ u &= u_D, & \text{on } \Gamma_D, \\ \left(\underline{\underline{\boldsymbol{\kappa}}} \nabla u\right) \cdot \widehat{\boldsymbol{n}} &= g_N, & \text{on } \Gamma_N, \\ \left(\underline{\underline{\boldsymbol{\kappa}}} \nabla u\right) \cdot \widehat{\boldsymbol{n}} + \alpha u &= g_R & \text{on } \Gamma_R. \end{cases}$$

To derive the variational formulation, we begin by multiplying the first equation in (2.1) by a weight function w and integrating over the whole domain Ω which yields

$$\int_\Omega -w \nabla \cdot (\underline{\underline{\boldsymbol{\kappa}}} \nabla u) + \int_\Omega w(\boldsymbol{\beta} \cdot \nabla u) + \int_\Omega cwu = \int_\Omega fw.$$

The first term in this equation can be rewritten using the identity

$$-w \nabla \cdot (\underline{\underline{\boldsymbol{\kappa}}} \nabla u) = \nabla w \cdot (\underline{\underline{\boldsymbol{\kappa}}} \nabla u) - \nabla \cdot (w \underline{\underline{\boldsymbol{\kappa}}} \nabla u),$$

which can be derived from the product rule. Substituting this expression in the integral equation results in

$$\int_\Omega \nabla w \cdot (\underline{\underline{\boldsymbol{\kappa}}} \nabla u) - \int_\Omega \nabla \cdot (w \underline{\underline{\boldsymbol{\kappa}}} \nabla u) + \int_\Omega w(\boldsymbol{\beta} \cdot \nabla u) + \int_\Omega cwu = \int_\Omega fw.$$

Using the divergence theorem in the second term of the above expression yields

$$\int_\Omega \nabla \cdot (w\underline{\underline{\boldsymbol{\kappa}}}\nabla u) = \int_\Gamma w(\underline{\underline{\boldsymbol{\kappa}}}\nabla u)\cdot \widehat{\boldsymbol{n}}.$$

Notice that with this operation we are getting rid of the second order derivatives appearing in the differential formulation and keeping only first order derivatives. To further simplify the expression, we can decompose the boundary of the domain Ω into the Dirichlet, Neumann, and Robin boundaries Γ_D, Γ_N, and Γ_R respectively.

Applying the boundary condition on Γ_R and Γ_N and setting $w = 0$ on Γ_D yields the variational (weak) formulation of the problem: If u solves the differential problem then

$$\begin{cases} u = u_D, \qquad \text{on } \Gamma_D \\ \underbrace{\displaystyle\int_\Omega \nabla w\cdot(\underline{\underline{\boldsymbol{\kappa}}}\nabla u) + \int_{\Gamma_R} \alpha w u + \int_\Omega w(\boldsymbol{\beta}\cdot\nabla u) + \int_\Omega cwu}_{=:a(u,w)} = \underbrace{\displaystyle\int_\Omega fw + \int_{\Gamma_R} g_R w + \int_{\Gamma_N} g_N w}_{=:\ell(w)}, \\ \qquad\qquad \forall w \in \{v : v = 0 \text{ on } \Gamma_D\}. \end{cases} \tag{2.17}$$

By imposing this constraint on the weight function w, it becomes necessary to enforce the Dirichlet conditions on the boundaries. Consequently, in this context, these conditions are commonly referred to as *essential* conditions. On the other hand, Robin and Neumann conditions are known as *natural* conditions since they are a consequence of the variational formulation.

It is worth noting that, while the variational formulation can have a solution, it may not necessarily satisfy the regularity requirements of the original problem. For instance, the *solution* may not have a continuous second derivative, which is required for the strong formulation. Hence, the name "weak" formulation, as it imposes fewer requirements for the solution. Nonetheless, some level of regularity is still needed for the weak formulation. In particular, the solution must have continuous first derivatives due to the presence of the gradient.

2.3.2 Discretization

The finite element method consists, in a nutshell, in replacing the continuous functions appearing in (2.17) by functions in P_h^m:

$$\begin{cases} P_h^m \ni u_h \approx u_D, \qquad \text{on } \Gamma_D \\ \underbrace{\displaystyle\int_\Omega \nabla w_h\cdot(\underline{\underline{\boldsymbol{\kappa}}}\nabla u_h) + \int_{\Gamma_R} \alpha w_h u_h + \int_\Omega w_h(\boldsymbol{\beta}\cdot\nabla u_h) + \int_\Omega cw_h u_h}_{=:a(u_h,w_h)} \\ \qquad\qquad = \underbrace{\displaystyle\int_\Omega fw_h + \int_{\Gamma_R} g_R w_h + \int_{\Gamma_N} g_N w_h}_{=:\ell(w_h)}, \\ \qquad\qquad \forall w_h \in \{v_h \in P_h^m : v_h = 0 \text{ on } \Gamma_D\}. \end{cases} \tag{2.18}$$

This *discretizes* the problem in the following sense: to solve (2.18), i.e., compute u_h, we just have to find the values of u_h at the non-Dirichlet nodes. Therefore the number of unknowns

is finite, which are determined by demanding the solution to satisfy a test when multiplied by the elements of (a subspace of) P_h^m, a finite-dimensional space.

Let us define `iD` and `inD` to be the vectors containing the global indices of Dirichlet (where the exact solution is known) and non-Dirichlet (where it is not) nodes respectively. That is, `iD` and `inD` are a partition of the indices of nodes,

$$\texttt{iD} \cup \texttt{inD} = \{1, 2, \ldots, \texttt{nNodes}\}, \quad \texttt{iD} \cap \texttt{inD} = \emptyset,$$

and

$$j \in \texttt{iD} \text{ if and only if } \boldsymbol{v}_j \in \Gamma_D.$$

The Finite Element Method seeks $u_h \in P_h^m$

$$u \approx u_h = u_{h,D} + u_{h,nD} = \sum_{k \in \texttt{iD}} u_k \varphi_k + \sum_{j \in \texttt{inD}} u_j \varphi_j \tag{2.19}$$

where $u_{h,D}(\boldsymbol{v}_j) = u_D(\boldsymbol{v}_j)$ for nodes $\boldsymbol{v_j}$ in Γ_D and zero at every other node. To obtain the non-Dirichlet nodal value the Galerkin method is used, where the weight function w is chosen from the basis $\{\varphi_i\}_{i \in \texttt{inD}}$. Therefore, the discretized version of the variational formulation seeks to find $u_{h,nD}$ satisfying

$$\begin{cases} u_h = u_{h,D}, & \forall \boldsymbol{v}_j \in \Gamma_D \\ a(u_{h,nD}, \varphi_i) = \ell(\varphi_i) - a(u_{h,D}, \varphi_i), & \forall i \in \texttt{inD}. \end{cases} \tag{2.20}$$

This means that the approximation of the solution u via the Finite Element Method transforms the problem to the resolution of the linear system

$$\sum_{j \in \texttt{inD}} (\boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},ij} + \boldsymbol{R}_{\alpha,ij} + \boldsymbol{A}_{\boldsymbol{\beta},ij} + \boldsymbol{M}_{c,ij}) u_j = (\boldsymbol{b}_{f,i} + \boldsymbol{t}_{n,i} + \boldsymbol{t}_{r,i}) - \sum_{k \in \texttt{iD}} (\boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},ik} + \boldsymbol{R}_{\alpha,ik} + \boldsymbol{A}_{\boldsymbol{\beta},ik} + \boldsymbol{M}_{c,ik}) u_k, \tag{2.21}$$

with $i \in \texttt{inD}$ and where

$$\begin{aligned} \boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},ij} &= \int_\Omega \nabla \varphi_i \cdot (\underline{\underline{\boldsymbol{\kappa}}} \nabla \varphi_j), & \boldsymbol{R}_{\alpha,ij} &= \int_{\Gamma_R} \alpha \varphi_i \varphi_j, \\ \boldsymbol{M}_{c,ij} &= \int_\Omega c \varphi_i \varphi_j, & \boldsymbol{A}_{\boldsymbol{\beta},ij} &= \int_\Omega \varphi_i (\boldsymbol{\beta} \cdot \nabla \varphi_j), \\ \boldsymbol{b}_{f,i} &= \int_\Omega f \varphi_i, & \boldsymbol{t}_{n,i} &= \int_{\Gamma_N} g_N \varphi_i, \\ \boldsymbol{t}_{r,i} &= \int_{\Gamma_R} g_R \varphi_i. \end{aligned}$$

The choice of the basis defined in (2.2.2) plays a crucial role in the efficiency of the method. In particular, the use of piecewise polynomial functions with compact support (nonzero in some tetrahedrons) over others yields a matrix where most of the entries are zero, known in the literature as **sparse** matrix. For example, $\boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},ij}$ and $\boldsymbol{M}_{c,ij}$ vanish unless the nodes $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$ belongs to the same tetrahedron K.

This sparsity property allows for significant savings in computational time and memory storage, and can be exploited by specially designed solvers to further enhance the efficiency of the method.

Note. It is possible to implement the Dirichlet conditions as a natural condition. Indeed, define on Γ_D the boundary condition

$$(\underline{\underline{\boldsymbol{\kappa}}}\nabla u)\cdot\widehat{\boldsymbol{n}}+\alpha u=\tilde{\alpha}u_D,$$

with $\alpha,\tilde{\alpha}\gg(\underline{\underline{\boldsymbol{\kappa}}}\nabla u)\cdot\widehat{\boldsymbol{n}}$. This results in approximating the value of u at the nodes with index `iD` with $u\approx u_D$.

2.3.3 System Assembly

In this section, we will discuss the general ideas behind the assembly process of the linear system, including the construction of local matrices and vectors, the use of quadrature rules to approximate integrals, and the assembly of the global system. To this end, we will provide a thorough explanation of the assembly of the mass matrix, while for the other terms, we will only provide details on any new computation required.

Mass matrix

Let us consider the computation of the element (i,j) of the mass matrix

$$\boldsymbol{M}_{c,ij}=\int_\Omega c\varphi_i\varphi_j.$$

This term integrates the basis functions associated to the nodes with global index i and j. From the linearity of the integral we obtain

$$\boldsymbol{M}_{c,ij}=\sum_{K\in\mathcal{T}_h}\int_K c\varphi_i\varphi_j.$$

As already observed, the only terms that contribute to this sum are those tetrahedrons containing both nodes i and j. Let us assume that one such term corresponds to the tetrahedron K. Furthermore, let us assume that the global nodes i and j are associated with the rth and sth local nodes, respectively, within the reference tetrahedron $\widehat{K}$. Applying the change of variables $\boldsymbol{x}=F_K(\widehat{\boldsymbol{x}})$ yields

$$\int_K cN_r^KN_s^K=\int_{\widehat{K}}(cN_r^KN_s^K\circ F_K)|\det\boldsymbol{J}_K|.$$

Here $\boldsymbol{J}_K$ is the Jacobian matrix associated to the tetrahedron K defined as

$$\boldsymbol{J}_K=\begin{bmatrix}\frac{\partial x}{\partial\widehat{x}}&\frac{\partial x}{\partial\widehat{y}}&\frac{\partial x}{\partial\widehat{z}}\\[2pt]\frac{\partial y}{\partial\widehat{x}}&\frac{\partial y}{\partial\widehat{y}}&\frac{\partial y}{\partial\widehat{z}}\\[2pt]\frac{\partial z}{\partial\widehat{x}}&\frac{\partial z}{\partial\widehat{y}}&\frac{\partial z}{\partial\widehat{z}}\end{bmatrix}\tag{2.22}$$

To obtain the Jacobian of F_K, the expression (2.12) can be expanded:

$$\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}(x_2-x_1)\widehat{x}+(x_3-x_1)\widehat{y}+(x_4-x_1)\widehat{z}+x_1\\(y_2-y_1)\widehat{x}+(y_3-y_1)\widehat{y}+(y_4-y_1)\widehat{z}+y_1\\(z_2-z_1)\widehat{x}+(z_3-z_1)\widehat{y}+(z_4-z_1)\widehat{z}+z_1\end{bmatrix}$$

and computing the derivatives yields

$$\boldsymbol{J}_K = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 & x_4 - x_1 \\ y_2 - y_1 & y_3 - y_1 & y_4 - y_1 \\ z_2 - z_1 & z_3 - z_1 & z_4 - z_1 \end{bmatrix},$$

that is, the Jacobian is exactly the matrix B_K defined for the affine mapping.

From section 2.2.5 recall that $N_r^K \circ F_K = \widehat{N}_r$. Therefore the integral we have to solve for this element is

$$|\det B_K| \int_{\widehat{K}} (c \circ F_K) \widehat{N}_r \widehat{N}_s. \tag{2.23}$$

An important consequence of this result is that we can evaluate all integrals in the reference element $\widehat{K}$. This eliminates the need to compute different integration limits for each $K \in \mathcal{T}_h$ and provides a consistent approach across all elements.

However, the main challenge lies in evaluating the function $(c \circ F_K)$, which may not be a simple polynomial function. Analytically computing these integrals might not be feasible in many cases. To overcome this challenge, we can use quadrature (or cubature) formulas to approximate these integrals.

A quadrature formula for the integral of a function $h = h(\boldsymbol{x})$ over a simplex T takes the form [12]

$$\int_T h \approx |T| \sum_{n=1}^{\mathtt{nQ}_D} \omega_n h(\boldsymbol{\lambda}_{T,n})$$

where $|T|$ is the area or volume of T. Clearly, for the references elements, $|\widehat{K}| = \frac{1}{6}$ (tetrahedra) and $|\widehat{A}| = \frac{1}{2}$ (triangle) respectively. The weights $\{\omega_n\}_{n=1}^{\mathtt{nQ}_D}$ satisfy the condition $\sum_{n=1}^{\mathtt{nQ}_D} \omega_n = 1$. Clearly, if $h = 1$ we obtain the area or volume of T. The integrand is evaluated at $\mathtt{nQ}_D$ nodes $\boldsymbol{\lambda}_{T,n}$ which are given in barycentric coordinates. For general functions h, the linear mapping (2.13) can be used to retrieve the coordinates of the nodes in Ω from the barycentric coordinates avoiding the need to invert the affine mapping. Using this to compute (2.23) yields

$$|\det B_K| \int_{\widehat{K}} (c \circ F_K) \widehat{N}_r \widehat{N}_s \approx \frac{1}{6} |\det B_K| \sum_{n=1}^{\mathtt{nQ}} c(\begin{bmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \boldsymbol{x}_3 & \boldsymbol{x}_4 \end{bmatrix} \boldsymbol{\lambda}_{T,n})(\widehat{N}_r \widehat{N}_s)(\boldsymbol{\lambda}_{T,n}).$$

This process should be performed for all elements K containing both nodes i and j, however, locating these elements is time-consuming. To get around this let us define the local mass matrix: for each tetrahedra $K \in \mathcal{T}_h$, let

$$\boldsymbol{M}_c^K = \begin{bmatrix} \int_K c\varphi_{i_1}\varphi_{i_1} & \cdots & \int_K c\varphi_{i_1}\varphi_{i_{\mathtt{dofK}}} \\ \vdots & \ddots & \vdots \\ \int_K c\varphi_{i_{\mathtt{dofK}}}\varphi_{i_1} & \cdots & \int_K c\varphi_{i_{\mathtt{dofK}}}\varphi_{i_{\mathtt{dofK}}} \end{bmatrix} = \begin{bmatrix} \int_K cN_1^K N_1^K & \cdots & \int_K cN_1^K N_{\mathtt{dofK}}^K \\ \vdots & \ddots & \vdots \\ \int_K cN_{\mathtt{dofK}}^K N_1^K & \cdots & \int_K cN_{\mathtt{dofK}}^K N_{\mathtt{dofK}}^K \end{bmatrix}$$

This local mass matrix has the advantage of including all the non-zero elements associated with tetrahedron K required for the mass matrix $\boldsymbol{M}_c$. The assembly of this matrix is summarized in the algorithm 1.

for $K \subset \Omega$ **do**
  Compute $\boldsymbol{M}_c^K$
  for $i = 1, \ldots,$ `dofK` **do**
    for $j = 1, \ldots,$ `dofK` **do**
      $\boldsymbol{M}_c(\boldsymbol{i}^K(i), \boldsymbol{i}^K(j)) = \boldsymbol{M}_c(\boldsymbol{i}^K(i), \boldsymbol{i}^K(j)) + \boldsymbol{M}_c^K(i,j)$
    end
  end
end

Algorithm 1: Mass matrix assembly.

Instead of locating the positions of the nodes in the mesh, we iterate through each tetrahedron. For each element, we compute the integrals of the basis functions associated with the nodes belonging to that element. To this end, we use the ordered row vector

$$\boldsymbol{i}^K = \begin{bmatrix} i_1 & \cdots & i_{\texttt{dofK}} \end{bmatrix}^\top \tag{2.24}$$

containing the global indices of the nodes in K following the local indexing. In other words,

$$F_K(\widehat{\boldsymbol{v}}_\ell) = \boldsymbol{v}_{i_\ell}, \quad \ell = 1, \ldots, \texttt{dofK}.$$

This allows us to place each term of the local matrix in its correct position within the global matrix. Note that by following this approach, once all elements containing a pair of nodes have been traversed, we will have computed the integrals associated with the corresponding basis functions.

Stiffness Matrix

The local stiffness matrix is defined as

$$\boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}}}^K = \begin{bmatrix} \int_K \nabla\varphi_{i_1} \cdot (\underline{\underline{\boldsymbol{\kappa}}}\nabla\varphi_{i_1}) & \cdots & \int_K \nabla\varphi_{i_1} \cdot (\underline{\underline{\boldsymbol{\kappa}}}\nabla\varphi_{i_{\texttt{dofK}}}) \\ \vdots & \ddots & \vdots \\ \int_K \nabla\varphi_{i_{\texttt{dofK}}} \cdot (\underline{\underline{\boldsymbol{\kappa}}}\nabla\varphi_{i_1}) & \cdots & \int_K \nabla\varphi_{i_{\texttt{dofK}}} \cdot (\underline{\underline{\boldsymbol{\kappa}}}\nabla\varphi_{i_{\texttt{dofK}}}) \end{bmatrix}$$

Let us choose the element of this matrix associated to the rth and sth local nodes. The integral we have to compute is

$$\int_K \nabla N_r^K \cdot (\underline{\underline{\boldsymbol{\kappa}}}\nabla N_s^K).$$

Applying the affine mapping to transform the integration over the reference tetrahedron gives:

$$\int_K \nabla N_r^K \cdot (\underline{\underline{\boldsymbol{\kappa}}}\nabla N_s^K) = |\det B_K| \int_{\widehat{K}} \nabla \widehat{N}_s \cdot \left[(\underline{\underline{\boldsymbol{\kappa}}} \circ F_K) \nabla \widehat{N}_s \right].$$

The derivatives of $\widehat{N}_k$ in the reference system can be obtained using the chain rule

$$\begin{aligned}
\frac{\partial \widehat{N}_k}{\partial \widehat{x}} &= \frac{\partial \widehat{N}_k}{\partial x}\frac{\partial x}{\partial \widehat{x}} + \frac{\partial \widehat{N}_k}{\partial y}\frac{\partial y}{\partial \widehat{x}} + \frac{\partial \widehat{N}_k}{\partial z}\frac{\partial z}{\partial \widehat{x}}, \\
\frac{\partial \widehat{N}_k}{\partial \widehat{y}} &= \frac{\partial \widehat{N}_k}{\partial x}\frac{\partial x}{\partial \widehat{y}} + \frac{\partial \widehat{N}_k}{\partial y}\frac{\partial y}{\partial \widehat{y}} + \frac{\partial \widehat{N}_k}{\partial z}\frac{\partial z}{\partial \widehat{y}}, \\
\frac{\partial \widehat{N}_k}{\partial \widehat{z}} &= \frac{\partial \widehat{N}_k}{\partial x}\frac{\partial x}{\partial \widehat{z}} + \frac{\partial \widehat{N}_k}{\partial y}\frac{\partial y}{\partial \widehat{z}} + \frac{\partial \widehat{N}_k}{\partial z}\frac{\partial z}{\partial \widehat{z}},
\end{aligned}$$

or in matrix notation

$$\widehat{\nabla}\widehat{N}_k = \nabla\widehat{N}_k \boldsymbol{J}_K \tag{2.25}$$

Here we are now using the usual matrix algebra notation that the gradient operator ∇ can be understood as a row vector. Since $\boldsymbol{J}_K = B_K$,

$$\nabla\widehat{N}_k = \widehat{\nabla}\widehat{N}_k B_K^{-1}.$$

The inverse of B_K can be computed using cross products:

$$B_K^{-1} = \frac{1}{\det B_K}\begin{bmatrix}[(\boldsymbol{x}_3-\boldsymbol{x}_1)\times(\boldsymbol{x}_4-\boldsymbol{x}_1)]^\top \\ [(\boldsymbol{x}_4-\boldsymbol{x}_1)\times(\boldsymbol{x}_2-\boldsymbol{x}_1)]^\top \\ [(\boldsymbol{x}_2-\boldsymbol{x}_1)\times(\boldsymbol{x}_3-\boldsymbol{x}_1)]^\top\end{bmatrix} \tag{2.26}$$

Note. This representation of the inverse will be relevant for vectorizing its computation.

Substituting the result obtained for the gradient in the integral above yields

$$|\det B_K|\int_{\hat{K}} \widehat{\nabla}\widehat{N}_r B_K^{-1}(\underline{\underline{\boldsymbol{\kappa}}}\circ F_K)B_K^{-\top}\widehat{\nabla}\widehat{N}_s^\top = \int_{\hat{K}} \widehat{\nabla}\widehat{N}_r\, C_K\, \widehat{\nabla}\widehat{N}_s^\top.$$

where

$$C_K = |\det B_K|\cdot B_K^{-1}(\underline{\underline{\boldsymbol{\kappa}}}\circ F_K)B_K^{-\top}$$

is a 3×3 symmetric matrix, as $C_K^\top = B_K^{-\top}(\underline{\underline{\boldsymbol{\kappa}}}\circ F_K)^\top B_K^{-1} = B_K^{-\top}(\underline{\underline{\boldsymbol{\kappa}}}\circ F_K)B_K^{-1} = C_K$. The matrix product can be expanded to give

$$\begin{aligned}\int_{\hat{K}} \widehat{\nabla}\widehat{N}_\ell C_K \widehat{\nabla}\widehat{N}_s^\top = &\int_{\hat{K}} \widehat{c}_{11}^K \frac{\partial\widehat{N}_r}{\partial\widehat{x}}\frac{\partial\widehat{N}_s}{\partial\widehat{x}} + \int_{\hat{K}} \widehat{c}_{12}^K\left(\frac{\partial\widehat{N}_r}{\partial\widehat{x}}\frac{\partial\widehat{N}_s}{\partial\widehat{y}} + \frac{\partial\widehat{N}_r}{\partial\widehat{y}}\frac{\partial\widehat{N}_s}{\partial\widehat{x}}\right)\\ &+\int_{\hat{K}} \widehat{c}_{13}^K\left(\frac{\partial\widehat{N}_r}{\partial\widehat{x}}\frac{\partial\widehat{N}_s}{\partial\widehat{z}} + \frac{\partial\widehat{N}_r}{\partial\widehat{z}}\frac{\partial\widehat{N}_s}{\partial\widehat{x}}\right) + \int_{\hat{K}} \widehat{c}_{22}^K\frac{\partial\widehat{N}_r}{\partial\widehat{y}}\frac{\partial\widehat{N}_s}{\partial\widehat{y}}\\ &+\int_{\hat{K}} \widehat{c}_{23}^K\left(\frac{\partial\widehat{N}_r}{\partial\widehat{y}}\frac{\partial\widehat{N}_s}{\partial\widehat{z}} + \frac{\partial\widehat{N}_\ell}{\partial\widehat{z}}\frac{\partial\widehat{N}_s}{\partial\widehat{y}}\right) + \int_{\hat{K}} \widehat{c}_{33}^K\frac{\partial\widehat{N}_r}{\partial\widehat{z}}\frac{\partial\widehat{N}_s}{\partial\widehat{z}},\end{aligned}$$

where $\widehat{c}_{ij}^K$ are the elements of the matrix C_K. Therefore the local stiffness matrix can be written as

$$\boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}}}^K = \boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},xx}^K + \boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},xy}^K + \boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},yx}^K + \boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},xz}^K + \boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},zx}^K + \boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},yy}^K + \boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},yz}^K + \boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},zy}^K + \boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},zz}^K$$

where

$$\boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}},x_ix_j}^K = |\det B_K|\left(\int_{\hat{K}} \widehat{c}_{ij}^K\frac{\partial\widehat{N}_\ell}{\partial\widehat{x}_i}\frac{\partial\widehat{N}_s}{\partial\widehat{x}_j}\right)_{r,s=1,\ldots,\texttt{dofK}},$$

for $x_i, x_j \in \{x, y, z\}$ are the matrices containing in the integrand the products of derivatives. Notice that from the symmetry of this matrix it follows that $\boldsymbol{S}_{x_ix_j} = \boldsymbol{S}_{x_jx_i}^\top$.

The assembly process is shown in Algorithm 2.

for $K \subset \Omega$ **do**
  Compute $C_K = |\det B_K| \cdot B_K^{-1}(\underline{\underline{\boldsymbol{\kappa}}} \circ F_K) B_K^{-\top}$
  Compute $\boldsymbol{S}^K_{\underline{\underline{\boldsymbol{\kappa}}}}$
  for $i = 1, \ldots,$ `dofK` **do**
    for $j = 1, \ldots,$ `dofK` **do**
      $\boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}}}(\boldsymbol{i}^K(i), \boldsymbol{i}^K(j)) = \boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}}}(\boldsymbol{i}^K(i), \boldsymbol{i}^K(j)) + \boldsymbol{S}^K_{\underline{\underline{\boldsymbol{\kappa}}}}(i,j)$
    end
  end
end

Algorithm 2: Stiffness matrix assembly.

The steps are now essentially the same, the only key difference is that for the assembly of the local matrix, more terms have to be computed. The integrals are again computed using cubature rules.

Advection Matrix

The local advection matrix is defined as:

$$\boldsymbol{A}^K_{\boldsymbol{\beta}} = \begin{bmatrix} \int_K \varphi_{i_1}(\boldsymbol{\beta} \cdot \nabla \varphi_{i_1}) & \cdots & \int_K \varphi_{i_1}(\boldsymbol{\beta} \cdot \nabla \varphi_{i_{\texttt{dofK}}}) \\ \vdots & \ddots & \vdots \\ \int_K \varphi_{i_{\texttt{dofK}}}(\boldsymbol{\beta} \cdot \nabla \varphi_{i_1}) & \cdots & \int_K \varphi_{i_{\texttt{dofK}}}(\boldsymbol{\beta} \cdot \nabla \varphi_{i_{\texttt{dofK}}}) \end{bmatrix}$$

Taking the rth and sth nodes of this matrix and moving to the reference element yields

$$\int_K N_r(\nabla N_s \boldsymbol{\beta}) = |\det B_K| \int_{\widehat{K}} \widehat{N}_r \left[\widehat{\nabla} \widehat{N}_s B_K^{-1}(\boldsymbol{\beta} \circ F_K) \right]$$

where we can denote $\boldsymbol{a}^K = |\det B_K| \cdot B_K^{-1}(\boldsymbol{\beta} \circ F_K)$, a column vector with components $\widehat{a}_i^K$. Expanding the matrix product gives

$$\int_K \widehat{N}_r \widehat{\nabla} \widehat{N}_s \boldsymbol{a}^K = \int_{\widehat{K}} \widehat{a}_1^K \widehat{N}_r \frac{\partial \widehat{N}_s}{\partial \widehat{x}} + \int_{\widehat{K}} \widehat{a}_2^K \widehat{N}_r \frac{\partial \widehat{N}_s}{\partial \widehat{y}} + \int_{\widehat{K}} \widehat{a}_3^K \widehat{N}_r \frac{\partial \widehat{N}_s}{\partial \widehat{z}}.$$

Therefore, the local advection matrix can be written as

$$\boldsymbol{A}^K_{\boldsymbol{\beta}} = \boldsymbol{A}^K_{\boldsymbol{\beta},x} + \boldsymbol{A}^K_{\boldsymbol{\beta},y} + \boldsymbol{A}^K_{\boldsymbol{\beta},z}$$

where

$$\boldsymbol{A}^K_{\boldsymbol{\beta},x_i} = |\det B_K| \left(\int_{\widehat{K}} \widehat{a}_i^K \widehat{N}_r \frac{\partial \widehat{N}_s}{\partial \widehat{x}_i} \right)_{r,s=1,\ldots,\texttt{dofK}}.$$

for $K \subset \Omega$ **do**
  Compute $\boldsymbol{a}^K = |\det B_K| \cdot B_K^{-1}(\boldsymbol{\beta} \circ F_K)$
  Compute $\boldsymbol{A}_{\boldsymbol{\beta}}^K$
  for $i = 1, \ldots,$ `dofK` **do**
    for $j = 1, \ldots,$ `dofK` **do**
      $\boldsymbol{A}_{\boldsymbol{\beta}}(\boldsymbol{i}^K(i), \boldsymbol{i}^K(j)) = \boldsymbol{A}_{\boldsymbol{\beta}}(\boldsymbol{i}^K(i), \boldsymbol{i}^K(j)) + \boldsymbol{A}_{\boldsymbol{\beta}}^K(i,j))$
    end
  end
end

Algorithm 3: Advection matrix.

Source term

The local source vector is given by

$$\boldsymbol{b}_f^K = \begin{bmatrix} \int_K f\varphi_{i_1} \\ \vdots \\ \int_K f\varphi_{i_{\texttt{dofK}}} \end{bmatrix}$$

The process is equivalent to the assembly of the mass matrix.

for $K \subset \Omega$ **do**
  Compute $\boldsymbol{b}_f^K$
  for $i = 1, \ldots,$ `dofK` **do**
    $\boldsymbol{b}_f(\boldsymbol{i}^K(i)) = \boldsymbol{b}_f(\boldsymbol{i}^K(i)) + \boldsymbol{b}_f^K(i)$
  end
end

Algorithm 4: Source vector assembly.

In practical applications it is rather common to have $f = \text{const}$, in this case the assembly of $\boldsymbol{b}_f$ can be done from the mass matrix $\boldsymbol{M}_{\text{const}}^K$ noticing that for the ith row of this matrix, if we add up all the terms we get

$$\sum_j \int_K \text{const}\ \varphi_i\varphi_j = \text{const} \int_K \varphi_i,$$

which is exactly the ith element of the source term. To verify this notice that adding up $p = \sum_{i=1}^{\texttt{dofK}} N_i^K$ yields a polynomial of degree m passing through the value 1 `dofK` times. Due to the uniqueness of polynomials, such polynomial must be the constant function $p = 1$.

Boundary Mass Matrix

For the boundary terms the only difference is that the integration are done over the triangles in Γ. Let us consider the element (i,j) of the boundary mass matrix and split it into the triangles conforming the Robin matrix

$$\boldsymbol{R}_{\alpha,ij} = \sum_{A \subset \Gamma_R} \int_A \alpha\varphi_i\varphi_j.$$

Considering one of the faces A containing both nodes i and j associated to the rth and sth local nodes in the reference triangle $\widehat{A}$. Similarly to the mass matrix case, after applying the surface parameterization $\boldsymbol{x} = F_A(\widehat{\boldsymbol{x}})$ yields the integral

$$\int_A \alpha N_r^A N_s^A = \int_{\widehat{A}} (\alpha \circ F_A)\widehat{N}_r\widehat{N}_s \left\| \frac{\partial F_A}{\partial \widehat{x}} \times \frac{\partial F_A}{\partial \widehat{y}} \right\|.$$

Computing the partial derivatives yields

$$\frac{\partial F_A}{\partial \widehat{x}} = \boldsymbol{x}_2 - \boldsymbol{x}_1,$$
$$\frac{\partial F_A}{\partial \widehat{y}} = \boldsymbol{x}_3 - \boldsymbol{x}_1,$$

which are precisely the edges of the boundary element. Therefore, the cross product yields the normal vector to the face A with module twice its area.

We can define the local mass boundary matrix as

$$\boldsymbol{R}_\alpha^A = \begin{bmatrix} \int_A \alpha\varphi_{i_1}\varphi_{i_1} & \cdots & \int_A \alpha\varphi_{i_1}\varphi_{i_{\texttt{dofA}}} \\ \vdots & \ddots & \vdots \\ \int_A \alpha\varphi_{i_{\texttt{dofA}}}\varphi_{i_1} & \cdots & \int_A \alpha\varphi_{i_{\texttt{dofA}}}\varphi_{i_{\texttt{dofA}}} \end{bmatrix}$$

As for the tetrahedral elements we can define a vector containing the global indices of the nodes belonging to the boundary element A as

$$\boldsymbol{i}^A = \begin{bmatrix} i_1 & \cdots & i_{\texttt{dofA}} \end{bmatrix}^\top. \tag{2.27}$$

The algorithm for the assembly of this matrix is shown below (see algorithm 5).

for $A \subset \Gamma_R$ **do**
  Compute $\boldsymbol{R}_\alpha^A$
  for $i = 1, \ldots, \texttt{dofA}$ **do**
    for $j = 1, \ldots, \texttt{dofA}$ **do**
      $\boldsymbol{R}_\alpha(\boldsymbol{i}^A(i), \boldsymbol{i}^A(j)) = \boldsymbol{R}_\alpha(\boldsymbol{i}^A(i), \boldsymbol{i}^A(j)) + \boldsymbol{R}_\alpha^A(i,j)$
    end
  end
end

Algorithm 5: Boundary mass matrix assembly.

Traction vector

The Robin and Neumann traction vectors are obtained exactly the same, as they are both related to the normal derivative in the boundary condition. The local traction vector can be defined as

$$\boldsymbol{t}_n^A = \begin{bmatrix} \int_A g\varphi_{i_1} \\ \vdots \\ \int_A g\varphi_{i_{\texttt{dofA}}} \end{bmatrix}$$

The assembly of this term is equivalent to the one for the boundary mass matrix and is summarized in the algorithm 6.

for $A \subset \Gamma_R$ *or* $A \subset \Gamma_N$ **do**
  Compute $\boldsymbol{t}_g^A$
  for $i = 1, \ldots, \texttt{dofA}$ **do**
    $\boldsymbol{t}_g(\boldsymbol{i}^A(i)) = \boldsymbol{t}_g(\boldsymbol{i}^A(i)) + \boldsymbol{t}_g^A(i)$
  end
end

Algorithm 6: Traction vector assembly.

Once again in practical applications it is usual to have constant Neumann or Robin data, in which case the traction vector can be computed from $\boldsymbol{R}_{\text{const}}$ by adding up all the columns

$$\sum_j \int_A \text{const}\, \varphi_i \varphi_j = \text{const} \int_A \varphi_i.$$

2.4 Further Applications

In the previous sections, we discussed the fundamentals of the finite element method (FEM). Now, we will introduce some necessary modifications to solve more complex problems numerically. Specifically, we will focus on an evolutionary problem and linear elasticity. It is important to note that these examples serve as illustrations to demonstrate general guidelines that can be applied to solve a wide range of problems.

2.4.1 Evolution Problems

An evolution problem is a partial differential equation that includes a temporal derivative, indicating that the system being studied has a transient behavior. A typical example is the heat conduction equation

$$\rho C_p \frac{\partial u}{\partial t} - \nabla \cdot \left(\underline{\underline{\boldsymbol{\kappa}}} \nabla u\right) = f,$$

where ρ is the density of the material and C_p its thermal capacity. This equation models the temperature $u(x, y, z, t)$ of each point (x, y, z) in the domain at time t ranging from $t = 0$ to $t = t_f$. To solve this problem we need once more the boundary conditions. For instance, let us consider the case where we have fixed temperature at the boundary

$$u(\boldsymbol{x}, t) = u_D(\boldsymbol{x}, t), \quad \boldsymbol{x} \in \Gamma.$$

Additionally, an initial condition, in particular a temperature distribution at $t = 0$ is needed:

$$u\left(\boldsymbol{x}, 0\right) = u_0, \qquad \forall \boldsymbol{x} \in \Omega.$$

To solve this problem numerically, a suitable time integrator is required to solve the evolution part of the problem coupled with a spatial discretization to handle the spatial derivatives and the boundary conditions. Building on the concepts of the FEM, the method can be extended by assuming that the mesh remains the same over the entire time interval. Therefore, the solution of the problem can be approximated as

$$u_h = \sum_{j=1}^{\texttt{nNodes}} u_j\left(t\right) \varphi_j,$$

where the values at the nodes u_j may change over time. This representation decomposes the function u_h into two different components: the basis functions, which depend only on the spatial coordinates, and the coefficients, which vary over time. Applying the same ideas presented in Section 2.3.1, the discretized variational formulation includes a new term due to the temporal derivative of the form:

$$\boldsymbol{M}_{\rho C_p, ij} \dot{u}_j(t) = \int_\Omega \rho C_p \varphi_i \varphi_j \dot{u}_j(t).$$

where $\dot{u}_j(t)$ is the approximation of $\frac{\partial u}{\partial t}$ at the jth node in the instant t. Applying the FEM discretization, we arrive at the linear system of equations

$$\begin{cases} \boldsymbol{M}_{\rho C_p}(t)\dot{\boldsymbol{u}} + \boldsymbol{A}_{\underline{\underline{\boldsymbol{\kappa}}}}(t)\,\boldsymbol{u} = \boldsymbol{b}_f(t)\,, \\ \boldsymbol{u}(0) = \boldsymbol{u}_{0,h}, \\ \boldsymbol{u} = \boldsymbol{u}_D(t), \qquad \boldsymbol{x} \in \Gamma, \end{cases}$$

where $\boldsymbol{u} = \left[u_1(t), u_2(t), \ldots, u_{\texttt{nNodes}}(t)\right]$ is the vector containing the time dependent functions at the nodes, $\boldsymbol{u}_{0,h}$ is the discretized initial condition. In this equation we have assumed that the diffusion and mass matrices and the source term are in general functions of time.

Two commonly used methods to numerically solve this problem are the implicit Euler method and the Crank-Nicolson method [24]. The implicit Euler method is a first-order scheme obtained by applying a finite difference scheme to the time derivative and evaluating the remaining terms at the next time step. This yields the following equation:

$$\boldsymbol{M}_{\rho C_p}(t_{n+1})\,\frac{\boldsymbol{u}_{n+1}-\boldsymbol{u}_n}{\tau_n} + \boldsymbol{A}_{\underline{\underline{\boldsymbol{\kappa}}}}(t_{n+1})\,\boldsymbol{u}_{n+1} = \boldsymbol{b}_f(t_{n+1})\,,$$

where $\boldsymbol{u}_n$ is the approximation of $\boldsymbol{u}$ at $t = t_n$ and $t_{n+1} = t_n + \tau_n$ where τ_n is the time step at the nth step. Rearranging the terms yields the following system of equations:

$$\left[\boldsymbol{M}_{\rho C_p}(t_{n+1}) + \tau_n \boldsymbol{A}_{\underline{\underline{\boldsymbol{\kappa}}}}(t_{n+1})\right]\boldsymbol{u}_{n+1} = \boldsymbol{M}_{\rho C_p}\boldsymbol{u}_n + \tau_n \boldsymbol{b}_f(t_{n+1})\,.$$

To include the Dirichlet boundary conditions, a similar procedure to the stationary case is done. At the instant t_n, the vector $\boldsymbol{u}_n$ is known data and as such, only the conditions at time t_{n+1} need to be specified. This results in the system of equations:

$$\begin{aligned}\left[\boldsymbol{M}_{\rho C_p}(t_{n+1}) + \tau_n \boldsymbol{A}_{\underline{\underline{\boldsymbol{\kappa}}}}(t_{n+1})\right](\texttt{inD},\texttt{inD})\boldsymbol{u}_{n+1}(\texttt{inD}) = \boldsymbol{M}_{\rho C_p}(t_n)\,(\texttt{inD},:)\boldsymbol{u}_n + \tau_n \boldsymbol{b}_f(t_{n+1})\,(\texttt{inD}) \\ - \left[\boldsymbol{M}_{\rho C_p}(t_{n+1}) + \tau_n \boldsymbol{A}_{\underline{\underline{\boldsymbol{\kappa}}}}(t_{n+1})\right](\texttt{inD},\texttt{iD})\boldsymbol{u}_{n+1}(\texttt{iD})\end{aligned}$$

where the notation " : " has been used to denote all the columns. This system needs to be solved at each step n until $t_{n+1} = t_f$.

An issue with the previous method is its low order in time, which implies that small time steps must be used to obtain errors comparable to the spatial discretization (if $\mathbb{P}_1$ elements are used). Mathematically, the Euler method has a first-order accuracy, which implies that in order to reduce the error by half, the time-step also needs to be halved. A better alternative is the second-order Crank-Nicholson method given by (we omit the details of how the Dirichlet conditions are treated):

$$\left[\boldsymbol{M}_{\rho C_p}(t_{n+1}) + \frac{1}{2}\tau_n \boldsymbol{A}_{\underline{\underline{\boldsymbol{\kappa}}}}(t_{n+1})\right]\boldsymbol{u}_{n+1} = \left[\boldsymbol{M}_{\rho C_p}(t_n) - \frac{1}{2}\tau_n \boldsymbol{A}_{\underline{\underline{\boldsymbol{\kappa}}}}(t_n)\right]\boldsymbol{u}_n + \frac{1}{2}\tau_n\left[\boldsymbol{b}_f(t_{n+1}) + \boldsymbol{b}_f(t_n)\right].$$

This method is second-order: meaning that halving the time step ideally reduces the error by a factor of four. Consequently, larger time steps can be employed while maintaining acceptable accuracy (if the solution is smooth enough).

2.4.2 Linear Elasticity

In the context of isotropic and homogeneous materials, the deformation of solids when subjected to load can be described with the Cauchy momentum equations. These equations relate the changes in the stress tensor to the internal forces acting on each point of the domain from the Newton's second law. Together with the boundary conditions, this results in the differential problem of finding the displacement vector $\boldsymbol{u}(\boldsymbol{x}) = \begin{bmatrix} u_x(\boldsymbol{x}) & u_y(\boldsymbol{x}) & u_z(\boldsymbol{x}) \end{bmatrix}^\top$ satisfying

$$\begin{cases} -\nabla \cdot \underline{\underline{\boldsymbol{\sigma}}}(\boldsymbol{u}) &= \boldsymbol{f}, & \boldsymbol{x} \in \Omega, \\ \underline{\underline{\boldsymbol{\sigma}}}(\boldsymbol{u})\,\widehat{\boldsymbol{n}} &= \boldsymbol{g}_N(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma_N, \\ \boldsymbol{u}(\boldsymbol{x}) &= \boldsymbol{u}_D(\boldsymbol{x}), & \boldsymbol{x} \in \Gamma_D, \end{cases} \tag{2.28}$$

where $\widehat{\boldsymbol{n}}$ is the outward unit normal vector to Γ, $\boldsymbol{f}$ are the body or volumetric forces in the material, typically set to zero or to $\rho\boldsymbol{g}$ if the weight of the material is considered. Here $\boldsymbol{g}$ is the gravity vector and ρ the density of the material.

The loads acting on the surface of the element are described with the Neumann data $\boldsymbol{g}_N$, which in case is positive, specifies a traction on Γ_N. For surfaces without any loads, this boundary condition becomes $\underline{\underline{\boldsymbol{\sigma}}}(\boldsymbol{u})\,\widehat{\boldsymbol{n}} = 0$.

In addition to Neumann conditions, Dirichlet conditions may also be specified. These conditions impose a prescribed displacement over some portion of the boundary. They are typically set to zero in structural mechanics to model fixed supports preventing the displacement in all three directions. It is also possible to prevent the motion along one or two directions.

In order to solve the elasticity problem given by Equation 2.28, we need to relate the stress tensor to the displacement vector $\boldsymbol{u}$. This relationship is described by the constitutive equation, also known as the generalized Hooke's law:

$$\underline{\underline{\boldsymbol{\sigma}}}(\boldsymbol{u}) = 2\mu\underline{\underline{\boldsymbol{\varepsilon}}}(\boldsymbol{u}) + \lambda\mathcal{I}_3 \operatorname{tr} \underline{\underline{\boldsymbol{\varepsilon}}}(\boldsymbol{u}), \tag{2.29}$$

where λ and μ are the Lamé parameters, and $\mathcal{I}_3$ is the 3×3 identity matrix. This equation expresses the dependence of the stress tensor on the strain tensor $\underline{\underline{\boldsymbol{\varepsilon}}}$ through the Lamé parameters. In vectorized notation, the constitutive equation can be written as:

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} 2\mu+\lambda & \lambda & \lambda & & & \\ \lambda & 2\mu+\lambda & \lambda & & & \\ \lambda & \lambda & 2\mu+\lambda & & & \\ & & & \mu & & \\ & & & & \mu & \\ & & & & & \mu \end{bmatrix} \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ 2\varepsilon_{23} \\ 2\varepsilon_{13} \\ 2\varepsilon_{12} \end{bmatrix}$$

where σ_{ij} and ε_{ij} are the components of the stress and strain tensor. Note that the tangential strains only affect the respective tangential stresses, whereas the axial stresses are affected by all the axial strains.

The Lamé parameters are material properties that depend on the elastic properties of the material. They can be related to the Young's modulus E and Poisson's ratio ν as follows:

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$$

and

$$\mu = \frac{E}{2(1+\nu)}.$$

The strain tensor is related to the displacement $\boldsymbol{u}$ assuming small deformations:

$$\underline{\underline{\boldsymbol{\varepsilon}}} = \frac{1}{2}\left(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^{\top}\right), \tag{2.30}$$

in matrix notation this becomes

$$\begin{bmatrix} \varepsilon_{11} & \varepsilon_{12} & \varepsilon_{13} \\ \varepsilon_{12} & \varepsilon_{22} & \varepsilon_{23} \\ \varepsilon_{13} & \varepsilon_{23} & \varepsilon_{33} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial u_x}{\partial x} & \dfrac{1}{2}\left(\dfrac{\partial u_x}{\partial y} + \dfrac{\partial u_y}{\partial x}\right) & \dfrac{1}{2}\left(\dfrac{\partial u_x}{\partial z} + \dfrac{\partial u_z}{\partial x}\right) \\ \dfrac{1}{2}\left(\dfrac{\partial u_y}{\partial x} + \dfrac{\partial u_x}{\partial y}\right) & \dfrac{\partial u_y}{\partial y} & \dfrac{1}{2}\left(\dfrac{\partial u_y}{\partial z} + \dfrac{\partial u_z}{\partial y}\right) \\ \dfrac{1}{2}\left(\dfrac{\partial u_z}{\partial x} + \dfrac{\partial u_x}{\partial z}\right) & \dfrac{1}{2}\left(\dfrac{\partial u_z}{\partial y} + \dfrac{\partial u_y}{\partial z}\right) & \dfrac{\partial u_z}{\partial z} \end{bmatrix}$$

To obtain its variational formulation of (2.28) we first note that the solution, i.e. the displacement, is a vector function. We can define the basis functions of the discrete space by considering each component of a node separately [4]:

$$\boldsymbol{\varphi}_j^1 = \begin{bmatrix} \varphi_j & 0 & 0 \end{bmatrix}, \qquad \boldsymbol{\varphi}_j^2 = \begin{bmatrix} 0 & \varphi_j & 0 \end{bmatrix}, \qquad \boldsymbol{\varphi}_j^3 = \begin{bmatrix} 0 & 0 & \varphi_j \end{bmatrix},$$

where φ_j is the scalar basis function associated to the jth node. In the rest of the section it will be assumed that all nodes are non-Dirichlet nodes so we can denote the basis as the set $\{\boldsymbol{\varphi}_j^r\}_{j=1,\ldots,\texttt{nNodes}}^{r=1,2,3}$. We can use this to approximate the displacement vector as

$$\boldsymbol{u} \approx \boldsymbol{u}_h = \sum_{j=1}^{\texttt{nNodes}} \left(u_{x,j}\boldsymbol{\varphi}_j^1 + u_{y,j}\boldsymbol{\varphi}_j^2 + u_{z,j}\boldsymbol{\varphi}_j^3\right). \tag{2.31}$$

After some manipulations, the variational formulation of the differential problem can be proved to be

$$\int_\Omega \underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{u}\right) : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^r\right) - \int_{\Gamma_N} \underbrace{\left(\underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{u}\right)\widehat{\boldsymbol{n}}\right)\cdot\boldsymbol{\varphi}_i^r}_{=\boldsymbol{g}_N\cdot\boldsymbol{\varphi}_i^r} = \int_\Omega \boldsymbol{f}\cdot\boldsymbol{\varphi}_i^r, \qquad \forall i = 1,\ldots,\texttt{nNodes},\ r = 1,2,3 \tag{2.32}$$

where

$$\boldsymbol{A} : \boldsymbol{B} = \sum_{i,j} a_{ij}b_{ij}$$

is the Frobenius product of two matrices of the same size. In the context of elasticity, this equation is named the **principle of virtual work** [10].

Following the same ideas as the scalar case, the next step is to write (2.32) in terms of the displacement, then

$$\begin{aligned} \sum_{j=1}^{\texttt{nNodes}} \int_\Omega \left[u_{x,j}\underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{\varphi}_j^1\right) : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^r\right) + u_{y,j}\underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{\varphi}_j^2\right) : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^r\right) + u_{z,j}\underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{\varphi}_j^3\right) : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^r\right)\right] = \\ \int_\Omega \boldsymbol{f}\cdot\boldsymbol{\varphi}_i^r + \int_{\Gamma_N} \boldsymbol{g}_N\cdot\boldsymbol{\varphi}_i^r. \end{aligned} \tag{2.33}$$

From the constitutive equation (2.29), the Frobenius product in the left-hand side can be expanded as

$$\underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{\varphi}_j^{r_1}\right) : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^{r_2}\right) = 2\mu\underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_j^{r_1}\right) : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^{r_2}\right) + \lambda\,\mathrm{tr}\,\underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_j^{r_1}\right)\left(\mathcal{I}_3 : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^{r_2}\right)\right), \tag{2.34}$$

where $r_1, r_2 \in \{1,2,3\}$. The second term can be simplified by noticing that it equals the sum of the diagonal terms of the strain tensor

$$\left(\mathcal{I}_3 : \underline{\underline{\varepsilon}}\left(\boldsymbol{\varphi}_i^{r_2}\right)\right) = \operatorname{tr} \underline{\underline{\varepsilon}}\left(\boldsymbol{\varphi}_i^{r_2}\right),$$

then from the form of the basis function we have $\operatorname{tr} \underline{\underline{\varepsilon}}\left(\boldsymbol{\varphi}_i^{r_2}\right) = \frac{\partial \varphi_i}{\partial x_{r_2}}$. The first term requires a little bit more of work. First, we can write the strain tensor in its most general form when evaluated by a displacement, say the virtual displacement $\boldsymbol{\varphi}_i^r$:

$$\varepsilon_{k\ell}\left(\boldsymbol{\varphi}_i^r\right) = \frac{1}{2}\left(\nabla \boldsymbol{\varphi}_i^r + \nabla \boldsymbol{\varphi}_i^{r\top}\right)|_{k\ell} = \frac{1}{2}\left(\frac{\partial \varphi_i}{\partial x_k}\delta_{r\ell} + \frac{\partial \varphi_i}{\partial x_\ell}\delta_{rk}\right),$$

where δ_{ij} is the Kronecker delta. With this, the Frobenius product of the strain tensors can be written using summation notation as:

$$\begin{aligned}
\underline{\underline{\varepsilon}}\left(\boldsymbol{\varphi}_j^{r_1}\right) : \underline{\underline{\varepsilon}}\left(\boldsymbol{\varphi}_i^{r_2}\right) &= \sum_{k,\ell=1}^{3} \frac{1}{2}\left(\frac{\partial \varphi_j}{\partial x_k}\delta_{r_1\ell} + \frac{\partial \varphi_j}{\partial x_\ell}\delta_{r_1 k}\right)\frac{1}{2}\left(\frac{\partial \varphi_i}{\partial x_k}\delta_{r_2\ell} + \frac{\partial \varphi_i}{\partial x_\ell}\delta_{r_2 k}\right) \\
&= \frac{1}{4}\sum_{k,\ell=1}^{3}\left(\frac{\partial \varphi_j}{\partial x_k}\delta_{r_1\ell}\frac{\partial \varphi_i}{\partial x_k}\delta_{r_2\ell} + \frac{\partial \varphi_j}{\partial x_k}\delta_{r_1\ell}\frac{\partial \varphi_i}{\partial x_\ell}\delta_{r_2 k}\right. \\
&\qquad \left. + \frac{\partial \varphi_j}{\partial x_\ell}\delta_{r_1 k}\frac{\partial \varphi_i}{\partial x_k}\delta_{r_2\ell} + \frac{\partial \varphi_j}{\partial x_\ell}\delta_{r_1 k}\frac{\partial \varphi_i}{\partial x_\ell}\delta_{r_2 k}\right).
\end{aligned}$$

The first and the fourth terms involves the product of the derivatives of $\boldsymbol{\varphi}_j^{r_1}$ and $\boldsymbol{\varphi}_i^{r_2}$ over different components. Since the sums over k and ℓ are carried out with the same indices, they can be interchanged without changing the overall value of the sum. Upon this interchange, it is apparent that the first and last terms are identical, and the same holds for the intermediate terms. Grouping similar terms results in

$$\begin{aligned}
\underline{\underline{\varepsilon}}\left(\boldsymbol{\varphi}_j^{r_1}\right) : \underline{\underline{\varepsilon}}\left(\boldsymbol{\varphi}_i^{r_2}\right) &= \frac{1}{2}\sum_{k,\ell=1}^{3}\left(\frac{\partial \varphi_j}{\partial x_k}\delta_{r_1\ell}\frac{\partial \varphi_i}{\partial x_k}\delta_{r_2\ell} + \frac{\partial \varphi_j}{\partial x_k}\delta_{r_1\ell}\frac{\partial \varphi_i}{\partial x_\ell}\delta_{r_2 k}\right) \\
&= \frac{1}{2}\delta_{r_1 r_2}\sum_{k=1}^{3}\left(\frac{\partial \varphi_j}{\partial x_k}\frac{\partial \varphi_i}{\partial x_k}\right) + \frac{1}{2}\frac{\partial \varphi_j}{\partial x_{r_2}}\frac{\partial \varphi_i}{\partial x_{r_1}},
\end{aligned}$$

where it has been noted that the first addend is nonzero only when $\ell = r_1 = r_2$ and the second addend is nonzero when $\ell = r_1$ and $k = r_2$. Inserting the previous terms in (2.34) results in

$$\underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{\varphi}_j^{r_1}\right) : \underline{\underline{\varepsilon}}\left(\boldsymbol{\varphi}_i^{r_2}\right) = \mu\left[\delta_{r_1 r_2}\sum_{k=1}^{3}\left(\frac{\partial \varphi_j}{\partial x_k}\frac{\partial \varphi_i}{\partial x_k}\right) + \frac{\partial \varphi_j}{\partial x_{r_2}}\frac{\partial \varphi_i}{\partial x_{r_1}}\right] + \lambda\frac{\partial \varphi_j}{\partial x_{r_1}}\frac{\partial \varphi_i}{\partial x_{r_2}}. \tag{2.35}$$

We can test all possible combinations of $\boldsymbol{\varphi}_j^{r_1}$ and $\boldsymbol{\varphi}_i^{r_2}$. Expressing the results in matrix notation yields:

- $\boldsymbol{\varphi}_i^1 = \begin{bmatrix}\varphi_i & 0 & 0\end{bmatrix}^\top$ and $\boldsymbol{\varphi}_j^1 = \begin{bmatrix}\varphi_j & 0 & 0\end{bmatrix}^\top$

$$\begin{aligned}
\underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{\varphi}_j^1\right) : \underline{\underline{\varepsilon}}\left(\boldsymbol{\varphi}_i^1\right) &= \begin{bmatrix}\frac{\partial \varphi_i}{\partial x} & \frac{\partial \varphi_i}{\partial y} & \frac{\partial \varphi_i}{\partial z}\end{bmatrix}\begin{bmatrix}\lambda+2\mu & & \\ & \mu & \\ & & \mu\end{bmatrix}\begin{bmatrix}\frac{\partial \varphi_j}{\partial x} \\ \frac{\partial \varphi_j}{\partial y} \\ \frac{\partial \varphi_j}{\partial z}\end{bmatrix} \\
&= \nabla\varphi_i A_{11}\nabla\varphi_j^\top
\end{aligned}$$

- $\boldsymbol{\varphi}_i^2 = \begin{bmatrix} 0 & \varphi_i & 0 \end{bmatrix}^\top$ and $\boldsymbol{\varphi}_j^2 = \begin{bmatrix} 0 & \varphi_j & 0 \end{bmatrix}^\top$

$$\begin{aligned} \underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{\varphi}_j^2\right) : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^2\right) &= \begin{bmatrix} \frac{\partial \varphi_i}{\partial x} & \frac{\partial \varphi_i}{\partial y} & \frac{\partial \varphi_i}{\partial z} \end{bmatrix} \begin{bmatrix} \mu & & \\ & \lambda + 2\mu & \\ & & \mu \end{bmatrix} \begin{bmatrix} \frac{\partial \varphi_j}{\partial x} \\ \frac{\partial \varphi_j}{\partial y} \\ \frac{\partial \varphi_j}{\partial z} \end{bmatrix} \\ &= \nabla \varphi_i A_{22} \nabla \varphi_j^\top \end{aligned}$$

- $\boldsymbol{\varphi}_i^3 = \begin{bmatrix} 0 & 0 & \varphi_i \end{bmatrix}^\top$ and $\boldsymbol{\varphi}_j^3 = \begin{bmatrix} 0 & 0 & \varphi_j \end{bmatrix}^\top$

$$\begin{aligned} \underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{\varphi}_j^3\right) : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^3\right) &= \begin{bmatrix} \frac{\partial \varphi_i}{\partial x} & \frac{\partial \varphi_i}{\partial y} & \frac{\partial \varphi_i}{\partial z} \end{bmatrix} \begin{bmatrix} \mu & & \\ & \mu & \\ & & \lambda + 2\mu \end{bmatrix} \begin{bmatrix} \frac{\partial \varphi_j}{\partial x} \\ \frac{\partial \varphi_j}{\partial y} \\ \frac{\partial \varphi_j}{\partial z} \end{bmatrix} \\ &= \nabla \varphi_i A_{33} \nabla \varphi_j^\top \end{aligned}$$

- $\boldsymbol{\varphi}_i^1 = \begin{bmatrix} \varphi_i & 0 & 0 \end{bmatrix}^\top$ and $\boldsymbol{\varphi}_j^2 = \begin{bmatrix} 0 & \varphi_j & 0 \end{bmatrix}^\top$

$$\begin{aligned} \underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{\varphi}_j^2\right) : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^1\right) &= \begin{bmatrix} \frac{\partial \varphi_i}{\partial x} & \frac{\partial \varphi_i}{\partial y} & \frac{\partial \varphi_i}{\partial z} \end{bmatrix} \begin{bmatrix} & \lambda & \\ \mu & & \\ & & \end{bmatrix} \begin{bmatrix} \frac{\partial \varphi_j}{\partial x} \\ \frac{\partial \varphi_j}{\partial y} \\ \frac{\partial \varphi_j}{\partial z} \end{bmatrix} \\ &= \nabla \varphi_i A_{12} \nabla \varphi_j^\top \end{aligned}$$

- $\boldsymbol{\varphi}_i^1 = \begin{bmatrix} \varphi_i & 0 & 0 \end{bmatrix}^\top, \boldsymbol{\varphi}_j^3 = \begin{bmatrix} 0 & 0 & \varphi_j \end{bmatrix}^\top$

$$\begin{aligned} \underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{\varphi}_j^3\right) : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^1\right) &= \begin{bmatrix} \frac{\partial \varphi_i}{\partial x} & \frac{\partial \varphi_i}{\partial y} & \frac{\partial \varphi_i}{\partial z} \end{bmatrix} \begin{bmatrix} & & \lambda \\ & & \\ \mu & & \end{bmatrix} \begin{bmatrix} \frac{\partial \varphi_j}{\partial x} \\ \frac{\partial \varphi_j}{\partial y} \\ \frac{\partial \varphi_j}{\partial z} \end{bmatrix} \\ &= \nabla \varphi_i A_{13} \nabla \varphi_j^\top \end{aligned}$$

- $\boldsymbol{\varphi}_i^2 = \begin{bmatrix} 0 & \varphi_i & 0 \end{bmatrix}^\top, \boldsymbol{\varphi}_j^3 = \begin{bmatrix} 0 & 0 & \varphi_j \end{bmatrix}^\top$

$$\begin{aligned} \underline{\underline{\boldsymbol{\sigma}}}\left(\boldsymbol{\varphi}_j^3\right) : \underline{\underline{\boldsymbol{\varepsilon}}}\left(\boldsymbol{\varphi}_i^2\right) &= \begin{bmatrix} \frac{\partial \varphi_i}{\partial x} & \frac{\partial \varphi_i}{\partial y} & \frac{\partial \varphi_i}{\partial z} \end{bmatrix} \begin{bmatrix} & & \lambda \\ & \mu & \\ & & \end{bmatrix} \begin{bmatrix} \frac{\partial \varphi_j}{\partial x} \\ \frac{\partial \varphi_j}{\partial y} \\ \frac{\partial \varphi_j}{\partial z} \end{bmatrix} \\ &= \nabla \varphi_i A_{23} \nabla \varphi_j^\top \end{aligned}$$

The remaining cases can be obtained noticing the roles of φ_i and φ_j will change and $A_{r_1 r_2} = A_{r_2 r_1}^\top$. Then, equation (2.33) can be rewritten as

$$\begin{cases} \sum_{j=1}^{\texttt{nNodes}} \int_\Omega u_{x,j}\nabla\varphi_i(A_{11}\nabla\varphi_j^\top) + u_{y,j}\nabla\varphi_i(A_{12}\nabla\varphi_j^\top) + u_{z,j}\nabla\varphi_i(A_{13}\nabla\varphi_j^\top) \\ \qquad = \int_\Omega \boldsymbol{f}\cdot\boldsymbol{\varphi}_i^1 + \int_{\Gamma_N} \boldsymbol{g}_N\cdot\boldsymbol{\varphi}_i^1, \\ \sum_{j=1}^{\texttt{nNodes}} \int_\Omega u_{x,j}\nabla\varphi_i(A_{21}\nabla\varphi_j^\top) + u_{y,j}\nabla\varphi_i(A_{22}\nabla\varphi_j^\top) + u_{z,j}\nabla\varphi_i(A_{23}\nabla\varphi_j^\top) \\ \qquad = \int_\Omega \boldsymbol{f}\cdot\boldsymbol{\varphi}_i^2 + \int_{\Gamma_N} \boldsymbol{g}_N\cdot\boldsymbol{\varphi}_i^2, \\ \sum_{j=1}^{\texttt{nNodes}} \int_\Omega u_{x,j}\nabla\varphi_i(A_{31}\nabla\varphi_j^\top) + u_{y,j}\nabla\varphi_i(A_{32}\nabla\varphi_j^\top) + u_{z,j}\nabla\varphi_i(A_{33}\nabla\varphi_j^\top) \\ \qquad = \int_\Omega \boldsymbol{f}\cdot\boldsymbol{\varphi}_i^3 + \int_{\Gamma_N} \boldsymbol{g}_N\cdot\boldsymbol{\varphi}_i^3, \end{cases}$$

for $i = 1\ldots,$ `nNodes` where we have rewritten the Frobenius product using the above equalities. The integrals involved in the construction of the system of equations are simply the stiffness matrix $\boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}}}$ with $\underline{\underline{\boldsymbol{\kappa}}} = A_{r_1r_2}$ (in this case the diffusion matrix is not symmetric); the source term $\boldsymbol{b}_f$ with $f = \boldsymbol{f}_r$, where $\boldsymbol{f}_r$ is the rth component of the volumetric force; and the traction vector $\boldsymbol{t}_N$ with $g_N = \boldsymbol{g}_{N,r}$. This system can be written in matrix notation as

$$\begin{bmatrix} \boldsymbol{S}_{11} & \boldsymbol{S}_{12} & \boldsymbol{S}_{13} \\ \boldsymbol{S}_{12}^\top & \boldsymbol{S}_{22} & \boldsymbol{S}_{23} \\ \boldsymbol{S}_{13}^\top & \boldsymbol{S}_{23}^\top & \boldsymbol{S}_{33} \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_x \\ \boldsymbol{u}_y \\ \boldsymbol{u}_z \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_x + \boldsymbol{t}_x \\ \boldsymbol{b}_y + \boldsymbol{t}_y \\ \boldsymbol{b}_z + \boldsymbol{t}_z \end{bmatrix}.$$

Note. When we consider the dynamic case of (2.28), the acceleration $\frac{\partial^2 \boldsymbol{u}}{\partial t^2}$ is no longer zero and we are in an elastodynamics case. The differential problem becomes

$$\rho\frac{\partial^2 \boldsymbol{u}}{\partial^2 t} - \nabla\cdot\underline{\underline{\boldsymbol{\sigma}}}(\boldsymbol{u}) = \boldsymbol{f}$$

The most general case of this problem can be treated using the ideas shown in the previous section, but with a suitable time integrator developed for solving the wave equation.

A special case of this problem can be considered when the displacement of the body modeled by Ω follows a harmonic motion with frequency $\omega = 2\pi f$. In this case the displacement can be written as

$$\boldsymbol{u} = \boldsymbol{U}(\boldsymbol{x})\cos wt = \boldsymbol{U}(\boldsymbol{x})\Re(e^{i\omega t}),$$

where $\Re(z)$ denotes the real part of a complex number z and $\boldsymbol{U}$ is the displacement field at $t = 0$. Substituting this in the differential problem yields the time independent problem

$$-\nabla\cdot\underline{\underline{\boldsymbol{\sigma}}}(\boldsymbol{U}) - \rho\omega^2\boldsymbol{U} = \boldsymbol{F}$$

where $\boldsymbol{F} = \boldsymbol{f}\Re(e^{-iwt})$. In this case, when we write the variational formulation, the term $\boldsymbol{\varphi}_i^{r_1}\cdot\boldsymbol{\varphi}_j^{r_2}$ will be zero unless the displacements are in the same direction. This result in an additional term from the mass matrix:

$$\begin{bmatrix} \boldsymbol{M}_{\rho\omega^2} & & \\ & \boldsymbol{M}_{\rho\omega^2} & \\ & & \boldsymbol{M}_{\rho\omega^2} \end{bmatrix}.$$

Chapter 3

FEM package

The `FEM3dclass` class is an essential part of the Finite element Method implementation in Matlab and Octave. It stores the information obtained from the mesh file generated by GMSH and provide functions to assemble the system of equations discussed in chapter 2. Additionally, it provides methods for computing the solution at points outside the mesh nodes. The `FEM3dclass` is based on a structure created with the `importGMSH3D` function, which extracts relevant information from the mesh file.

The fields of an object of this class are the following:

- mesh: structure data type. It contains the information of the mesh with $\mathbb{P}_m$ elements and relevant data for implementing the finite element method as well as data necessary for post-processing and error estimations.
- Nj3D: cell array of handle functions. It contains the Lagrange basis $\widehat{N}_i$ of degree m in the reference tetrahedral element as functions of the barycentric coordinates.
- Nj2D: cell array of handle functions. It contains the the Lagrange basis $\widehat{N}_i$ of degree m in the reference triangular element as functions of the barycentric coordinates.
- gradNj3D: cell array of handle functions. It contains the gradient of the Lagrange basis $\widehat{\nabla}\widehat{N}$ in the reference tetrahedral element as functions of the barycentric coordinates.
- ComplementaryInformation: map container. It has the physical tags and the numerical label of the entities belonging to it.

An object of this class can be generated in two ways: passing a file with extension `.msh` using `T=FEM3Dclass('file')` or passing a mesh structure with `T=FEM3Dclass(mesh)`. In the case of generating an object from a file, the `importGMSH3D` function takes care of creating the mesh structure and setting the appropriate fields. On the other hand, when generating an object from a mesh structure, it is necessary to ensure that all the fields defined in section 3.4 have been properly defined to guarantee the correct functioning of the methods in the class. In this case, no `ComplementaryInformation` data is given.

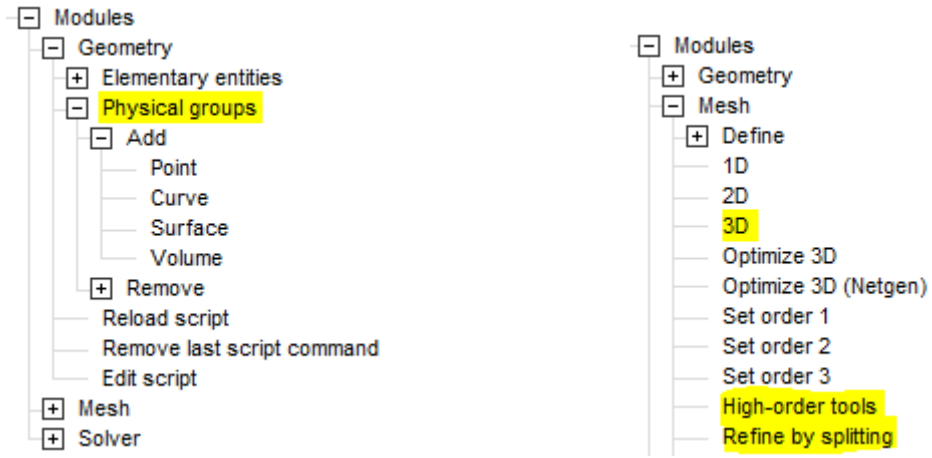
The elementary methods for this class are defined as follows:

- `basisNj`: Contains all the Lagrange basis up to degree 4 defined in the previous chapter and the gradient of the function.
- `quadRule2D`: Contains the weights and nodes needed for the quadrature rule inside the reference triangular element.
- `quadRule3D`: Contains the weights and nodes needed for the cubature rules inside the reference tetrahedral element.

- `local3DMatrices`: Contains the exact local matrices assuming constant functions inside the tetrahedral reference element.
- `localRobinMass`: Contains the exact local boundary mass matrix assuming constant function inside the triangular reference element.
- `femMassMatrix`, `femStressMatrix`, `femStiffnessMatrix`, `femAdvectionMatrix`, `femRobin`, `femSourceTerm`: They assemble each term of the system of equations in the FEM formulation.
- `evalFEM3DUh`: It performs the computation of the FEM solution to non-nodal points inside the mesh.

In the following sections, we will present a brief introduction to GMSH, followed by an explanation of the code developed for constructing the structure and an explanation of the specific fields and how to gather relevant information from them. Finally we will give some details of the methods developed for this class.

3.1 GMSH



(a) Physical Groups. (b) Mesh module.

Figure 3.1: GMSH modules of interest.

To generate a finite element mesh in GMSH, it is necessary to first define the properties of the domain and its boundaries by classifying the entities (points, lines, surfaces, and volumes) of the geometry. For boundary value problems in three dimensions the entities of interest are the surfaces and the volumes. Surfaces can then be grouped according to their properties or the boundary type they belong to (Dirichlet, Neumann or Robin boundaries). They may also be grouped if additional outputs are needed in some sub-boundaries. Similarly, we can decompose the domain into several volumes.

The option `Physical groups` located in the geometry module is used to perform this operation (see Figure 3.1a). Four options are available: points, curves, surfaces, volumes. Choosing one of these options, a new window opens as shown in Figure 3.2. In this window, the Physical group can be named and the entities belonging to it can be selected, in which case they are highlighted in red. If `Automatic` is checked, a numerical label is created, otherwise, a numerical label is chosen by the user. Once the selection is over, pressing `e` saves the physical group if it wasn't previously created. If it was, the surfaces are appended to the existing group.

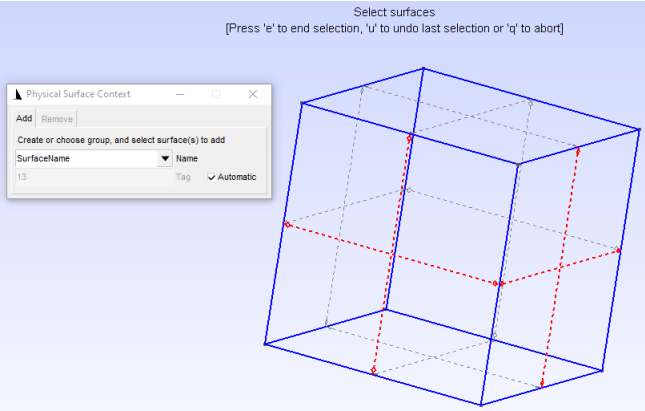


Figure 3.2: Grouping of surfaces into a physical surface.

The meshing process can be done via its corresponding module (see Figure 3.1b). The most important options in the interface are highlighted there. The option `3D mesh` generates a three-dimensional mesh with elements of order equal to the value selected. This can be modified in `HighOrdertools` → `PolynomialOrder` or in `Tools` → `Options` → `Mesh` → `General` → `ElementOrder`.

Remark Before meshing the mesh you should verify the orientation of the surfaces. You can go to `Tools` → `Options` → `Geometry` → `Visibility` and check if the normals to the boundary points outwards. If it is not the case, modify the orientation of the surfaces.

3.2 Mesh File

After the mesh has been constructed, the file can be saved by clicking on `File` →`Export` →`type:.msh`. Upon selecting the `Save` button, a new window will open where additional options may be chosen (see Figure 3.3). There are several file formats available: `Version 1, Version 2 ASCII, Version 2 Binary, Version 4 ASCII` and `Version 4 Binary`. The first three formats are older versions, and their use is no longer recommended according to the documentation.

The version `ASCII 4.1` is the latest as of the writing of this project, and this must be chosen for the code to work. Additional options such as `Save all elements` and `Save parametric coordinates` may also be selected. If the former is checked, all elements including points, one-dimensional, two-dimensional and three-dimensional elements will be saved. Otherwise, the mesh file will include all the elements contained in the entities belonging to any of the Physical Groups created. Its selection is optional, but if chosen, the files may be too large for fine meshes. The latter option yields the parametric coordinates of the nodes in a mesh, but it is not yet implemented.

After saving the mesh file, a plain text file is obtained with parts of interest summarized in table 3.1. Each section contains information about the generated file and relevant data about the obtained mesh. In the following sections, a summary of important features for each section of the file is provided along with information from the official documentation (see [8, Section 9.1]).

In `Version 4 ASCII`, the sections are delimited by `$SectionName`. $\cdots$ `$EndSectionName`. For clarity on the structure of this version, a general explanation and an example using the test case shown in Figure 3.4 will be provided. The test case consists of the reference tetrahedron.

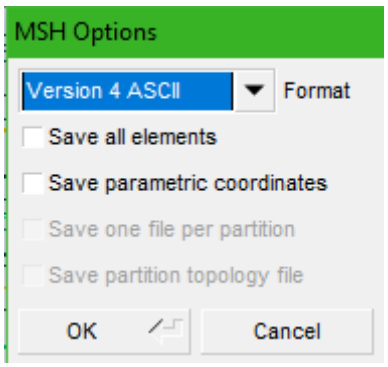


Figure 3.3: File format selection.

Mesh format
Physical Tags
Entities
Nodes
Elements

Table 3.1: Parts of mesh file.

The volume has a physical tag `Volume` and the surfaces are grouped into two different Physical Groups: `Surface1` and `Surface2`. The surface entities belonging to the former are the entities 2, 3 and 4 and for the latter only the surface entity 1 belongs to it. After performing the mesh, only one element was created, coinciding exactly with the reference element. As every surface and volume has an associated Physical Group, the option `Save all Elements` was not checked.

3.2.1 Mesh format

```
$MeshFormat
version(ASCII double; currently 4.1)
file-type(ASCII int; 0 for ASCII mode, 1 for binary mode)
    data-size(ASCII int; sizeof(size_t))
    < int with value one; only in binary mode,
    to detect endianness >
$EndMeshFormat
```

In this section, some properties of the file are indicated, including the version and the way in which it has been saved.

```
$MeshFormat
4.1 0 8
$EndMeshFormat
```

For the test case, it can be seen the file version is 4.1, It is followed by a zero, indicating that the file is saved in `ASCII` and not binary. The last number represents the number of bits needed to represent each character.

3.2.2 Physical tags

```
$PhysicalNames
numPhysicalNames(ASCII int)
```

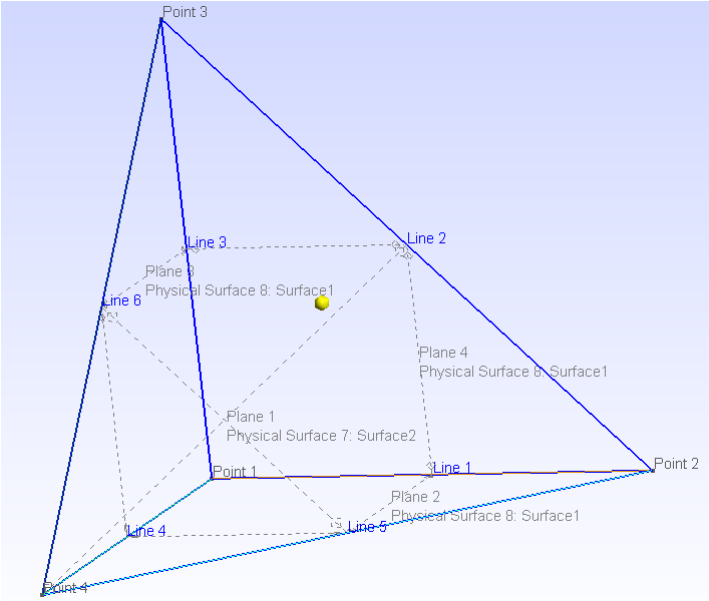


Figure 3.4: Test case.

```
dimension(ASCII int) physicalTag(ASCII int) "name"
    (127 characters max)
...
$EndPhysicalNames
```

This section displays the physical tags. These may correspond to grouping of points, lines, surfaces and/or volumes of the geometry. Once again, we remark that only surface and volume physical tags are of interest. In this section, the number of physical tags created is displayed and a list with three columns containing the dimension of the physical tag, its numerical label and the string label under which it was saved. The order of appearance is based on the group dimension and its numerical label. Unless otherwise specified, default labels are ordered sequentially. Also, any two tags of the same dimension must have different labels, but if they have different dimensions they may be the same.

```
$PhysicalNames
3
2 7 "Surface2"
2 8 "Surface1"
3 9 "Volume"
$EndPhysicalNames
```

From the test case it can be seen that three physical tags were generated. Each row shows the dimension of the physical tag followed by its numerical label and the string label given by the user. It is noticeable that the three tags are divided into two surface tags and one volume tag with numerical labels 7, 8 and 9 respectively.

3.2.3 Entities

```
$Entities
numPoints(size_t) numCurves(size_t)
    numSurfaces(size_t) numVolumes(size_t)
pointTag(int) X(double) Y(double) Z(double)
    numPhysicalTags(size_t) physicalTag(int) ...
```

```
...
curveTag(int) minX(double) minY(double) minZ(double)
    maxX(double) maxY(double) maxZ(double)
    numPhysicalTags(size_t) physicalTag(int) ...
    numBoundingPoints(size_t) pointTag(int) ...
...
surfaceTag(int) minX(double) minY(double) minZ(double)
    maxX(double) maxY(double) maxZ(double)
    numPhysicalTags(size_t) physicalTag(int) ...
    numBoundingCurves(size_t) curveTag(int) ...
...
volumeTag(int) minX(double) minY(double) minZ(double)
    maxX(double) maxY(double) maxZ(double)
    numPhysicalTags(size_t) physicalTag(int) ...
    numBoundngSurfaces(size_t) surfaceTag(int) ...
...
$EndEntities
```

This section stores all entities created for the geometry generation including points, curves, surfaces and volumes. In the first row, a list shows the number of generated entities following the order: `numPoints, numCurves, numSurfaces` and `numVolumes`.

All entities are then listed according to their dimension (points are considered to have dimension zero) and numerical label in increasing order. First all points are shown followed by curves and so on. Each row shows the label of the entity, extreme coordinates, the number of associated physical tags followed by a list containing their numerical labels and finally the number of entities of a smaller dimension limiting it.

```
$Entities
4 6 4 1
1 0 0 0 0
2 1 0 0 0
3 0 1 0 0
4 0 0 1 0
1 0 0 0 1 0 0 0 2 1 -2
2 0 0 0 1 1 0 0 2 2 -3
3 0 0 0 0 1 0 0 2 3 -1
4 0 0 0 0 0 1 0 2 1 -4
5 0 0 0 1 0 1 0 2 2 -4
6 0 0 0 0 1 1 0 2 3 -4
1 0 0 0 1 1 1 1 7 3 6 -5 2
2 0 0 0 1 0 1 1 8 3 4 -5 -1
3 0 0 0 0 1 1 1 8 3 3 4 -6
4 0 0 0 1 1 0 1 8 3 3 1 2
1 0 0 0 1 1 1 1 9 4 1 3 4 2
$EndEntities
```

The geometry in the test case comprises four points, six curves, four surfaces, and one volume. Each entity is assigned a numerical label, with the first value of each row indicating its label. These labels correspond to the vertices, edges, faces, and the tetrahedron itself. From this example it is apparent that any two entities may have the same label as long as they differ in dimension. Also, only surfaces and volume contains physical tags; in this case only one per

entity by this is not compulsory. Additionally, we observe that all surfaces are bounded by three curves, and the volume is bounded by four. The last number displays the numerical label of those entities.

3.2.4 Nodes

```
$Nodes
numEntityBlocks(size_t) numNodes(size_t)
    minNodeTag(size_t) maxNodeTag(size_t)
entityDim(int) entityTag(int) parametric(int; 0 or 1)
    numNodesInBlock(size_t)
nodeTag(size_t)
...
x(double) y(double) z(double)
    < u(double; if parametric and entityDim >= 1) >
    < v(double; if parametric and entityDim >= 2) >
    < w(double; if parametric and entityDim == 3) >
...
...
$EndNodes
```

This section of the file stores all the coordinates of the nodes that were generated in the mesh. The format used for storage involves a classification of the nodes into blocks, where each block contains the nodes inside a generated entity (excluding the nodes on the boundaries), thus preventing the occurrence of repeated nodes.

Each node will be assigned a numerical index. Although the numbering is increasing by default, this is not always the case. For example, if the generated mesh is the result of a mesh refinement, the sorting does not have to be increasing and there may be jumps between indices [8, Section 10.1].

Each block is organized in ascending order according to its dimension and the numerical label of the associated entity. For every block, a line is presented, indicating the dimension and label of the entity. It also includes a value of 1 or 0 to indicate whether the coordinates are provided in parametric form or not, along with the number of nodes in the block. Subsequently, the numerical indices of the nodes within the block are listed followed by their corresponding coordinates in the same sequential order.

```
$Nodes
15 20 1 20
0 1 0 1
1
0 0 -0
0 2 0 1
2
1 0 -0
0 3 0 1
3
0 1 -0
0 4 0 1
4
0 0 1
```

```
1 1 0 2
5
6
0.3333333333324915 0 0
0.6666666666657831 0 0
1 2 0 2
7
8
0.6666666666671625 0.3333333333328375 0
0.3333333333341937 0.6666666666658063 0
1 3 0 2
9
10
0 0.6666666666668164 0
0 0.3333333333341704 0
1 4 0 2
11
12
0 0 0.3333333333324915
0 0 0.6666666666657831
1 5 0 2
13
14
0.6666666666671625 0 0.3333333333328375
0.3333333333341937 0 0.6666666666658063
1 6 0 2
15
16
0 0.6666666666671625 0.3333333333328375
0 0.3333333333341937 0.6666666666658063
2 1 0 1
17
0.3333333333342374 0.3333333333333333 0.3333333333324293
2 2 0 1
18
0.3333333333332102 0 0.3333333333323062
2 3 0 1
19
0 0.3333333333341142 0.3333333333323062
2 4 0 1
20
0.3333333333332102 0.3333333333332102 0
3 1 0 0
$EndNodes
```

By examining the test case, we can observe the presence of fifteen blocks, comprising a total of twenty nodes numbered from one to twenty. It is important to note that nodes situated at the vertices or faces of the tetrahedrons do not appear multiple times. This is because the nodes are considered to be part of an entity only if they reside within its interior.

3.2.5 Elements

```
$Elements
numEntityBlocks(size_t) numElements(size_t)
    minElementTag(size_t) maxElementTag(size_t)
entityDim(int) entityTag(int) elementType(int)
    numElementsInBlock(size_t)
elementTag(size_t) nodeTag(size_t) ...
...
...
$EndElements
```

In GMSH, all objects generated during the meshing process with the same dimension as the containing entity are called elements, this includes, points, lines, triangles and tetrahedrons. Similar to the previous subsection, the elements are classified in blocks according to their location. On each block, the dimension of the entity, its label, the element type and the number of elements in that block are shown.

To obtain the element type, the developers have assigned each element a unique numerical code. The codes of the elements of interest, namely tetrahedra and triangular elements up to degree four, are collected in Table 3.2.

Degree	Triangle	Tetrahedron
1	2	4
2	9	11
3	21	29
4	23	30

Table 3.2: Element code.

```
$Elements
5 5 1 5
2 1 21 1
1 2 3 4 7 8 15 16 14 13 17
2 2 21 1
2 1 4 2 11 12 14 13 6 5 18
2 3 21 1
3 1 4 3 11 12 16 15 9 10 19
2 4 21 1
4 1 2 3 5 6 7 8 9 10 20
3 1 29 1
5 1 3 4 2 10 9 15 16 12 11 6 5 13 14 7 8 19 20 18 17
$EndElements
```

On the test case, the option `Save all elements` was not chosen and the physical tags created were only for surfaces and volumes, hence the absence of point and line elements in the file. We can see that all surface elements are $\mathbb{P}_3$ elements with numerical code 21 and the same is true for the tetrahedron.

3.3 Mesh File reading

This section aims to explain the essential elements of the code responsible for creating the data structure containing the mesh data, namely `importGMSH3D`. This function has two outputs: `T` is a structure with the mesh parameters defined inside it; `Complementary information` is a map containing the physical tags generated during the preprocessing and the numerical labels of the entities associated to each tag.

The primary concept behind this function involves reading each section of the mesh separately and gathering the relevant data. After verifying that the file has the correct format, the next step is to read all the sections.

```
71 if ~isnan(sPhysicalNames*ePhysicalNames)
72     %% Insert Physical objects
73     surfacesNum = []; surfacesPTags = {};
74     volumesNum  = []; volumesPTags  = {};
75     nPhysicalNames=str2num(meshFile{sPhysicalNames+1});
76     for r = 1: nPhysicalNames
77         aux    = regexp(meshFile{sPhysicalNames+1+r}, ' ', 'split');
78         aux{1} = str2num(aux{1}); aux{2} = str2num(aux{2});
79         tag = '';
80         for i=3:length(aux)
81             tag = [tag ' ' aux{i}];
82         end
83         if tag(end-1)=='"'
84             tag = tag(3:end-2);
85         else
86             tag = tag(3:end-1);
87         end
88         ComplementaryInformation(tag)=[];
89         if aux{1} == 2
90             surfacesNum(end+1)   = aux{2};
91             surfacesPTags{end+1} = tag;
92         elseif aux{1} == 3
93             volumesNum(end+1)   = aux{2};
94             volumesPTags{end+1} = tag;
95         end
96     end
97 end
```

Lines 71-97 are responsible for reading the `PhysicalNames` section. In these lines, the numerical label and physical tags for both surfaces and volumes are stored in separate arrays and cells, respectively, while storing them in the same order to maintain their relationship. The keys of the `map` are initialized with empty arrays.

```
99  if ~isnan(sEntities*eEntities)
100     aux = str2num(meshFile{sEntities+1});
101
102     % Surfaces
103     sSurfaces = sEntities+aux(1)+aux(2)+2;
104     eSurfaces = sSurfaces + aux(3)-1;
105     %Volumes
106     sVolumes  = sEntities+aux(1)+aux(2)+aux(3)+2;
107     eVolumes  = sVolumes+aux(4)-1;
108
109     % Surfaces
110     for i=sSurfaces:eSurfaces
111         v = str2num(meshFile{i});
112         nPhysicalTags = v(8);
```

```
113         if  nPhysicalTags > 0
114             for j =1:nPhysicalTags
115                 ComplementaryInformation(surfacesPTags{surfacesNum == v(8+j)
    }) = ...
116                     [ComplementaryInformation(surfacesPTags{surfacesNum == v
    (8+j)}) v(1)];
117             end
118         end
119     end
120     % Volumes
121     for i=sVolumes:eVolumes
122         v = str2num(meshFile{i});
123         nPhysicalTags = v(8);
124         if  nPhysicalTags >0
125             for j =1:nPhysicalTags
126                 ComplementaryInformation(volumesPTags{volumesNum == v(8+j)})
     = ...
127                     [ComplementaryInformation(volumesPTags{volumesNum == v
    (8+j)}) v(1)];
128             end
129         end
130     end
131 end
```

This section of the code handles the entities information. In lines 102-107, the first and last lines associated to surfaces and volumes are stored based on the information obtained from the `aux` variable in line 100, which contains the number of entities of each dimension in the geometry.

Starting at line 112, the number of tags associated with each surface entity is stored, and a loop is performed through all their numerical labels. At lines 115-116, the numerical label of the surface entity is stored. A similar process is repeated for all volumes in the geometry at lines 121-130. This approach allows for easy identification of the surfaces and volumes and their associated physical tags.

```
133 if ~isnan(sNodes*eNodes)
134     v = meshFile{sNodes+1};
135     v = str2num(v);
136     T.coord = nan(v(4),3);
137     j = sNodes+2;
138     while j<eNodes
139         aux = str2num(meshFile{j});
140         if aux(3) ~=0
141             error('Parametric nodes are not supported')
142         end
143         % read the positions
144         if aux(4)~=0
145             indNodes = j+(1:aux(4));
146             indCoord = indNodes(end)+(1:aux(4));
147 
148             indNodes = cellfun(@str2num, meshFile(indNodes));
149             auxCoord = meshFile(indCoord);
150             str = strjoin(auxCoord,'\n');
151             str = textscan(str,'%f');
152             Coord = reshape(str{1}, [], numel(auxCoord))';
153             T.coord(indNodes,:) = Coord;
154         else
155             warning(['Domain without proper nodes: ' num2str(aux)])
156         end
```

```
157         j = j+2*aux(4)+1;
158     end
159 end
```

Lines 133-159 contain the code for storing the mesh nodes. In line 136, the matrix `T.mesh.coord` is initialized with rows equal to the maximum node index. A while loop is used to iterate through all the nodes associated with each entity until no more entities remain. In line 144, the code checks whether there is any node in an entity. If there are nodes, their numeration is stored in line 145. Subsequently, their coordinates are stored in line 146 and added to `T.mesh.coord`.

```
161 %%%%% Elements
162 disp('Checking Elements')
163 nBlocks = str2num(meshFile{sElements+1});
164 nBlocks = nBlocks(1);
165
166 pointerLine = sElements+2;
167 % supported Elements
168 supElements = [2  9  21   23 ;  % P1,P2, P3, P4 triangles
169                4 11  29   30 ]; % P1, P2, P3, P4 tetrahedra
170 %Elements' fields initialization
171 T.trB   = []; T.domBd  = [];
172 T.ttrh  = []; T.domain = [];
173 T.faces = []; T.ttrh2faces = []; T.faces2ttrh = [];
174
175 for j=1:nBlocks
176     aux = str2num(meshFile{pointerLine});
177     deg = aux(3);
178     supElementsTest = (supElements ==deg);
179     indElements     = pointerLine+(1:aux(4));
180     auxElm = meshFile(indElements);
181     str = strjoin(auxElm,'\n');
182     str = textscan(str,'%f');
183     elements = reshape(str{1}, [], numel(auxElm))';
184     elements(:,1)   =[];
185     if sum(supElementsTest(1,:))~=0  % 2D -> Face
186         T.trB    = [T.trB; elements];
187         T.domBd  = [T.domBd; ones(size(elements,1),1)*aux(2)];
188     elseif sum(supElementsTest(2,:))~=0
189         T.ttrh   = [T.ttrh; elements];
190         T.domain = [T.domain; ones(size(elements,1),1)*aux(2)];
191     end
192     pointerLine  = pointerLine+aux(4)+1;
193 end
```

The following section of the mesh file reads the elements within the mesh. A loop iterates through all the entity blocks from lines 175 to 193 in order to store the elements in the mesh. At line 176, a logical equality check is executed to verify the code of the elements belonging to the block. Subsequently, lines 185 and 191 are used to determine whether the element is a triangle or a tetrahedron, which is then used to store the elements in `T.mesh.trB` or `T.mesh.ttrh`, respectively.

```
196 % Check if there are "ghost nodes"
197 nNodes     = max(max(T.ttrh));
198 nodesTtrh  = unique(T.ttrh(:));
199 nodesTrBd  = unique(T.trB(:));
200 if ~isempty(setdiff(nodesTrBd,nodesTtrh))
201     error('Nodes in triangles which are not in ttrh')
```

```
202 end
203 [~, indghostNodes]= setdiff(1:nNodes,nodesTtrh);
204 if ~isempty(indghostNodes)
205     % Fixing
206     warning('ghost nodes... fixing')
207     p       = nan(1,nNodes);
208     p(nodesTtrh) = 1:length(nodesTtrh);
209     T.coord(indghostNodes,:)=[];
210     T.ttrh = p(T.ttrh);
211     T.trB  = p(T.trB);
212     nNodes = max(max(T.ttrh));
213 end
214 T.nNodes = nNodes;
215 switch(size(T.ttrh,2))
216     case(4)
217         T.deg =1;
218     case(10)
219         T.deg =2;
220     case(20)
221         T.deg =3;
222     case(35)
223         T.deg =4;
224 end
```

This section of the code is responsible for removing any inconsistencies in node numbering. First, at line 197, the maximum index associated with a node is determined. Then, at lines 198-199, the nodes belonging to tetrahedral and triangular elements are sorted in ascending order using the `unique` function. To ensure the mesh is three-dimensional or that every volume is correctly meshed, line 200 checks if there is any occurrence of a boundary node that does not appear in any tetrahedron.

It is possible that there are rows in `T.mesh.coord` without any nodes. This is because some node indices may be missing, as seen in 3.2.4. Line 203 stores the missing node numbers in a vector to remove any jumps in numbering. At lines 207-208, a vector `p` is created, which contains the ordered numeration of nodes at the positions where there is an existing node index. The rows in `T.mesh.coord` without a defined node are then removed at line 209. Finally, lines 210 and 211 use the vector `p` to correct the numbering in `T.mesh.ttrh` and `T.mesh.trB`. The actual number of nodes in the mesh is defined at lines 212-214, and the degree of the mesh is determined at lines 215-224.

```
228 %%% Boundary faces
229 v21 = T.coord(T.trB(:,1),:) - T.coord(T.trB(:,2),:);
230 v31 = T.coord(T.trB(:,1),:) - T.coord(T.trB(:,3),:);
231 T.trBNormal = cross(v21, v31, 2);
```

In the last portion of the code, additional data regarding the faces, and the size of the elements are calculated at each point.Lines 229-230 compute the edge vectors $\boldsymbol{x}_1 - \boldsymbol{x}_2$ and $\boldsymbol{x}_1 - \boldsymbol{x}_3$ respectively, where $\boldsymbol{x}_i$ are the components of the vertices of the triangle in a boundary.

```
v = cross(A, B, dim),
```

which performs the cross product of the vectors in both matrices contained in the dimension `dim`. If the value of this parameter is 2, the cross product is performed over the row vectors in the same position of the matrices `A` and `B`. This computation works as the numeration of the vertices in the global coordinates of the triangle are defined such that the normals points outwards.

```
233 %%% Tetrahedron faces
234
235 nElement = size(T.ttrh,1);
236 indAux   = [2 3 4; 1 4 3; 1 2 4; 1 3 2];
237 T.faces  = [T.ttrh(:,indAux(1,:)); ...
238             T.ttrh(:,indAux(2,:)); ...
239             T.ttrh(:,indAux(3,:)); ...
240             T.ttrh(:,indAux(4,:))];
241 % We use only the faces' vertices
242 facesAux = sort(T.faces(:,1:3),2);
243 [~,p1, p3]  = unique(facesAux,'rows', 'first');
244 [~,p2]      = unique(facesAux,'rows', 'last');
245 T.faces     = T.faces(p1,:);
246 ind = zeros(nElement,4); ind(:) = 1:numel(T.ttrh(:,1:4));
247 T.ttrh2faces  = p3(ind);
248 p1  = mod(p1,nElement)    ; p1(p1==0) = nElement;
249 p2  = mod(p2,nElement)    ; p2(p2==0) = nElement;
250 T.faces2ttrh = [p1, p2];
251 ind = p1==p2;
252 T.faces2ttrh(ind,2) = nan;
```

Lines 235-252, performs the computation of geometric data pertaining to all faces of the tetrahedra in the mesh. Specifically, line 236 stores, at each row, the vertices associated with each face in an ordered fashion such that their outward normal points away from the corresponding tetrahedron (check Figure 2.5). Subsequently, at line 237, the matrix `T.mesh.faces` is generated, containing all faces arranged as

$$\begin{bmatrix} \mathtt{f}_{11} \\ \vdots \\ \mathtt{f}_{1,\mathtt{nttrh}} \\ \vdots \\ \mathtt{f}_{41} \\ \vdots \\ \mathtt{f}_{4,\mathtt{nttrh}} \end{bmatrix}$$

where $\mathtt{f}_{ij}$ is a row vector listing the global numeration of the vertices comprising the `i`th face of the `j`th tetrahedron. However, some faces with identical sets of vertices may appear multiple times in this matrix, albeit with different orders of the vertices. To overcome this challenge, an auxiliary variable `facesAux` is introduced, which disregards the ordering of the vertices and instead takes into account only their membership to each respective face.

At line 243, the first appearance of each unique face is computed using the command `unique` with the arguments `'first'` and `'rows'`. This command internally sorts the rows of the matrix by the smallest values of the first column, and in case of a tie, by the values of the next columns. The resulting sorted matrix is

$$\mathtt{uniqueSortedFaces} = \begin{bmatrix} \tilde{\mathtt{f}}_1 \\ \vdots \\ \tilde{\mathtt{f}}_{\mathtt{nFaces}} \end{bmatrix}$$

where $\tilde{\mathtt{f}}_i$ are the unique faces with sorted vertices. This matrix is not stored to avoid ruining the orientation of the faces. Instead, the indices of the first occurrence of each face are stored in the

variable `p1`, which contains the indices of the rows in `T.mesh.faces` that satisfy the condition `facesAux`$(\texttt{p1}, :) ==$ `uniqueSortedFaces`.

The variable `p3` contains the location of each face in the sorted matrix `uniqueSortedFaces`, such that `facesAux` $==$ `uniqueSortedFaces`$(\texttt{p3}, :)$. At line 244, the latest occurrence of each face is computed using the `unique` command with the argument '`last`', and their location is stored in the variable `p2`.

Line 245 modifies the field `T.mesh.faces` to contain the unique faces with the correct ordering of the vertices but with the rows permuted (due to the reordenation of the `unique` command). At lines 246-247, the field `T.mesh.facesttrh` is created. To assemble it, the matrix `ind` enumerates the faces of the tetrahedra, as given at line 237. Then with `p3`, the indices are readjusted to the unique faces.

Lines 248-252 compute the tetrahedra that contain each face. Starting from line 237, each face is initially located in $(\texttt{i}-1)\texttt{nElement}+\texttt{j}$, where `j` denotes the index of the tetrahedron, and `i` denotes the index of the `i`th face within that tetrahedron. Line 248-249 applies the modulus operation $(\texttt{i}-1)\texttt{nElement}+\texttt{j} \equiv \texttt{j}(\mod \texttt{nElement})$ to determine the index of the tetrahedron containing the face, except when `j` is equal to `nElement`. In this case, `j` represents the last face of the tetrahedron and must be treated specially. The operation returns a value of 0, which is then corrected to `nElement`.

Line 250 appends the tetrahedra containing the first and last occurrence of each face to a matrix. However, if a face appears only once and its first and last occurrence are the same, the tetrahedral information is redundant. The program detects and handles this situation using lines 251-252.

```
255 v21 = T.coord(T.faces(:,1),:) - T.coord(T.faces(:,2),:);
256 v31 = T.coord(T.faces(:,1),:) - T.coord(T.faces(:,3),:);
257 T.facenormal = cross(v21, v31, 2);
258
259 %%% Elements
260
261 % edges' vectors
262 v12 = T.coord(T.ttrh(:,2),:)-T.coord(T.ttrh(:,1),:);
263 v13 = T.coord(T.ttrh(:,3),:)-T.coord(T.ttrh(:,1),:);
264 v14 = T.coord(T.ttrh(:,4),:)-T.coord(T.ttrh(:,1),:);
265
266 % Determinant
267 T.detBk = dot(v12, cross(v13, v14, 2),2);
```

The computation of normal vectors for all faces is carried out in the same manner as for the boundary elements. Additionally, the determinant of the affine matrices, for all the elements, is computed at once using the triple product formula.

3.4 Data structure

To check the physical tags, the command `T.ComplementaryInformation.keys` can be used. This returns a cell array with the physical tags created. The entities belonging to each physical tag are returned with the command `T.ComplementaryInformation('String Label')` can be used, this returns a vector with the numerical labels of each entity belonging to that physical groups as defined in the `Entities` section of the mesh file. If all entities associated to a physical tag are to be displayed, we can use `T.ComplementaryInformation.values`.

The mesh structure `T.mesh` contains the following fields:

- `nNodes`: number of nodes in the mesh.
- `deg`: Degree of the elements used for the mesh.
- `coord`: `nNodes`×`3` matrix. Each row contains the three components of the nodes in a mesh.

$$\texttt{T.mesh.coord} = \begin{bmatrix} x_1 & y_1 & z_1 \\ & \vdots & \\ x_{\texttt{nNodes}} & y_{\texttt{nNodes}} & z_{\texttt{nNodes}} \end{bmatrix}$$

- `trB`: `ntrB`×`dofA` matrix. Each row contains the `dofA` pointers to the `coord` field, that is, the nodes inside each triangular element in the boundary.

$$\texttt{T.mesh.trB} = \begin{bmatrix} \boldsymbol{i}^1 \\ \vdots \\ \boldsymbol{i}^{\texttt{ntrB}} \end{bmatrix}$$

  where $\boldsymbol{i}^r$ is the vector of global indices belonging to the rth boundary element in a mesh, say $\begin{bmatrix} i_{r1} & \cdots & i_{r,\texttt{dofA}} \end{bmatrix}$.
- `domBd`: `ntrB`×`1` column vector. Contains the numerical label of the surface to which each elements in `trB` belongs.
- `trBNormal`: `ntrB`×`3` matrix. The rth row contains the normal vector to the rth boundary element in the mesh. This vector has magnitude equal to twice the area of the triangular element.

$$\texttt{T.mesh.trBNormal} = \begin{bmatrix} n_{x,1} & n_{y,1} & n_{z,1} \\ & \vdots & \\ n_{x,\texttt{ntrB}} & n_{y,\texttt{ntrB}} & n_{z,\texttt{ntrB}} \end{bmatrix}$$

- `ttrh`: `nTtrh`×`dofK` matrix. Each row contains the `dofK` pointers to `coord`, that is, the nodes inside each tetrahedral element.

$$\texttt{T.mesh.ttrh} = \begin{bmatrix} \boldsymbol{i}^1 \\ \vdots \\ \boldsymbol{i}^{\texttt{nTtrh}} \end{bmatrix}$$

  where $\boldsymbol{i}^r$ is the vector of global indices belonging to the rth tetrahedron in a mesh, say $\begin{bmatrix} i_{1r} & \cdots & i_{r,\texttt{dofK}} \end{bmatrix}$.
- `domain`: `nTtrh`×`1` column vector. Contains the numeric label of the volume to which each element in `ttrh` belongs.
- `detBk`: `nTtrh`×`1` column vector. The rth row of this vector contains the Jacobian determinant associated to the affine mapping of the rth tetrahedron in the mesh F_r.
- `faces`: `nFaces`×`3` matrix. Contains pointers to `coord` of the vertices of all faces belonging to a tetrahedron in the mesh, with no repetition.

$$\texttt{T.mesh.faces} = \begin{bmatrix} i_{11} & i_{12} & i_{13} \\ & \vdots & \\ i_{\texttt{nFaces},1} & i_{\texttt{nFaces},2} & i_{\texttt{nFaces},3} \end{bmatrix}$$

- `ttrh2faces`: `nTtrh`×4 matrix. Each row has pointer to the `faces` field. The rth row contain the four faces belonging to the rth tetrahedron.

$$\texttt{T.mesh.ttrh2faces} = \begin{bmatrix} \ell_{11} & \ell_{12} & \ell_{13} & \ell_{14} \\ & \vdots & \vdots & \\ \ell_{\texttt{nTtrh},1} & \ell_{\texttt{nTtrh},2} & \ell_{\texttt{nTtrh},3} & \ell_{\texttt{nTtrh},4} \end{bmatrix}$$

- `faces2ttrh`: `nFaces`×2 matrix. Each row contains the pointers to the tetrahedra containing each face. The rth row contains the two tetrahedrons sharing a common face.If a face belongs to the boundary then the second row is set to `NaN`.

$$\texttt{T.mesh.faces2ttrh} = \begin{bmatrix} \ell_{11} & \ell_{12} \\ \vdots & \vdots \\ \ell_{\texttt{nFaces},1} & \ell_{\texttt{nFaces},2} \end{bmatrix}$$

- `facenormal`: `nFaces`×3 matrix. The rth row contains the normal vector to the rth face in the mesh. This vector has magnitude equal to twice the area of the triangular element.

$$\texttt{T.mesh.trBNormal} = \begin{bmatrix} n_{x,1} & n_{y,1} & n_{z,1} \\ & \vdots & \\ n_{x,\texttt{ntrB}} & n_{y,\texttt{nFaces}} & n_{z,\texttt{nFaces}} \end{bmatrix}$$

The field `T.mesh.coord` gives the nodes and their global numeration; to obtain the coordinates of a node with global numeration `i` it is as simple as using `T.mesh.coord(i, :)`. If the nodes inside an element `k` are needed, for example a triangular element, the command `T.mesh.trB(k, :)` gives the nodes in the triangle $\begin{bmatrix} i_{k1} & \cdots & i_{k,\texttt{dofA}} \end{bmatrix}$. Then, the command `T.mesh.coord(T.mesh.trB(k,:),:)` returns a matrix of size `dofA` $\times$ 3 of the form

$$\begin{bmatrix} x_{i_{\texttt{k}1}} & y_{i_{\texttt{k}1}} & z_{i_{\texttt{k}1}} \\ x_{i_{\texttt{k}2}} & y_{i_{\texttt{k}2}} & z_{i_{\texttt{k}2}} \\ & \vdots & \\ x_{i_{\texttt{k,dofA}}} & y_{i_{\texttt{k,dofA}}} & z_{i_{\texttt{k,dofA}}} \end{bmatrix}$$

Similarly, the nodes in a tetrahedron `k` can be accessed using `T.mesh.coord(T.mesh.ttrh(k, :), :)`. If instead multiple triangular elements are indexed simultaneously, for example two at the same time, then `T.mesh.coord(T.mesh.trB([k1,k2],:),:)` returns

$$\begin{bmatrix} x_{i_{\texttt{k}_1},1} & y_{i_{\texttt{k}_1},1} & z_{i_{\texttt{k}_1},1} \\ x_{i_{\texttt{k}_2},1} & y_{i_{\texttt{k}_2},1} & z_{i_{\texttt{k}_2},1} \\ x_{i_{\texttt{k}_1},2} & y_{i_{\texttt{k}_1},2} & z_{i_{\texttt{k}_1},2} \\ x_{i_{\texttt{k}_2},2} & y_{i_{\texttt{k}_2},2} & z_{i_{\texttt{k}_2},2} \\ & \vdots & \\ x_{i_{\texttt{dofA}}^{\texttt{k}_1}}^{\texttt{k}_1} & y_{i_{\texttt{dofA}}^{\texttt{k}_1}}^{\texttt{k}_1} & z_{i_{\texttt{dofA}}^{\texttt{k}_1}}^{\texttt{k}_1} \\ x_{i_{\texttt{dofA}}^{\texttt{k}_2}}^{\texttt{k}_2} & y_{i_{\texttt{dofA}}^{\texttt{k}_2}}^{\texttt{k}_2} & z_{i_{\texttt{dofA}}^{\texttt{k}_2}}^{\texttt{k}_2} \end{bmatrix}$$

That is, the first nodes of each element are given first; in the next rows, the coordinates of the node with local numeration equal to two are given and so on. This is because Matlab/Octave reshapes internally the matrix `T.mesh.trB([k1,k2],:)` to a column vector, hence it is equivalent to indexing the rows

$$\begin{bmatrix} i_{\mathtt{k}_1,1} & i_{\mathtt{k}_2,1} & \cdots & i_{\mathtt{k}_1,\mathtt{dofA}} & i_{\mathtt{k}_2,\mathtt{dofA}} \end{bmatrix}^\top.$$

The triangular elements in the interior of an specific entity can be accessed with the help of the command `T.mesh.domBd==surfaceLabel`. This returns a logical column vector of the same size as `domBd` that equals `1` if an element is inside the surface and `0` otherwise. The command `ismember(T.mesh.domBd,` $\begin{bmatrix}\mathtt{surfaceLabel}_1 & \ldots & \mathtt{surfaceLabel}_p\end{bmatrix}$`)` can be used to check multiple entities at the same time. An example of this case is to use the field `T.ComplementaryInformation` to gather the elements with an associated boundary condition. The pointer to the nodes belonging to each triangular element is accessed using `T.mesh.trB(T.mesh.trB==surfaceLabels),:)`. Notice that the indexes on the rows is a logical vector, Matlab returns the rows with a logical value of `true`. The same process can be performed if a set of tetrahedrons are needed instead but using the field `domain`.

To obtain the Jacobian of the parameterization of the triangular surface element we use `vecnorm(T.mesh.trBNormal,2,2)`.

The command `T.mesh.faces(T.mesh.ttrh2faces(k,`ℓ`),:)` can be used to return the vertices of ℓth face of the `k`th tetrahedron. Given a face `k` of the matrix `T.mesh.face`, a pointer to a tetrahedron can be obtained using the command `T.mesh.ttrh(T.mesh.faces2ttrh(k,`ℓ_i`),:)`, where $i = 1, 2$.

The function `trisurf` can be used to plot a triangular mesh from the vector components of the nodes conforming a triangular mesh and a matrix with the pointers to the vertices of the triangles. For instance, the vectors and the matrix can be chosen from the fields `coord` and `trB`. This yields a representation of the surface of the mesh. The command to represent this function in this case is given by

```
trisurf(T.mesh.trB(:,1:3),T.mesh.coord(:,1),T.mesh.trB(:,2),T.mesh.trB(:,3),
    u,'facecolor','interp')
```

If an additional vector `u` of the same size as `coord` is given, then the function colors the surface element according to the values at the vertices and applies a linear interpolation to produce a smooth representation of the solution within each triangular element. For higher order elements, it is necessary to define subtriangles inside each element by connecting the nodes. For arbitrary surfaces, a pointer similar to `trB` (with the vertices) is needed. From a set of arbitrary points, the command `delaunay(x,y,z)` allows creating such a pointer matrix.

3.5 Lagrange Basis

The Lagrange basis in the reference element can be accessed with

```
[Ns2D,Ns3D,gradNs3D] = FEM3Dclass.basisNj(deg)
```

It contains all the basis functions shown in Section 2.2.4 stored in a cell array. The variable `gradNs3D` contains a cell array of size $\mathbb{R}^{3\times \mathtt{dofK}}$, with the partial derivatives of the Lagrange basis. Check appendix A to see the code developed to obtain these terms.

3.6 Integration formulas

The quadrature and cubature formulas for approximating the integrals needed in the assembly process can be accessed using `FEM3Dclass.quadRule2D` and `FEM3Dclass.quadRule3D` respectively. The weights and the evaluation nodes were taken from the python package `quadpy`[22]

for the triangle (T2) and tetrahedron (T3) respectively. The rules obtained on this package are given directly in barycentric coordinates, so the application of the formula is as shown in section 2.3.3. For the quadrature rules the formulas chosen have degree 3, 5, 7 and 9. For the cubature rules, additional formulas are stored, they have degree 3, 5, 6, 7 and 9.

3.7 Local Matrices

In certain cases, it may be useful to use exact local matrices in the finite element method (FEM). This is particularly true when the functions involved in Problem (2.1), such as $\underline{\underline{\boldsymbol{\kappa}}}$, α or c, are constant, or piecewise constant on each tetrahedra, which is a rather common scenario in practical applications. In such cases, the mass matrix $\boldsymbol{M}_c^K$, the stiffness matrix $\boldsymbol{S}_{\underline{\underline{\boldsymbol{\kappa}}}}^K$, the boundary mass matrix $\boldsymbol{R}_\alpha$, and the advection matrix $\boldsymbol{A}_{\boldsymbol{\beta}}^K$ can be constructed exactly up to computer precision.

Although approximate formulas will be used for the integrals in the implementation, the exact matrices have been computed using exact integration and are already stored for the sake of completeness. The command to obtain the local advection, mass and stiffness matrices is

```
[M,Sxx,Sxy,Sxz,Syy,Syz,Szz, Ax, Ay, Az] = FEM3Dclass.local3DMatrices(deg)
```

and to obtain the boundary mass matrix we use

```
R = FEM3Dclass.localRobinMass(deg)
```

3.8 Assembly of the system of equations

For the assembly of the linear system we require to define the function associated to each term of the system of equations. They may be given in two forms:

```
r = @(x,y,z)   r(x,y,z)
```

or as

```
 r = @(x,y,z, dom)   r1 (x,y,z).* (dom==1)+ r2(x,y,z).*(dom==2) ...
```

where `dom` may be either `T.mesh.domain` or `T.mesh.domBd` (for Robin conditions it is a subset of this vector containing only the elements in the Robin or Neumann boundary) and `1`, `2` and so on are the numerical labels of the entities. Naturally, other logical operators may be used. For constant functions we will use

```
r(x,y,z)=cte+x*0;
```

Doing this we ensure that function `r` returns a constant (`cte`) vector of the same size as the input (`x`).

3.8.1 Mass matrix

The assembly of the mass matrix is done with the function

```
Mc = femMassMatrix(obj, c)
```

where `c` is a handle function and `obj` is an object of the class.

Assume then we work with the cubature formula:

$$\int_{\widehat{K}} \widehat{f} \approx \frac{1}{6} \sum_{\ell=1}^{\texttt{nQ}} w_\ell \widehat{f}(\widehat{\boldsymbol{\lambda}}_\ell).$$

```
12 % Geometry Data
13 T            = obj.mesh;
14 nTtrh        = size(T.ttrh,1);
15 nNodes       = T.nNodes;
16 dofK = size(T.ttrh,2);
17
18 %%% Cubature rule
19 degQuad       = max(2*T.deg-1,3);
20 quadRuleTtrh = obj.quadRule3D;
21 nodesQuad     = quadRuleTtrh{degQuad}.nodes;
22 weights       = quadRuleTtrh{degQuad}.weights;
23 nQ            = length(weights);
24
25 % Evaluate Interpolation functions
26 PkValues =zeros(dofK, nQ);
27 for i = 1:length(obj.Nj3D)
28     PkValues(i,:) = obj.Nj3D{i}(nodesQuad(1,:),nodesQuad(2,:), ...
29                                 nodesQuad(3,:),nodesQuad(4,:));
30 end
31 PkProd = repelem(PkValues,dofK,1).*repmat(PkValues,dofK,1);
```

Lines 19-23 define the nodes use in the cubature formula which is then used in lines 26-30 for computing the individual Lagrange basis evaluated at each cubature node, which are then stored in $\texttt{PkValues} \in \mathbb{R}^{\texttt{dofK}\times\texttt{nQ}}$. Specifically, at $\texttt{PkValues}(\ell, \texttt{r})$, the value of $\widehat{N}_\ell$ evaluated at the $\texttt{r}$th cubature node $\widehat{\boldsymbol{\lambda}}_r$ is stored. Subsequently, at line 31, all combinations of the product $\widehat{N}_i\widehat{N}_j(\widehat{\boldsymbol{\lambda}}_r)$ are computed and stored in the matrix $\texttt{PkProd}$, defined as:

$$\texttt{PkProd} = \begin{bmatrix} \widehat{N}_1\widehat{N}_1(\widehat{\boldsymbol{\lambda}}_1) & \cdots & \widehat{N}_1\widehat{N}_1(\widehat{\boldsymbol{\lambda}}_{\texttt{nQ}}) \\ \vdots & \ddots & \vdots \\ \widehat{N}_1\widehat{N}_{\texttt{dofK}}(\widehat{\boldsymbol{\lambda}}_1) & \cdots & \widehat{N}_1\widehat{N}_{\texttt{dofK}}(\widehat{\boldsymbol{\lambda}}_{\texttt{nQ}}) \\ & \vdots & \\ \widehat{N}_1\widehat{N}_{\texttt{dofK}}(\widehat{\boldsymbol{\lambda}}_1) & \cdots & \widehat{N}_1\widehat{N}_{\texttt{dofK}}(\widehat{\boldsymbol{\lambda}}_{\texttt{nQ}}) \\ \vdots & \ddots & \vdots \\ \widehat{N}_{\texttt{dofK}}\widehat{N}_{\texttt{dofK}}(\widehat{\boldsymbol{\lambda}}_1) & \cdots & \widehat{N}_{\texttt{dofK}}\widehat{N}_{\texttt{dofK}}(\widehat{\boldsymbol{\lambda}}_{\texttt{nQ}}) \end{bmatrix}$$

Furthermore, the function c must also be evaluated at the cubature nodes in the physical domain.

```
33 % Cubature nodes in physical domain
34 px = T.coord(:,1); py = T.coord(:,2); pz = T.coord(:,3);
35 nodesX = px(T.ttrh(:,1:4))*nodesQuad;
36 nodesY = py(T.ttrh(:,1:4))*nodesQuad;
37 nodesZ = pz(T.ttrh(:,1:4))*nodesQuad;
38
39 if nargin(c) == 3
40     val = c(nodesX, nodesY, nodesZ);
41 elseif nargin(c) == 4
42     val = c(nodesX, nodesY, nodesZ, T.domain*ones(1,nQ));
43 end
44 val = val.*T.detBk;
```

The x, y and z components in the physical domain of each cubature node are stored in `nodesX, nodesY, nodesZ` respectively using Equation(2.13) for each individual component. Lines 39-43 compute the value of c at each cubature node and store it in `val`. At line 44 the computation $\det B_K c_{\mathtt{r}}$ is done for every tetrahedron and every cubature node and stored in

$$\mathtt{val} = \begin{bmatrix} \det B_1 c_1(\widehat{\boldsymbol{\lambda}}_1) & \cdots & \det B_1 c_1(\widehat{\boldsymbol{\lambda}}_{\mathtt{nQ}}) \\ \vdots & \ddots & \vdots \\ \det B_{\mathtt{nttrh}} c_{\mathtt{nttrh}}(\widehat{\boldsymbol{\lambda}}_1) & \cdots & \det B_{\mathtt{nttrh}} c_{\mathtt{nttrh}}(\widehat{\boldsymbol{\lambda}}_{\mathtt{nQ}}) \end{bmatrix}$$

```
46 % Cubature formula
47 PkProd = repmat(PkProd, nTtrh, 1);
48 val    = repelem(val, dofK^2, 1);
49 val    = val.*PkProd;
50 val    = val*weights(:)/6;
```

In these lines the cubature formula is applied. Line 47 construct the block column vector

$$\mathtt{PkProd} = \begin{bmatrix} \mathtt{PkProd} \\ \vdots \\ \mathtt{PkProd} \end{bmatrix} \in \mathbb{R}^{(\mathtt{nttrh}\cdot\mathtt{dofK}^2)\times \mathtt{nQ}}$$

and line 48, on the other hand, repeats $\mathtt{dofK}^2$ times each value obtained in `val`, resulting in

$$\mathtt{val} = \begin{bmatrix} \det B_1 c_1(\widehat{\boldsymbol{\lambda}}_1) & \cdots & \det B_1 c_1(\widehat{\boldsymbol{\lambda}}_{\mathtt{nQ}}) \\ \vdots & \ddots & \vdots \\ \det B_1 c_1(\widehat{\boldsymbol{\lambda}}_1) & \cdots & \det B_1 c_1(\widehat{\boldsymbol{\lambda}}_{\mathtt{nQ}}) \\ & \vdots & \\ \det B_{\mathtt{nttrh}} c_{\mathtt{nttrh}}(\widehat{\boldsymbol{\lambda}}_1) & \cdots & \det B_{\mathtt{nttrh}} c_{\mathtt{nttrh}}(\widehat{\boldsymbol{\lambda}}_{\mathtt{nQ}}) \\ \vdots & \ddots & \vdots \\ \det B_{\mathtt{nttrh}} c_{\mathtt{nttrh}}(\widehat{\boldsymbol{\lambda}}_1) & \cdots & \det B_{\mathtt{nttrh}} c_{\mathtt{nttrh}}(\widehat{\boldsymbol{\lambda}}_{\mathtt{nQ}}) \end{bmatrix} \in \mathbb{R}^{(\mathtt{nttrh}\cdot\mathtt{dofK}^2)\times \mathtt{nQ}}.$$

At line 49, the componentwise product is performed to obtain all the integrands of the local mass matrix for every tetrahedron and cubature node. Finally, at line 50 the cubature formula is implemented simultaneously for all elements on the last line noticing the summation can be written as

$$\sum_{\ell} w_\ell r(\widehat{\boldsymbol{\lambda}}_\ell) = \begin{bmatrix} r(\widehat{\boldsymbol{\lambda}}_1) & \cdots & r(\widehat{\boldsymbol{\lambda}}_{\mathtt{nQ}}) \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_{\mathtt{nQ}} \end{bmatrix}.$$

Denoting with $\boldsymbol{M}_c^K$ the local mass matrix of the tetrahedron K and $\boldsymbol{M}_c^K(:)$ its vectorization (collapsing the matrix into a column vector) then the matrix obtained is

$$\mathtt{val} \approx \begin{bmatrix} \boldsymbol{M}_c^1(:) \\ \vdots \\ \boldsymbol{M}_c^{\mathtt{nttrh}}(:) \end{bmatrix}.$$

```
52 % Mass matrix assembly
53 indi = repmat(T.ttrh,1,dofK);  indi = indi';
54 indj = repelem(T.ttrh,1,dofK); indj = indj';
55 Mc   = sparse(indi(:), indj(:), val(:), nNodes, nNodes);
```

The last step is to assembly the matrix. Lines 53-54 uses the field `ttrh` containing the indices vector of each tetrahedron to obtain all pairs of rows and columns (i, j) of the local matrix for each tetrahedron. This is used in line 55 to assemble the mass matrix using the command `sparse` to assembly a sparse matrix. This command receives as inputs a list of row indices and column indices where the nonzero elements are localized. The corresponding elements in `val(:)` are stored in those positions of a `nNodes` $\times$ `nNodes` matrix. This command works such that if any pair (i, j) appears multiple times then the results are added. This behavior allows adding the contribution of each tetrahedron containing nodes i and j to the associated term in the mass matrix.

3.8.2 Stiffness Matrix

Two separate functions have been implemented for the assembly of the stiffness matrix. This is motivated by the fact that when $\underline{\underline{\boldsymbol{\kappa}}}$ is equal to the identity matrix $\mathcal{I}_3$, the matrix $C_K = B_K^{-1} B_K^{-\top}$ can be computed exactly up to rounding errors and the matrix assembly from it can be implemented without any further complication. The function

```
S = femStressMatrix(obj)
```

has been specifically developed to handle this scenario. The only input required is an object of the class.

```
9  % Geometry Data
10 T            = obj.mesh;
11 nttrh        = size(T.ttrh,1);
12 nNodes       = T.nNodes;
13 dofK = size(T.ttrh,2);
14
15 [~,Sxx,Sxy,Sxz,Syy,Syz,Szz] = obj.local3DMatrices(T.deg);
16
17
18 %Ck matrix elements obtention
19 v12 = T.coord(T.ttrh(:,2),:)-T.coord(T.ttrh(:,1),:);
20 v13 = T.coord(T.ttrh(:,3),:)-T.coord(T.ttrh(:,1),:);
21 v14 = T.coord(T.ttrh(:,4),:)-T.coord(T.ttrh(:,1),:);
22
23 c11 =  v13(:,3).^2.*(v14(:,1).^2+v14(:,2).^2)-          ...
24        2.*v13(:,1).*v13(:,3).*v14(:,1).*v14(:,3)-2.*   ...
25        v13(:,2).*v14(:,2).*(v13(:,1).*v14(:,1)+        ...
26        v13(:,3).*v14(:,3))+v13(:,2).^2.*(v14(:,1).^2+  ...
27        v14(:,3).^2)+v13(:,1).^2.*(v14(:,2).^2+         ...
28        v14(:,3).^2);
29 c11 = c11./T.detBk;
30 c22 =  v12(:,3).^2.*(v14(:,1).^2+v14(:,2).^2)-          ...
31        2.*v12(:,1).*v12(:,3).*v14(:,1).*v14(:,3)-2.*   ...
32        v12(:,2).*v14(:,2).*(v12(:,1).*v14(:,1)+        ...
33        v12(:,3).*v14(:,3))+v12(:,2).^2.*(v14(:,1).^2+  ...
34        v14(:,3).^2)+ v12(:,1).^2.*(v14(:,2).^2+        ...
35        v14(:,3).^2);
36 c22 = c22./T.detBk;
37 c33 =  v12(:,3).^2.*(v13(:,1).^2+v13(:,2).^2)-          ...
38        2.*v12(:,1).*v12(:,3).*v13(:,1).*v13(:,3)-2.*   ...
```

```
39          v12(:,2).*v13(:,2).*(v12(:,1).*v13(:,1)+          ...
40          v12(:,3).*v13(:,3))+v12(:,2).^2.*(v13(:,1).^2+    ...
41          v13(:,3).^2)+ v12(:,1).^2.*(v13(:,2).^2+          ...
42          v13(:,3).^2);
43  c33 = c33./T.detBk;
44  c12 =  -v12(:,3).*v13(:,3).*(v14(:,1).^2+v14(:,2).^2)+...
45          v12(:,3).*(v13(:,1).*v14(:,1)+                    ...
46          v13(:,2).*v14(:,2)).* v14(:,3)+                   ...
47          v12(:,2).*v14(:,2).*(v13(:,1).*v14(:,1)+          ...
48          v13(:,3).*v14(:,3))-v12(:,2).*v13(:,2).*(         ...
49          v14(:,1).^2+v14(:,3).^2)+                         ...
50          v12(:,1).*(v13(:,2).*v14(:,1).*v14(:,2)+          ...
51          v13(:,3).*v14(:,1).*v14(:,3)-                     ...
52          v13(:,1).*(v14(:,2).^2+v14(:,3).^2));
53  c12 = c12./T.detBk;
54  c13 = v12(:,3).*(v13(:,1).*v13(:,3).*v14(:,1)+            ...
55        v13(:,2).*v13(:,3).*v14(:,2)-                       ...
56        v13(:,1).^2.*v14(:,3)- v13(:,2).^2.*v14(:,3))+      ...
57        v12(:,1).*(-v13(:,2).^2.*v14(:,1)-                  ...
58        v13(:,3).^2.*v14(:,1)+                              ...
59        v13(:,1).*v13(:,2).*v14(:,2)+                       ...
60        v13(:,1).*v13(:,3).*v14(:,3))+                      ...
61        v12(:,2).*(v13(:,1).*v13(:,2).*v14(:,1)-            ...
62        v13(:,1).^2.*v14(:,2)+                              ...
63        v13(:,3).*(-v13(:,3).*v14(:,2)+v13(:,2).*v14(:,3)));
64  c13 = c13./T.detBk;
65  c23 = -v12(:,3).^2.*(v13(:,1).*v14(:,1)+                  ...
66        v13(:,2).*v14(:,2))+                                ...
67        v12(:,1).*v12(:,3).*(v13(:,3).*v14(:,1)+            ...
68        v13(:,1).*v14(:,3))+                                ...
69        v12(:,2).*(v12(:,1).*v13(:,2).*v14(:,1)+            ...
70        v12(:,1).*v13(:,1).*v14(:,2)+                       ...
71        v12(:,3).*v13(:,3).*v14(:,2)+                       ...
72        v12(:,3).*v13(:,2).*v14(:,3))-                      ...
73        v12(:,2).^2.*(v13(:,1).*v14(:,1)+                   ...
74        v13(:,3).*v14(:,3))-                                ...
75        v12(:,1).^2.*(v13(:,2).*v14(:,2)+                   ...
76        v13(:,3).*v14(:,3));
77  c23 = c23./T.detBk;
78
79  Sk = kron(c11', Sxx)+kron(c22', Syy)+kron(c33', Szz)+ ...
80       kron(c12', Sxy+Sxy')+ kron(c13',Sxz+Sxz')+       ...
81       kron(c23', Syz+Syz');
```

Lines 23-77 compute the coefficients of the matrix C_K that appears in the expression for the stiffness matrix. These expressions have been precomputed using Mathematica and transformed into Matlab/Octave code using the package ToMatlab [18].

Finally, lines 79-81 use the Kronecker product operator to assemble the global stiffness matrix $\boldsymbol{S}$. The Kronecker product of two matrices $A \in \mathbb{R}^{n\times m}$ and $B \in \mathbb{R}^{p\times q}$ is defined as

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1m}B \\ \vdots & \ddots & \vdots \\ a_{n1}B & \cdots & a_{nm}B \end{bmatrix} \in \mathbb{R}^{np\times mq}.$$

Here, the Kronecker product is used to stack the local stiffness matrices $\boldsymbol{S}^K$ for each

tetrahedron K into a block matrix `Sk` with dimensions `dofK` $\times$ `dofK` $\cdot$ `nttrh` as

$$\texttt{Sk} = \begin{bmatrix} \boldsymbol{S}^1 & \cdots & \boldsymbol{S}^{\texttt{nttrh}} \end{bmatrix}$$

Finally, the assembly is done with the following lines:

```
% Stiffness matrix assembly
[j,i] = meshgrid(1:dofK, 1:dofK);
indi  = zeros(dofK, dofK*nttrh); indj = indi;
indi(:) = T.ttrh(:,i)';
indj(:) = T.ttrh(:,j)';
S = sparse(indi, indj, Sk, nNodes, nNodes);
```

The most general case where $\underline{\underline{\boldsymbol{\kappa}}}$ is a matrix whose components are in general dependent of the position vectors is addressed with the function

```
[S] = femStiffnessMatrix(obj,K)
```

The input arguments are an object `obj` of the class and a 3×3 cell array of function handles `K`.

In order to compute the stiffness matrix $\boldsymbol{S}$, cubature rules are required once again. However, the integrands in this case involve the products of derivatives of the Lagrange basis functions. To evaluate these derivatives at the quadrature nodes, the code compute and stores them individually using the following expressions:

```
% Geometry data
T           = obj.mesh;
nttrh       = size(T.ttrh,1);
nNodes      = T.nNodes;
dofK = size(T.ttrh,2);

%Cubature Rule
degQuad      = max(2*T.deg-1,3);
quadRuleTtrh = obj.quadRule3D;
nodesQuad    = quadRuleTtrh{degQuad}.nodes;
weights      = quadRuleTtrh{degQuad}.weights;
nQ           = length(weights);

% Evaluate gradient of Interpolation function
gradPhi  = zeros(3*dofK,nQ);
for j=1:3
    for i=1:dofK
        gradPhi((j-1)*dofK+(i-1)+1,:) = ...
        obj.gradNj3D{j,i}(nodesQuad(1,:), nodesQuad(2,:), ...
                          nodesQuad(3,:), nodesQuad(4,:));
    end
end
Sx = gradPhi(1:dofK,:);
Sy = gradPhi(dofK+1:2*dofK,:);
Sz = gradPhi(2*dofK+1:end,:);
```

The variables `Sx`, `Sy`, `Sz` $\in \mathbb{R}^{\texttt{dofK}\times\texttt{nQ}}$ contain at each row the derivative of the Lagrange basis with respect to $\widehat{x}, \widehat{y}$ and, $\widehat{z}$ respectively. The next step is to compute the inverses of B_K.

```
% Bk^-1.sqrt(T.detBk)
v12 = T.coord(T.ttrh(:,2),:)-T.coord(T.ttrh(:,1),:);
v13 = T.coord(T.ttrh(:,3),:)-T.coord(T.ttrh(:,1),:);
v14 = T.coord(T.ttrh(:,4),:)-T.coord(T.ttrh(:,1),:);
```

```
51
52 b1 = cross(v13,v14,2);
53 b2 = cross(v14,v12,2);
54 b3 = cross(v12,v13,2);
55
56 ind = 1:nttrh;
57 ind32 = 3*ind(:)-2;
58 ind31 = 3*ind(:)-1;
59 ind30 = 3*ind(:);
60 indices = [ind32 ind31 ind30];
61 rowIndex = repmat(indices(:),3,1);
62 colIndex = reshape(repmat(indices,3,1),[],1);
63
64 Bkp   = [b1; b2; b3];
65 Bkp   = Bkp./sqrt(repmat(T.detBk,3,1));
66 BsInv = sparse(rowIndex, colIndex, Bkp);
```

This portion of the code compute the inverses of the Jacobian using Equation(2.26). Here, a factor of $\frac{1}{\det B_K}$ is missing in the inverse to ease the assembly of C_K. These matrices are stored in `BsInv` as a block diagonal matrix

$$\texttt{BsInv} = \begin{bmatrix} \texttt{Bs1} & & \\ & \ddots & \\ & & \texttt{Bsnttrh} \end{bmatrix} \in \mathbb{R}^{(3\cdot\texttt{nttrh})\times(3\cdot\texttt{nttrh})},$$

where `Bs1, Bs2` and so on, are the inverses (up to the previously mentioned factor).

The computation of the diffusion matrix at the cubature nodes in the physical domain is computed using:

```
68 % Cubature nodes in physical domain
69 px = T.coord(:,1); py = T.coord(:,2); pz = T.coord(:,3);
70 nodesX = px(T.ttrh(:,1:4))*nodesQuad;
71 nodesY = py(T.ttrh(:,1:4))*nodesQuad;
72 nodesZ = pz(T.ttrh(:,1:4))*nodesQuad;
73
74 checkNargin = cellfun(@nargin,K);
75 if sum(sum(checkNargin~=checkNargin(1)))
76     error(['The components of the diffusion matrix K should have' ...
77         ' the same number of inputs'])
78 elseif nargin(K{1,1})==3
79     K11 = K{1,1}(nodesX, nodesY, nodesZ);
80     K12 = K{1,2}(nodesX, nodesY, nodesZ);
81     K13 = K{1,3}(nodesX, nodesY, nodesZ);
82     K21 = K{2,1}(nodesX, nodesY, nodesZ);
83     K22 = K{2,2}(nodesX, nodesY, nodesZ);
84     K23 = K{2,3}(nodesX, nodesY, nodesZ);
85     K31 = K{3,1}(nodesX, nodesY, nodesZ);
86     K32 = K{3,2}(nodesX, nodesY, nodesZ);
87     K33 = K{3,3}(nodesX, nodesY, nodesZ);
88 elseif nargin(K{1,1})==4
89     K11 = K{1,1}(nodesX, nodesY, nodesZ,T.domain);
90     K12 = K{1,2}(nodesX, nodesY, nodesZ,T.domain);
91     K13 = K{1,3}(nodesX, nodesY, nodesZ,T.domain);
92     K21 = K{2,1}(nodesX, nodesY, nodesZ,T.domain);
93     K22 = K{2,2}(nodesX, nodesY, nodesZ,T.domain);
94     K23 = K{2,3}(nodesX, nodesY, nodesZ,T.domain);
95     K31 = K{3,1}(nodesX, nodesY, nodesZ,T.domain);
96     K32 = K{3,2}(nodesX, nodesY, nodesZ,T.domain);
```

```
    K33 = K{3,3}(nodesX, nodesY, nodesZ,T.domain);
end
```

Here the most general case of $\underline{\underline{\boldsymbol{\kappa}}}$ not being a symmetric matrix is considered to enable the assembly of a wider class of problems.

To apply the cubature formula for the computation of the stiffness matrix $\boldsymbol{S}$, a for loop is used to address potential memory issues that may arise when working with meshes that contain a large number of elements.

```
% Cubature formula
Sval = 0;
for i=1:nQ
    KK   = [K11(:,i) K12(:,i) K13(:,i); ...
            K21(:,i) K22(:,i) K23(:,i); ...
            K31(:,i) K32(:,i) K33(:,i)];

    Cs   = sparse(rowIndex, colIndex, KK);
    Cs   = BsInv*Cs*BsInv';
    aux  = diag(Cs);
    C11  = full(aux(1:3:end));
    C22  = full(aux(2:3:end));
    C33  = full(aux(3:3:end));
    aux  = diag(Cs,1);
    C12  = full(aux(1:3:end));
    C23  = full(aux(2:3:end));
    aux  = diag(Cs,-1);
    C21  = full(aux(1:3:end));
    C32  = full(aux(2:3:end));
    aux  = diag(Cs,2);
    C13  = full(aux(1:3:end));
    aux  = diag(Cs,-2);
    C31  = full(aux(1:3:end));

    Sval = Sval +  1/6*weights(i)*(   ...
        kron(C11',Sx(:,i)*Sx(:,i)') + ...
        kron(C22',Sy(:,i)*Sy(:,i)') + ...
        kron(C33',Sz(:,i)*Sz(:,i)') + ...
        kron(C12',Sx(:,i)*Sy(:,i)') + ...
        kron(C13',Sx(:,i)*Sz(:,i)') + ...
        kron(C21',Sy(:,i)*Sx(:,i)') + ...
        kron(C23',Sy(:,i)*Sz(:,i)') + ...
        kron(C31',Sz(:,i)*Sx(:,i)') + ...
        kron(C32',Sz(:,i)*Sy(:,i)')   ...
    );
end
```

Inside the loop, the matrix `Cs` is constructed to contain the matrices C_K in the block diagonal matrix

$$\begin{bmatrix} \texttt{C1} & & \\ & \ddots & \\ & & \texttt{Cnttrh} \end{bmatrix} = \begin{bmatrix} \texttt{Bs1} \cdot \texttt{KK1} \cdot \texttt{Bs1}^\top & & \\ & \ddots & \\ & & \texttt{Bsnttrh} \cdot \texttt{KKnttrh} \cdot \texttt{Bsnttrh}^\top \end{bmatrix}$$

where each element in the block has the form

$$\texttt{Ci} = \begin{bmatrix} C_{11}^{\texttt{i}} & C_{12}^{\texttt{i}} & C_{13}^{\texttt{i}} \\ C_{21}^{\texttt{i}} & C_{22}^{\texttt{i}} & C_{23}^{\texttt{i}} \\ C_{31}^{\texttt{i}} & C_{32}^{\texttt{i}} & C_{33}^{\texttt{i}} \end{bmatrix} \in \mathbb{R}^{3\times 3}.$$

The next lines collect the corresponding coefficients using the built-in Matlab function `diag(A,k)`.

This function returns the `k`th diagonal of the matrix `A` in vector form. The default value of `k` is zero, which returns the main diagonal of the matrix. If `k>0`, it returns the `k`th diagonal above the main diagonal, and if `k<0`, it returns the diagonal below the main diagonal. By using this function, the coefficients `Cij` of every tetrahedron are stored.

Line 125 adds the term of the cubature formula using a similar idea to the `femStressMatrix` case. The resulting matrix `Sval` has the same structure as the previous function and the assembly is done exactly the same.

3.8.3 Advection Matrix

The computation of the advection matrix is done with

```
[A] = femAdvectionMatrixfemAdvectionMatrix(obj, v)
```

The input to this function are: a 3×1 cell array `v` of functions handles and an object `obj` from the class.

```
% Geometry data
T = obj.mesh;
dofK = size(T.ttrh,2);
nNodes = T.nNodes;
nttrh=length(T.ttrh);

%Cubature Rule
degQuad      = max(2*T.deg-1,3);
quadRuleTtrh = obj.quadRule3D;
nodesQuad    = quadRuleTtrh{degQuad}.nodes;
weights      = quadRuleTtrh{degQuad}.weights;
nQ           = length(weights);

% Evaluate gradient of Interpolation function
gradPhi  = zeros(3*dofK,nQ);
for j=1:3
    for i=1:dofK
        gradPhi((j-1)*dofK+(i-1)+1,:) = ...
                obj.gradNj3D{j,i}(nodesQuad(1,:), nodesQuad(2,:), ...
                nodesQuad(3,:), nodesQuad(4,:));
    end
end
Sx = gradPhi(1:dofK,:);
Sy = gradPhi(dofK+1:2*dofK,:);
Sz = gradPhi(2*dofK+1:end,:);
clear gradPhi

% Evaluate Interpolation functions
PkValues = zeros(dofK, nQ);
for j = 1:length(obj.Nj3D)
    PkValues(j,:) = ...
            obj.Nj3D{j}(nodesQuad(1,:),nodesQuad(2,:), ...
            nodesQuad(3,:),nodesQuad(4,:));
end
```

The local matrix is assembled by computing the Lagrange functions and their gradients evaluated at the cubature nodes. Each component of the gradient function is stored

in $\texttt{Sx}, \texttt{Sy}, \texttt{Sz} \in \mathbb{R}^{\texttt{dofK}\times\texttt{nQ}}$, and the values of the Lagrange basis are stored in `PkValues`.

```
% Bk^-1.(T.detBk)
v12 = T.coord(T.ttrh(:,2),:)-T.coord(T.ttrh(:,1),:);
v13 = T.coord(T.ttrh(:,3),:)-T.coord(T.ttrh(:,1),:);
v14 = T.coord(T.ttrh(:,4),:)-T.coord(T.ttrh(:,1),:);

b1 = cross(v13,v14,2);
b2 = cross(v14,v12,2);
b3 = cross(v12,v13,2);

ind = 1:nttrh;
ind32 = 3*ind(:)-2;
ind31 = 3*ind(:)-1;
ind30 = 3*ind(:);
indices = [ind32 ind31 ind30];
rowIndex = repmat(indices(:),3,1);
colIndex = reshape(repmat(indices,3,1),[],1);

Bkp   = [b1; b2; b3];
BsInv = sparse(rowIndex, colIndex, Bkp);
```

The components of B_K up to a missing factor of $\frac{1}{\det B_K}$ are stored in the diagonals of the block matrix $\texttt{BsInv} \in \mathbb{R}^{3\cdot\texttt{nttrh}\times 3\cdot\texttt{nttrh}}$. The next step is to evaluate the vector `v` at the cubature nodes in the physical domain.

```
% Cubature nodes in physical domain
px = T.coord(:,1); py = T.coord(:,2); pz = T.coord(:,3);
nodesX = px(T.ttrh(:,1:4))*nodesQuad;
nodesY = py(T.ttrh(:,1:4))*nodesQuad;
nodesZ = pz(T.ttrh(:,1:4))*nodesQuad;

checkNargin = cellfun(@nargin,v);
if sum(sum(checkNargin~=checkNargin(1)))
    error(['The components of the velocity vector v should have' ...
        ' the same number of inputs'])
elseif nargin(v{1})==3
    vValx = v{1}(nodesX, nodesY, nodesZ);
    vValy = v{2}(nodesX, nodesY, nodesZ);
    vValz = v{3}(nodesX, nodesY, nodesZ);
elseif nargin(v{1})==4
    vValx = v{1}(nodesX, nodesY, nodesZ, T.domain);
    vValy = v{2}(nodesX, nodesY, nodesZ, T.domain);
    vValz = v{3}(nodesX, nodesY, nodesZ, T.domain);
end
vVal = [vValx vValy vValz];
```

With this code, all the components are stored in

$$\texttt{vVal} = \begin{bmatrix}\texttt{vValx} & \texttt{vValy} & \texttt{vValz}\end{bmatrix} = \begin{bmatrix} \texttt{v}^1_{\texttt{X},1} & \cdots & \texttt{v}^1_{\texttt{X},\texttt{nQ}} & \texttt{v}^1_{\texttt{Y},1} & \cdots & \texttt{v}^1_{\texttt{Y},\texttt{nQ}} & \texttt{v}^1_{\texttt{Z},1} & \cdots & \texttt{v}^1_{\texttt{Z},\texttt{nQ}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \texttt{v}^{\texttt{nttrh}}_{\texttt{X},1} & \cdots & \texttt{v}^{\texttt{nttrh}}_{\texttt{X},\texttt{nQ}} & \texttt{v}^{\texttt{nttrh}}_{\texttt{Y},1} & \cdots & \texttt{v}^{\texttt{nttrh}}_{\texttt{Y},\texttt{nQ}} & \texttt{v}^{\texttt{nttrh}}_{\texttt{Z},1} & \cdots & \texttt{v}^{\texttt{nttrh}}_{\texttt{Z},\texttt{nQ}} \end{bmatrix} \in \mathbb{R}^{\texttt{nttrh}\times(3\cdot\texttt{nQ})}.$$

Then, the cubature formula is applied following a similar idea to the assembly of the stiffness matrix.

```
Aval = 0;
```

```
93  for i=1:nQ
94      vnQ = sparse( indices, [ind ind ind], vVal(:, i:nQ:end), ...
95                                                  3*nttrh, 3*nttrh);
96      vnQ = BsInv*vnQ;
97      vnQ = vnQ(:,ind);
98
99      a1 = full(diag(vnQ(indices(:,1),:)));
100     a2 = full(diag(vnQ(indices(:,2),:)));
101     a3 = full(diag(vnQ(indices(:,3),:)));
102
103     Aval = Aval+ 1/6*weights(i)*( ...
104                 kron(a1', PkValues(:,i)*Sx(:,i)') + ...
105                 kron(a2', PkValues(:,i)*Sy(:,i)') + ...
106                 kron(a3', PkValues(:,i)*Sz(:,i)'));
107 end
```

At line 94, the block matrix

$$\texttt{vnQ} = \begin{bmatrix} \texttt{v}_\texttt{i}^1 & & \\ & \ddots & \\ & & \texttt{v}_\texttt{i}^\texttt{nttrh} \end{bmatrix}$$

is computed. Here $\texttt{v}_\texttt{i}^K$ is a column vector containing the components of the velocity function evaluated at the ith cubature node inside the tetrahedron K, this allows to perform at line 96 the block matrix product of the inverses and the velocity function

$$\begin{bmatrix} \texttt{Bs1} \cdot \texttt{v}_\texttt{i}^1 & & \\ & \ddots & \\ & & \texttt{Bsnttrh} \cdot \texttt{v}_\texttt{i}^\texttt{nttrh} \end{bmatrix}$$

Line 99-101 stores the components of the resultant column vectors in $\texttt{a1}, \texttt{a2}, \texttt{a3} \in \mathbb{R}^{\texttt{nttrh}}$. Lastly the line 103-106 adds the term of the individual component using the kronecker product. The assembly process is the same as for $\boldsymbol{S}$.

```
110 % Advection matrix assembly
111 [j, i]  = meshgrid(1:dofK,1:dofK);
112 indi    = zeros(dofK, dofK*nttrh); indj = indi;
113 indi(:) = T.ttrh(:,i)';
114 indj(:) = T.ttrh(:,j)';
115 A       = sparse(indi, indj, Aval, nNodes,nNodes);
```

3.8.4 Source term

The assembly of the source vector is performed using the function

```
b = femSourceTerm(obj, f)
```

where `f` is a function handle and `obj` is an object from the class.

```
12  % Geometry data
13  T = obj.mesh;
14  nTtrh  = size(T.ttrh,1);
15  nNodes = T.nNodes;
16  dofK = size(T.ttrh,2);
17
18  % Cubature rule
```

```
19 degQuad      = max(2*T.deg-1,3);
20 quadRuleTtrh = obj.quadRule3D;
21 nodesQuad    = quadRuleTtrh{degQuad}.nodes;
22 weights      = quadRuleTtrh{degQuad}.weights;
23 nQ           = length(weights);
24
25 % Evaluate Interpolation functions
26 PkValues = zeros(dofK, nQ);
27 for i = 1:length(obj.Nj3D)
28     PkValues(i,:) = obj.Nj3D{i}(nodesQuad(1,:),nodesQuad(2,:), ...
29                                 nodesQuad(3,:),nodesQuad(4,:));
30 end
31
32 % Cubature nodes in physical domain
33 px = T.coord(:,1); py = T.coord(:,2); pz = T.coord(:,3);
34 nodesX = px(T.ttrh(:,1:4))*nodesQuad;
35 nodesY = py(T.ttrh(:,1:4))*nodesQuad;
36 nodesZ = pz(T.ttrh(:,1:4))*nodesQuad;
37
38
39 if nargin(f) == 3
40     val = f(nodesX, nodesY, nodesZ);
41 elseif nargin(f) == 4
42     val = f(nodesX, nodesY, nodesZ, T.domain*ones(1,nQ));
43 end
44 val = val.*T.detBk;
45
46 % Cubature Formula
47 PkValues = repmat(PkValues,nTtrh,1);
48 val = repelem(val,dofK,1);
49 val = val.*PkValues;
50 val = val*weights(:)/6;
```

Similar to the mass matrix, line 26 compute the Lagrange basis evaluated at each cubature node and lines 39-44 compute the value of `f` at the cubature nodes inside each tetrahedron, which is then multiplied by the determinant of the same element in line 44.

To apply the cubature formula, line 47 repeats all the interpolating functions for every tetrahedron and line 48 repeats the values of `f` for every tetrahedron. This is used in line 49 to obtain all the integrands of the source vector over each tetrahedron and evaluated at each cubature node. Finally, line 50 computes the cubature rule

```
52 % Source vector assembly
53 indi = T.ttrh';
54 indi = indi(:);
55 b =  accumarray(indi, val, [nNodes, 1]);
```

Unlike the previous cases, here `accumarray` is used to assemble the vector. The functionality is similar to the `sparse` function. It creates a vector with `nNodes` components with entries equal to the values of `val` located in the positions as given by `indi`. As with the other function, in case an index appears multiple times the results are added.

3.8.5 Robin Boundary condition

The Robin boundary condition can be implemented by constructing the `nNodes` $\times$ `nNodes` boundary matrix $\boldsymbol{R}_\alpha$ and the `nNodes` traction vector $\boldsymbol{t}_r$. The assembly of these two terms are performed with the function

```
[t,MR] = femRobin(obj, robin, gN, varargin)
```

This function requires as inputs: `Robin`, a logical vector with `ntrB` components indicating which boundary elements belong to the Robin/Neumann boundary (i.e. the vector contains one for elements in those boundaries and zeros otherwise). The argument `gN` is a function handle with the boundary condition which may have two different forms: a scalar function handle or a function handle with vector input, i.e. `gN = @(x,y,z) [gNX(x,y,z), gnY(x,y,z), gnZ(x,y,z)]` (it may also have an optional fourth argument). Additionally, there are several optional arguments that can be used, as shown below:

```
p = inputParser;
addRequired(p,'obj')
addRequired(p,'robin')
addRequired(p, 'gN')
addParameter(p, 'alpha', @(x,y,z) 0.*x);
addParameter(p, 'exact', nan);
parse(p, obj, robin, gN, varargin{:});

% Obtention of robin parameters
alpha = p.Results.alpha;
u     = p.Results.exact;
```

The implementation of optional arguments in the function `femRobin` enables the passing of these arguments in any order by specifying the string `'alpha'` or `'exact'` as the first argument. If `'alpha'` is specified, the next parameter is expected to be a function handle in one of two forms: `@(x,y,z) alpha(x,y,z)` or `@(x,y,z,domBd) alpha(x,y,z,domBd)`. If this argument is not specified, the default value is set to `alpha = @(x,y,z) 0*x`, that is, we assume we are working with Neumann conditions. If `'exact'` is specified the next parameter is a function handle with the exact solution to the differential problem. This second term is only required if the boundary condition is robin and the user has only available the computation of the Neumann term as a function handle of either a scalar or a vector function.

```
% Geometry Data
T             = obj.mesh;
trBR          = T.trB(robin,:);
domBdR        = T.domBd(robin);
ntrBR         = size(trBR,1);
normal        = T.trBNormal(robin,:);
detAk         = vecnorm(normal,2,2);
nNodes        = T.nNodes;
dofA   = size(T.trB,2);


%%% Quadrature rule
degQuad     = max(2*T.deg-1,3);
quadRuleTr  = obj.quadRule2D;
nodesQuad   = quadRuleTr{degQuad}.nodes;
weights     = quadRuleTr{degQuad}.weights; weights = weights(:);
nQ          = length(weights);

% Evaluate Interpolation functions
PkValues = zeros(dofA, nQ);
for j = 1:length(obj.Nj2D)
    PkValues(j,:) = obj.Nj2D{j}(nodesQuad(1,:),...
                                nodesQuad(2,:),nodesQuad(3,:));
end

% Quadrature nodes in physical domain
```

```
72 px = T.coord(:,1); py = T.coord(:,2); pz = T.coord(:,3);
73 nodesX = px(trBR(:,1:3))*nodesQuad;
74 nodesY = py(trBR(:,1:3))*nodesQuad;
75 nodesZ = pz(trBR(:,1:3))*nodesQuad;
76
77 %%%  Traction term
78 if nargin(gN) == 3
79     val = gN(nodesX, nodesY, nodesZ);
80 elseif nargin(gN) == 4
81     val = gN(nodesX, nodesY, nodesZ, domBdR*ones(1,nQ));
82 end
83
84 if size(val,2)==nQ*3
85     normal  = kron(normal, ones(1, size(val,2)/3));
86     val     = val.*normal;
87     val     = val(:,1:nQ)+val(:,nQ+1:2*nQ)+val(:,2*nQ+1:end);
88 elseif size(val,2)==nQ
89     val     = val.*detAk;
90 end
91 if isa(u,'function_handle')
92     if nargin(p.Results.exact)==3
93         val = val + alpha(nodesX, nodesY, nodesZ).*...
94                     u(nodesX, nodesY, nodesZ).*detAk;
95     elseif nargin(p.Results.exact)==4
96         val = val + alpha(nodesX, nodesY, nodesZ, domBdR).*...
97                     u(nodesX, nodesY, nodesZ, domBdR).*detAk;
98     end
99 end
100 val = repmat(PkValues,ntrBR,1).*repelem(val,dofA, 1);
101 val = val*weights/2;
```

The computation of the Lagrange function is done as in the previous cases but now using their expressions restricted to the faces. The next step is to compute the function `gN` at the quadrature nodes. If it was given as a function handle with a vector function then

$$\texttt{val} = \begin{bmatrix} \texttt{gN}^1_{\texttt{X},1} & \cdots & \texttt{gN}^1_{\texttt{X},\texttt{nQ}} & \texttt{gN}^1_{\texttt{Y},1} & \cdots & \texttt{gN}^1_{\texttt{Y},\texttt{nQ}} & \texttt{gN}^1_{\texttt{Z},1} & \cdots & \texttt{gN}^1_{\texttt{Z},\texttt{nQ}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \texttt{gN}^{\texttt{ntrBR}}_{\texttt{X},1} & \cdots & \texttt{gN}^{\texttt{ntrBR}}_{\texttt{X},\texttt{nQ}} & \texttt{gN}^{\texttt{ntrBR}}_{\texttt{Y},1} & \cdots & \texttt{gN}^{\texttt{ntrBR}}_{\texttt{Y},\texttt{nQ}} & \texttt{gN}^{\texttt{ntrBR}}_{\texttt{Z},1} & \cdots & \texttt{gN}^{\texttt{ntrBR}}_{\texttt{Z},\texttt{nQ}} \end{bmatrix} \in \mathbb{R}^{\texttt{ntrBR}\times(3\cdot\texttt{nQ})}$$

is a block matrix where at each block there is each component of the gradient vector evaluated at each quadrature node. On the other hand, if it is a scalar function then

$$\texttt{val} = \begin{bmatrix} \texttt{gN}^1_1 & \cdots & \texttt{gN}^1_{\texttt{nQ}} \\ \vdots & \ddots & \vdots \\ \texttt{gN}^{\texttt{ntrBR}}_1 & \cdots & \texttt{gN}^{\texttt{ntrBR}}_{\texttt{nQ}} \end{bmatrix} \in \mathbb{R}^{\texttt{ntrBR}\times\texttt{nQ}}.$$

The first case is considered at lines 84-87. Here

$$\texttt{normal} = \begin{bmatrix} n^1_x & \cdots & n^1_x & n^1_y & \cdots & n^1_y & n^1_z & \cdots & n^1_z \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ n^{\texttt{ntrBR}}_x & \cdots & n^{\texttt{ntrBR}}_x & n^{\texttt{ntrBR}}_y & \cdots & n^{\texttt{ntrBR}}_y & n^{\texttt{ntrBR}}_z & \cdots & n^{\texttt{ntrBR}}_z \end{bmatrix} \in \mathbb{R}^{\texttt{ntrBR}\times\texttt{nQ}},$$

is a matrix containing at each row the components of the normal vectors to each element. Then, lines 86-87 computes the dot product between the two vectors for each quadrature node. Here, the normal vectors contain the Jacobian of the parameterization, this is the reason why there

is no need to add the Jacobian to this term. The case where `gN` is a scalar function handle is handled at line 89.

In case `exact` is given, the term αu is added to the previous term at lines 91-99. The quadrature rule is implemented using:

```
100 val = repmat(PkValues,ntrBR,1).*repelem(val,dofA, 1);
101 val = val*weights/2;
```

Finally, to assemble the traction vector, the idea is the same as for the source vector.

```
103 % Traction vector assembly
104 id  = trBR';
105 t   = accumarray(id(:), val, [nNodes, 1]);
```

The computation of the matrix $\boldsymbol{R}_\alpha$ is done the exact same way as $\boldsymbol{M}_c$ but with the Lagrange basis in the triangular element.

```
103 %%% Mass Boundary matrix
104 % Product of interpolation function at every quadrature node
105 PkProd = repmat(PkValues, dofA, 1).*repelem(PkValues, dofA, 1);
106 PkProd = repmat(PkProd, ntrBR, 1);
107
108
109 if nargin(alpha) == 3
110     val = alpha(nodesX, nodesY, nodesZ);
111 elseif nargin(alpha) == 4
112     val = alpha(nodesX, nodesY, nodesZ, domBdR*ones(1,nQ));
113 end
114 val = val.*detAk;
115
116 val = repelem(val, dofA^2, 1);
117 val = val.*PkProd;
118 val = val*weights/2;
119
120 % Boundary mass matrix assembly
121 indi = repmat(trBR, 1, dofA);  indi=indi';
122 indj = repelem(trBR, 1, dofA); indj=indj';
123 MR   = sparse(indi(:), indj(:), val(:), nNodes, nNodes);
```

3.9 Finite element evaluation

After obtaining the solution of the FEM problem at the nodes, it may be necessary to evaluate the solution at other points in the domain. This can be accomplished easily if the tetrahedron that contains each point is known, as the solution can be evaluated using the barycentric coordinates of the point and Equation(2.13). This is done with the function

```
[val,Meval,barPt,indTtrh,indPtError] = evalFEM3DUh(uh,obj,pt,varargin)
```

The function takes as inputs the finite element defined by its values at the nodes (`uh`), an object from the class `fem3Dclass`, a matrix `pt` containing at each row the coordinates of the points where the solution is to be interpolated, and an optional parameter `list` containing the indices of the tetrahedra that might contain the points. If `list` is passed, the function will locate the points in `pt` over these tetrahedra. Otherwise, the computation is performed for all tetrahedra in the mesh. This allows for computations of multiple problems in case we have the same domain without having to recompute everything each time.

The output includes a vector `val` with the finite element evaluated at the input points, a matrix `Meval` containing the basis functions evaluated at each point, a matrix `barPt` with the barycentric coordinates with respect to the tetrahedron each point belongs to, a vector `indTtrh` where the jth entry points to the tetrahedron (in the field `ttrh`) the jth point belongs to, and a logical vector `indPtError` with the points where it was not possible to find a tetrahedron containing it. If the input `uh` is an empty array `[]`, then the output `val` has the value `NaN`. For the points where `indPtError=true`, the obtained values are extrapolations to the nearest tetrahedron. If those values are not of interest, then `val(~indPtError)` retrieves only the rows belonging to nodes inside the domain. The steps performed in this function are shown below.

```
42 %% Find Elements containing the points
43 x = obj.mesh.coord(:,1);
44 y = obj.mesh.coord(:,2);
45 z = obj.mesh.coord(:,3);
46
47 x = x(obj.mesh.ttrh(list,1:4).');
48 y = y(obj.mesh.ttrh(list,1:4).');
49 z = z(obj.mesh.ttrh(list,1:4).');
50
51 % Block diagonal matrix to compute the barycentric coordinates
52 nT = length(list);
53 jaux = 1:nT*4;
54 jaux = [1 1 1 1]'*jaux(:)';
55
56 iaux = 1:4:nT*4;
57 iaux = kron(iaux, [1 1 1 1]);
58 iaux= bsxfun(@plus,[0 1 2 3]',iaux);
59
60
61 matrix = sparse( iaux(:), jaux(:),[ones(1,4*nT); x(:)'; y(:)';z(:)'], 4*nT
       ,4*nT);
62
63
64
65 nPt    = size(pt,1);
66 bt = kron(ones(nT,1),[pt(:,1).^0   pt ]');
67
68 aux = matrix\bt;
```

At lines 43-49, the coordinates of the vertices of the tetrahedra where the points are going to be tested against, are stored in a matrix of size `nPt` $\times$ `nT`, where `nPt` is the number of points and `nT` is the length of `list`. Then, at line 61, the matrix

$$\texttt{matrix} = \begin{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix}_1 & & \\ & \ddots & \\ & & \begin{bmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix}_{\texttt{nttrh}} \end{bmatrix} \in \mathbb{R}^{4\cdot\texttt{nttrh}\times 4\cdot\texttt{nttrh}}$$

is created, which contains the coefficient matrix needed to compute the barycentric coordinates of a point as given in Equation(2.6) for every tetrahedron. At line 66, the right-hand side of this Equation is computed for every point and is stored in

$$\texttt{bt} = \begin{bmatrix} \mathbf{1}_{\texttt{nPt}}^{\top} \\ \texttt{pt}^{\top} \\ \vdots \\ \mathbf{1}_{\texttt{nPt}}^{\top} \\ \texttt{pt}^{\top} \end{bmatrix} \in \mathbb{R}^{4\cdot \texttt{nttrh} \times \texttt{nPt}}$$

where $\mathbf{1}_{\texttt{nPt}}^{\top}$ is a row vector full of ones with length `nPt`.

Line 68 solves a block system of equations to find the barycentric coordinates of each point for all tetrahedra in `list`. Each block in this system corresponds to a single tetrahedron in the mesh and is responsible for finding the barycentric coordinates of every point solving

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix}_{\texttt{i}}^{-1} \begin{bmatrix} 1 & \cdots & 1 \\ \texttt{pt}_1 & \cdots & \texttt{pt}_{\texttt{nPt}} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix}_{\texttt{i}}^{-1} \begin{bmatrix} 1 \\ \texttt{pt}_1 \end{bmatrix} & \cdots & \begin{bmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix}_{\texttt{i}}^{-1} \begin{bmatrix} 1 \\ \texttt{pt}_{\texttt{nPt}} \end{bmatrix} \end{bmatrix}.$$

Once the system is solved, the resulting matrix has the form

$$\texttt{aux} = \begin{bmatrix} \boldsymbol{\lambda}^{\mathbf{1}}(\texttt{pt}_1) & \cdots & \boldsymbol{\lambda}^{\mathbf{1}}(\texttt{pt}_{\texttt{nPt}}) \\ & \vdots & \\ \boldsymbol{\lambda}^{\texttt{nttrh}}(\texttt{pt}_1) & \cdots & \boldsymbol{\lambda}^{\texttt{nttrh}}(\texttt{pt}_{\texttt{nPt}}) \end{bmatrix} \in \mathbb{R}^{4\cdot \texttt{nttrh} \times \texttt{nPt}}.$$

This block matrix contains at each row the barycentric coordinates of all points for a specific tetrahedron. We can use this to determine the location of each point.

```
69 ind = abs(aux(1:4:end,:))+abs(aux(2:4:end,:)) + ...
70     abs(aux(3:4:end,:))+abs(aux(4:4:end,:));
71
72 [aux2,indTtrh] = min(abs(ind-1));
73
74 Meval = sparse(nPt,obj.mesh.nNodes);
75 ind3= sub2ind(size(ind),indTtrh,1:nPt);
76
77 % barycentric coordinates
78 barPt= [aux(4*(ind3-1)+1); ...
79         aux(4*(ind3-1)+2); ...
80         aux(4*(ind3-1)+3); ...
81         aux(4*(ind3-1)+4)];
82 % in tetrahedra indTtrh
```

At line 69, the sum over the absolute values of the barycentric coordinates of each point is computed and stored in a matrix $\texttt{ind} \in \mathbb{R}^{\texttt{nttrh} \times \texttt{nPt}}$, where each row represents a tetrahedron and each column contains the sum of the barycentric coordinates of a point. For the sum to be greater than one it must mean that at least one coordinate is negative and is therefore outside the element, therefore the only way a point is inside the element is for the sum be equal to one.

This is checked at line 72, and the tetrahedra containing each point are stored in `indTtrh`. If a point is outside the mesh, the value in `aux2` will not equal zero, and the *closest* tetrahedron is stored instead. At line 75, the location of every tetrahedron is transformed to linear indices to obtain the tetrahedra using only one index, which is used in line 78 to store the barycentric coordinates of the points inside the tetrahedra containing them in

$$\texttt{barPt} = \left[\widehat{\boldsymbol{\lambda}}^{i_1}(\texttt{pt}_1) \quad \cdots \quad \widehat{\boldsymbol{\lambda}}^{i_{\texttt{nPt}}}(\texttt{pt}_{\texttt{nPt}})\right] \in \mathbb{R}^{4\times\texttt{nPt}}.$$

```
93  aux = zeros(size(obj.mesh.ttrh,2),nPt);
94  for j = 1:length(obj.Nj3D)
95      aux(j,:)= obj.Nj3D{j}(barPt(1,:),barPt(2,:),...
96                            barPt(3,:),barPt(4,:));
97  end
98  % index
99  indi = kron(1:nPt,ones(1,size(obj.mesh.ttrh,2))) ;
100 indj = obj.mesh.ttrh(list(indTtrh),:)'; indj = indj(:);
101 Meval = sparse(indi,indj,aux(:),nPt,obj.mesh.nNodes);
102
103 % Evaluation:
104
105 val = Meval*uh;
```

The Lagrange basis functions are computed at lines 93-97 using the barycentric coordinates. Then, at lines 99-101, the matrix $\texttt{Meval} \in \mathbb{R}^{\texttt{nPt}\times\texttt{nNodes}}$ is assembled, where each row contains the basis function values φ_i evaluated at a point. Most of these entries are zero, except for those functions associated with the nodes contained in the element where the point belongs. This matrix is used to compute the solution at the specified points via a matrix-vector product, and the results are stored in $\texttt{val} \in \mathbb{R}^{\texttt{nPt}}$.

Chapter 4

Numerical examples

In this section, we present a series of numerical experiments conducted using the package introduced in the previous chapter. First, we examine the errors obtained for various orders of approximation when compared to a known manufactured solution. This analysis is performed on multiple meshes to assess the accuracy and convergence properties of our package. Next, we showcase the capabilities of our package by solving two different problems. Through these examples, we demonstrate the utilization of specific functionalities developed within our package.

4.1 Error analysis

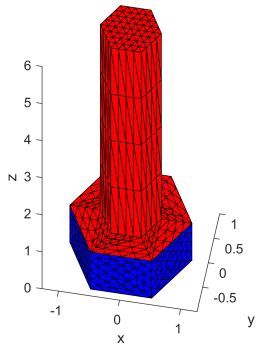


Figure 4.1: Domain of study.

We consider the advection-diffusion-reaction equation

$$\begin{cases} -\nabla \cdot \left(\underline{\underline{\boldsymbol{\kappa}}} \nabla u\right) + \boldsymbol{\beta} \cdot \nabla u + cu &= f, & \text{in } \Omega, \\ u &= u_D, & \text{on } \Gamma_D, \\ \left(\underline{\underline{\boldsymbol{\kappa}}} \nabla u\right) \cdot \widehat{\boldsymbol{n}} + \alpha u &= g_R & \text{on } \Gamma_R, \end{cases} \tag{4.1}$$

with diffusion matrix $\underline{\underline{\boldsymbol{\kappa}}} = \mathcal{I}_3$, i.e. the identity matrix, the velocity field is defined as

$$\boldsymbol{\beta} = \begin{bmatrix} 0.5(1+\cos xy) & 0.5(1-\cos xz) & 0.5(1+\sin yz) \end{bmatrix}^\top$$

and reaction term $c = 0.5e^{-xyz}$. The source term f is chosen such that the solution to this problem is given by $u = \cos \pi x \cos \pi y \cos \pi z$. The coefficient α in the Robin condition is chosen as $\alpha = 0.1 + x^2 + y^2$. For the boundary conditions, the Dirichlet boundary data u_D is obtained from the exact solution, and the Robin data g_R is obtained from $\left(\underline{\underline{\boldsymbol{\kappa}}} \nabla u\right) \cdot \widehat{\boldsymbol{n}} + \alpha u = g_N + \alpha u = g_R$.

In Figure 4.1, we can see the polygonal domain used in this study. The boundaries are color-coded: the Dirichlet boundary Γ_D is represented in red, and the Robin boundary Γ_R is shown in blue. This choice of a polyhedra domain is done to prevent geometric errors that arise from approximating a curved domain Ω. By using a polyhedral domain, any errors obtained in the resolution are solely attributed to the approximation of the exact solution with piecewise polynomials.

Since the solution u is known in advance, the errors can be computed exactly by comparing the results obtained at the nodes. For this case, we use the max-norm, which is computed by obtaining the largest error in absolute value from the exact solution:

$$\|u - u_h\|_\infty = \max_j \{|u(\boldsymbol{v}_j) - u_h(\boldsymbol{v}_j)|\}.$$

In this example, we have tested the behavior of all elements considered in this dissertation, ranging from $\mathbb{P}_1$ to $\mathbb{P}_4$ elements. For each element, we have used four meshes of different sizes. For this purpose, we have specified the maximum size allowed for the elements in GMSH and allowed the software to individually mesh each case. Consequently, instead of obtaining the same meshes with nodes appropriately located for elements of each degree, we obtained different meshes. This does not pose a problem for the subject of study, as this test only aims to show the convergence of u_h to the solution u. The conduction of this experiment was performed with the 128GB Turing computer in Tudela, running Ubuntu 220.04 LTS with a processor Intel Xeon E5-2630 v4 (10 cores) at 3.1GHz.

4.1.1 MSH file reading

We start by examining the time required for the function `importGMSH3D` to create the `T.mesh` structure. To evaluate this, we compared the time taken in relation to the number of elements and the number of nodes. The results of this comparison are presented in Figure 4.2.

From the results, we observe that the function exhibits similar performance when the number of nodes in the mesh remains constant regardless of the order of the elements. However, when considering the number of tetrahedra in the mesh, there is a significant difference in the size of the meshes generated. This indicates that the distribution of nodes in the mesh, as determined by the number of elements, as well as the degree of the elements, is more crucial than the overall number of nodes for the function performance.

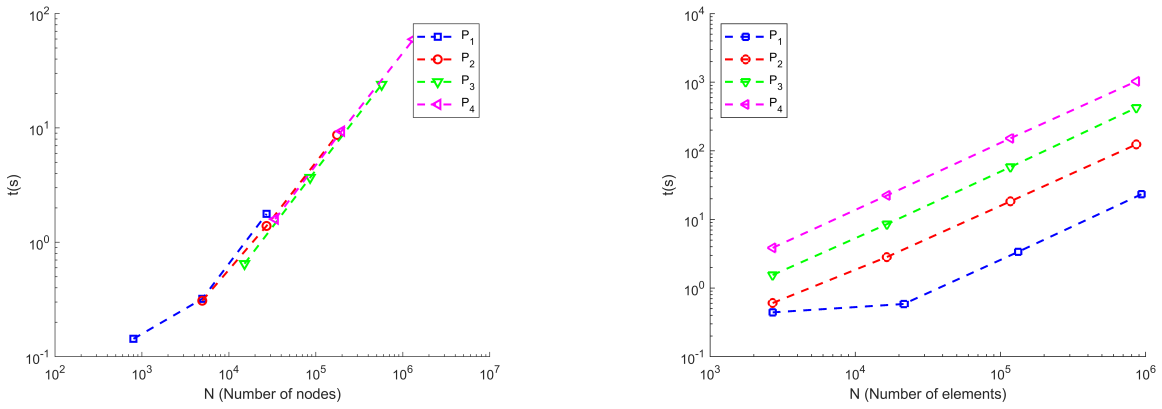


Figure 4.2: File reading time.

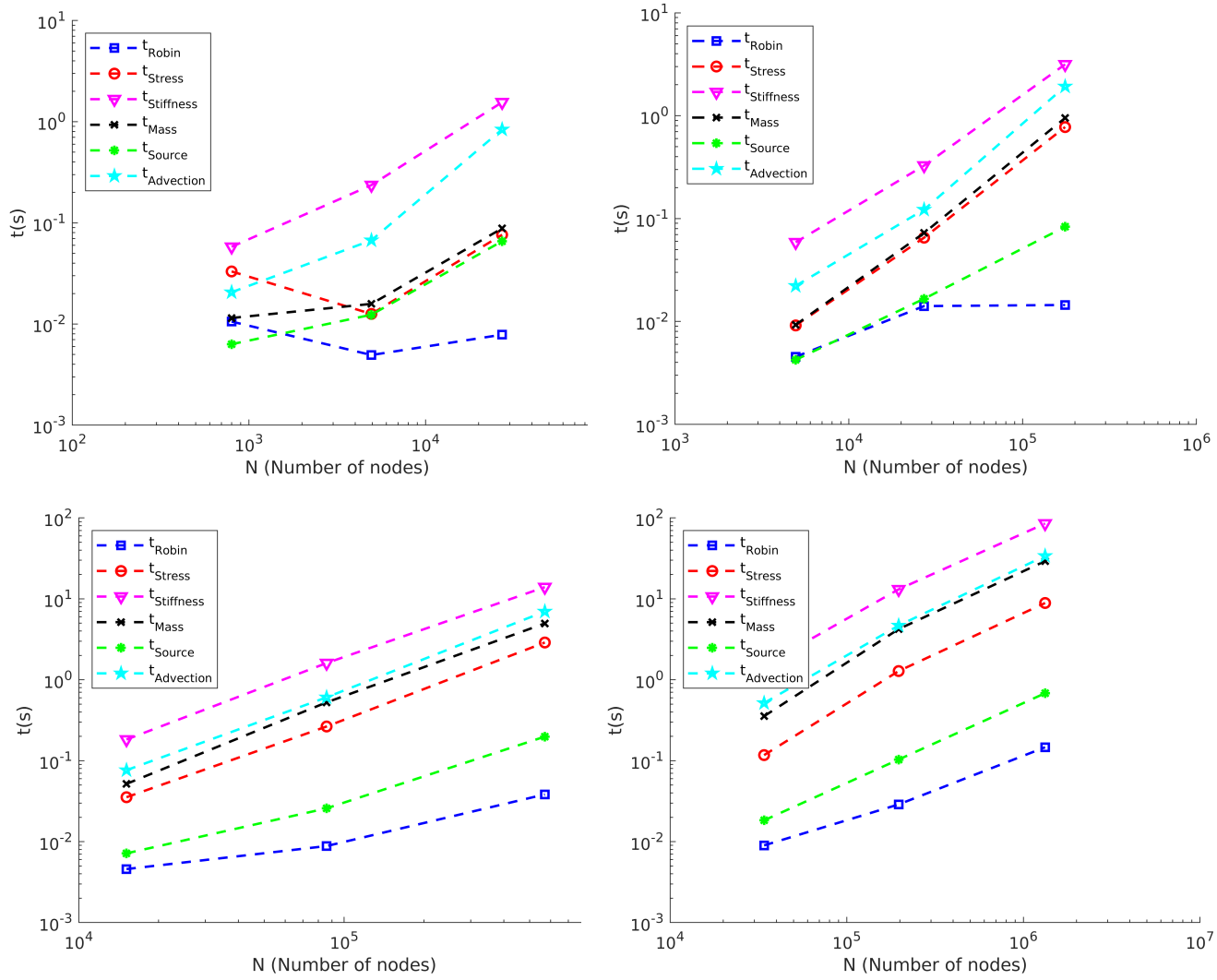


Figure 4.3: Assembly time of each term for $\mathbb{P}_1$ elements (above-right), $\mathbb{P}_2$ elements (above-left), $\mathbb{P}_3$ elements (below-left) and $\mathbb{P}_4$ elements (below-right).

4.1.2 Assembly time

Next, we examine the time taken to assemble each term of the system of equations. Since the diffusion matrix is the identity, we can use both functions `femStiffnessMatrix` and `femStressMatrix` to assemble the stiffness matrix. Figure 4.3 illustrates the time required to construct each term. In the example only the computation for the first three meshes are shown.

We observe that the assembly time for the Robin mass matrix and the traction vector is significantly shorter than for the other terms. This is expected since the number of elements on the boundary is, in general, much smaller than the number of tetrahedral elements, resulting in a reduced number of computations.

The assembly of the source term follows, and as we increase the order of the elements, the difference in assembly time compared to the other terms becomes more noticeable. This can be attributed to the fact that higher-order elements involve a greater number of Lagrange basis functions and nodes for the cubature formulas. Also, this term requires working with each individual Lagrange basis, consequently this function requires fewer operations compared to performing products of these terms, leading to a faster assembly time.

The stiffness matrix constructed with `femStressMatrix` follows, which makes sense as it does not involve the use of cubature rules. Next in terms of computational time are the assembly of the mass matrix, advection matrix, and the stiffness matrix. As expected, the stiffness matrix takes the longest, since it involves a larger number of operations. However, we encountered issues with the assembly of the mass matrix. Specifically, the assembly of `PkProd` leads to large matrices beings constructed, resulting to noticeable memory problems. Nevertheless, this is not a problem unless very fine meshes are used.

Overall, the time required for assembling the different terms of the system of equations varies as we increase the number of nodes. However, this time remains at reasonable values, even for large systems. For some of the tested cases the time required for the assembly takes longer than the resolution of the system, but even in those cases the computing time is small.

4.1.3 Convergence analysis

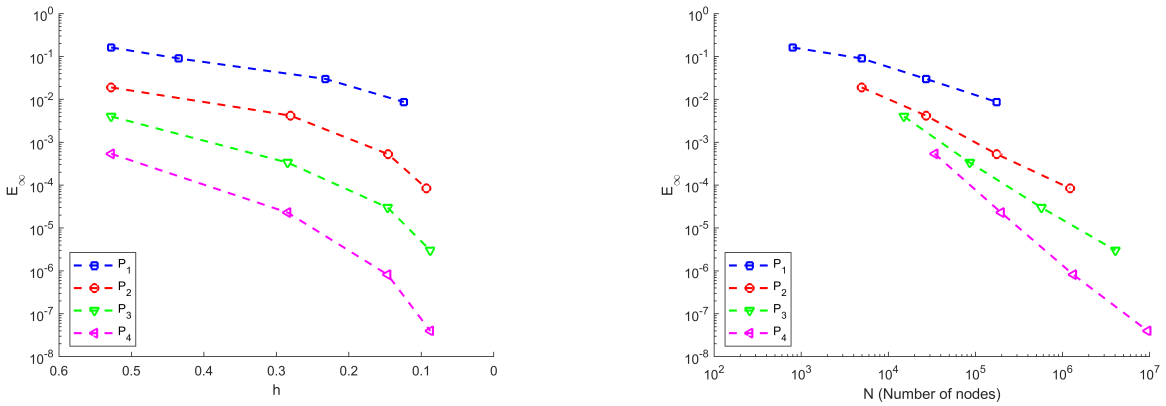


Figure 4.4: Errors in the maximum norm versus the diameter of the mesh (left) and the number of nodes(right).

For solving the system of equations we have used the `backslash` command in Matlab except for the second and third refinement. In these cases we have solved the system by a Krylov (iterative) solver, specifically the biconjugate gradient stabilized method with a stopping criteria (tolerance) for the relative residual equal to 1×10^{-11}. The method is preconditioned with an incomplete LU factorization. Despite the very small tolerance used in the calculations, this preconditioner is shown to significantly reduce the number of iterations to less than 1,000 in this particular experiment. Let us point out that the purpose of selecting such a small tolerance, which is several orders of magnitude smaller than the expected error of the finite element solution itself, is done with the aim of ensuring that the iterative method has a negligible impact on the error in the finite element approximation.

Table 4.1 presents the diameter of each tested mesh, which is defined as the length of the largest edge in the mesh. With this, we have a measure on the size of the mesh. From the table, we can see that for two consecutive meshes, there is no a factor of two in the size of the mesh. This is because, unlike triangles, a tetrahedron cannot be split into several similar tetrahedrons. The variation in the mesh size is attributed to the fact that GMSH generates non-structured meshes. Once again, we point out that in this study, less emphasis was placed on carefully controlling the mesh generation process, which contributed to the observed differences in mesh structures.

The errors obtained in the maximum norm are visually represented in Figure 4.4 and quantitatively summarized in Table 4.2. These results show that the numerical solution does in

fact converge to the exact solution, as noticed from the fact that the error is reduced as we use finer meshes. As anticipated, increasing the degree of the polynomial basis leads to improved approximations, resulting in smaller errors.

	No-refinement	First refinement	Second refinement	Third refinement
$\mathbb{P}_1$	5.2793×10^{-1}	4.3481×10^{-1}	2.3237×10^{-1}	1.2414×10^{-1}
$\mathbb{P}_2$	5.2793×10^{-1}	2.8048×10^{-1}	1.4582×10^{-1}	9.2956×10^{-2}
$\mathbb{P}_3$	5.2793×10^{-1}	2.8458×10^{-1}	1.4654×10^{-1}	8.7633×10^{-2}
$\mathbb{P}_4$	5.2793×10^{-1}	2.8458×10^{-1}	1.4654×10^{-1}	8.7633×10^{-2}

Table 4.1: Mesh diameter h.

	No-refinement	First refinement	Second refinement	Third refinement
$\mathbb{P}_1$	1.6214×10^{-1}	8.9875×10^{-2}	2.9961×10^{-2}	8.6810×10^{-3}
$\mathbb{P}_2$	1.8983×10^{-2}	4.1761×10^{-3}	5.2820×10^{-4}	8.2146×10^{-5}
$\mathbb{P}_3$	3.9931×10^{-3}	3.3593×10^{-4}	3.0034×10^{-5}	3.0094×10^{-6}
$\mathbb{P}_4$	5.3841×10^{-4}	2.2974×10^{-5}	8.1202×10^{-7}	3.9850×10^{-8}

Table 4.2: Errors in the maximum norm.

4.2 Loaded Connecting rod

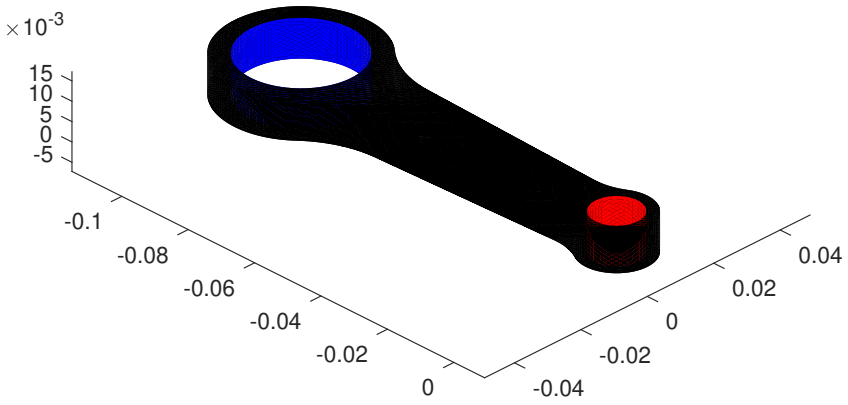


Figure 4.5: Connecting rod geometry.

The connecting rod is a vital mechanical component within the crank-connecting rod mechanism. Its primary function is to convert a reciprocating motion into rotational motion of the crank, and vice versa. This mechanism is widely employed in internal combustion engines, serving as a means to convert the thermal energy generated during the combustion process into mechanical energy. This mechanism operates on the principle that when combustion takes place, the piston moves in a downward direction, transmitting this motion to the connecting rod. Consequently, the connecting rod converts this linear motion into circular motion, transferring it to the crankshaft. The resulting circular motion can then be employed to drive the transmission system, ultimately powering the wheels and propelling the vehicle forward.

This study is dedicated to examining the displacement of the rod when subjected to a uniform pressure load of $P = 20\text{MPa}$ applied to the smaller end of the road. This load represents

the force transmitted by the piston to the cylindrical area. For the purpose of our analysis, we have chosen to focus on steel as the material of interest, with Young's modulus of $E = 200\text{GPa}$ and a Poisson's ratio of $\nu = 0.3$.

To study the static state of the rod it is necessary to prevent any translational or rotational motion. For this reason we have considered the larger end, which is connected to the crankshaft, to be a fixed support. By adopting this configuration, we can effectively analyze and understand the behavior of the rod under the applied pressure load.

The geometry considered is shown in Figure 4.5. In blue, we can see the portion of the connecting rod that is connected to the crankshaft. As mentioned before, this surface is modelled as a fixed support, therefore the displacements of all nodes belonging to it are set to zero. On the other hand, in red, we have the cylindrical surface that will be in contact with the piston (or more specifically the needle bearing between these two elements). When modeling the load on the smaller end, it is necessary to consider that the load varies across different triangles in the mesh due to variations in both the area and orientation of the elements. The compressive load vector experienced by a triangle A can be defined as:

$$P_A = -P\frac{\boldsymbol{n}_A}{\|\boldsymbol{n}_A\|},$$

where $\boldsymbol{n}_A$ is the normal to the triangle. This expression allows computing the contribution of the load to each individual element in the boundary.

For the numerical test we have used a mesh with $\mathbb{P}_2$ elements comprised of $129,218$ nodes, $83,333$ tetrahedra, and $18,732$ triangles on the boundary. The displacements obtained are represented in Figure 4.6. The total displacement was computed using the usual Euclidean norm

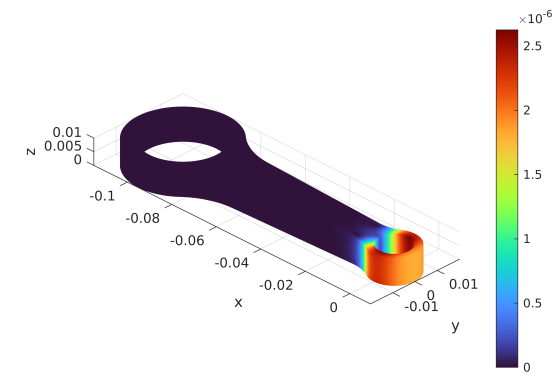
$$\|\boldsymbol{u}\| = \sqrt{\boldsymbol{u}_x^2 + \boldsymbol{u}_y^2 + \boldsymbol{u}_z^2}.$$

The results shows that the smaller end undergoes the most significant displacement, with a maximum value of $\|\boldsymbol{u}\| \approx 2.5\mu\text{m}$.

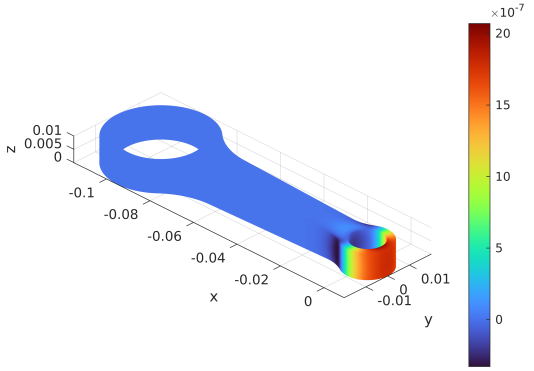
For the remaining displacements, their magnitudes and directions are represented. In particular, for the $\boldsymbol{u}_x$ displacement, it can be observed that the outermost part experiences the most movement in this orientation, with a displacement $\boldsymbol{u}_x \approx 2.2\mu\text{m}$. This result is expected, as this is the zone where the compression loads points the most towards the x-axis. From this result it can be seen that this displacement is positive, indicating that this portion tends to stretch.

As a consequence of this stretching, there is also an elongation in the neighbor points, albeit to a lesser extent. This attenuation occurs due to the combined influence of the load acting upon those points and the body of the connecting rod, which counters the stretching motion by exerting a force in the opposite direction to maintain the equilibrium. Consequently, it can be observed that the displacement diminishes significantly as one moves away from the smaller end.

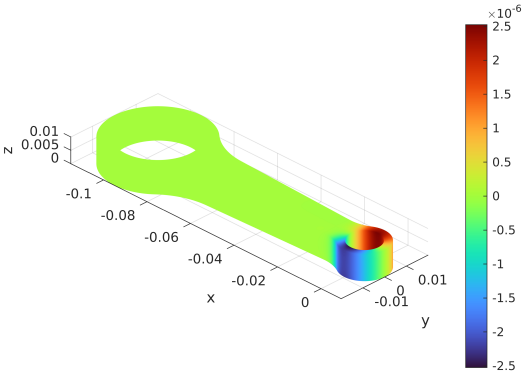
Regarding the displacement $\boldsymbol{u}_y$, it can be observed that the sections with the most displacement are those where the compressive load points in the direction of the y-axis. From their sign, it is apparent that those regions are stretching. The maximum values attained are similar to before, with a displacement of $\boldsymbol{u}_y \approx 2.4\mu\text{m}$. This result is consistent with what we would be expected in reality, as the plot shows that the cylinder tends to expand symmetrically around its center due to the compressive load in the interior.



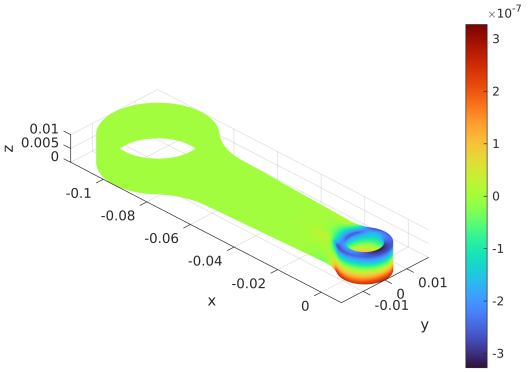
(a) Total displacement.



(b) Displacement in the x-direction.



(c) Displacement in the y-direction.



(d) Displacement in the z-direction.

Figure 4.6: Displacements in the connecting rod.

Computing the displacement, u_z we can see that nonzero values appear in the smaller end. This occurrence can be attributed to the discretization of the mesh. Since the surface is approximated using triangles, the normal vectors possess slight tilts away from the xy-plane. Consequently, when the load is projected onto these elements, a nonzero component in the z direction appears. However, it is important to note that the obtained displacements for u_z are still one order of magnitude smaller compared to the other displacements and, therefore, do not significantly influence the overall results.

4.3 Heat equation

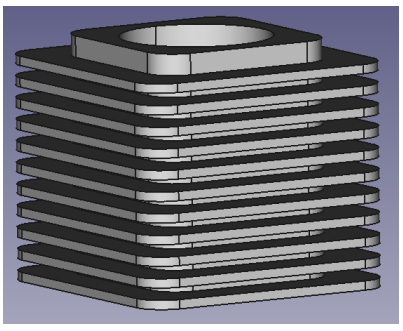


Figure 4.7: Finned block cylinder.

In this example we study a finned cylinder engine shown in Figure 4.7. The cylinder is a

component in automobiles subject to high temperatures due to the combustion in its interior. For this reason, fins are usually placed in the exterior; to increase the contact area with the surrounding air and hence improving the heat transfer [2, 3].

The geometry of study Figure 4.7, which is based on the geometry presented in the reference [3]. The necessary data for the properties of the cylinder were also extracted from this source. Gray Cast iron was selected as the material of choice, exhibiting a density of $\rho = 7200$ kg/m^3, a specific heat capacity of $C_p = 447$ J/kgK, and a thermal conductivity of $\kappa = 52$ W/mK. In this case the material is considered to be isotropic and homogeneous, therefore all values are assumed to be uniform throughout the whole volume.

For the boundary conditions we take the inner wall to have a fixed temperature of $T = 300°$C. In the exterior surfaces we assume a heat transfer between stagnant air and the surface of the material with a convection coefficient of $h = 5$W/m^{2}K.

To begin, let us compare the temperatures of the cylinder with and without fins in the stationary regime, that is, $\frac{\partial T}{\partial t} = 0$. In both cases, we utilize $\mathbb{P}_1$ elements. The mesh for the former configuration consists of $9,229$ nodes and a total of $28,939$ tetrahedra, while the latter configuration employs $2,144$ nodes and $7,361$ tetrahedra. The difference in element count arises from the inclusion of fins, which, due to their small thickness, requires an increased number of tetrahedra in that particular region. Nevertheless, the number of tetrahedra in the cylinder portion of the mesh remains comparable, enabling a meaningful comparison between the results obtained.

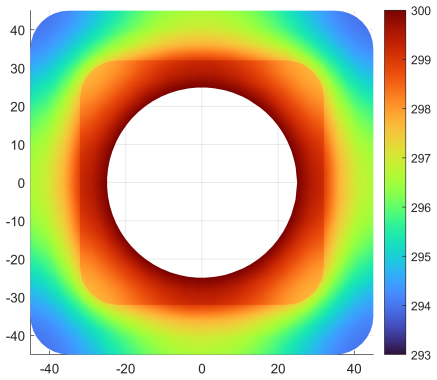
In Figure 4.8, we present the temperature fields obtained for both cases. Notably, in the case with fins, the results obtained exhibit similarity to those reported in [3], validating the correctness of our results. It is apparent that the temperature of the finned cylinder is slightly lower than that of the cylinder without fins. This effect can be attributed to the presence of fins, which increase the contact area between the air and the material from $A = 23,177.83$m^2 to $A = 114,722.71$m^2. Consequently, there is a small enhancement in heat rate transfer.

Based on these results, it becomes evident that the utilization of fins plays a crucial role. By reducing the working temperature of the material, not only are the mechanical properties enhanced, but the overall service life of the component is also increased.

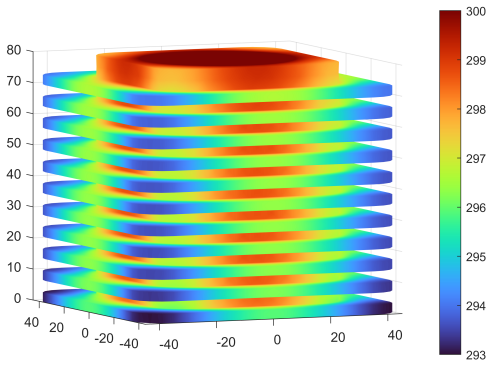
Next, we shift our focus to the transient study of the finned cylinder, which initially has a temperature of $T_0 = 20°$C. The boundary condition remain the same as before. To conduct this experiment, we employ the second-order Crank-Nicolson scheme with a fixed time step of $\tau = 0.05$s. This scheme allows us to accurately analyze the temporal behavior of the temperature field and observe its convergence towards the stationary state.

In Figure 4.9, the temperature fields are displayed at different instant of times. They clearly show that the overall temperature of the cylinder gradually increases over time. This phenomenon is due to the limited capacity of the surrounding air to dissipate all the heat resulting from the temperature gradient between the inner surface and the rest of the cylinder. However, as the temperatures in the remaining regions of the cylinder approach the temperature of the inner surface, the thermal gradient diminishes in magnitude. Eventually, a thermal equilibrium is achieved, where the heat transfer with the air reaches a balance with the conduction heat transfer, leading to a nearly constant temperature.

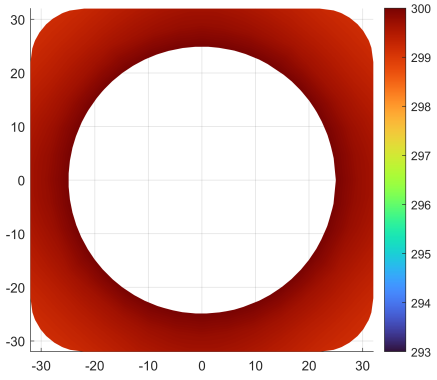
This observation is further supported by Figure 4.10, which represents the maximum temperature variation ΔT in the cylinder at each instant of time. The plot clearly illustrates that around $t \approx 40$s, the temperature begins to stabilize to the stationary case.



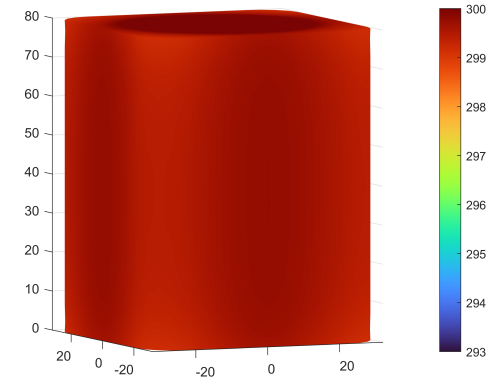
(a) Top view.



(b) Perspective.



(c) Top view.



(d) Perspective.

Figure 4.8: Temperature field in a cylinder with fins (above) and without fins (below).

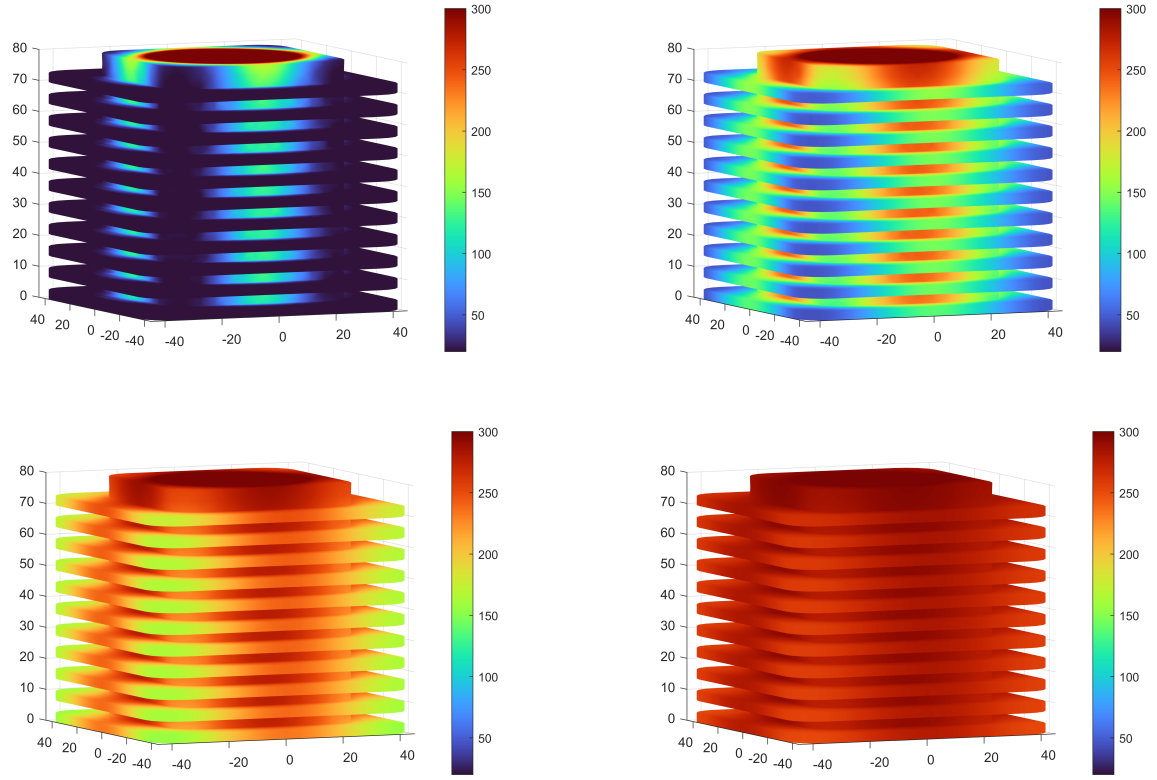


Figure 4.9: Temperature field of the finned cylinder block at $t = 1$s (top-left), $t = 8s$ (top-right), $t = 20$s (bottom-left) and $t = 40$s (bottom-right).

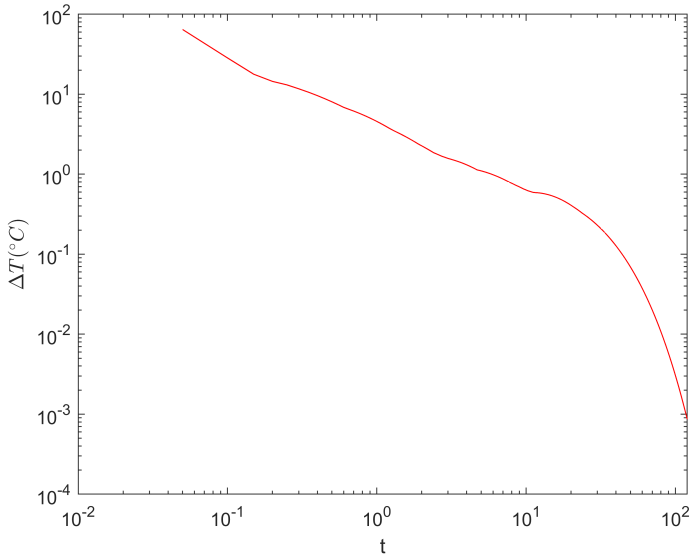


Figure 4.10: Maximum temperature variation.

Chapter 5

Conclusions and future Work

In this project, we have successfully developed a robust and efficient package for solving complex Finite Element problems over arbitrary domains. Currently, our package is capable of effectively handling tetrahedral meshes up to degree four, which surpasses the limitations of many commercial codes that usually support only up to degree two.

Recognizing the importance of optimizing code efficiency, we have tried to achieve high-performance in our current implementation. However, we acknowledge that there is always room for improvement. Future modifications can focus on optimizing memory consumption and computation time, enabling users to tackle even larger and more challenging problems efficiently.

To further enhance the package, we can explore the inclusion of additional element types, such as hexahedral elements or edge elements. By incorporating these ideas, we can extend the range of problems that can be effectively solved and enhance the accuracy of simulations in complex scenarios.

Additionally, the integration of postprocessing capabilities would significantly enhance the practical utility of our package. While our project did not extensively explore these features, the incorporation of a-posteriori error estimation methods and other relevant postprocessing techniques can offer valuable insights and validate the accuracy of the obtained results. It is worth noting that certain fields within our project were intentionally designed to generate relevant data that can be utilized for performing these computations.

In conclusion, this work presents a package to solve Finite Element problems. Throughout this project, I have not only learned how to model and solve problems but also gained a deeper understanding of the computations involved in implementing these methods. This knowledge has provided me with a solid foundation for comprehending the internal routines of commercial solvers used in various engineering applications.

By continually refining and expanding this package capabilities, we aim to provide engineers and scientists with a valuable tool for addressing a diverse range of challenges without the need to spend money to do so.

Bibliography

[1] M. Emam A. M. Radwan and M. Ahmed. "Comparative Study of Active and Passive Cooling Techniques for Concentrated Photovoltaic Systems". In: Oct. 2017, pp. 475–505.

[2] C.R. Sonawane et al. "Numerical simulation to evaluate the thermal performance of engine cylinder Fins: Effect of fin geometry and fin material". In: *Materials Today: Proceedings* 49 (2022). GC-RASM 2021, pp. 1590–1598.

[3] S. Durgam et al. "Thermal analysis of fin materials for engine cylinder heat transfer enhancement". In: 1126.1 (Mar. 2021), p. 012071.

[4] J. Alberty et al. "Matlab Implementation of the Finite Element Method in Elasticity". In: *Computing* 69 (2002), pp. 239–263.

[5] M. S. Alnaes et al. "The FEniCS Project Version 1.5". In: *Archive of Numerical Software* 3 (2015).

[6] A. Ern and J.L. Guermond. *Theory and Practice of Finite Elements*. Applied Mathematical Sciences.

[7] FreeCAD Community. *FreeCAD*. https://www.freecadweb.org/. 2023.

[8] C. Geuzaine and J.F. Remacle. *Gmsh*. Version 4.6.0. June 22, 2020. URL: http://gmsh.info/.

[9] D. Gilbarg and N.S. Trudinger. *Elliptic Partial Differential Equations of Second Order*. Classics in Mathematics. Springer Berlin, Heidelberg, 2001.

[10] P.L. Gould. "Introduction to Linear Elasticity". In: Springer, 2018.

[11] M.A. Herrero. "Reaction-diffusion systems: a mathematical biology approach". In: 2003.

[12] J. Jaśkowiec and N. Sukumar. "High-order cubature rules for tetrahedra". In: *International Journal for Numerical Methods in Engineering* 121.11 (2020), pp. 2418–2436.

[13] S. Koukal and M. Křížek. "Curved affine quadratic finite elements". In: *Journal of Computational and Applied Mathematics* 63.1 (1995). Proceedings of the International Symposium on Mathematical Modelling and Computational Methods Modelling 94, pp. 333–339.

[14] P. Lahuerta. "FEM Modelization Using Gmsh and GetDP: Examples and Guidelines". Bachelor's Thesis. Universitat Rovira I Virgili, 2015.

[15] G. Larson and F. Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Springer Publishing Company, Incorporated, 2013.

[16] M.A Londoño and M. Hebert. "Optimal shape parameter for meshless solution of the 2D Helmholtz equation". In: *CT&F Ciencia, Tecnología y Futuro* (2019).

[17] J.H. Tang M.J. Grote F. Nataf and P.H. Tournier. "Parallel controllability methods for the Helmholtz equation". In: *Computer Methods in Applied Mechanics and Engineering* 362 (2020), p. 112846.

[18] H. Ojanen. *Mathematica Expression to Matlab m-file Converter*. URL: https://library.wolfram.com/infocenter/MathSource/577/.

[19] OpenAI. *ChatGPT: [Large language model]*. https://openai.com. 2023.

[20] Paharuddin et al. “Numerical solutions to Helmholtz equation of anisotropic functionally graded materials”. In: *Journal of Physics: Conference Series* 1341.8 (2019), p. 082012.

[21] F.J. Sayas. *A gentle introduction to the Finite Element Method*. 2015.

[22] N. Schlömer. *quadpy*. https://github.com/sigma-py/quadpy.

[23] Y. Sun, A.S. Jayaraman, and G.S. Chirikjian. “Approximate solutions of the advection–diffusion equation for spatially variable flows”. In: *Physics of Fluids* 34.3 (2022).

[24] V. Thomee. *Galerkin Finite Element Methods for Parabolic Problems*. Springer Series in Computational Mathematics. Springer Berlin Heidelberg, 2013.

[25] A.N. Tikhonov and A.A. Samarski. *Equations of Mathematical Physics*. Dover Books on Physics. Dover Publications.

[26] L.N. Trefethen. *Approximation Theory and Approximation Practice, Extended Edition*. Society for Industrial and Applied Mathematics, 2019.

[27] Y.A. Çengel and A.J. Ghajar. *Heat and Mass Transfer: Fundamentals & Applications*. Asia Higher Education Engineering/Computer Science Mechanica. McGraw Hill Education.

Appendix A

Lagrange Basis

For computing the basis elements, it was necessary to create a tetrahedral reference element in GMSH. It should be obvious that the global numbering of the element coincide with the local numbering. However, it was not possible to create a mesh such that this was the case. Instead, while the node placement satisfied the rules presented in 2.2.4, the node 2 did not lie in the x-axis. This was taken into account when computing the functions $\widehat{N}_i^{\widehat{K}}$ in barycentric coordinates. The procedure followed is shown below.

```
degree = 4;

switch degree

    case 1
        vertices = ...
           [0 0 0;
            0 1 0;
            0 0 1;
            1 0 0];

        disp('P1')
        disp('==')
        disp('')
        % P1

        barP1 = ...
           [1 0 0 0;...
            0 1 0 0;...
            0 0 1 0;...
            0 0 0 1];
        barP = barP1;

        disp('The tempative order')
        PsOurs = barP1*vertices;
        disp(PsOurs)


        % GMSH
        t = [1 2 3 4];
        PsGMSH= ...
           [1 0 0 0
            2 0 1 0
            3 0 0 1
            4 1 0 0];
```

```
39
40
41          PsGMSH= PsGMSH(t,2:4);
42          disp('GMSH')
43          disp(PsGMSH)
44
45          disp('Check')
46          disp(PsGMSH-PsOurs)
47
48      case 2
49          disp('P2')
50          disp('==')
51          % P2
52
53          barP2 = ...
54             [1         0          0          0;...
55              0         1          0          0;...
56              0         0          1          0;...
57              0         0          0          1;...
58              1/2       1/2        0          0;...
59              0         1/2        1/2        0;...
60              1/2       0          1/2        0;...
61              1/2       0          0         1/2;...
62              0         0         1/2         1/2;...
63              0         1/2        0           1/2];
64          barP = barP2;
65          disp('The tempative order')
66          PsOurs= barP2*vertices;
67          disp(PsOurs)
68
69          % GMSH
70
71          t = [1 2 3 4 5 6 7 8 10 9];
72          PsGMSH = [...
73              1 0 0 0      ;
74              2 0 1 0      ;
75              3 0 0 1      ;
76              4 1 0 0      ;
77              5 0 0.5 0    ;
78              6 0 0.5 0.5  ;
79              7 0 0 0.5    ;
80              8 0.5 0 0    ;
81              9 0.5 0.5 0  ;
82              10 0.5 0 0.5 ];
83
84          PsGMSH = PsGMSH(t,2:4);
85
86          disp('GMSH')
87          disp(PsGMSH)
88
89          disp('Check')
90          disp(PsGMSH-PsOurs)
91
92      case 3
93          % P3
94
95          barP3 = ...
96             [1    1          0          0          0;...
97              2    0          1          0          0;...
98              3    0          0          1          0;...
```

```
            4    0          0           0          1;...
            %  Edges
            %  y = 0, z = 0;
            5    2/3        1/3         0          0;...
            6    1/3        2/3         0          0;...
            % z = 0, x+y =1;
            7    0          2/3         1/3        0;...
            8    0          1/3         2/3        0;...
            % x = 0,z  =0;
            9    1/3        0           2/3        0;...
            10   2/3        0           1/3        0;...
            %    x = 0, y  =0;
            11   1/3        0           0          2/3;...
            12   2/3        0           0          1/3;...
            %    x = 0, y+z  =0;
            13   0          0           1/3        2/3;...
            14   0          0           2/3        1/3;...
            %    y = 0, x+z  =1;
            15   0          1/3         0          2/3;...
            16   0          2/3         0          1/3;...
            %
            % internal faces nodes
            % z=9
            17   1/3         1/3     1/3            0;...
            % y=0
            18   1/3        1/3        0          1/3;...
            % x=0
            19   1/3          0       1/3          1/3;...
            % x+y+z=1
            20   0          1/3       1/3          1/3;...
            ];
        barP3  =barP3(:,end-3:end);
        barP = barP3;

        disp('The tempative order')
        PsOurs= barP3*vertices;
        disp(PsOurs)


        PsGMSH =  [...
            1               0               0               0        ;...
            2               0               1               0        ;...
            3               0               0               1        ;...
            4               1               0               0        ;...
            5               0               1/3             0        ;...
            6               0               2/3             0        ;...
            7               0               2/3             1/3      ;...
            8               0               1/3             2/3      ;...
            9               0               0               2/3      ;...
            10               0               0               1/3      ;...
            11               1/3             0               0        ;...
            12               2/3             0               0        ;...
            13               1/3             2/3             0        ;...
            14               2/3             1/3             0        ;...
            15               1/3             0               2/3      ;...
            16               2/3             0               1/3      ;...
            17               0               1/3             1/3      ;...
            18               1/3             1/3             0        ;...
            19               1/3             1/3             1/3      ;...
            20               1/3             0               1/3   ];
```

```
159
160         t = [1 2 3 4 5 6 7 8 9 10 12 11 16 15 14 13 17 18 20 19];
161         PsGMSH = PsGMSH(t,2:4);
162
163
164         disp('GMSH')
165         disp(PsGMSH)
166
167         disp('Check')
168         disp(PsGMSH-PsOurs)
169
170     case 4
171         % P4
172
173
174         barP4 = ...
175            [1    1          0          0          0;...
176             2    0          1          0          0;...
177             3    0          0          1          0;...
178             4    0          0          0          1;...
179             %  edges
180             %  y = z = 0
181             5    3/4        1/4        0          0;...
182             6    2/4        2/4        0          0;...
183             7    1/4        3/4        0          0;...
184             %    z = 0, x+y=1
185             8    0          3/4        1/4        0;...
186             9    0          2/4        2/4        0;...
187             10   0          1/4        3/4        0;...
188             %         x = z =0
189              11    1/4          0        3/4         0;...
190              12    2/4          0        2/4        0;...
191              13    3/4          0        1/4        0;...
192             %         x = 0, y  =0;
193             14   1/4        0          0          3/4;...
194             15   2/4        0          0          2/4;...
195             16   3/4        0          0          1/4;...
196             %         x= 0, y+z=0
197             17   0          0        1/4          3/4;...
198             18   0          0        2/4          2/4;...
199             19   0          0        3/4          1/4;...
200             %          y=0, x+z=1
201             20   0        1/4          0       3/4;...
202             21   0        2/4          0        2/4;...
203             22   0        3/4          0        1 /4;...
204             % internal faces z=0
205             23   2/4        1/4        1/4          0;...
206             24   1/4        1/4        2/4          0;
207             25   1/4        2/4        1/4          0;  ...
208             % internal faces y =0
209             26   2/4        1/4          0      1/4      ;...
210             27   1/4        2/4          0      1/4      ;...
211             28   1/4        1/4          0      2/4      ;...
212             % internal  faces x= 0
213             29              2/4 0      1/4          1/4          ;...
214             30              1/4 0       1/4        2/4      ;...
215             31              1/4 0       2/4        1/4      ;...
216             % internal faces x+y+z=1 OK
217             32   0        1/4          1/4        2/4      ;...
218             33   0        2/4          1/4        1/4      ;...
```

```
        34  0       1/4          2/4       1/4     ;...
        % internal  nodes
        35  1/4       1/4      1/4         1/4;...
        ];

barP4 = barP4(:,end-3:end);
barP = barP4;


disp('The tempative order')
PsOurs= barP4*vertices;
disp(PsOurs)


PsGMSH = [ ...
        1 0 0 0            ;
        2 0 1 0            ;
        3 0 0 1            ;
        4 1 0 0            ;
        5 0 0.25 0         ;
        6 0 0.5 0          ;
        7 0 0.75 0         ;
        8 0 0.75 0.25      ;
        9 0 0.5 0.5        ;
        10 0 0.25 0.75     ;
        11 0 0 0.75        ;
        12 0 0 0.5         ;
        13 0 0 0.25        ;
        14 0.25 0 0        ;
        15 0.5 0 0         ;
        16 0.75 0 0        ;
        17 0.25 0.75 0     ;
        18 0.5 0.5 0       ;
        19 0.75 0.25 0     ;
        20 0.25 0 0.75     ;
        21 0.5 0 0.5       ;
        22 0.75 0 0.25     ;
        23 0 0.25 0.25     ;
        24 0 0.5 0.25      ;
        25 0 0.25 0.5      ;
        26 0.25 0.25 0     ;
        27 0.25 0.5 0      ;
        28 0.5 0.25 0      ;
        29 0.25 0.5 0.25   ;
        30 0.25 0.25 0.5   ;
        31 0.5 0.25 0.25   ;
        32 0.25 0 0.25     ;
        33 0.5 0 0.25      ;
        34 0.25 0 0.5      ;
        35 0.25 0.25 0.25 ];

t = [1 2 3 4 5 6 7 8 9 10 11 12 13 16 15 14 22 21 20 19 18 17 ...
     23 25 24 26 27 28 32 33 34 31 29 30 35];
PsGMSH = PsGMSH(t,2:4);



disp('GMSH')
disp(PsGMSH)

```

```
        disp('Check')
        disp(PsGMSH-PsOurs)
end


% We try to compute the basis now for P4. First, barycentric functions



h =1/degree;
disp('Nodes in barycentric coordinates')
barP
disp('Basis')

lambda{1} = @(l1,l2,l3,l4) l1;
lambda{2} = @(l1,l2,l3,l4) l2;
lambda{3} = @(l1,l2,l3,l4) l3;
lambda{4} = @(l1,l2,l3,l4) l4;

syms l1 l2 l3 l4
clear N
% Choose the degree (up to four)

for i=1:length(barP)
    p =sym('1');
    for j = 1:4
        for r = 1:degree
            if barP(i,j)>=r/degree
                coef = lambda{j}(barP(i,1),barP(i,2),barP(i,3),barP(i,4))-(r
   -1)*h;
                p = p*(lambda{j}(l1,l2,l3,l4)-(r-1)*h)/coef ...
                    ;
            end
        end
    end
    N{i} = matlabFunction(p,'vars',{l1,l2,l3,l4});
end
Ns =  [];
for j =1:length(barP)
    Ns = [Ns;...
          %j
          N{j}(l1,l2,l3,l4)];
end
disp('Basis functions')
Ns
% Check
checkingBasis = zeros(length(barP));
for j = 1:length(checkingBasis)
    checkingBasis(:,j) = N{j}(barP(:,1),barP(:,2),barP(:,3),barP(:,4));
end
disp('Next matrix MUST be the identity')
checkingBasis

barP
Ns

% clf;
hold on
for j=1:length(barP)
  plot3(barP(j,2),barP(j,3),barP(j,4),'ro')
```

```
    xlabel('x')
    ylabel('y')
    zlabel('z')
  text(barP(j,2)+0.03,barP(j,3)+0.03,barP(j,4)+0.03,num2str(j))
end



% bases
```

The partial derivatives of $\widehat{N_i}$ can be computed using the chain rule, this results in

$$\frac{\partial \widehat{N}_i}{\partial \widehat{x_k}} = \sum_{i=1}^{4} \frac{\partial \widehat{N}_i}{\partial \lambda_i} \frac{\partial \lambda_i}{\partial \widehat{x}_k},$$

for $\widehat{x}_k \in \{\widehat{x}, \widehat{y}, \widehat{z}\}$. To handle this operation, it has to be noticed that $\frac{\partial \lambda_i}{\partial \widehat{x}_k} \in \{-1, 0, 1\}$, that is, the derivatives are constant functions. Thus, $\widehat{N_i}$ can be derived, and each term can be multiplied by the corresponding constant factor. Keep in mind the resulting expressions for the gradient must be given in barycentric coordinates. This was accomplished using the following code.

```
syms x y z l1(x,y,z) l2(x,y,z) l3(x,y,z) l4(x,y,z)


[~, Ns3D1] = FEM3Dclass.basisNj(1);
[~, Ns3D2] = FEM3Dclass.basisNj(2);
[~, Ns3D3] = FEM3Dclass.basisNj(3);
[~, Ns3D4] = FEM3Dclass.basisNj(4);

Ns3Dk = {Ns3D1, Ns3D2, Ns3D3, Ns3D4};
gradNs3Dk = cell(4,1);


for j =1:4
    Ns3D = Ns3Dk{j};
    gradNs3D = cell(3, length(Ns3D));
    for i=1:length(Ns3D)
       Ni = Ns3D{i};

       Nix = string(diff(Ni(l1(x,y,z),l2(x,y,z),l3(x,y,z),l4(x,y,z)),x));
       Nix = strrep(Nix, 'diff(l1(x, y, z), x)','-1');
       Nix = strrep(Nix, 'diff(l2(x, y, z), x)','1');
       Nix = strrep(Nix, 'diff(l3(x, y, z), x)','0');
       Nix = strrep(Nix, 'diff(l4(x, y, z), x)','0');
       Nix = strrep(Nix, 'l1(x, y, z)','l1');
       Nix = strrep(Nix, 'l2(x, y, z)','l2');
       Nix = strrep(Nix, 'l3(x, y, z)','l3');
       Nix = strrep(Nix, 'l4(x, y, z)','l4');

       Niy = string(diff(Ni(l1(x,y,z),l2(x,y,z),l3(x,y,z),l4(x,y,z)),y));
       Niy = strrep(Niy, 'diff(l1(x, y, z), y)','-1');
       Niy = strrep(Niy, 'diff(l2(x, y, z), y)','0');
       Niy = strrep(Niy, 'diff(l3(x, y, z), y)','1');
       Niy = strrep(Niy, 'diff(l4(x, y, z), y)','0');
       Niy = strrep(Niy, 'l1(x, y, z)','l1');
       Niy = strrep(Niy, 'l2(x, y, z)','l2');
       Niy = strrep(Niy, 'l3(x, y, z)','l3');
       Niy = strrep(Niy, 'l4(x, y, z)','l4');
```

```
38
39          Niz = string(diff(Ni(l1(x,y,z),l2(x,y,z),l3(x,y,z),l4(x,y,z)),z));
40          Niz = strrep(Niz, 'diff(l1(x, y, z), z)','-1');
41          Niz = strrep(Niz, 'diff(l2(x, y, z), z)','0');
42          Niz = strrep(Niz, 'diff(l3(x, y, z), z)','0');
43          Niz = strrep(Niz, 'diff(l4(x, y, z), z)','1');
44          Niz = strrep(Niz, 'l1(x, y, z)','l1');
45          Niz = strrep(Niz, 'l2(x, y, z)','l2');
46          Niz = strrep(Niz, 'l3(x, y, z)','l3');
47          Niz = strrep(Niz, 'l4(x, y, z)','l4');
48
49
50          Nix = str2sym(Nix); Nix = string(Nix);
51          Niy = str2sym(Niy); Niy = string(Niy);
52          Niz = str2sym(Niz); Niz = string(Niz);
53
54
55          Nix = append('@(l1,l2,l3,l4)', Nix);
56          Niy = append('@(l1,l2,l3,l4)', Niy);
57          Niz = append('@(l1,l2,l3,l4)', Niz);
58
59          if j==1
60              Nix = append(Nix, '+0*l1');
61              Niy = append(Niy, '+0*l1');
62              Niz = append(Niz, '+0*l1');
63          end
64          if Nix == '@(l1,l2,l3,l4)0'
65              Nix = append(Nix, '+0*l1');
66          end
67          if Niy == '@(l1,l2,l3,l4)0'
68              Niy = append(Niy, '+0*l1');
69          end
70          if Niz == '@(l1,l2,l3,l4)0'
71              Niz = append(Niz, '+0*l1');
72          end
73
74          Nix = strrep(Nix,'*','.*');
75          Niy = strrep(Niy,'*','.*');
76          Niz = strrep(Niz,'*','.*');
77
78
79          gradNs3D{1,i} = str2func(Nix);
80          gradNs3D{2,i} = str2func(Niy);
81          gradNs3D{3,i} = str2func(Niz);
82      end
83      gradNs3Dk{j} = gradNs3D;
84  end
```

It is worth noting that the procedure commented above is not implemented in Matlab, as usually writing the dependence of one set of variables on another means that the resultant expression is in terms of the latter set. Therefore, the barycentric coordinates were written implicitly in terms of $(\widehat{x}, \widehat{y}, , \widehat{z})$, and the terms $\frac{\partial \lambda_i}{\partial \widehat{x_k}}$ were handled using string manipulations.

Appendix B

Example codes

B.1 Connecting rod

```
%% Displacement analysis of a connecting rod.

% Mesh file
T = FEM3Dclass('ConnectingRod_order2.msh');
% T = FEM3Dclass('ConnectingRod.msh'); % Comment lines 136-139. Uncomment
    140

%% Geometry Visualization
Loaded = T.ComplementaryInformation('Loaded Surface');
Loaded = T.mesh.domBd==Loaded;
LoadedSurface = T.mesh.trB(Loaded, 1:3);

Fixed = T.ComplementaryInformation('Fixed Support');
Fixed = T.mesh.domBd==Fixed;
FixedSurface = T.mesh.trB(Fixed, 1:3);

Free = T.ComplementaryInformation('Free Surface');
Free = ismember(T.mesh.domBd,Free);
FreeSurface = T.mesh.trB(Free, 1:3);

%
figure()
hold on
trimesh(LoadedSurface, T.mesh.coord(:,1),T.mesh.coord(:,2), ...
    T.mesh.coord(:,3),'FaceColor', 'red', 'EdgeColor','black')
trimesh(FixedSurface, T.mesh.coord(:,1),T.mesh.coord(:,2), ...
    T.mesh.coord(:,3),'FaceColor', 'blue', 'EdgeColor','black')
trimesh(FreeSurface, T.mesh.coord(:,1),T.mesh.coord(:,2), ...
        T.mesh.coord(:,3),'FaceColor', '#808080', 'EdgeColor','black')

axis('equal')
view(32.6577,18.1538)
%% Data

%%% Material Properties
% Young Modulus
E = 2e11; %Pa
% Poisson Coefficient
nu = 0.3;
%%% Lame Parameters
lambda = E*nu/(1+nu)/(1-2*nu);
mu = E/(1+nu)/2;
%% Load
q = 20;
```

```
q = q*1e6; %MPa



% Director cosines
LoadedUnitNormal = T.mesh.trBNormal(Loaded,:);
LoadedUnitNormal = LoadedUnitNormal./vecnorm(LoadedUnitNormal,2,2);
cosThetaX = LoadedUnitNormal(:,1);
cosThetaY = LoadedUnitNormal(:,2);
cosThetaZ = LoadedUnitNormal(:,3);

qX =@(x,y,z) -q.*cosThetaX + x.*0;
qY =@(x,y,z) -q.*cosThetaY + x.*0;
qZ =@(x,y,z) -q.*cosThetaZ + x.*0;

%% Boundary Condition nodes
% Fixed Nodes
fixedNodes    = T.mesh.trB(Fixed,:); fixedNodes = unique(fixedNodes);
% Load
loadedNodes = T.mesh.trB(Loaded,:); loadedNodes = unique(loadedNodes);


nNodes = T.mesh.nNodes;
iD = fixedNodes; iD2 = [iD; iD+nNodes; iD+nNodes*2];
inD = (1:length(T.mesh.coord)*3)'; inD = setdiff(inD, iD2);

%% System Construction

%%% Assembly

% Stiffness Matrix
A = {@(x,y,z) mu+2*lambda+x.*0, @(x,y,z) 0+x.*0,  @(x,y,z) 0+x.*0;
     @(x,y,z) 0+x.*0,           @(x,y,z) mu+x.*0, @(x,y,z) 0+x.*0;
     @(x,y,z) 0+x.*0,           @(x,y,z) 0+x.*0,  @(x,y,z) mu+x.*0};
S11 = femStiffnessMatrix(T,A);

A = {@(x,y,z) mu+x.*0, @(x,y,z) 0+x.*0,           @(x,y,z) 0+x.*0;
     @(x,y,z) 0+x.*0,  @(x,y,z) lambda+2*mu+x.*0, @(x,y,z) 0+x.*0;
     @(x,y,z) 0+x.*0,  @(x,y,z) 0+x.*0,           @(x,y,z) mu+x.*0};
S22 = femStiffnessMatrix(T,A);

A = {@(x,y,z) mu+x.*0, @(x,y,z) 0+x.*0,  @(x,y,z) 0+x.*0;
     @(x,y,z) 0+x.*0,  @(x,y,z) mu+x.*0, @(x,y,z) 0+x.*0;
     @(x,y,z) 0+x.*0,  @(x,y,z) 0+x.*0,  @(x,y,z) lambda+2*mu+x.*0};
S33 = femStiffnessMatrix(T,A);

A = {@(x,y,z) 0+x.*0,  @(x,y,z) lambda+x.*0, @(x,y,z) 0+x.*0;
     @(x,y,z) mu+x.*0, @(x,y,z) 0+x.*0,      @(x,y,z) 0+x.*0;
     @(x,y,z) 0+x.*0,  @(x,y,z) 0+x.*0,      @(x,y,z) 0+x.*0};
S12 = femStiffnessMatrix(T,A);

A = {@(x,y,z) 0+x.*0,  @(x,y,z) 0+x.*0, @(x,y,z) lambda+x.*0;
     @(x,y,z) 0+x.*0,  @(x,y,z) 0+x.*0, @(x,y,z) 0+x.*0;
     @(x,y,z) mu+x.*0, @(x,y,z) 0+x.*0, @(x,y,z) 0+x.*0};
S13 = femStiffnessMatrix(T,A);

A = {@(x,y,z) 0+x.*0, @(x,y,z) 0+x.*0,  @(x,y,z) 0+x.*0;
     @(x,y,z) 0+x.*0, @(x,y,z) 0+x.*0,  @(x,y,z) lambda+x.*0;
     @(x,y,z) 0+x.*0, @(x,y,z) mu+x.*0, @(x,y,z) 0+x.*0};
S23 = femStiffnessMatrix(T,A);
```

```
104
105 matrix = [S11  S12  S13
106           S12' S22  S23
107           S13' S23' S33];
108
109 % Neglect body forces
110 F = zeros(nNodes*3,1);
111
112 % Load
113 tX = femRobin(T, Loaded, qX);
114 tY = femRobin(T, Loaded, qY);
115 tZ = femRobin(T, Loaded, qZ);
116 t = [tX; tY; tZ];
117
118
119 % Solution initialization
120 u = zeros(nNodes*3,1);
121
122 % System Resolution
123 F = F+t; F = F(inD);
124 F = F-matrix(inD, iD2)*u(iD2);
125 matrix = matrix(inD, inD);
126 u(inD) = matrix\F;
127
128 %%
129 uX = u(1:nNodes);
130 uY = u(nNodes+1:2*nNodes);
131 uZ = u(2*nNodes+1:3*nNodes);
132 %% PLOT SOLUTION
133 SurfaceSubTriangles = [1 4 6; 2 5 4; 3 6 5; 6 4 5];
134
135 trB = [T.mesh.trB(:,SurfaceSubTriangles(1,:));
136                 T.mesh.trB(:,SurfaceSubTriangles(2,:));
137                 T.mesh.trB(:,SurfaceSubTriangles(4,:))];
138                 T.mesh.trB(:,SurfaceSubTriangles(3,:));
139 % trB = T.mesh.trB;
140
141 figure;
142 trisurf(trB, T.mesh.coord(:,1),T.mesh.coord(:,2),T.mesh.coord(:,3),...
143             vecnorm([uX uY uZ],2,2), 'FaceColor','interp','EdgeColor','none'
    );
144 colorbar
145 colormap turbo
146 xlabel('x')
147 ylabel('y')
148 zlabel('z')
149 axis('equal')
150 view(45,30)
151
152 fig = figure;
153 trisurf(trB, T.mesh.coord(:,1),T.mesh.coord(:,2),T.mesh.coord(:,3),...
154             uX, 'FaceColor','interp','EdgeColor','none');
155 colorbar
156 colormap turbo
157 xlabel('x')
158 ylabel('y')
159 zlabel('z')
160 axis('equal')
161 view(45,30)
162
```

```
figure;
trisurf(trB, T.mesh.coord(:,1),T.mesh.coord(:,2),T.mesh.coord(:,3),...
            uY, 'FaceColor','inter p','EdgeColor','none');
colorbar
colormap turbo
xlabel('x')
ylabel('y')
zlabel('z')
axis('equal')
view(45,30)

figure;
trisurf(trB, T.mesh.coord(:,1),T.mesh.coord(:,2),T.mesh.coord(:,3),...
            uZ, 'FaceColor','interp','EdgeColor','none');
colorbar
colormap turbo
xlabel('x')
ylabel('y')
zlabel('z')
axis('equal')
view(45,30)
```

B.2 Finned cylinder

```
T = FEM3Dclass('CylinderFins.msh');
%% Properties

% density
rho = 7200;
% Thermal Conductivity
k = 52;
% Heat capacity
Cp = 447;


% Interior surface Temperature
TInt = 300; TInt = TInt+273;
% Convection
h = 5;
TInf = 27; TInf = TInf+273;
gR = @(x,y,z) h*TInf+x*0;
alpha = @(x,y,z) h+x*0;

% Initial condition
T0 = 20; T0 = T0+273;
% time step
tau = 0.05;
t0 = 0;
tf = 120;


%% Show geometry

FixedTempSur = T.ComplementaryInformation('Interior');
FixedTempSur = T.mesh.trB(ismember(T.mesh.domBd, FixedTempSur),:);

ConvectSur =  T.ComplementaryInformation('Exterior');
robin = ismember(T.mesh.domBd, ConvectSur);
ConvectSur = T.mesh.trB(ismember(T.mesh.domBd, ConvectSur),:);
```

```
36
37
38 figure
39 hold on
40 axis 'equal'
41 trimesh(FixedTempSur, T.mesh.coord(:,1), T.mesh.coord(:,2), ...
42         T.mesh.coord(:,3),'EdgeColor','k')
43 trimesh(ConvectSur, T.mesh.coord(:,1), T.mesh.coord(:,2), ...
44         T.mesh.coord(:,3),'EdgeColor','k')
45
46 %% Geometry Scaling
47 Length = 1000; %mm->m
48 T.mesh.coord = T.mesh.coord/Length;
49 T.mesh.detBk  = T.mesh.detBk/Length^3;
50 T.mesh.trBNormal  = T.mesh.trBNormal/Length^2;
51
52 %%
53 iD = unique(FixedTempSur(:));
54 inD = (1:T.mesh.nNodes)'; inD = setdiff(inD,iD);
55 Uend = zeros(T.mesh.nNodes, 1);
56
57 % Initial conditions
58 Uend = Uend+T0;
59 % Dirichlet conditions
60 Uend(iD) = TInt;
61
62 disp('System assembly')
63 M = femMassMatrix(T, @(x,y,z) 1+x*0); M = rho*Cp*M;
64 S = femStressMatrix(T); S = k*S;
65 [t,MR] = femRobin(T, robin, gR, 'alpha',alpha); t = t(inD);
66 A  = M+0.5*tau*(S+MR); AnD =  A(inD,inD);
67 A2 = M-0.5*tau*(S+MR);
68 disp('done')
69
70 % Can be used to decrease time computation of the system
71 % [l,d,p] = ldl(AnD);
72 clear  M S
73
74 maxDiff = [];
75 checktimes = [1 8 20 40];
76 for tn1 = t0+tau:tau:tf
77     r   = A2(inD,:)*Uend+0.5*tau*(t+t)-A(inD,iD)*Uend(iD);
78     Uend2 = AnD\r;
79     maxDiff = [maxDiff max(abs(Uend(inD)-Uend2))];
80     Uend(inD) = Uend2;
81     if any(abs(tn1- checktimes)<1e-14)
82
83         fig = figure;
84         trisurf(T.mesh.trB, T.mesh.coord(:,1)*Length,...
85               T.mesh.coord(:,2)*Length,T.mesh.coord(:,3)*Length,...
86               Uend-273, 'FaceColor','interp','EdgeColor','none');
87         colormap turbo
88         caxis([20 300])
89         colorbar
90         axis equal
91         view(-23.5629,5.4)
92
93 %         saveas(fig,['heat_time' num2str(round(tn1,2)) '_top'],'epsc')
94
95         fig = figure;
```

```
96          trisurf(T.mesh.trB, T.mesh.coord(:,1)*Length,...
97                T.mesh.coord(:,2)*Length,T.mesh.coord(:,3)*Length,...
98                Uend-273, 'FaceColor','interp','EdgeColor','none');
99          colormap turbo
100         caxis([20 300])
101         colorbar
102         axis equal
103         view(0, 90)
104         ttt = true;
105 %         saveas(fig,['heat_time' num2str(round(tn1,2)) '_pers'],'epsc')
106     end
107 end
108
109 %%
110 loglog(t0+tau:tau:tf, maxDiff, 'r-', 'linewidth',0.6)
111 ylabel('$\Delta T(^\circ C)$', 'interpreter','Latex')
112 xlabel('t')
```

Index