UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
DEPARTAMENTO DE INGENIERÍA MECÁNICA

VIRTUAL ELEMENT METHOD FOR LINEAR ELASTICITY PROBLEMS IN MODIFIABLE MESHES

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERA CIVIL EN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERA CIVIL MECÁNICO

CATALINA PAZ ÁLVAREZ INOSTROZA

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

PROFESOR COGUÍA:
ALEJANDRO ORTIZ BERNARDIN

MIEMBROS DE LA COMISIÓN:
BENJAMIN BUSTOS CARDENAS
ELENA ATROSHCHENKO
LUIS SALINAS CARRASCO

Este trabajo ha sido parcialmente financiado por CONICYT

SANTIAGO DE CHILE
2017

Resumen

Los métodos numéricos son una valiosa herramienta en las ciencias y la ingeniería, ya que permiten obtener soluciones a muchos problemas difíciles. La elasticidad lineal, el estudio de como los objetos se deforman y cargan dado a cargas, es un problema en que los métodos numéricos tienden a usarse. El Método de Elemento Finito (FEM) es el método más usado para resolver problemas de elasticidad lineal; tiene muchas ventajas: es estable, es fácil aumentar el orden de los elementos para mejorar las aproximaciones, entre otros. Tiene, sin embargo, un buen número de desventajas: en general se debe usar con mallas de triángulos o cuadriláteros, menos flexibles que las de polígonos, y la precisión de la solución depende de la calidad de la malla. Estas dos desventajas hacen que FEM no sea la mejor opción para aplicaciones en que la calidad de la malla no está asegurada dado que cambia en el momento, como por ejemplo mecánica de fractura o análisis con mallas adaptivas. Nosotros teorizamos que una técnica novedosa, el Método de Elemento Virtual (VEM), es mejor para esta clase de aplicaciones; sin embargo, esta idea debe ser probada.

Considerando los problemas anteriores, este trabajo presenta un estudio del uso de VEM para aplicaciones en que las mallas cambian en tiempo real. Para testear la hipótesis presentada, se implementa: una librería para la generación eficiente de mallas poligonales, basadas en el diagrama de Voronoi restringido; una extensión a dicha librería, incluyendo operaciones para modificar las mallas; y una librería final, que implementa VEM y FEM para elasticidad lineal. Hacemos énfasis en que nuestra implementación de VEM es la primera de código abierto disponible.

Usando las herramientas implementadas, presentamos experimentos validando la convergencia numérica de los dos métodos; los resultados son satisfactorios, por lo que se procede con las pruebas para validar la hipótesis principal de esta tesis. Presentamos una comparación de los errores nodales para VEM y FEM cuando las mallas son sometidas a distintos cambios y concluimos que VEM se comporta mejor cuando las mallas cambian, incluso logrando tasas de error similares a las obtenidas cuando no se aplica ningún cambio. De esta forma, concluimos que VEM es una herramienta valida para la resolución de problemas de elasticidad lineal, en particular cuando las mallas presentan cambios imprevistos. Analizándo geométricamente las mismas pruebas, concluimos que las mallas de polígonos dan elementos de mejor calidad, para las operaciones probadas, en comparación con triangulaciones.

Finalmente, se presenta la complejidad teórica de los algoritmos, y se compara contra resultados experimentales; también se presentan ejemplos mostrando las funcionalidades logradas, concluyendo con los aspectos relacionados al trabajo futuro.

Abstract

Numerical methods are a valuable tool in many areas of science and engineering, as they allow obtaining of approximate solutions to many hard problems. Linear elasticity, the study of how object deform and are stress due to loading conditions, is one problem when numerical methods are commonly used. The Finite Element Method(FEM) is the most common numerical method used for linear elasticity; it has a great number of advantages, it is stable, it is easy to add higher order elements to improve the accuracy of the approximation, among others. It comes, however, with its own disadvantages: it is usually used with quadrilateral or triangular meshes, which are less flexible than polygonal ones, and the accuracy of the solution depends on the quality of the mesh. These two disadvantages make FEM not the best option for applications in which the mesh quality cannot be assured as it changes online, such as fracture simulation or adaptive mesh analysis. We theorize that a novel technique known as the Virtual Element Method (VEM) is better for this kind of applications; however, this thesis must be tested.

Considering the previous issues, this work presents a study of the Virtual Element Method for applications with online, or what we call "modifiable" meshes. To test the hypothesis presented, we implement a library for the efficient generation of polygonal meshes, based the constrained Voronoi diagram; an extension to this library, that includes operations to modify the mesh; and a final library, which implements the Virtual Element Method and Finite Element Method for linear elasticity. We do note that ours is the first open source implementation of the Virtual Element Method available.

Using the implemented tools, we present experiments validating the numerical convergence of the two methods; the results are satisfactory, so we proceed with the tests to validate this thesis' main hypothesis. We present a comparison of the nodal error value for both VEM and FEM when their respective meshes are modified in different ways, including refining and breaking, and conclude the VEM behaves better when the meshes change, even presenting error rates like the ones obtained when no changes are applied. This way, we conclude that the Virtual Element Method is a valid, and valuable, tool for the solution of linear elasticity problems, in particular, in meshes with non-staged changes. Analyzing geometrically the same tests, we conclude that polygonal meshes give better quality elements, for the tested modifiable operations, when compared to triangulations.

Finally, the theoretical complexity of the algorithms is presented, and compared against experimental results; we also present examples showing the functionalities achieved, concluding with aspects left as future work.

“To absent friends, lost loves, old gods, and the season of mists;
and may each and every one of us always give the devil his due”

Agradecimientos

A mi familia, que siempre se siente junto a mí a pesar de la distancia: a mi madre por todas las preguntas y a mi padre por no necesitar hacerlas. A mi hermana, que no solo soportó mis quejas, sino que me ayudó a aprender cómo terminar todo esto.

A todos aquellos que me enseñaron que tengo amigos, les agradezco estar allí y darme el ánimo de tomar el gran paso que termina aquí.

A la profesora Nancy Hitschfeld por enseñarme la valiosa lección de que a veces hay que detenerse, y al profesor Alejandro Ortiz por siempre recibirme cuando lo necesitaba.

A Michel, lo sé.

Contents

List of Figures

Glossary

N	Number of points in a mesh
P	Number of polygons in a mesh
n	Number of points of a polygon
FEM	Finite Element Method
PFEM	Polygonal Finite Element Method
VEM	Virtual Element Method
u	Displacement
ε	Strain
σ	Stress
E	Young module
ν	Poisson ratio
C	Material matrix, also called Stress/Strain matrix
$\boldsymbol{u}^h$	Approximated displacement, also $\boldsymbol{\varepsilon}^h$ is the approximated strain and $\boldsymbol{\sigma}^h$
K	Stiffness matrix, $\boldsymbol{K}_{\mathrm{e}}$ is the element-wise matrix, while $\boldsymbol{K}_g$ is the global one
b	Body force vector
f	Load vector, $\boldsymbol{f}_{\mathrm{e}}$ is the element-wise vector, $\boldsymbol{f}_g$ is the global vector

Chapter 1

Introduction

Linear elasticity, a simplified version of solid mechanics, is the study of how solid objects react and deform when loaded, but with several assumptions that make the problem more tractable. It is relatively simple, especially when compared with solid mechanics, but still intensively used in mechanical engineering, both because of its simplicity, and because the assumptions it takes are reasonable for many engineering scenarios. The problem, in simple words, consists in finding the displacements and internal forces of each point of a given domain, when certain load conditions are applied. However, analytical solutions exist only for a bounded number of combinations of loads and domain, hence the need of numerical approaches.

We refer to as numerical methods to all techniques that are used to find numerical approximate solutions to different problems, usually ordinary or partial differential equations. Numerical methods are used when the analytical solution to a given problem is too hard to obtain, extremely complex, or downright nonexistent. Instead of finding the continuous solution to the problem, represented by a function, numerical methods compute the approximate value the function takes in a certain number of points. This way, using that information, one can reconstruct a solution that, possibly not exact, is a good representation of the solution.

The points used in numerical methods are usually obtained from what is called a mesh: a collection of vertices, edges and faces that define the shape of a polygonal object in 2D, and a polyhedral one in 3D. Although the study of meshes, which is an important subject in computational geometry, is not a concern in numerical analysis per se, it is closely related, and sometimes studied.

The most common method used to solve linear elasticity problems is called finite element method (FEM); however, as applications get more varied, so does the method, giving birth to many variants. Classic FEM has been used successfully since its conception, but has several restrictions that make its use in all applications unsuitable; for example, FEM, without extensions, is both unable to work on arbitrary polygon shapes, being restricted mostly to triangles and quadrilaterals, and it requires a certain element quality to give acceptable results. As in the practice of engineering, differently from theoretical examples, it is usual to encounter situations in which it is necessary to obtain solutions in arbitrary domains, the concerns that could arise from the FEM restrictions are important.

When using FEM in domains of complex shape, one can either obtain extremely inaccurate results, due to bad quality, or note the computations take a long time to solve, due to the high number of required elements. Other approaches in the finite element family, such as the polygonal finite element method (PFEM) solve the problem of working with arbitrary polygons, which are more flexible and can adapt better to details of the domain with less elements; these techniques somehow help the previous issue, but still require certain element quality for the simulations to be accurate.

The exact same phenomenon appears when using this method to solve problems in which meshes are not completely static, meaning, those which change after being generated; examples of such problems are fracture mechanics, or computation using adaptive mesh refinement. Of course, one can always solve this problem generating the mesh again, including whatever changes were included; however, remeshing is an expensive procedure that greatly increases the cost of the already costly numerical simulation, besides the obvious fact that it likely increases the number of elements of the mesh.

Considering the two previous points, it can be readily noted that there is space for improvement in the resolution of linear elasticity problems in arbitrary domains, or in cases the meshes require changes after being generated.

The previous conclusion is the main motivation behind this thesis: we propose the use of a different technique to solve linear elasticity problems on complex non- static domains, called Virtual Element Method (VEM). Differently from FEM, VEM is theoretically suited for arbitrary polygon meshes with no quality requirements whatsoever, so our hypothesis is that VEM used in combination with arbitrary polygon meshes, it is better suited to solve problems in complex non-static domains.

1.1 Thesis description

As it has been mentioned before, this thesis' main idea is to use virtual element method in arbitrary non-static meshes to solve linear elasticity problems in complex domains. We want to test whether this approach can improve the accuracy or computation time in such conditions, in comparison with the classic FEM, which needs to use a great number of triangles to both mesh the same domain, and comply with the quality restrictions of the method. Our motivation to study this subject is that linear elasticity is a commonly used model throughout all mechanical engineering, and, as such, improvement in either accuracy or computation time can have an important impact in the industry, helping engineers to design safer and more efficient machinery and structures, among other applications.

In the next sections we will break down this thesis, including details of its motivation, objectives, and methodology, which will give the view necessary to understand the work presented later.

1.2 Motivation

As mentioned before, linear elasticity problems are an important subject in mechanical engineering. This model is used in applications as varied as the design of piping lines, in which it is necessary to determine pressure resistance, to analysis of possible failure in machine parts. Therefore, improvement in the accuracy or computation time of the numerical solution of linear elasticity opens new possibilities to the industry, and could possibly lead to future advances in the area.

Besides the improvements in the quantitative measures mentioned in the previous paragraph, we are also concerned about the possible domains in which the linear elasticity problem can be solved. We note that most approaches to the problem are mostly focused on accuracy, but sometimes ignore the fact that being able to obtain good quality solutions in almost arbitrary domains is also relevant to the actual practice of engineering. When we add to this the fact that the performance of classic methods is significantly worse when the meshes are not known beforehand, it is easy to understand why, even today, numerical solutions of the seemingly simple linear elasticity problem are still being studied.

Based on the ideas mentioned in the two previous paragraphs, we now define the concrete motivation behind this thesis: to present and validate a new methodology that improves the computation time and accuracy of the results of linear elasticity numerical solutions in non-static complex domains. We name as complex all arbitrary, but not self-intersecting, polygonal domains in which many triangles are required to obtain good numerical results using standard methods such as FEM; on the other hand, we use the term non- static, to refer to meshes which structure is changed after being generated, either adding, or deleting, vertices or edges.

Concretely, we present as main hypothesis: the use of virtual element method, in combination with arbitrary polygonal meshes, should be an improvement (in either accuracy or time) in the resolution of linear elasticity problems in arbitrary non-static domains, in comparison with finite element method in the same domains. Hence, we present a series of investigation questions that we desire to answer in this thesis:

1. Does VEM yield results at least as good as FEM for linear elasticity problems?
2. Does VEM yield good results when compared with FEM when the meshes are modified and not remeshed?
3. Does VEM require less elements when including additional segments on the mesh?
4. Is the time efficiency of VEM similar, or better, than FEM?

We present, alongside this main hypothesis, related ones, focused on the geometrical aspects of the simulations, which we will summarize in a second set of investigation questions:

1. Are polygonal meshes better suited, when compared with triangulations, to online changes?
2. Is it possible, from known tools, to obtain a robust and flexible polygonal mesh generator?

From the previous hypothesis, we can list the concrete goals of this thesis. First, we will implement a library to create and handle non-static polygonal meshes, study its performance and contribute to the scientific community with a new way of generating and manipulating polygon meshes. Second, as VEM is a relatively new technique, currently there are only barebones implementations of the method, so we also expect to contribute with a new complete implementation of a new technique. Hence, part of the motivation behind this thesis is to contribute to the development of software in both computational geometry and numerical analysis, leaving behind well designed and extensible open source code.

Finally, it was decided to use Voronoi diagrams as basis for the polygonal meshes; we justify this choice with three arguments. First, Voronoi diagrams have a dual structure, known as the Delaunay tessellation, which can both be used to compute the Voronoi diagram, and to obtain good quality triangulations, that are commonly found in finite element simulations; this way, we propose using Voronoi diagrams as it also provides us triangulations that can be used to other ends. Second, there are several efficient open source implementations of Delaunay triangulation generators, such as [37] and [11], thus it was relatively easy to integrate into the project and from it derive the Voronoi diagram. Third, Voronoi diagrams have been studied over the years to such a degree that its properties are well known; however, Voronoi diagrams are naturally unbounded and is not common to find applications that use one restricted to a region in space (constrained Voronoi diagram) while keeping most of their properties; this, in combination with the desire to innovate, led to its use in this work.

1.3 Objectives

In this section we present the objectives pursued by this thesis, divided in the main objective, the main guideline of the work, and a list of specific objectives, which structure the progress.

1.3.1 Main objective

As it has been implied in previous sections, this thesis main objective is to design and implement a new library for both a) the generation and manipulation of general non- static polygonal meshes, b) numerical resolution of linear elasticity problems using virtual element method.

1.3.2 Specific objectives

In this section we will list the specific objectives pursued, that will structure all the work done throughout this thesis. It is important to note that there is no order whatsoever and that several of these objectives have aspects that overlap in their concrete form.

- Design and implement a novel polygon mesh library, including generation and non-static functionalities, fulfilling the conditions declared before.

- Design and implement a novel elasticity numerical solver, using the virtual element method.
- Solve the linear elasticity problem using the virtual element method, in various load conditions and domains; the domains will be meshed using arbitrary polygonal meshes.
- Solve the linear elasticity problem, in the same conditions and domains, using the finite element method; now the domains are triangulated.
- Obtain measures of error and computation time, and compare the performance of the proposed method with the standard one.
- Generate high quality open source code that can be further extended.

1.4 Methodology

The methodology employed in the thesis is a practical one, that is to say, design, development and testing are pivotal to the work; this will be reflected as the thesis is presented. As was commented before, it was decided to use Voronoi diagrams generated from Delaunay triangulations; specifically, Triangle[37] was chosen because of its legacy, stability and known usage in the scientific community. Hence, taking Triangle, implemented in C, as a base and looking for as much efficiency as possible, it was decided to code in C++, considering that object oriented programming is a recognized paradigm, known for its advantages, such as extensibility, maintainability and encapsulation.

That being said, the main phases of this work will be:

1. Review of methods used in mesh generation, numerical solution of linear elasticity problems in polygonal meshes, and operations on polygonal meshes, such as refining and cutting.
2. Design and implementation of a polygonal mesh generation library, using the bounded Voronoi diagram as a base.
3. Design and implementation of a virtual element method for elasticity library.
4. Design and implementation of an extension to the polygonal mesh generation library, including new functionalities related to mesh changes.
5. Analyze complexity of the most relevant algorithms and conclude about theoretical performance.
6. Solve linear elasticity problems in several cases using both the proposed methodology and the standard finite element method.
7. Compare both methodologies in both result accuracy and computation time.
8. Design and implement a set of numerical examples, that will be left public, to compare accuracy of the methods.
9. Generate examples of polygonal meshes in different conditions, including complex non-convex domain, testing our polygonal mesh generation library.
10. Study results and conclude about performance, contrasting with theoretical results.

1.5 Thesis content

This thesis is organized as follows: Chapter 2 presents the most relevant background information needed to understand this work, covering from the geometrical concepts and algorithms, to a small summary of linear elasticity theory and both relevant numerical methods: finite element and virtual element method. Chapter 3 features the literature review done about methods to solve linear elasticity problems on polygonal meshes, plus some related techniques that were either investigated as alternatives to the final solution or used as inspiration. Chapter 4 describes the high-level design of the solution presented, including the experimental design, while Chapter 5 shows its concrete implementation. Chapter 6 displays the obtained results, both theoretical and experimental. Chapter 6 presents a few examples of the geometrical results obtained, that helps us prove our geometrical hypothesis, while the final chapter summarizes our work, including conclusions obtained from it and aspects left open as future work.

Chapter 2

Background

This chapter presents the knowledge necessary to understand this thesis, divided in two main areas: the first covers the computational geometry concepts used to model the polygonal meshes, while the second features the problem of linear elasticity itself, including information on the techniques used to numerically solve the problem.

2.1 Geometry

Geometry is an important part of every numerical simulation, as they all need to discretize a continuous domain to be able to compute whatever variable is needed. We will first explore the necessary concepts, data structures and algorithms that are both needed to completely understand this work, and are used later to design a solution to the problem at hand.

2.1.1 Basic concepts

First, we will introduce a set of definitions relevant to the understanding of the concepts detailed in following sections.

Definition (Circumcenter)**.** *The circumcenter of a triangle is the center of the circumcircle, namely, the circle which passes through all its vertices.*

Definition (Convex Hull)**.** *The convex hull of a set of points S is defined as the smallest convex polygon that contains all the points in S. By definition, a polygon is convex if the vector formed between two points inside the polygon is completely inside the polygon as well.*

Definition (Triangulation)**.** *A triangulation is a subdivision of a planar object into non-overlapping triangles, on which any triangle edge is shared by a maximum of two triangles.*

Definition (Centroid)**.** *The centroid is the geometric center of a polygon. Intuitively, the point where a thin slice of the same shape as the polygon could be perfectly balanced.*

Definition (Minimum bounding Box)**.** *The minimum bounding box of a point set S is the expression of the maximum extends of the set within its coordinate system, in other words, the maximum and minimum values in all axis.*

2.1.2 Delaunay and Voronoi diagrams

A triangulation T of a set of points S is Delaunay if there is no point in S inside the circumcircle of any triangle in T; this property is called empty circle property. The empty circle property is maintained even when new points are added, as is the case with many of the algorithms to generate the diagram itself, using the flip edge operation, shown in Figure 2.1. Delaunay triangulations have a set of interesting properties, such as maximizing the smallest of all the angles of the triangles and having a bounded number of elements, dependent on the number of points in the original set.

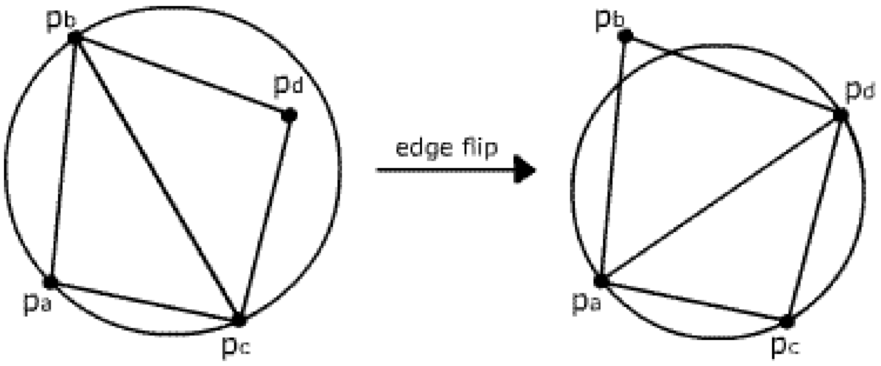


Figure 2.1: *Flip edge operation*

It is important to note that the Delaunay triangulation of a set of points S tessellates the convex hull of S; this point is relevant because otherwise modified, the Delaunay triangulation will ignore non- convexities and tessellate the convex hull, as it is shown in Figure 2.2 which could be problematic in some cases. The constrained Delaunay triangulation was created to tessellate non- convex regions or force edges to be in the triangulation; this structure satisfies the empty circle property in most triangles, but can leave some nonconformities to respect constrained edges, as the triangles in said segments cannot be flipped. Since the quality of the triangles is no longer ensured, it can be risky to use in simulations. To avoid this problem and still be able to constrain edges, the *conforming Delaunay triangulation* includes points outside the original set, called Steiner points, on the constrained edges until the empty circle property is satisfied.

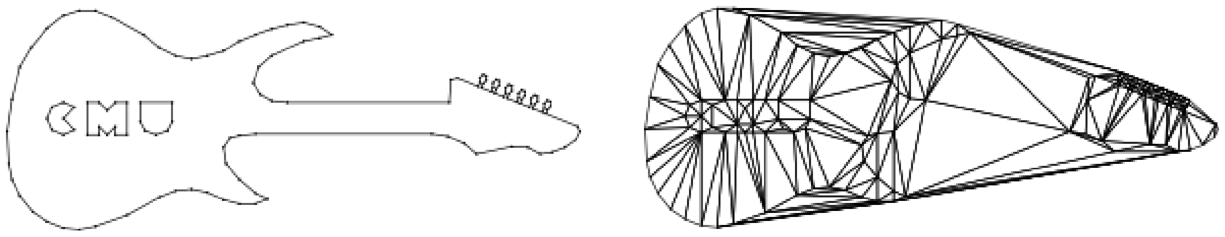


Figure 2.2: *Delaunay triangulation[37]*

The Voronoi diagram is the dual of the Delaunay triangulation, meaning, there is a linear mapping between the two structures, shown in Figure 2.3. Formally, given a set of points S (called a set of seeds or sites) in a space H, the Voronoi diagram or Voronoi tessellation of S

corresponds to the partition of H into regions, one for each point in S. Each region, called Voronoi cell, contains all the points p in space that are closer to the corresponding site than to any other. Unlike Delaunay triangulations, Voronoi diagrams tessellate the entire space, thus the existence of the constrained Voronoi diagram; computation of this diagram can be done starting from the unbounded Voronoi diagram[24][15] or using the duality Delaunay-Voronoi.

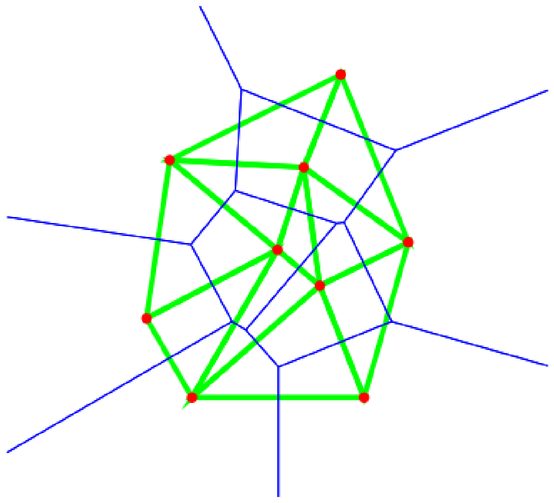


Figure 2.3: *Delaunay-Voronoi duality*

The Centroidal Voronoi Tessellation (CVT) is a special variety of Voronoi diagram which, besides complying with the Voronoi properties, has the additional constraint that the location of the generating points is the center of mass (centroid) of each Voronoi region; it can be viewed as an optimal partition of the space with the optimal distribution of the generators. The CVT is used in several applications because of its uniformity properties[36]. The difference between a Voronoi diagram and a Centroidal Voronoi diagram can be seen in Figure 2.4.

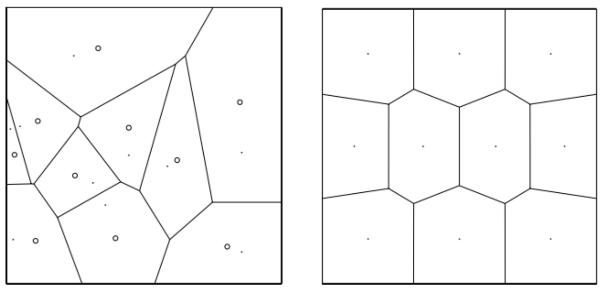


Figure 2.4: *Voronoi and Centroidal Voronoi Tessellations[36]*

The most popular algorithm for the construction of a CVT is known as Lloyd's algorithm, which iterates computing Voronoi diagrams and moving the centroids until it finds the CVT; the complexity of this algorithm is dependent on the distribution of the initial generating points, ranging from linear in the best case to super polynomial in the worst. Moreover, the convergence rate of this algorithm is known to be slow even in simple cases[35], making it unsuitable for several applications.

2.1.3 Data structures

In this section are discussed the various ways of representing polygon meshes as data structures, including structures for storing geometrical information, relationships between elements, among others. As it is a broad area, it is important to make explicit some restrictions to the meshes that will limit the scope of the investigation; these constraints will be valid throughout all this thesis, so further sections may refer to this one. Also, since there are numerous representations for polygon meshes, depending on the properties, operations and restrictions imposed (either memory space, complexity or both), we will consider, and refer to, only those that are relevant to this project.

It is important to remark on the fact that the library Triangle[37], used for the generation of conforming Delaunay triangulation, does not use any of the following described structures, but rather creates a custom triangle-based structure; however, as we are working with general polygons, that structure is not useful for our purposes.

In this subsection we will first address the assumptions taken about the properties of the polygons, following by a small summary of three commonly used data structures, including their advantages and disadvantages.

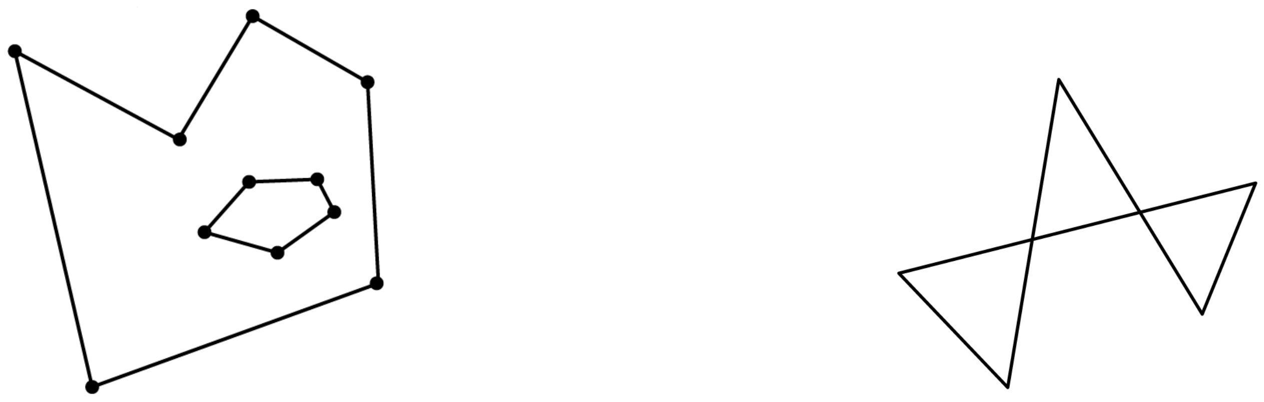
2.1.3.1 Assumptions

Before all else, the most relevant point to clarify is that the work is limited to two dimensions; extension to three dimensions is out of the scope of this thesis and will not be discussed.

The rest of the assumptions taken are related to the properties of the meshes: First, there is no restriction whatsoever in the convexity of the polygons, so convex or concave are equally valid, but polygons with holes, like the one shown in Figure 2.5a are not included; this is because polygons are represented as a single list of points, so it is impossible to have two separated set of segments. Next, points have real numbers for coordinates, so precision errors are a concern that must be addressed. Third, self-intersecting polygons, such as the one seen in Figure 2.5b, are not considered.

2.1.3.2 Vertex-Vertex

The simplest possible representation, consisting of a list of points coordinates that all other structure reference; polygons are an ordered list of point indexes, and segments are implicit. It is cheap in memory, but every operation on edges or faces require traversing through the list of points, making the cost at least linear. However, this structure is used because of memory restrictions and the ability to allow polygon changes at low cost.



(a) Polygon with hole **(b)** Self intersecting polygon

Figure 2.5: *Examples of invalid polygons*

2.1.3.3 Face-Vertex

Face-vertex meshes represent a polygon as a set of vertices; each polygon contains references to the indexes of its vertices and neighbours (other polygons), and edges are left implicit. For easier traversal of the mesh, each point keeps a reference to all the faces that share it, hence, location of neighbours is average constant time for a Delaunay tessellation. However, edges are not explicit, so dynamic operations such as splitting or merging of polygons are costly.

2.1.3.4 Winged-Edge

Winged-edge meshes explicitly represent vertices, faces and edges of a mesh. This structure contains three lists, polygons, edges and vertex. Polygons are represented by the list of edges that compose them; vertices keep their coordinates and references to their incident edges; lastly, edges keep information of the vertices that compose them, faces that share them and four pointers: previous and next edges when traversing in clockwise order and counterclockwise, respectively, as it is seen in Figure 2.6. It is widely used in modeling programs because it provides the greatest flexibility in changing the mesh, but with the downside of large memory consumption and increased complexity programming the algorithms due to the maintenance of many indexes.

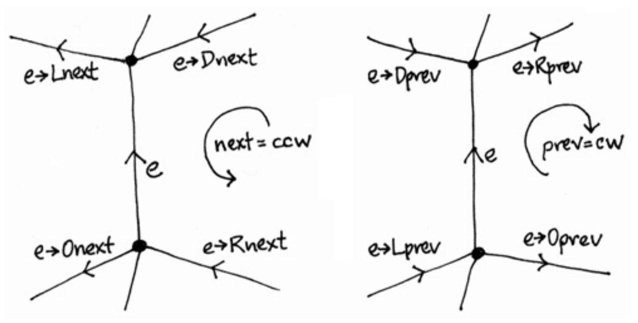


Figure 2.6: *Winged-edge data structure*

2.1.4 Key algorithms

In this section we will review some important algorithms that, although are pivotal to this thesis, are well known and obtained directly from the literature, so they can be considered theoretical background. These algorithms will be explained in detail here and referenced in following sections.

2.1.4.1 Segment intersection

A convenient way of representing line segments is through parametric equations; a line segment from $p_0 = (x_0, y_0)$ to $p_1 = (x_1, y_1)$ can be described as $p = p_0 + v * t$, with t in the interval $[a, b]$, and where $v = (x_1 - x_0, y_1 - y_0)$. This representation is general, as an infinite line goes from $a = -\infty$ to $b = \infty$, while a line segment from $a = 0$ to $b = 1$.

With the parametric equations at hand, two-line segments can be seen as a two-equation system, and its intersection can be determined from the solution; the only difference with the common intersection is that, since we are dealing with line segments, the intersection is valid if the parameter obtained is in $[0, 1]$. We can see the implementation of this algorithm in Line 1.

Algorithm 1: Line segment intersection

```
Data: Two-line segments ($p_1, p_2$) and ($o_1, o_2$)
Result: True if they intersect, false if not; intersection point $i$ if existent
$s1 \leftarrow p_2$ - $p_1$
$s2 \leftarrow o_2$ - $o_1$
Solve for t and s,
    $p = p_1 + s_1 * t$
    $p = o_1 + s_2 * s$
if $s$ in $[0, 1]$ and $t$ in $[0, 1]$ then
    $i \leftarrow p_1 + t * s_1$
    return True
end
return False
```

This algorithm is clearly $O(1)$ since the entire algorithm relies in formulas; solving a linear system can be done by standard method such as substitution, which only need sums and multiplications. It is important to note, however, when implementing this algorithm, strictly comparing the results is not an option, as the precision errors sometimes yield results slightly over one, or under zero, that still need to be considered as valid; this problem, first presented here, will be an important subject throughout this entire work.

2.1.4.2 Point on segment

To determine if a point p is on a line segment (p_1, p_2) the simplest, and more efficient, way is to verify if the point is between the endpoints and it is aligned to the segment. The first check can be done with simple comparison of coordinates, but the second requires computing the cross product between the vectors $p_1 - p_2$ and $p - p_1$ and checking if it is equal to zero.

Algorithm 2: Point on segment

```
Data: Point $p$, Line segment $(p_1, p_2)$
Result: Boolean indicating if $p$ lies on segment
return  $(p \geqslant p_1$ and $p \leqslant p_2$ or $p \geqslant p_2$ and $p \leqslant p_1)$ and
        $|(p.y - p_1.y)(p_2.x - p_1.x) - (p.x - p_1.x)(p_2.y - p_1.y)| < \varepsilon$
```

The order of the tests in Algorithm 2 is relevant, since the first calculations are likely to discard many of the candidates and leave the more expensive computation of the cross product last, when it is still possible to get a positive answer. Still, this algorithm is $O(1)$, as all operations are sums, multiplications and comparisons.

2.1.4.3 Polygon area

To calculate the area of any convex polygon, it is enough to choose any vertex, triangulate and obtain the total area as the sum of the area of each triangle, as shown in Figure 2.7 and Equation (2.1).

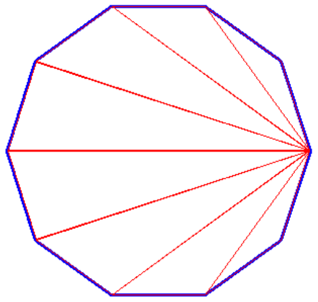


Figure 2.7: *Triangulation to calculate polygon area*

$$A(p) = A_t(p, v_0, v_1) + A_t(p, v_1, v_2) + ... + A_t(p, v_{n-2}, v_{n-1}) + A_t(p, v_{n-1}, v_0) \tag{2.1}$$

where p is any point in the plane and $A_t(a, b, c)$ is the signed area of the triangle defined by the points a, b and c, which is calculated using Algorithm 3.

```
Algorithm 3: Area of a triangle
  Data: Points $p_1$, $p_2$ and $p_3$
  Result: Area of the triangle defined by the three points
  $v_1 \leftarrow p_2 - p_1$
  $v_2 \leftarrow p_3 - p_1$
  $area \leftarrow \frac{v_1.x*v_2.x-v_1.y*v_2.y}{2}$
```

The methodology shown before can be applied to general polygons, but the problem with non-convex ones is that the process of actually obtaining the triangulation is costly; to bypass this and use the simple method, one can triangulate considering that the polygon is convex and work with the signed area, calculated so that it is positive when the triangle is a counterclockwise path and negative when clockwise. Hence, the area of any convex or non-convex polygon with vertices v_0, v_1,..., v_{n-1} labeled counterclockwise can be calculated as Algorithm 4.

```
Algorithm 4: Area of a general polygon
  Data: Polygon $p$ with vertices $p_1, ..., p_n$
  Result: Area of the polygon $p$
  $area \leftarrow 0$
  foreach i in $[1, n]$ do
      $area \leftarrow area + triangleArea(p_\mathrm{i}, p_{\mathrm{i}+1}, p_0)$
  end
```

As it is obvious from the previous procedure, the computation of the area of any polygon (either convex or non-convex) is always done in time $O(n)$.

2.1.4.4 Polygon centroid

The computation of the centroid of any closed non- self-intersecting polygon of n vertices $(x_0, y_0), ..., (x_n, y_n)$ numbered counterclockwise is as simple as replacing the vertices in their order of apparition in the equations shown in Equation (2.2)[10], where the coordinates of the centroid are (C_x, C_y), and the area of the polygon A is calculated using the methodology shown in the previous section. As this is just a formula, it is $O(1)$.

$$\begin{aligned} C_x &= \frac{1}{6A}\sum_{\mathrm{i}=0}^{n-1}(x_\mathrm{i} + x_{\mathrm{i}+1})(x_\mathrm{i} y_{\mathrm{i}+1} - x_{\mathrm{i}+1} y_\mathrm{i}) \\ C_y &= \frac{1}{6A}\sum_{\mathrm{i}=0}^{n-1}(y_\mathrm{i} + y_{\mathrm{i}+1})(x_\mathrm{i} y_{\mathrm{i}+1} - x_{\mathrm{i}+1} y_\mathrm{i}) \end{aligned} \tag{2.2}$$

2.1.4.5 Polygon diameter

The diameter of a polygon is the largest distance between any pair of its vertices. There are various ways of dealing with this problem, but all of them are bounded to convex polygons, and one of our first assumptions is that we would not constraint convexity; hence, we first must find a way to get over this issue. It is decided to first calculate the convex hull of the polygon and then use one of the classic algorithms to compute the diameter of a convex polygon; it is important to note that this approach is accurate, since the convex hull represents the polygon and does not add extra points.

In our case, we work in two dimensions, so simple and efficient ($\mathcal{O}(n \log n)$) algorithms for the computation of the convex hull are readily available; for its simplicity and ease of understanding, we choose Andrew's Monotone Chain Algorithm, shown in Algorithm 5. An important part of this algorithm is orientation, a simple test that checks the order (clockwise or counter clockwise) of three points; this is done using vector cross product, and checking whether the result is positive (counter clockwise) or negative (clockwise).

Algorithm 5: Monotone chain algorithm

```
Data: List of points $P$, ordered first by x value, and then by y value
Result: Two lists of points $U$ and $L$, representing upper and lower convex hull
U ← [ ]
L ← [ ]
foreach $p \in P$ do
    n ← Size of U
    while $n > 1$ and orientation(U[n-2], U[n-1], p) $\geqslant 0$ do
        remove last element of U
    end
    insert $p$ into U
    m ← Size of L
    while $m > 1$ and orientation(L[m-2], L[m-1], p) $\leqslant 0$ do
        remove last element of L
    end
    insert $p$ into L
end
return U,L
```

After computing the convex hull, to determine the diameter we use an algorithm known as Rotating Calipers[45], which name comes from the analogy of rotating a caliper around a convex polygon; it can be used to solve several problems, including bounding boxes, minimum distance, polygon diameter, among others. The advantage of this algorithm is that works extremely well with monotone chain, since having upper and lower convex hulls avoids the calculations of angles typical to this method, and allows to directly determine antipodal pairs (points that are diametrically opposite of each other); using this pairs, determining the diameter is just a matter of searching for the pair of points that are the farthest from each other, procedure shown in Algorithm 7. The pseudo code for the rotating calipers algorithm using Monotone Chain for convex hull is presented in Algorithm 6; for further clarification, a schematic representation of the advance of the rotating caliper around a polygon is shown in Figure 2.8.

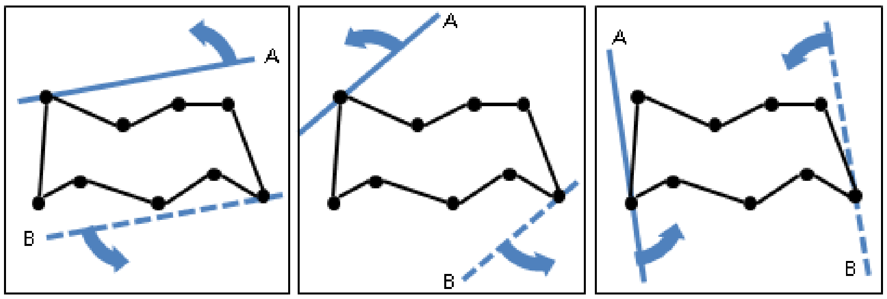


Figure 2.8: *Rotating calipers algorithm*

Algorithm 6: Rotating calipers using monotone chain

```
Data: Upper and lower convex hulls (U, L)
Result: List of antipodal pairs $A$

A ← [ ]
i ← 0
j ← Size of (U) - 1

while $i <$ (Size of (U) - 1) or $j > 0$ do
    insert ($P$[i], $P[j]$) into $A$
    if $i$ equals Size of (U) then
        j ← j - 1
    else if $j$ equals 0 then
        i ← i + 1
    else if Slope of tangent of upper hull > Slope of tangent of lower hull then
        i ← i + 1
    else
        j ← j - 1
    end
end
```

Algorithm 7: Determination of the diameter of a polygon from antipodal pairs

```
Data: List of antipodal pairs $A$
Result: Diameter d

$max \leftarrow -1$
foreach Pair $p$ in $A$ do
    distance ← norm of ($p$.first, $p$.second)
    if distance > $max$ then
        $max$ ← distance
    end
end
return $max$
```

2.1.4.6 Point in polygon

Testing whether a given point p lies inside or outside a polygon is a well-known problem that has many solutions, depending on the restrictions imposed. In our case, those constraints lead us to choose an algorithm called ray crossing because of its generality (we need an algorithm that works with non-convex polygons) and efficiency.

Ray crossing is based in a simple concept: counting the times one enters and exits the polygon when walking in an arbitrary direction can tell us if the point is inside or not, as can be seen in Figure 2.9. Despise the simplicity, the implementation should be extremely careful of a few border cases that can arise: the ray can cross through a vertex, be collinear with an edge or the point can lie in the edges of the polygon. To avoid these issues, a single supposition is enough: to count an edge as crossed, one of its endpoints has to be strictly above the given point. On the other hand, to be able to handle points in the border, this specific test is done separately at the end of the computation, as show in Algorithm 8.

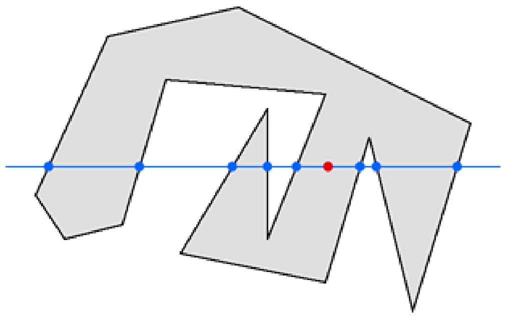


Figure 2.9: *Ray crossing*

Algorithm 8: Point in polygon

```
Data: Polygon poly with vertices $v_0$,...,$v_{n-1}$ and point $p$
Result: True if $p$ is inside poly
odd ← False
$j \leftarrow n - 1$
for i ← 0 to $n - 1$ do
    $p_i \leftarrow v_i$
    $p_j \leftarrow v_j$
    if $p$ is above $p_i$ and below $p_j$ or above $p_j$ and below $p_i$ then
        if $p$ intersects segment $(p_i, p_j)$ then
            odd ← ¬odd
        end
    end
    $j \leftarrow i$
end
if odd then
    return True
end
return inEdges(poly, $p$)
```

The last check, seemingly redundant, is done to avoid checking if the point is in the edges of the polygon when it is already determined that it is inside. The algorithm named *inEdges* just iterates through all the edges of the polygon and checks if the point is contained in each one; the test is done using the algorithm explained in Section 2.1.4.2. Ray crossing is $O(n)$ with n the number of vertices when no assumptions are made on the characteristics of the polygons, as it is our case; however, it but can be optimized up to order $O(logn)$ if the polygon is convex, which we cannot apply, hence, we need to assume a linear cost.

2.2 Numerical solution of linear elasticity

The other important subject of this thesis is numerical linear elasticity; we present the necessary information to understand this aspect of the work, including basic concepts related to the theory of elasticity, and numerical resolution of the problem itself, including both finite element and virtual element methods.

2.2.1 Basic concepts

In the first place, we will go through some important concepts that constantly throughout this chapter.

Definition (Strain)**.** *Strain is a measure of deformation, representing the displacement between particles relative to a reference length (usually a measure of the size of the specimen). It is closely related to displacement, u, and denoted* ε.

Definition (Stress)**.** *Stress is a physical quantity that expresses the internal forces that particles exert on each other. It is denoted* σ.

Definition (Isotropy)**.** *A material is called isotropic if its mechanical properties are the same in all orientations; a material that does not satisfy this condition is called anisotropic.*

Definition (Young modulus)**.** *Also called the modulus of elasticity, it is the mechanical property that describes the tendency of an object to deform along an axis when opposing forces are applied along a particular axis. It is denoted by* E *in this work.*

Definition (Poisson's ratio)**.** *Poisson's ratio represents the tendency of the materials to expand or contract in directions perpendicular to the direction of compression, phenomenon called the Poisson effect. It is the ratio of the strain perpendicular to P to the strain parallel to P, as shown in Figure 2.10. It is denoted by* ν.

2.2.2 Linear elasticity

Linear elasticity is the mathematical study on how solid objects deform when loading conditions are applied; it is the simplest, and widely used, model for structural analysis and engineering design. Linear elasticity is a branch of the theory of elasticity, but with several assumptions that greatly simplify the problem, but are still reasonable enough for engineering applications.

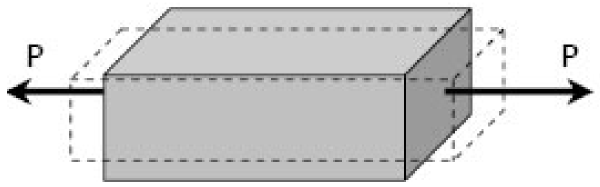


Figure 2.10: *Representation of the Poisson effect*

- Small deformations, usually with respect to the size of the domain.
- Linear relationship between stress and strain. This assumption is valid for most materials provided sufficiently small stresses.
- The material returns to its original shape when the loads are removed; in other words, it is only valid without yielding.

The laws governing this problem have different forms, all of them representing the same phenomena; since we will next use them in the following sections, but heavily modifying their form, we choose the most general and compact form, as shown in Equation (2.3).

$$\begin{aligned} &\nabla \cdot \boldsymbol{\sigma} + \boldsymbol{b} = 0 \quad in \quad \Omega \\ &\boldsymbol{\varepsilon} = \frac{1}{2}\left[\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T\right] \\ &\boldsymbol{\sigma} = \boldsymbol{C} : \boldsymbol{\varepsilon} \end{aligned} \tag{2.3}$$

$\boldsymbol{u}$ is the vector with the displacements of the system; in 2D problems, it is usual to refer to each component of the vector $\boldsymbol{u}$ independently, u for the x-displacement and v for the y-displacement. $\boldsymbol{\sigma}$ is a vector that contains the stresses in all directions. To avoid overloading the notation, stresses perpendicular to the face on which they are applied, called normal stresses, are denoted as σ with two sub-indexes: the first represents the face containing the stress and the second is the stress direction. On the other hand, stresses parallel to their faces, shear stresses, and denoted with τ; their sub-indexes have the same interpretation as normal stresses. All stresses of a 3D body can be seen in Figure 2.11.

$\boldsymbol{C}$ is called the stiffness tensor, and its form depends heavily on the properties of the material and the conditions of the problem. In our case, all work is restricted to 2D, so several simplifications can be made, as will be shown next.

Elasticity problems fall into the category of boundary value problem, meaning that the problem itself may not be solvable without boundary values; these boundary conditions are shown in their generic form in Equation (2.4). In linear elasticity, there are two types of conditions commonly found (which are also referred to as constraints): the first type is called Neumann, or natural boundary conditions (as they are included in the system before its resolution), while the second is referred to as Dirichlet, or essential (they are literal solutions

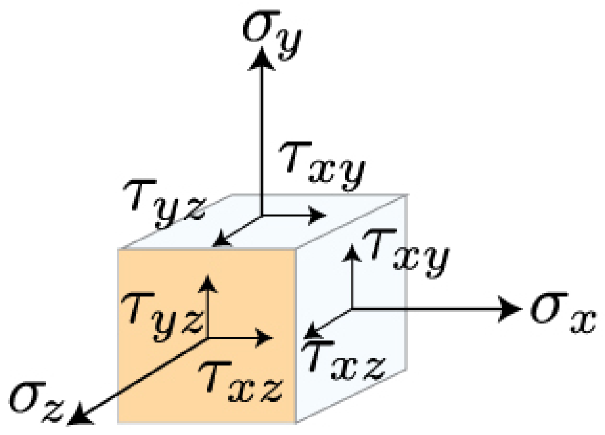


Figure 2.11: *Stress state of a generic 3D body*

to certain degrees of freedom) conditions; both can be seen in Figure 2.12.

$$\begin{aligned} \boldsymbol{n} \cdot \boldsymbol{\sigma} = \bar{\boldsymbol{t}} \quad in \quad \Gamma_t \\ \boldsymbol{u} = \bar{\boldsymbol{u}} \quad in \quad \Gamma_u \end{aligned} \tag{2.4}$$

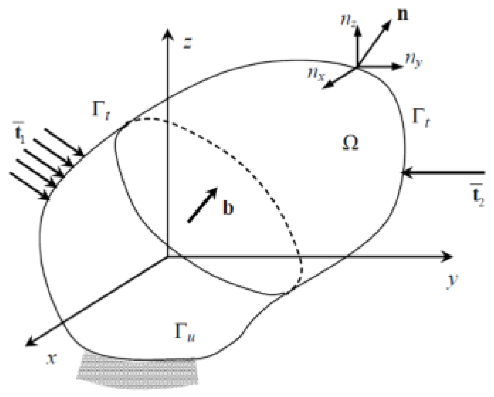


Figure 2.12: *Arbitrary 3D loaded body*

Plane strain and plane stress As stated previously, the matrix $\boldsymbol{C}$ takes different forms depending on the conditions of the problem. In the case of 2D analysis, there are two important simplifications that can be made, called plane stress and plane strain.

A material is considered under plane stress when one of the components of the stress vector is zero across a particular plane (or at least considerably smaller than the other directions), as is often the case for thin plates; in this case, the stress analysis is simpler, as the stress and strains vectors are now 3×3 and $\boldsymbol{C}$ takes the value:

$$\boldsymbol{C} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \tag{2.5}$$

On the other hand, plane strain can be applied to in situations when one direction is "infinitely" large in comparison to the others (as in several orders of magnitude larger at least), so that forces in the studied plane are practically independent of the third axis; this is applicable, for example, in beams, dams, tunnels and other big structures. As it is the case with plane stress, in plane strain the vectors are of size 3×3 and $\boldsymbol{C}$ takes a particular value:

$$\boldsymbol{C} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \tag{2.6}$$

2.2.3 Numeral integration

Before continuing with numerical methods for solving elasticity problems, we need to touch first the important subject of numerical integration, which is extremely relevant as finite element methods are based on solving the integral form of differential equations, and a good part of precision in the solutions is lost if the integrals are not exact.

The basic problem in numerical integration is to compute an approximate solution to a definite integral

$$\int_a^b f(x)\mathrm{d}x \tag{2.7}$$

to a given degree of precision; depending on the properties of $f(x)$ there are many methods to approximate this value or even obtaining its exact solution. In our case, we are only concerned with the methodology commonly used in Finite Element Analysis (FEA), known as numerical quadrature or quadrature rule.

A quadrature rule approximates the desired integral as a weighted sum of the values of function f evaluated at specified points (called quadrature points) within the domain of integration, as seen in Equation (2.8). The index i refers to the i-th quadrature point, w_i to its corresponding weight, and $f(x_\mathrm{i})$ to the function evaluated at the quadrature point; it is important to note that functions with singularities, especially in the quadrature points values, tend to give extremely bad results.

$$\int_{-1}^{1} f(x)\mathrm{d}x \approx \sum_{\mathrm{i}=1}^{n} w_\mathrm{i} f(x_\mathrm{i}) \tag{2.8}$$

A n-point Gaussian quadrature rule is a specific type of quadrature rule constructed to yield an exact result for polynomial functions up to degree $2n-1$. The domain of integration of a Gaussian quadrature (and most of its variants) is $[-1, 1]$, so a change of interval is required for computing integral in general domains, as seen in Equation (2.9).

$$\int_a^b f(x)\mathrm{d}x \approx \frac{b-a}{2} \sum_{\mathrm{i}=1}^{n} w_\mathrm{i} f\left(\frac{b-a}{2} x_\mathrm{i} + \frac{a+b}{2}\right) \tag{2.9}$$

There are many gaussian integration rules, such as Gauss-Legendre, Gauss-Hermite and Chebyshev-Gauss; quadrature weights (and the corresponding quadrature points) can be analytically computed, but can be prohibitively costly, so they are usually kept at hand in quadrature tables to minimize computation cost.

Gaussian quadratures are heavily used in finite element simulations since most integrals needed for the method are polynomials. However, some FEM variations, such as polygonal finite elements or extended finite element, use non- polynomial functions, which have to be integrated, giving origin to the so-called intrinsic integration errors.

Finally, Gain et. al[1] present a simple first order approximation for the computation of integrals without using any quadrature rule whatsoever, at the expense of only being able to integrate exactly up to linear order; the surface integral over a face F with m vertices can be approximated by

$$\int_F f(x)\mathrm{d}s \approx \sum_{j=1}^{m} w_j f(x_j) \tag{2.10}$$

where $x_1, ..., x_m$ are the vertices of the face F, and the weights w_j are the areas of the quadrilateral formed by x_j, the midpoint of the edges incident on x_j, and the centroid of F. We call this technique a **nodal integration scheme**.

2.2.4 Numerical methods for elasticity

In this subsection we will explore in detail numerical methods to solve the linear elasticity problem. We will first review the Finite Element Method, its theory, the standard procedure for general FEA, and how it is implemented at a high level. Then, we will dedicate ourselves to Virtual Element Method, which we will explore in the same order: starting with the theory, we then review the implementation, using as a basis the finite element procedure shown first.

2.2.4.1 Finite Element Method

As mentioned before, this thesis uses a novel technique called Virtual Element Method to solve the linear elasticity problem; however, most of the concepts found in VEM are better understood if one is familiar with finite elements, hence, before presenting VEM we will briefly summarize the most relevant concepts of FEM to assure that any reader not familiar with the subject can easily comprehend the information presented in the next section.

FEM background In most applications, a physical phenomenon is presented as a partial differential equation with boundary conditions, like the linear elasticity equations shown in Equation (2.3). This notation, known commonly as classic or strong form, is convenient for human beings as it is compact, easy to understand and exact, but most of the time problems are hard or downright impossible to solve. Nonetheless, we need solutions, so several methods have surfaced that use a modified version of the strong form, obtained using variational calculus; through this technique, known generally as Galerkin method, one can obtain the so-called weak form of the problem, an integral equation equivalent to the differential one. Finite element methods make use of the method of mean weighted residuals (MWR), a particular kind of Galerkin method.

As all Galerkin methods, MWR assumes that an exact solution may not be possible to obtain and instead tries to find the best approximate solution that satisfies essential boundary conditions but might not do the same with natural ones. The residual $R(x)$ is defined as the difference between the real solution and the approximate one, the measure of the error in the solution. Naturally, the aim of the method is to minimize this residual; when the solution is exact this value is zero, and, for an approximation to be considered good, R has to be small in all parts of the domain.

After defining the residual, the first step in the finite element method (and in most Galerkin methods actually) is to assume an approximation, which is usually of the form

$$\tilde{\boldsymbol{u}} = \sum_{\mathrm{i}=1}^{n} w_{\mathrm{i}} \boldsymbol{\varphi}_{\mathbf{i}} \tag{2.11}$$

where φ_{i} are the trial functions and w_{i} are the approximation coefficients to be calculated. The choice of trial functions for different problems is a whole area of study in finite elements, outside the scope of this thesis; the choice of trial functions in virtual element method does concern us though, but will be discussed later.

MWR gets its name from the fact that its objective is to minimize the weighted residual, computed as shown in Equation (2.12).

$$\int_E W(x) R(x) \tag{2.12}$$

If the value in Equation (2.12) equals zero for any weight function $W(x)$, then $R(x)$ approaches zero and the approximated solution will approach the exact solution.

Finally, in the case of linear elasticity, the weak form of Equation (2.3) is shown in Equation (2.13); this is the form that later we will reference when detailing virtual element methods.

$$\int_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) : \boldsymbol{\varepsilon}(\boldsymbol{v}) = \int_\Omega \boldsymbol{b} \cdot \boldsymbol{v} \mathrm{d}\boldsymbol{x} + \int_{\Gamma_t} \boldsymbol{t} \cdot \boldsymbol{v} \mathrm{d}\boldsymbol{s} \tag{2.13}$$

As we will be working with what is called a Galerkin method, we will further simplify the previous integral equation (Equation (2.13)) taking that $\boldsymbol{u}$ and $\boldsymbol{v}$ (trial and weight function, respectively) are equal. We must remark on the fact that, in other variants of the method, this supposition is not necessarily true.

Finally, the solution to the linear elasticity problem that we wish to solve is represented by the value $\boldsymbol{u^h}$ that approximates the real solution $\boldsymbol{u}$ in each point of the domain; from those approximates values one can obtain the value on all points through linear interpolation. More formally, consider a domain Ω discretized with two dimensional elements (commonly triangles or quadrilaterals), then the displacements $\boldsymbol{u}$ at any point of the domain can be approximated as,

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} N_1 & 0 & N_2 & 0 & \cdots \\ 0 & N_1 & 0 & N_2 & \cdots \end{bmatrix} \begin{bmatrix} u_{x_1} \\ u_{y_1} \\ u_{x_2} \\ u_{y_2} \\ \vdots \end{bmatrix} \tag{2.14}$$

where the values u_{x_1}, u_{y_1}, $\cdots$, u_{x_n}, u_{y_n} are the approximated displacements, called nodal values, and N_i are named shape functions.

One last remark is that the previous definitions are presented in function of the global, or real, coordinates of each element; however, using these coordinates directly has several disadvantages:

- The construction of consistent shape functions for higher order elements with curved boundaries is extremely complicated.
- The finite element method requires to numerically evaluate integrals inside the elements, which, as they can be arbitrarily shaped, is inconvenient and complicated.

To solve the previous issues, the concepts of isoparametric elements appeared. The term isoparametric is derived from the fact that the same shape functions are used to define the element shape and the displacements within the element. Isoparametric elements are formulated using a natural (or intrinsic) coordinate system that is defined by the element geometry and not by the element orientation in the global coordinate system. To transform any element of a certain structure, one simply maps the coordinates to the isoparametric system, using the standard formula

$$\begin{aligned} x &= N_1 x_1 + N_2 x_2 + \cdots \\ y &= N_1 y_1 + N_2 y_2 + \cdots \end{aligned} \tag{2.15}$$

Shape functions As seen in the previous section, the approximated solution to the linear elasticity problem can be obtained from the nodal values and the shape functions; the first are obtained solving the linear system that is obtained through the finite element approximated; the second, though, are constructed from the required geometry so that they follow a set of properties. The isoparametric shape function N_i^e associated with node i of element e must satisfy the following conditions:

- Takes a unit value at node i, and is zero in all other nodes.
- Vanishes over any element boundary that does not contain node i.
- Is C^0 continuous between adjacent elements over any boundary that includes node i.
- The set of all shape functions can interpolate any field that is a linear polynomial in x and y; in particular, a constant value.
- The sum of the shape functions evaluated in any point is identically one. This property is called *partition of unity* in the literature.

Finally, there is a great number of finite elements, each with their own shape functions; however, most are out of the scope of this thesis, as we will use FEM to solve linear elasticity on triangular meshes, so we will focus only on triangular elements. Nevertheless, the implementation of FEM with triangular elements requires knowledge of the one-dimensional linear element, basically a line, to compute certain values, so we will start with this element, and then study the triangular ones.

As we are working on two dimensions, all isoparametric shape functions are presented as a function of (ξ, η), which can be seen as the isoparametric representation of (x, y); the only exception is the explicitly named one dimensional linear element, which, as it represents the boundary of the element, is only function of a single variable, s, which traverses it.

One dimensional linear element This element, shown in Figure 2.13, is a line segment with end points x_1 and x_2; the variable ξ represents the natural coordinate that traverses the element, going from -1 to +1. This way, the shape functions are:

$$\begin{aligned} N_1 &= \frac{1-\xi}{2} \\ N_2 &= \frac{1+\xi}{2} \end{aligned} \tag{2.16}$$

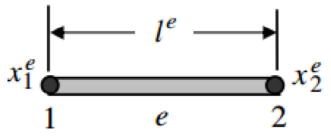


Figure 2.13: *One dimensional linear element for FEM*

Linear triangular element This element, also called constant strain triangle o CST, can be seen in Figure 2.14; as it is two dimensional, its shape functions are a function of (ξ,η) can be simply represented as

$$\begin{aligned} N_1 &= \xi \\ N_2 &= \eta \\ N_3 &= 1-\xi-\eta \end{aligned} \tag{2.17}$$

Quadratic triangular element Although called triangular, this element, seen in Figure 2.15, has six nodes and does not necessarily preserve the triangular shape after deformed; it is alternatively named as linear strain triangle or LST. Its six shape functions are

$$\begin{aligned} N_1 &= \xi(2\xi-1) \\ N_2 &= \eta(2\eta-1) \\ N_3 &= \zeta(2\zeta-1) \\ N_4 &= 4\xi\eta \\ N_5 &= 4\eta\zeta \\ N_6 &= 4\zeta\xi \end{aligned} \tag{2.18}$$

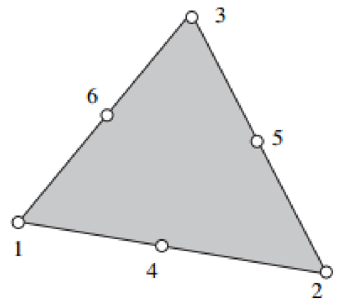


Figure 2.14: *Linear triangular element for FEM*

where $\zeta = 1 - \xi - \eta$.

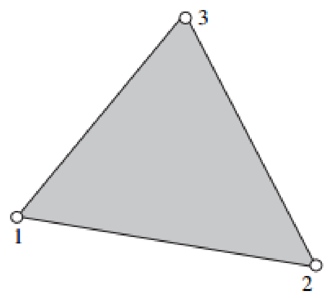


Figure 2.15: *Quadratic triangular element for FEM*

FEM process In this section we will go through the process necessary for a generic application of the finite element method and explain the important points as they seem, because, curious as it may appear, the whole process is completely parallel to VEM with differences in how to compute some terms. The steps necessary to a generic 2D FEM application are described as follows:

1. Mesh the geometric domain; in standard FEM the mesh must be made of triangles or quadrilaterals, but there are some variations that allow more general polygons. Each of the polygons define what is called an *element* of the numerical system, a small and atomic version of the problem to solve, while every vertex is called a *node*; this way, in a triangulation, every triangle has an associated element, while each element has three nodes.
2. Calculate the elemental stiffness matrix (universally denoted as K_{e}), that relates the variable to be calculated (either temperature, deformation, electric charge, depending on the problem to solve) in every point with all the others, also known as internal relationships of the element. The name "stiffness" comes from elasticity, as it was the original problem that FEM was intended to solve, but the term was adopted for all applications. The matrix $\boldsymbol{K}_{\mathrm{e}}$ is of size $n \times n$, where n is the number of "degrees of freedom" of the element (independent variables to calculate); in our case, elasticity in 2D, each node of the element has a horizontal and vertical displacement, so n equals two times the number of vertices.

3. Calculate the elemental load vector (represented with $\boldsymbol{f}_{\mathrm{e}}$), that relates the body forces imposed in the problem itself and natural conditions imposed externally to the degrees of freedom of the system.
4. To obtain the linear system that represents the problem to solve, the elemental matrix and load vector are "assembled". This process consists in creating an empty global stiffness matrix (or load vector, as the procedure is analogous) that has a space for each degree of freedom, and then fill in the spaces with every elemental matrix; since most degrees of freedom are present in more than one element, the contribution from the different elemental matrices is added. The complete matrix assembly algorithm is detailed in its own section next.
5. Essential boundary conditions are directly imposed in the system.
6. Solve the problem $\boldsymbol{Ku} = \boldsymbol{f}$ for $\boldsymbol{u}$ to obtain the results using standard matrix algebra.

Stiffness matrix and load vector We purposely left the computation of the stiffness matrix and load vector vague in the previous section as it is the most remarkable difference between the finite element and virtual element methods. In this subsection we will detail how both values are computed in FEM; we will leave all computed values dependent on N_{i}, as they are dependent on the chosen element.

To actually obtain the expressions for the stiffness there are still several operations that we need to do; taking the shape functions and nodal values defined previously, we can express the displacement function on the element. Now, remembering the definition given in Equation (2.3), we can use the displacement function to obtain the deformation as a function of the shape functions and nodal values

$$\boldsymbol{\varepsilon} = \begin{bmatrix}\varepsilon_{xx} & \varepsilon_{yy} & \gamma_{xy}\end{bmatrix} = \boldsymbol{\varepsilon}^{\mathrm{e}} = \nabla_S \boldsymbol{u}^{\mathrm{e}} = \nabla_S N^{\mathrm{e}} \mathbf{d}^{\mathrm{e}} = \boldsymbol{B}^{\mathrm{e}} \mathbf{d}^{\mathrm{e}} \tag{2.19}$$

Where $\mathbf{d}^{\mathrm{e}}$ is the vector of the nodal displacements $\mathbf{d}^{\mathrm{e}} = [u_1^{\mathrm{e}} v_1^{\mathrm{e}} u_2^{\mathrm{e}} v_2^{\mathrm{e}} \cdots]$, N_{e} is the element shape function matrix

$$\begin{bmatrix} N_1 & 0 & N_2 & 0 & \cdots \\ 0 & N_1 & 0 & N_2 & \cdots \end{bmatrix} \tag{2.20}$$

and $\boldsymbol{B}^{\mathrm{e}}$ is defined as

$$\begin{bmatrix} \frac{\partial N_1^{\mathrm{e}}}{\partial x} & 0 & \frac{\partial N_2^{\mathrm{e}}}{\partial x} & 0 & \cdots \\ 0 & \frac{\partial N_1^{\mathrm{e}}}{\partial y} & 0 & \frac{\partial N_2^{\mathrm{e}}}{\partial y} & \cdots \\ \frac{\partial N_1^{\mathrm{e}}}{\partial y} & \frac{\partial N_1^{\mathrm{e}}}{\partial x} & \frac{\partial N_2^{\mathrm{e}}}{\partial y} & \frac{\partial N_2^{\mathrm{e}}}{\partial x} & \cdots \end{bmatrix} \tag{2.21}$$

This way, replacing the value of $\boldsymbol{\varepsilon}$ in Equation (2.3) we obtain the stress in function of the shape functions and nodal values; finally, those two values can be inserted in Equation (2.13), which gives us the final form which we finally use to obtain the stiffness matrix and load vector on an element e,

$$\int_{\Omega^{\mathrm{e}}} \mathbf{B}^{\mathrm{e}T} \mathbf{D}^{\mathrm{e}} \mathbf{B}^{\mathrm{e}} \mathrm{d}\Omega = \int_{\Omega^{\mathrm{e}}} \mathbf{N}^{\mathrm{e}T} b \mathrm{d}\Omega + \int_{\Gamma_t^{\mathrm{e}}} \mathbf{N}^{\mathrm{e}T} \bar{t} \mathrm{d}\Gamma \tag{2.22}$$

From the previous equation we obtain the stiffness matrix and load vector directly.

- **Element stiffness matrix**:

$$\mathbf{K}^{\mathrm{e}} = \int_{\Omega^{\mathrm{e}}} \mathbf{B}^{\mathrm{e}T}\mathbf{D}^{\mathrm{e}}\mathbf{B}^{\mathrm{e}}\mathrm{d}\Omega \tag{2.23}$$

- **Elemental load vector**: It is denoted as $\mathbf{f}^{\mathrm{e}}$, and it is composed of two parts, the body force vector ($\mathbf{f}^{\mathrm{e}}_{\Omega}$) and the traction vector($\mathbf{f}^{\mathrm{e}}_{\Gamma}$).

$$\begin{aligned}\mathbf{f}^{\mathrm{e}}_{\Omega} &= \int_{\Omega^{\mathrm{e}}} \mathbf{N}^{\mathrm{e}T} b\mathrm{d}\Omega \\ \mathbf{f}^{\mathrm{e}}_{\Gamma} &= \int_{\Gamma^{\mathrm{e}}_{t}} \mathbf{N}^{\mathrm{e}T} \bar{t}\mathrm{d}\Gamma \\ \mathbf{f}^{\mathrm{e}} &= \mathbf{f}^{\mathrm{e}}_{\Omega} + \mathbf{f}^{\mathrm{e}}_{\Gamma}\end{aligned} \tag{2.24}$$

Matrix assembly The complete assembly algorithm is detailed in Algorithm 9.

Algorithm 9: Matrix assembly algorithm

```
Data: Elemental stiffness matrices $K_1 ... K_{n_e}$
Result: Global stiffness matrix $K_{global}$
foreach Element e do
    foreach Node $n_1$ of e do
        foreach Degree of freedom i of $n_1$ do
            foreach Node $n_2$ of e do
                foreach Degree of freedom $j$ of $n_2$ do
                    $i_{global} \leftarrow n_{dof} \times (connectivity(n_1, e) - 1) + i$
                    $j_{global} \leftarrow n_{dof} \times (connectivity(n_2, e) - 1) + j$
                    $K_{global}(i_{global}, j_{global}) \leftarrow$
                      $K_{global}(i_{global}, j_{global}) + K_e(n_{dof}(n_1 - 1) + i, n_{dof}(n_2 - 1) + j)$
                end
            end
        end
    end
end
```

The connectivity function used before in Algorithm 9 relates the index of a degree of freedom in the element (local index) to the corresponding global index, as can be seen in Figure 2.16. How this function is implemented depends heavily on the data structures, type of mesh, memory restrictions, etc.

2.2.4.2 Virtual Element Method

Virtual Element Method (VEM) is a variation of the classic Finite Element Method (FEM), with elements extracted from the Mimetic Difference Method[28]; the name "virtual" comes from the fact that, through mathematical manipulation, the basis functions that approximate

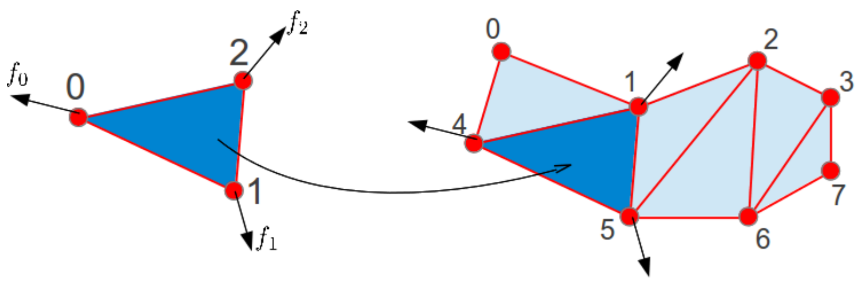


Figure 2.16: *Relationship between local and global node index*

the solution do not have to be computed numerically (they are "virtual") and all the necessary information is extracted from element geometry. As the stiffness matrix can be obtained analytically, many of the common integration errors in FEA are avoided, giving solutions potentially more accurate.

VEM has solid mathematical foundations that we will not explore deeply; our focus will be on the aspects relevant to the method in linear elasticity, but it is important to note that in no way this represents all the complexity of the method.

In this subsection we will first explore the theory behind the method, including only the minimum necessary to understand it, followed by details in how the method is implemented at a high level.

Virtual element theory We consider a linear elastic body like the one seen in Figure 2.12; since we are bounded to two dimensional problems, we discretize the domain Ω into a set of disjoint non-overlapping polygons F, each with area $|f|$ and centroid $\bar{x}$. Each element $f \in F$ has n_f vertices $x_1, ..., x_{n_f}$; note that we do not require f to be convex or have any other specific property. Let us denote by $\mathcal{E}_\mathrm{i}$ the set of edges that include x_i, and by $\mathcal{E}_\mathrm{i}^c$ the remaining edges.

Given our partition of Ω, we define the space $\mathcal{V}_h$, which consists of continuous displacement fields which restriction to the element F belong to the space $\mathcal{W}(F)$ of smooth functions; the space $\mathcal{W}(F)$ contains the deformation states that can be represented by the element F, presented as rigid body motions and constant states of strain that together can represent any state of the body.

For virtual element method, we will construct $\mathcal{W}(F)$ using the canonical basis $\boldsymbol{\varphi}_1, ..., \boldsymbol{\varphi}_{2n}$ of the form

$$\boldsymbol{\varphi}_{2\mathrm{i}-1} = [\varphi_\mathrm{i}, 0]^T, \quad \boldsymbol{\varphi}_{2\mathrm{i}} = [0, \varphi_\mathrm{i}]^T \tag{2.25}$$

where $\varphi_1, ..., \varphi_{2n}$ constitutes a set of barycentric coordinates; examples of barycentric coordinates can be found aplenty in related literature[49][18]. Barycentric coordinates satisfy three important properties:

1. They satisfy the Kronecker-delta property, meaning $\varphi_\mathrm{i}(x_j) = \delta_{\mathrm{i}j}$ where $\delta_{\mathrm{i}j}$ is the Kronecker delta, defined as

$$\delta_{\mathrm{i}j} = \begin{cases} 0 & if \quad \mathrm{i} \neq j \\ 1 & if \quad \mathrm{i} = j \end{cases} \tag{2.26}$$

2. $\boldsymbol{\varphi}_{\mathrm{i}}$ varies linearly along the edges of F and vanishes on $\mathcal{E}_{\mathrm{i}}^{c}$, i.e., the edges that do not contain the associated vertex x_{i}.
3. The variation of φ_{i} on $\mathcal{E}_{\mathrm{i}}$ is determined uniquely by the geometry of those faces, and independent of the shape of the element.
4. Barycentric coordinates can interpolate linear fields exactly.

Stiffness matrix and load vector We explored previously how the stiffness matrix and load vector are computed in finite elements so we It is important to comment that one appeal of this new method is that the stiffness matrix can be analytically obtained, as will be explored in this section; the load vector still involves integration most of the time, since it depends on the external loads applied to the domain, outside the method's control.

The elemental stiffness matrix of an element E, denoted $\boldsymbol{K}_E$ in Virtual Element Method, is obtained through projection of the deformation states over the nodal values of E; the deformation states refer to the analogue to the stress states shown in Figure 2.11. This projection defines two projection matrices $\boldsymbol{P}_R$ and $\boldsymbol{P}_C$ that represent, respectively, rigid body motion and constant states of strain, that need to be calculated; the total projection $\boldsymbol{P}_P$, representing the whole change in the domain, is calculated adding both matrices. Through mathematical manipulation, both matrices can be decomposed in two parts $\boldsymbol{N}$ and $\boldsymbol{W}$, that are easily computed using geometrical information. The two projection matrices can be calculated as

$$\begin{aligned}\boldsymbol{P}_R &= \boldsymbol{N}_R\boldsymbol{W}_R^T\\ \boldsymbol{P}_C &= \boldsymbol{N}_C\boldsymbol{W}_C^T\end{aligned} \tag{2.27}$$

Using the matrices defined in Equation (2.27), the stiffness matrix can be obtained using Equation (2.28). The only two terms not described until now are $\boldsymbol{D}$ and $\boldsymbol{S}_E$: the first term is the stress/strain matrix, also known as material matrix, previously presented as $\boldsymbol{C}$; on the other hand, the is second more complex and will be explored in detail next.

$$\boldsymbol{K}_E = |E|\boldsymbol{W}_C\boldsymbol{D}\boldsymbol{W}_C^T + (\boldsymbol{I}-\boldsymbol{P}_P)^T\boldsymbol{S}_E(\boldsymbol{I}-\boldsymbol{P}_P) \tag{2.28}$$

Hence, we only need explicit formulas for all matrices, which deduction is, again, part of the theory of the method that we will not concern ourselves with. Both $\boldsymbol{N}$ and $\boldsymbol{W}$ matrices are of size dependent on the number of vertices of the element, as each vertex contributes with a block of size 2×3 (when working in two dimensions) that together form the whole matrix. The matrices $\boldsymbol{N}_R$ and $\boldsymbol{N}_C$ are given in Equation (2.29), for a vertex of local index i. We denote the first component of a given vector $\boldsymbol{x}$ as $\boldsymbol{x}_{(1)}$, while its second component as $\boldsymbol{x}_{(2)}$.

$$\begin{aligned}N_R &= \begin{bmatrix}1 & 0 & (\boldsymbol{x}_{\mathrm{i}}-\bar{\boldsymbol{x}})_{(2)}\\ 0 & 1 & -(\boldsymbol{x}_{\mathrm{i}}-\bar{\boldsymbol{x}})_{(1)}\end{bmatrix}\\ N_C &= \begin{bmatrix}(\boldsymbol{x}_{\mathrm{i}}-\bar{\boldsymbol{x}})_{(1)} & 0 & (\boldsymbol{x}_{\mathrm{i}}-\bar{\boldsymbol{x}})_{(2)}\\ 0 & (\boldsymbol{x}_{\mathrm{i}}-\bar{\boldsymbol{x}})_{(2)} & (\boldsymbol{x}_{\mathrm{i}}-\bar{\boldsymbol{x}})_{(1)}\end{bmatrix}\end{aligned} \tag{2.29}$$

On the other hand, the matrices $\boldsymbol{W}$ need an extra definition to simplify its presentation. For every vertex i we define the vector $\boldsymbol{q}_\mathrm{i}$ as

$$\boldsymbol{q}_\mathbf{i} = \frac{1}{2|E|}\int_{\partial E} \varphi_\mathrm{i}\hat{\boldsymbol{n}}\mathrm{d}\boldsymbol{s} \tag{2.30}$$

Since we do not desire to numerically calculate any integral, we need to use the properties of the barycentric coordinates .First, φ_i have zero values in $\mathcal{F}_\mathrm{i}^c$, so can write

$$\boldsymbol{q}_\mathrm{i} = \frac{1}{2|E|}\sum_{F\in\mathcal{F}_\mathrm{i}}(\int_F \varphi_\mathrm{i}\mathrm{d}s)\hat{\boldsymbol{n}}_{F,E} \tag{2.31}$$

Where $\hat{n}_{F,E}$ is the unitary norm of the face F pointing outwards with respect to element E.

Also, using the Kronecker-delta property and the fact that this work is limited to two dimensions, this integral can be computed exactly, shown in Equation (2.32). Using this value, one can easily compute the vector $\boldsymbol{q}_\mathbf{i}$ without any integration error.

$$\int_F \varphi_\mathrm{i}\mathrm{d}s = \frac{|f|}{2} \tag{2.32}$$

With $\boldsymbol{q}_\mathrm{i}$ at hand, the matrices $\boldsymbol{W}_R$ and $\boldsymbol{W}_C$ are defined directly, as

$$\boldsymbol{W}_R = \begin{bmatrix} \frac{1}{n} & 0 & (\boldsymbol{q}_\mathrm{i})_{(2)} \\ 0 & \frac{1}{n} & -(\boldsymbol{q}_\mathrm{i})_{(1)} \end{bmatrix} \quad \boldsymbol{W}_C = \begin{bmatrix} 2(\boldsymbol{q}_\mathrm{i})_{(1)} & 0 & (\boldsymbol{q}_\mathrm{i})_{(2)} \\ 0 & 2(\boldsymbol{q}_\mathrm{i})_{(2)} & (\boldsymbol{q}_\mathrm{i})_{(1)} \end{bmatrix} \tag{2.33}$$

Finally, the last term of Equation (2.28) left is $\boldsymbol{S}_E$, related to the stability term of the equation. There are different approaches for its computation depending on the referenced source ([1], [33], [20]); we choose to work with Gain et al.[1] to be consistent with the previous terms. This matrix is defined in Equation (2.34), where $\boldsymbol{I}$ is the identity matrix of adequate size to fit in Equation (2.28).

$$\boldsymbol{S}_E = \alpha^E\boldsymbol{I} \tag{2.34}$$

The coefficient α^E represents a scaling of linear modes to higher-order ones and can be approximated as

$$\alpha^E = \gamma\alpha_*^E \tag{2.35}$$

where

$$\alpha_*^E = \frac{|E|trace(\boldsymbol{D})}{trace(\boldsymbol{N}_C^T\boldsymbol{N}_C)} \tag{2.36}$$

The coefficient γ is included in [1] to study the influence of α^E in the numerical solutions; it is independent of E and is generally left as one, but may change depending of the experimentation.

2.2.5 Error estimates

To actually validate the quality of the numerical results obtained, it is necessary to define some measures to assess the accuracy of the method. It is important to note that, to actually be able to compare the results, most error measures require knowledge of the analytical solution of the problem to study; we will select a few such cases and use them to prove our implementation, as it is standard in the literature.

In the following subsections we present the three different methodologies that will be used as error estimates.

2.2.5.1 Comparing nodal values

The simplest way of measuring the error of the approximation is to simple compare the obtained value in each node with the analytical solution in the exact same position. This approach has the clear advantage that it does not require further processing to obtain, but is deficient showing that the method does converge, as the error is obtained on a single mesh, and cannot be mixed with other results; it is, however, more intuitive to understand than the following measures.

2.2.5.2 Energy norms

In numerical analysis, and specially in finite element, it is usual to study the convergence rate of a given method using its energy norm; said norms are used to study the error of a solution due to having to discretize the mesh instead of computing the solution, in other words, the error that appears because we settle for a discrete approach instead of the continuous solution.

In this thesis, we will consider two of the most commonly used energy norms, the L2 norm and the H1 norm. It is important to note that, although the norms have the same definition in both considered methods (VEM and FEM), as the shape functions in VEM are virtual, the computation of the norms is done differently; we first present the generic formula, which is used in FEM, and after each one, the form it takes in VEM.

L2 norm Given the analytic displacement function $\boldsymbol{u}$, and the approximated solution $\boldsymbol{u}^h$, the L2-norm is defined as:

$$\left\|\boldsymbol{u}-\boldsymbol{u}^h\right\|_{L2} = \left[\frac{\sum\int_E \left\|\boldsymbol{u}-\boldsymbol{u}^h\right\| \mathrm{d}E}{\sum\int_E \left\|\boldsymbol{u}\right\| \mathrm{d}E}\right]^{\frac{1}{2}} \tag{2.37}$$

where $\|\cdot\|$ is the norm of the vector.

In VEM, we use the nodal displacement values in combination with the nodal integration scheme presented in Section 2.2.3 to compute the L2 norm.

H1 norm The H1 norm, differently from the L2, requires knowledge about the strains and strains. Given analytical solutions of the strain and stress fields $\boldsymbol{\varepsilon}$ and $\boldsymbol{\sigma}$, respectively, and the approximated values, computed from $\boldsymbol{u}^h$, $\boldsymbol{\varepsilon}^h$ and $\boldsymbol{\sigma}^h$, the H1 norm is defined as:

$$\left\|\boldsymbol{u}-\boldsymbol{u}^h\right\|_{H1} = \left[\frac{\sum\int_E \left(\boldsymbol{\varepsilon}-\boldsymbol{\varepsilon}^h\right)\cdot\left(\boldsymbol{\sigma}-\boldsymbol{\sigma}^h\right)\mathrm{d}E}{\sum\int_E \boldsymbol{\varepsilon}\cdot\boldsymbol{\sigma}\mathrm{d}E}\right]^{\frac{1}{2}} \tag{2.38}$$

As the approximated strain and stress require the shape functions to be computed, there is no trivial approximate in VEM. Hence, there are several approaches to the computation of the H1 norm, some using shape functions from other methods, others using the definitions presented in the theory of Virtual Elements to derive a value similar to the strain and stress, and others approximating the norm using only nodal values; we choose the latest option as it is closest to the spirit of VEM. We use the alternative presented by Chi et al.[22], shown in Equation (2.39).

$$\left\|\boldsymbol{u}-\boldsymbol{u}^h\right\|_{H1} = \left[\sum_E h_E \sum_{\mathrm{e}\in E} \frac{\left[\mathbf{e}(\boldsymbol{X}_{v_1^{\mathrm{e}}})-\mathbf{e}(\boldsymbol{X}_{v_2^{\mathrm{e}}})\right]\cdot\left[\mathbf{e}(\boldsymbol{X}_{v_1^{\mathrm{e}}})-\mathbf{e}(\boldsymbol{X}_{v_2^{\mathrm{e}}})\right]}{|\mathrm{e}|}\right]^{\frac{1}{2}} \tag{2.39}$$

where $\mathbf{e}=\boldsymbol{u}-\boldsymbol{u}^{\boldsymbol{h}}$ is the displacement error, h^E is the diameter of element E, and v_1^{e} and v_2^{e} are the two endpoints of edge e.

Chapter 3

State of the art

This chapter summarizes the literature review done on techniques used to solve the problem of linear elasticity on polygonal meshes; we use this study to present the advantages and disadvantages of such methodologies, and explain why we decided to work with Virtual Element Method. We also include a review on the subject of mesh generation for those techniques, that were used as further confirmation to our decision of using Voronoi diagrams and Delaunay triangulations.

We also present reviews on several subjects that were investigated and used as inspiration of the solution finally implemented; among those subjects, we can name: adaptive meshes and generation of the Voronoi diagram.

3.1 Polygonal finite element methods

Polygonal finite elements is the name given to a family of techniques that use the same kind of discretization as the classic finite element method, but using a generalization that allows them to work on n-sided polygons; naturally, this methodology is known as Polygonal Finite Element (PFEM). The technique dates back to 1975[49], when Wachspress presented what he called a **rational** finite element basis; from then, there has been an important advancement in the area, with a good number of different coordinates for PFEM, each with its own set of advantages and disadvantages. It is important to note that, although the method dates from the 70's, it did not see use in finite element simulations until recently, which explains why the development is not as advanced as one could expect.

Most barycentric coordinates, such as the one presented by Wachspress[49], require the polygons to be convex; as we want to work on general polygons without imposing conditions on the convexity, these functions are not an option. On the other side, coordinates such as Mean-Value[18] can handle non convex polygons, even with collinear vertices. In the following subsections, we will present a small review of the most common barycentric coordinates used in the PFEM.

Wachpress coordinates

The rational shape functions, also known Wachpress functions, were developed by Wachpress[49]. They can be defined by the formula:

$$\begin{aligned}\varphi_i(X) &= \frac{w_i(x)}{\sum_{j=1}^{n} w_j(x)} \\ w_i(x) &= \frac{A(v_{i-1}, v_i, v_{i+1})}{A(x, v_{i-1}, v_i)A(x, v_i, v_{i+1})}\end{aligned} \tag{3.1}$$

where $A(x_1, x_2, x_3)$ denotes the signed area of the triangle with vertices x_1,x_2,x_3, computed as seen in the geometry background. The graphical definition of the Wachpress coordinates can be seen in Figure 3.1. It is important to note that the Wachpress coordinates are limited to convex polygons, as the previous formulas (shown in Equation (3.1)) require the signed areas of the triangles to be positive, which is only the case when the polygon is convex.

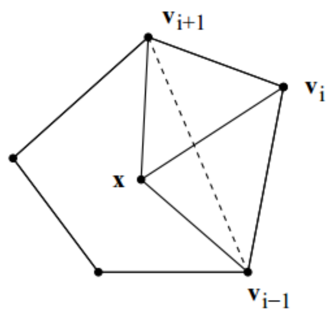


Figure 3.1: *Definition of the Wachpress coordinates[49]*

Mean-Value coordinates

As we saw in the previous subsection, Wachpress coordinates are relatively simple functions that lead to well-behaved coordinates; they are, however, limited to convex polygons. An alternative set of coordinates for convex polygons is the mean-value coordinates, which have a simple generalization to non-convex polygons.

For an n-sided polygon E with i-th vertex located at X_i, its Mean-Value coordinate for vertex i is defined as:

$$\varphi_i(X) = \frac{w_i(X)}{\sum_{j=1}^{n} w_j(X)} \tag{3.2}$$

with w_i given by

$$w_i(X) = \frac{\tan\left[\frac{\beta_{i-1}(X)}{2}\right] + \tan\left[\frac{\beta_i(X)}{2}\right]}{\|X - X_i\|} \tag{3.3}$$

where the angle $\beta_i(X)$ is the angle defined in Figure 3.2.

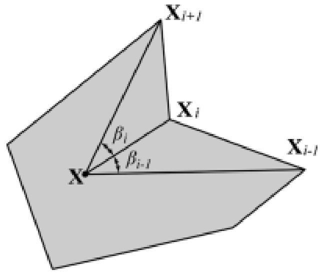


Figure 3.2: *Definition of angles for Mean-Value coordinates[23]*

Conclusion

PFEM has been used successfully to solve linear elasticity problems in polygonal meshes[23][43], but has a number of drawbacks that makes natural the search for alternatives: the integration of the basis functions, necessary as PFEM is a FEM variant, gives an intrinsic error as all barycentric coordinates are non-linear, and possibly highly irregular depending on the properties of the polygonal elements, so the method is never completely exact. Besides, the integration itself can be tricky as we can be dealing with general polygons, as, differently from triangles and quadrilaterals, there is no standard quadrature rule available. The literature proposed the use of triangulation or quadrangulation schemes, which subdivide each polygon of the mesh into triangles or quadrilaterals[16]; this procedure adds further cost to PFEM.

We conclude that, to obtain truly accurate results on polygonal meshes, we should investigate on other techniques to solver linear elasticity problems; we present next the investigation on Virtual Element Method, which, even if it is not exactly a finite element variant, we include in the same section as it could be thought as an alternative of sorts to PFEM.

3.1.1 Virtual Element Method

Although we commented on VEM theory and implementation in the background chapter, we choose to present a small review of the original, uses, advantages of the technique to give further context to its choice. Virtual Element Method can be seen, simultaneously, as extension of the Finite Element Method and a variety of Finite Volume method, with special ties to the Mimetic Finite Differences (MFD) methods; indeed, for many problems, the final system matrices are identical for VEM and MFD.

Given a boundary value problem to solve, for example the Poisson or elasticity equations, the Mimetic Finite Differences method schemes are constructed so that fundamental calculus relations (such as integration by parts, positivity of solutions, etc) involving the discrete versions of the operators divergence, gradient and curl on a grid are exactly met. MFD has been successfully used to solve different partial differential equations, such as electromagnetism, continuum mechanics, elasticity, among others.

Recently, MFD evolved into the virtual element method, which uses the same mimetic discretization, but now on a variational setting. The general idea of the Virtual Element Method is that it is much simple to assume that the degrees of freedom of the problem at hand (the unknowns in MFD) were, as it is the case with FEM, attached to trial or test functions inside the elements. At first sight, this idea puts VEM closer to methods such as PFEM or extended FEM, which attempt to generalize the standard finite elements on polygons; however, differently from VEM, this methods rely on adding particular shape functions to take care of the singularities of the exact solution, while VEM seeks to preserve the polynomial accuracy one has on triangles and quadrilaterals, but while working on polygons.

The core ideas behind virtual element method are simple, and can be summarized as follows:

- The trial and test functions, that here coincide, contain, on each element, all polynomials of degree $\leqslant k$, plus some other functions that are not necessarily polynomials.
- When computing the elemental stiffness matrix, special care is taken with the cases were one of the two entries of the bilinear form is a polynomial. The degrees of freedom are chosen so that one can compute the exact result using only the degrees of freedom that in general will not be a polynomial.
- It can be shown that for the remaining part of the bilinear form (when one non polynomial encounters another), it is only required to produce a result with right order of magnitude and stability properties.

One important point to establish is that, as a novel method, there is only one simple open source VEM implementation currently available[42]; this implementation solves a generic Poisson problem, similar to the one required for linear elasticity, but with several simplifications. Therefore, this work presents the first open source object oriented implementation of the virtual element method for linear elasticity.

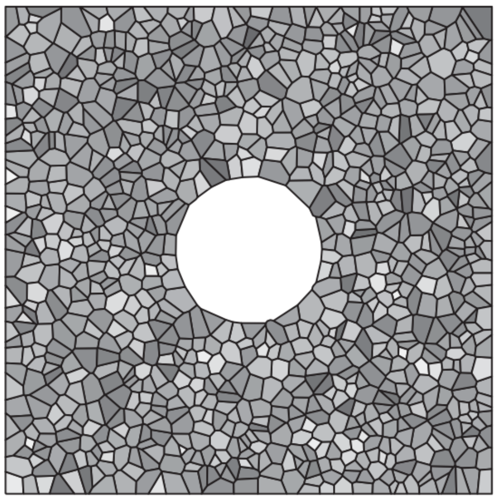
Regarding VEM applications, most importantly for this work, the use of virtual element method for linear elastic and inelastic solid mechanics problems has been proven successful by at least two authors; we follow the guidelines presented by Gain et al.[1], but da Veiga et al.[4] have also shown extremely good results in the area. Virtual element has been also applied to solve problems as varied as fluid dynamics[46] and discrete networks[31], on which its ability to handle general polygons allows obtaining of effective solutions.

3.1.2 Mesh generation for PFEM

As implied by the fact that FEM is the currently most common technique for linear elasticity problems, the meshes commonly used in this application are mostly triangular and quadrilateral. However, in combination with PFEM, there are some precedents on the use of polygonal meshes, especially in those cases the domains are complex enough to make a triangulation inconvenient; in this subsection we will explore the subject of the mesh generation techniques for polygonal finite element method, in particular, we do note that almost all application use the Voronoi diagram or a closely related structure.

First, Sukumar et al.[40] use the Voronoi diagram as a means of both discretizing the domain with polygons and to define a different type of barycentric coordinates, called the *natural neighbours coordinates.* Natural neighbours coordinates take advantage of the properties of the Voronoi diagram and its dual Delaunay to determine what are called the "natural neighbours" of a given point p; a node p_i is said to be a natural neighbour of p if the Voronoi cells that contains both points share at least one edge. From the natural neighbours, the coordinates are defined as a weighted function of the distance between the point p and all natural neighbours. Natural neighbour coordinates do present an advantage when compared with the standard barycentric coordinates, but are still non linear, so the intrinsic integration errors common to all polygonal finite element methods are still present; this way, its direct use is discarded.

Second, Talischi et al.[12] present the Voronoi diagram as an effective tool to generate irregular polygonal meshes; they value particularly the property of the method of generating randomness and higher levels of geometric isotropy as the byproduct of the ability of using arbitrary seed placement. However, as the authors are using polygonal finite elements, the meshes they use need to fulfill certain quality requirements, which is not a concern in our case; nevertheless, they do present a number of ideas that are deemed useful to our work. The authors impose quality on their generated meshes using two different tools: First, they use the centroidal Voronoi diagram (CVT) to obtain regular well-formed meshes, which, as commented in Section 2.1.2 is not an option in our case as this structure does not generate non convex polygons, which is one of the properties we require our meshes to have. Second, they use what they call "quasi-random" meshes, which are generated from arbitrary seeds, but imposing a minimum distance between the points; this measure prevents the appearance of small polygons, but still can produce distorted elements not suitable for finite element analysis. The differences between the three types of Voronoi meshes presented in Talischi et al's work can be seen in Figure 3.3 and Figure 3.4.


(a) Random mesh

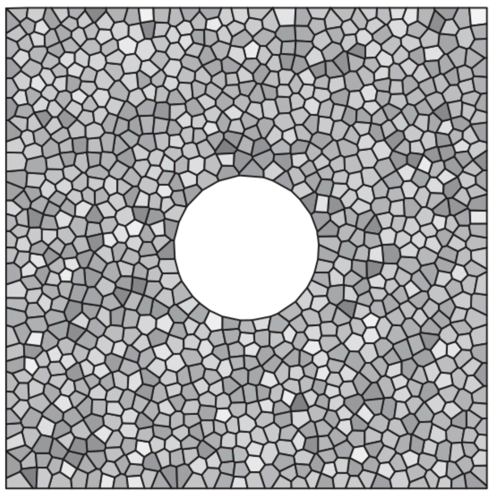

(b) Quasi-random mesh

Figure 3.3: *Random and quasi-random Voronoi meshes for PFEM*

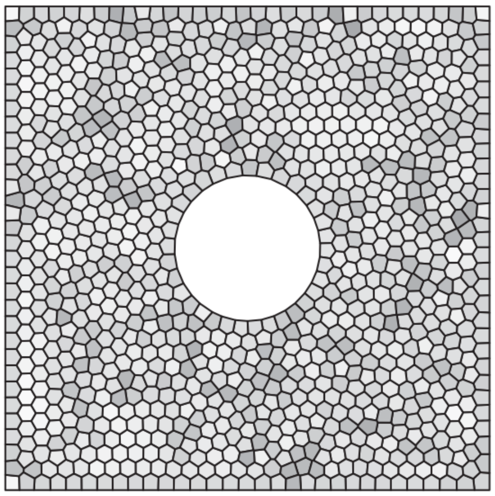


Figure 3.4: *Centroidal Voronoi for PFEM*

In this aspect, the work of Talischi et al. gives further support to our proposal of using Voronoi diagrams to generate arbitrary polygonal meshes; also, we do rescue the idea of generating quasi-random meshes by restricting the minimum distance between seed points.

Finally, there is additional precedence on the use of the centroidal Voronoi diagram for polygonal finite elements in open-source software such as PolyMesher[13], and in other related works. We remark on Sieger et al's[14] work, as they present an idea that, although applied to CVT, we think it could be used in standard Voronoi; they present a technique to optimize CVT meshes for polygonal finite element computations by removing all short edges of the diagram, which is a source of numerical instability for finite elements.

3.2 Adaptive meshes

We present a small summary about the concept of adaptive meshes, idea that was one of the main inspirations of this work. We first go through the history of the concept, its emergence, and advantages; we finalize with applications of adaptive meshes closer to the subject of this thesis.

In numerical analysis, adaptive meshes are commonly used to obtain solutions accurate to a certain degree of precision in sensitive regions of the domain, to deal with phenomena such as turbulence, discontinuities or shocks, dynamically adapting mesh size during the computation of the solution. The adaptive meshes paradigm is completely opposite to the classic one, which consists in creating a mesh, obtaining results, analyzing their quality, adjusting the mesh according to the outcome, and iterating until the desired solution precision is obtained; the problem with this is that the mesh is refined without taking in consideration that certain areas are simple to approximate, so they do not require further refinement, leaving a mesh that is unnecessarily fine in most areas, in which calculating can be prohibitively expensive.

Adaptive mesh have huge advantages in processing time, solution accuracy and memory usage when compared to classical methods, but are consistently more complicated to implement than static meshes, as step by step changes require geometry updates (a problem on its own, specially if using a complicated data structure), and mapping previous results to a new mesh, process particularly relevant in time-dependant problems.

The most classic algorithm for numerical analysis using adaptive meshes is the one presented by Berger and Oliger[8], that work recursively refining parts of a mesh made of rectangles, as seen in Figure 3.5; the mesh is represented as a tree, where neighbours are sub parts of the mesh that have the same level of refinement. Since this algorithm is restricted to rectangles and we work with general polygons, it is not completely relevant to our subject, but it is mentioned since it first presents the idea of computing using meshes that can change during computation as seen in Figure 3.5, an important point in this thesis.

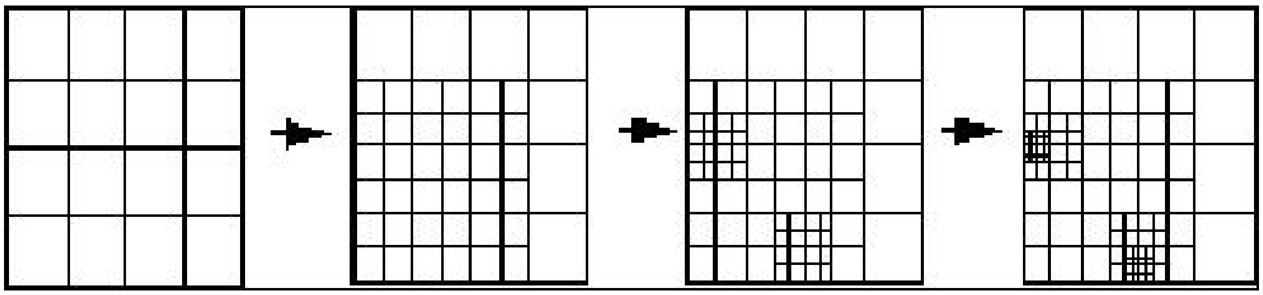


Figure 3.5: *Steps of adaptive mesh refinement*

Adaptive mesh simulations have seen intensive study in the industry, reaching the point of being considered a standard tool when simple simulations do not suffice; there are a number of robust open-source alternatives available[6][34] should one choose to use this approach. Finally, it is relevant to note that adaptive mesh refinement is a computational expensive process, which explains why the latest topic of study in the area is the implementation of the algorithm in parallel; as working in parallel is out of the scope of this thesis, we prefer to stick to the concept and do not investigate further in its use to solve the problem at hand.

3.3 Generation of the Voronoi diagram

In this subsection we will give a review of all study related to mesh generation and manipulation, subject that can be considered the second half of this thesis. As we mentioned in the introduction, we decided to use Voronoi diagrams because of: the dual structure of the Voronoi diagram, the Delaunay triangulation, can be both used to generate the Voronoi diagram and to obtain good quality triangulations for the finite element simulations that will be used to compare the obtained results. As we commented in Section 2.1.2, the Voronoi diagram is the dual structure to the Delaunay triangulation; the second structure is finite, as it triangulates the convex hull of the input points, but the first one is, by definition, infinite. As numerical methods solve a given problem in a bounded domain, the Voronoi diagram per se is not useful; hence the need to compute a Voronoi diagram with boundaries. This subsection gives a summary on the investigated techniques to generate a Voronoi diagram with finite boundaries.

3.3.1 ShaPo - A framework for generating 2D Voronoi Meshes

ShaPo[26] is a cross platform C++ library developed by Kitware to generate 2D Voronoi tessellations; although it is open-source, it is currently in development and has not been released to the public. Moreover, it was not presented to the public until after the investigation step of this thesis was finished; this way, although ShaPo does exactly what it is required for this work, we decide to implement our own library. We do, however, take some important lessons from ShaPo, that we either include in our own library, or leave as future work. We now proceed to list the most relevant features that we use in our own library:

- ShaPo makes the difference between the clipped Voronoi diagram, in which the boundary points are not cell generators, and the constrained Voronoi diagram, in which they are. The first diagram keeps the Voronoi properties for all the points of the mesh, but it is more expensive to compute, while the second is simpler. As we use the Voronoi diagram for its flexibility in the generation of the mesh, and not because of its specific properties, we decide to use the clipped Voronoi diagram, but leave as future work the implementation of the constrained diagram.
- ShaPo provides a number of operations that allows the generation of extremely high quality meshes. For example, it can merge small edges (following the same idea presented by [14]), it can compute the centroidal Voronoi diagram, and it allows remeshing part of the existing mesh.
- ShaPo uses the winged-edge data structure, as it provides functionalities beyond computing Voronoi.

3.3.2 Computing the clipped Voronoi diagram

Considering that ShaPo was not available at the time of this thesis development, we investigated on techniques to compute the clipped Voronoi diagram. In this subsection, we review two different techniques for the computation of the clipped Voronoi diagram and present their advantages and disadvantages.

Computing from the Delaunay triangulation

This alternative relies on the theory behind Voronoi and Delaunay triangulations to compute the sites inside the given domain, and only closes the boundary cells. Algorithm to compute the standard Voronoi diagram that takes advantage of the Delaunay-Voronoi duality are well known and documented in the literature[50][3]. All algorithms work basically the same way: they all use the fact that the Voronoi cell associated to point p can be obtained by uniting the circumcenters of the triangles that contain p. Although the literature tends to leave boundary cells open, modifications to close them are direct, and only require small modifications to the algorithm.

There is, however, one issue that could appear that makes implementation more complex. In the case the boundary triangles have their longest edge on the boundary, or, equivalently, the angle opposite to the boundary is obtuse, the circumcenter falls outside of the boundary, which distortions the Voronoi diagram. Although there are techniques specific to solve this issue[25], checking this case make implementation more complex and prone to errors.

Working from the Voronoi diagram

The first, and most obvious, alternative to obtained the clipped Voronoi diagram is to generate the standard version and then clip all boundary cells. An efficient algorithm to compute the clipped Voronoi diagram was presented by Yan et al[15]. They approach the complex problem of clipping a given Voronoi diagram with an arbitrary input domain simply by choosing not to do it directly, but rather preprocess the diagram and detect those cells that intersect the domain; this way, the complex problem is limited to detecting those cells and clipping them. The algorithm is simple: it first computes the Delaunay triangulation from the Voronoi diagram, which it then uses to identify neighbour Voronoi sites; using this information, it starts traversing the diagram, marking the boundary edges; it finally uses a standard clipping algorithm (such as Vatti's[47] or Sutherland-Hodgman[41]) to compute the final cell.

This approach has the advantage of being more intuitive and flexible, as one could do a number of operations to the diagram while we search for the boundary cells, or change the way the boundary cells are clipped (for example, instead of just clipping, we could add additional points to fulfill a certain property). The disadvantages are not as obvious, and only appear when this technique is evaluated in comparison with the alternative shown in the previous subsection:

- The advantage of working directly with the Voronoi diagram, instead of first computing the Delaunay triangulation, disappears, as we still need the triangulation.
- The clipping algorithm is a costly one. This extra cost is specially important when one considers that it has no equivalent in the alternative method. Hence, it is expected that this approach is more costly than the alternative.

Chapter 4

Design

In this chapter we present the most relevant aspects of the high-level design of the implemented solution. We divide this chapter in four parts, the first dealing with the requirements and objectives sought with the development. Next, we present an analysis of the different alternatives considered, detailing advantages and disadvantages, explaining each of our choices. The third part includes the software architecture, including, but not limited to, how the software is divided, and how each component was designed. Finally, as we need some way of validating the results, the last part features the experiments designed.

4.1 Requirements

In this section we list the high-level requirements that the system should fulfill; these requirements were later used to analyze the possible solutions and determine the most appropriate one.

We define that the complete designed the system should:

- Solve the problem of linear elasticity on arbitrary polygonal meshes.
- Allow the following operations on polygonal meshes: inclusion of a segment ("breaking" the mesh), refinement of specific polygons and erasing points or polygons.
- Be modular, to allow the different components to be used for different ends.
- Be extensible, as this work is considered exploratory, so it is expected that it serves as a base for further development.
- Allow changes in case new tools need to be integrated into the work.
- Be easy to integrate with other software, as this work lacks, for example, a visualization scheme.
- Be as efficient in both processing time and memory usage as possible, prioritizing the first; we will consider that we have space for some redundancies in information if they are necessary to speed up some operations, but will try to minimize the space used.

- Consider the problem of machine precision and handle the associated errors that could appear.
- Work limited to two dimensions, without necessarily leaving open the possibility of extension to three.
- Work with polygonal meshes, which contain polygons of arbitrary shape that satisfy the constraints presented in Section 2.1.3.1.

4.2 Analysis of possible solutions

In this section we will explore the alternatives considered to solve the problem of linear elasticity on complex domains. We start by summarizing the problem this thesis desires to solve, and, from then, we present the analysis of the possible solutions. To simplify the design and development, we choose to divide the work in two separated parts: the generation of the initial mesh and the numerical analysis itself. The differentiation between geometric and numerical modeling is classic in almost all numerical simulation applications; moreover, the division will be self-explained as the alternatives are considered, as a separation of responsibilities naturally arises from the analysis presented next.

4.2.1 Problem definition

The problem this thesis presents is how to improve the accuracy and/or computation time of linear elasticity problems in complex modifiable meshes. Linear elasticity is commonly solved using FEM, which requires triangular or quadrilateral meshes; it is a satisfactory solution most of the time, but as meshes are modified or the domains get more complex, the harder it gets to preserve the quality that FEM requires. Most approaches simply increase the number of elements of the mesh to preserve quality and be able to adapt to the complex shapes; however, as the algorithms are dependent on the number of elements of the mesh, this solution is not scalable. Therefore, we propose using arbitrary polygons to mesh complex domains and easily adapt to whatever changes the users could include on the mesh. We justify this choice in the fact that arbitrary polygons are more flexible; hence, we do need to investigate on approaches to solve the linear elasticity problem on polygonal meshes, how to generate those polygonal meshes, and how to modify them.

4.2.2 Mesh generation

As we mentioned previously, the most common type of polygonal mesh used in linear elasticity are Voronoi diagrams; as they are well studied, relatively easy to implement (as the algorithms are well known) and have good properties, we decide to follow the standard and use Voronoi diagrams. We will study elasticity on a bounded domain, so the standard Voronoi diagram with infinite regions does not suffice; we require the clipped Voronoi diagram. We approach this problem directly at first, searching for external tools to generate the clipped Voronoi diagram, which, while now existent and open source[26], were not available at the time of the background investigation and development of this thesis. This way, as no tool for the generation of the clipped Voronoi was available during the design stage of this thesis, we were faced with the choice of how to compute it ourselves: we have the option of generating the standard Voronoi diagram and then clip it[15], or use the duality between Voronoi and Delaunay to derive it.

We study both alternatives: we first investigate tools for the generation of the Voronoi diagram that allow non-convex domains that can include holes; we study Qhull[7], but as it does not allow restricted segments on Voronoi or Delaunay diagrams, it was quickly discarded. Also, while simultaneously investigating clipping algorithms for Voronoi diagrams, it was noted that most of them made use of information from the associated Delaunay triangulation[15], hence, it is deemed more efficient to use the duality Voronoi-Delaunay directly.

Next, we search for libraries that generate the Delaunay triangulation and study how to use the duality to generate constrained Voronoi; as we desire to use the constrained Voronoi tessellation, we naturally investigate on external tools that help us generate the constrained Delaunay triangulation. We check on Triangle[37], an old but stable C library, that not only generates constrained Delaunay triangulation, but can compute conforming Delaunay (see Section 2.1.2). Conforming Delaunay has the important advantage that forces all triangles to be Delaunay, even those on the constrained segments, which ensures that all Voronoi points lie inside the domain. Without Voronoi points outside the domain, the constrained Voronoi diagram can be correctly constructed.

Finally, because of the ability to generate the conforming Delaunay triangulation from any PSLG, we decide to use Triangle to generate the Delaunay triangulation, and implement our own algorithm to use that tessellation to compute the constrained Voronoi diagram.

4.2.3 Linear elasticity

As seen in Chapter 3, the classic approach to linear elasticity on polygonal meshes is to use PFEM, with barycentric coordinates dependent on the characteristics of the mesh: for a mesh with only convex elements, Wachpress coordinates are recommended, while the mean-value coordinates are useful when non-convexities are present; in the case a Voronoi tessellation is used, one could consider natural neighbour coordinates. However, all types of barycentric coordinates present the same problem: they are inaccurate because of the intrinsic integration errors their non-linearity generates. Hence, PFEM is considered insufficient on their own for this thesis objective, as we seek to improve the accuracy of the state of the art.

We choose to investigate on alternatives to the Polygonal Finite Element Method, and find the Virtual Element Method, which, because of its mathematical formulation, does not have the intrinsic errors of PFEM. Therefore, we propose its use as an improvement in the accuracy of the solution of linear elasticity problems on polygonal meshes.

Finally, one important aspect of the problem we require to present now, as it affects the implementation, is how the results of VEM will be evaluated. As we have implied previously, we choose to compare to FEM on triangular meshes, as it is considered the standard solution to the problem; we use the same conforming Delaunay triangulation that is generated for the computation of the Voronoi diagram, both because it is readily available, and due to its good properties for FEA. Next, we need to decide whether FEM will be included through an external library or if we will implement it ourselves. We search for alternatives implementing the Finite Element Method for linear elasticity, and find alternatives aplenty[6][2][48]; we do, however, consider that most external libraries include optimizations that either speed up calculations or make it more accurate. This way, we decide that, to make comparisons as fair as possible, we decide to implement the Finite Element Method using the same tools and structure that we will use for Virtual Element Method; this way, we expect to make comparisons as fair as possible.

4.2.4 Modifiable meshes operations

We decide to separate the investigation on mesh generation and mesh modification as we consider that they are distinct problems, as a user could very well just require mesh generation for a numerical application and never use its modifiable functions. This way, including modifiable functions on the mesh generation library represents an important overhead to all applications that are concerned with only generation, and unnecessarily makes the library more complex. Instead, we choose to implement the modifiable functions as an extension to the mesh generation library, so to separate functionalities; moreover, we take advantage of our decision of using an object-oriented paradigm for easier extension, further justifying our previous choice.

After deciding to separate the modifiable functions from the generation of the mesh, we have two important points to work on: which functions will be included in the library, and investigate on algorithms and implementations of those functionalities. First, we decide to get inspiration on real life applications of the finite element method in fracture mechanics simulations; fracture mechanics, and crack growth, require an especially complex functionality of the meshes, the ability to include new segments on the mesh, breaking the elements the segment pierces through. Hence, we decide to include the possibility of breaking the mesh because of two reasons: first, it is considered interesting in the case one could desire to explore the possibility of using the Virtual Element Method for fracture mechanics; second, this kind of changes are particularly bad for classic Finite Element Method applications, as they require good quality meshes, so demonstrating that VEM works best in this case represents an advantage when compared to more standard methods.

Second, we take inspiration on adaptive meshes applications, which are known as being extremely useful in numerical simulations in complex domains. They are popular as they can give a higher degree of control over the mesh depending on the accuracy of the obtained results, without unnecessarily increasing the number of elements. This way, we consider adaptive related functionalities for our modifiable meshes, both because we see value in adaptive meshes as a solution to the problem of linear elasticity in complex meshes, and based on the fact that implementation of an easy to use library that can be adapted to this application is a valuable contribution to the scientific community. We decide to include two operations that can be easily used to implement an adaptive mesh application from our implementation: refining elements and coarsening the mesh. The first operation is more or less straightforward to understand

Finally, we have commented both in the introductory and background chapter how VEM can be used in meshes with non-convex elements, but we decide to prove this statement. Voronoi meshes can generate non-convex elements when the domain is non-convex, but, to be able to compare how VEM works in convex and non-convex meshes we require a way of transforming from one type of mesh to the other. We investigate on the subject and find several algorithms that partition non-convex polygons to convex ones, all with different characteristics, so we decide that it is easier, and more intuitive, to transform non-convex to convex and not the opposite. This way, we include a "fix" operation in our modifiable meshes, that is meant to represent all functionalities that change aspects that could be considered "wrong", such as the presence of non-convex elements. In particular, we implement as one fix operation the ability to take a mesh and return another, but without non-convex elements.

4.3 Proposed software architecture

In this section we will go through the design of the software architecture, that will be used to implement a solution to the problem we desire to solve. We will first analyze the principal software engineering choices that shape the solution, and then proceed to specify the design itself, starting with the external tools that will be used as support, finishing with the design of three libraries that will present the concrete implementation of the proposed methodology, including the process and classes design.

4.3.1 General architecture

In this subsection we will detail the points that led us to the choice of style of software design, including the justification of each one.

The first important point to note is that, as mentioned in the requirements, our solution needs to be easily integrated into external software; hence, we decide to develop our system as a library, so that any user can include it in their own projects. The choice of a library instead, for example, of a classic piece of software carries several implications:

- Most of the internal implementation is hidden from the user, which is usually the case with most numerical simulation software, since users tend to be more interested in results.
- We only give certain interfaces for input or output of information, which allows any user to abstract from the implementation, and just apply the obtained results in any desired application.

- Makes us carefully consider separation of responsibilities, so to minimize the duplication of code and allow to change complete parts of the implementation without the complete system failing.
- Implementing a library allows us to easily share our code in a modular fashion, encouraging its use in other applications.

Once established that the concrete implementation of the proposal will be done as a library, we now analyze how to better separate the implementation. At the start of this chapter, we noted that the problem can be naturally separated in two parts, one concerned the mesh generation, and the other with linear elasticity itself. We keep this previous separation, but also note that the problem of non-static meshes is separated from mesh generation, as we could get initial meshes from a different source, and still require them to be non-static.

This way, we design a system composed by three libraries: first, we have a mesh generation library that generates and models static polygonal meshes. Second, we create a VEM for linear elasticity library that contains the previous one, and uses it to both generate a mesh and to model the geometry aspects of the problem, and adds the computation of the displacements on a static polygonal mesh. Third, we develop a library to manipulate and modify meshes; this last library can be considered an extension to the first one, although it can be used on its own, as it contains it.

Using the previous division in three libraries, we now will detail how each one was designed, including: their exact responsibilities, their input and output interfaces, and how their most relevant algorithms work at a high level. The concrete implementation will be detailed in the next chapter.

4.3.2 General implementation choices

In this subsection we will go through several important details that concern all the three libraries. We comment on: the programming language and paradigm that will be used, and some external tools (third party libraries) that will help us in the implementation.

4.3.2.1 Programming language

As we commented in the analysis of the possible solutions, for the generation of the Voronoi diagram we decide to use Triangle, a C library, which can either be compiled and called using text files for input and output, or included in the code directly. The first approach has the advantage that it can be used from basically all programming languages; while the second is restricted to those language that can call C code directly. The disadvantages are more decisive, as the first procedure requires reading and writing from disk, operations that are known to be several orders of magnitude costlier that any operation in principal memory. This way, we decide to call Triangle directly as a C library, which restricts our choice of programming language.

A second important point is that, as mentioned in the requirements, we desire the system to be modular and extensible. Hence, we decide to discard the use of C directly, as a procedural language makes it harder to divide responsibilities cleanly. However, as we want to be as efficient in processing time as possible, we also decide not to stray too far from C, known to be among the more efficient languages, and choose C++, which can call C directly, but has object-oriented and generic programming features that help code be more modular.

4.3.2.2 External tools

Having decided to implement the libraries in C++, we start investigating for tools that can help us with aspects of the implementation that are outside the scope of this thesis work. The most important external tool is, as mentioned before, Triangle, which choice was already justified, so it will not be repeated.

Apart from Delaunay triangulation generation, there are several aspects of the work that, while complex on their own, are incidental to our objectives. We detect three such instances: first, VEM requires some linear algebra tools, particularly the resolution of a linear system; second, the generation of the domains that will be meshed could require computing the intersection of arbitrary polygons when the user decides to include holes; finally, we work on non-convex meshes, but choose to implement a tool that allows any user to eliminate all non-convex polygons in the mesh, through the use of the convex partitioning.

Concerning the first point, the implementation of linear algebra operations is not straightforward at all, and, incorrectly done, can greatly increase their cost of and be prone to errors, therefore justifying the choice of using and external library. Among available implementations, and prioritizing those implemented in C++ because of efficiency and the ease of inclusion in a project coded in the same language, we choose Eigen[5], a template library for linear algebra, known to be efficient, well documented and with an active community. Eigen provides operation on vectors and matrices, including addition, subtraction, multiplication and matrix inversion, all of which will be intensively used; furthermore, Eigen's interface is quite similar to MATLAB, a programming language commonly used in FEA and famous for its linear algebra module. In our case, we will use both the operations and some of the structures provided by Eigen in a good part of the numerical implementation, so from this point forward we will consider the linear algebra operations known and will not remark on them.

On the other hand, regarding the second point, the problem of intersection of polygons is a complex one, and prone to errors, as it has many border cases. Hence, we prefer to use an already available solution in the form of Clipper Library[27], an open source, efficient and well documented library for C++ that uses Vatti's algorithm[47] for intersection, union, difference and exclusive-or of general polygons. We choose this library because it is lightweight when compared with other options such as Boost[9] and CGAL[17], and easy to integrate in our project, as it is a header library. The only drawback of our decision is that Clipper works with integer coordinates, which is an important difficulty, since our project works with floating point precision; moreover, scaling up to integer point can only be done when the number of decimals is small (as they grow this method is impractical or downright impossible), which explains why we use Clipper for polygon intersection when the user adds holes to the problem domain, but for further uses, such as union for fracture growth, is preferred to implement a custom algorithm specific to our case.

Finally, regarding the third point, we decide that polygon partitioning, being a complex problem of its own, can be better approached using an external tool. We investigate alternatives, including again big libraries such as Boost and CGAL, which we discard because of their size, and some lighter ones; in particular, we decide to include PolyPartition[19], because it is specialized on the problem, lightweight, and presents several algorithms. The last point is particularly relevant as, depending on the obtained results, we could require the use of different ways of generating convex polygons from non convex ones.

4.3.3 Libraries design

In this section we will go through the general design of each of our libraries, making their respective responsibilities and input and output information clear, so that we can easily show the more detailed design in the next sections.

As mentioned in the general design section, our proposed solution consists in a series of three libraries that separate responsibilities: the generation of polygonal meshes, changing these same meshes, and solving the linear elasticity problem.

Finally, although we decide to implement libraries, meaning, a self-contained piece of software which input, and output are objects that can be further used, for various purposes all our libraries have procedures to print their outputs in text files; this functionality is mostly used in the debugging of the project, as it allows for an easier visualization, and is it decided to leave as a feature of the libraries.

Voronoi generation library

Although this library is named for its main responsibility, the generation of the initial mesh, it is also in charge of the following tasks:

- Give all necessary elements to model a polygonal mesh, such as points, segments, polygons, etc.
- Obtain polygon measures such as area, diameter and centroid.
- Allow the input of a domain, including holes, and process both when necessary.
- Allow both the input of seed points or the choice of generating rules, as starting points of the Delaunay triangulation.
- Generate the clipped Voronoi diagram from the conforming Delaunay triangulation.
- Allow the computation of the constrained Delaunay triangulation.

Considering the responsibilities mentioned before, we can list the input information of this library:

- The domain on which to generate the Voronoi diagram,
- any number of holes, either completely inside or intersecting the boundary of the domain,
- either a list of seed points generated externally by the user,
- or two pairs of generating function and number of points desired, one for each axis in 2D.

and its output:

- The conforming Delaunay triangulation of the seed points,
- the clipped Voronoi diagram,
- the constrained Delaunay triangulation, all inside the given domain.

Virtual Element Method library

Differently from the previous library, this one is extremely self-explanatory, as its main responsibility is the computation of the displacements using the Virtual Element Method; however, to that end, there are several other secondary responsibilities that this library needs to fulfill:

- Allow the input of conditions of a linear elasticity problem: essential and natural boundary conditions, material and body forces.
- Receive an external polygonal mesh, or use the previous Voronoi library to generate it. Either way, it uses the Voronoi library to model the polygonal mesh.
- Create the linear system representing the elasticity problem defined by the user, creating an element for each polygon of the mesh.
- Compute the displacements on the system when the loads and constraints are applied, using the Virtual Element Method.
- Receive an external triangulation and compute the displacements when loads and constraints are applied, using the Finite Element Method.

Again, we list the input of the library considering the tasks it oversees:

- A polygonal mesh or a triangulation, defined using the models of the Voronoi library,
- a list of essential and natural constraints,
- a body force, if necessary to model the elasticity problem,
- a material to assign to the domain.

and its output:

- A vector with the displacements of each point of the mesh.

Modifiable meshes library

This final library oversees the algorithms that modify the meshes. It can be seen as an extension to the polygonal mesh generation, and is implemented as such:

- Receive either a polygonal or triangular mesh and initialize the data structures required.
- Receive a series of configuration parameters that decide how the operations must be done.
- Allow the inclusion of a segment, or a series of segments, on the mesh, and process the broken polygons as required. From this point, we call this process "breaking" the mesh.
- Allow meshes with non-convex elements, and include an operation to partition those elements if required, so to obtain a mesh with only convex elements.
- Allow the refinement of a given mesh element.
- Allow merging two or several mesh elements.

Next, we list the input of the library:

- A polygonal mesh, following the conventions of the Voronoi library,
- or a triangulation, following the same conventions.

and the output:

- The same mesh, but modified as desired by the user.

4.3.4 Process design

After detailing the responsibilities of each library, we design the processes required to fulfill those tasks. In the following subsections, we will present the most relevant processes of this library, among which we can mention: the generation of the Voronoi diagram, the initialization of the numerical system (that represents the problem of linear elasticity), the computation of the displacements through VEM and FEM, and the two most complex modifiable mesh operations: breaking the mesh, and merging elements.

Voronoi generation library

We now present in Figure 4.1 the details of the central algorithm to the first library, starting with the input of the domain to the generation of the polygonal mesh.

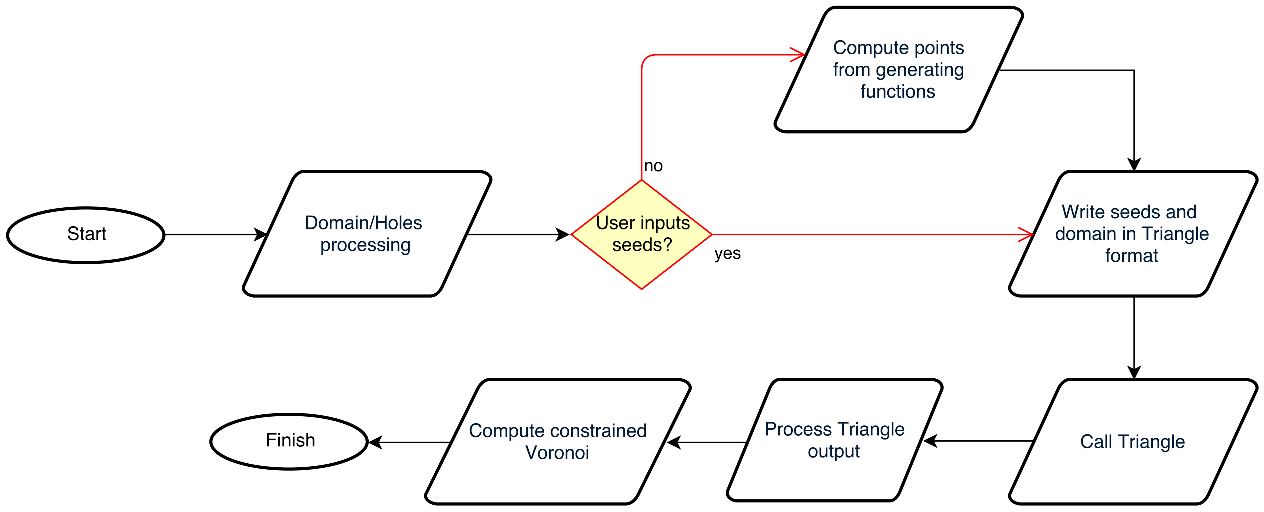

Start
Domain/Holes processing
User inputs seeds?
no
yes
Compute points from generating functions
Write seeds and domain in Triangle format
Call Triangle
Process Triangle output
Compute constrained Voronoi
Finish

Figure 4.1: *Flow diagram of Voronoi diagram generation*

Virtual Element Method library

In this subsection we present the two processes related to the VEM library, which correspond to the Initialization of the numerical system and the computation of the displacements, shown, respectively, in Figure 4.2 and Figure 4.3.

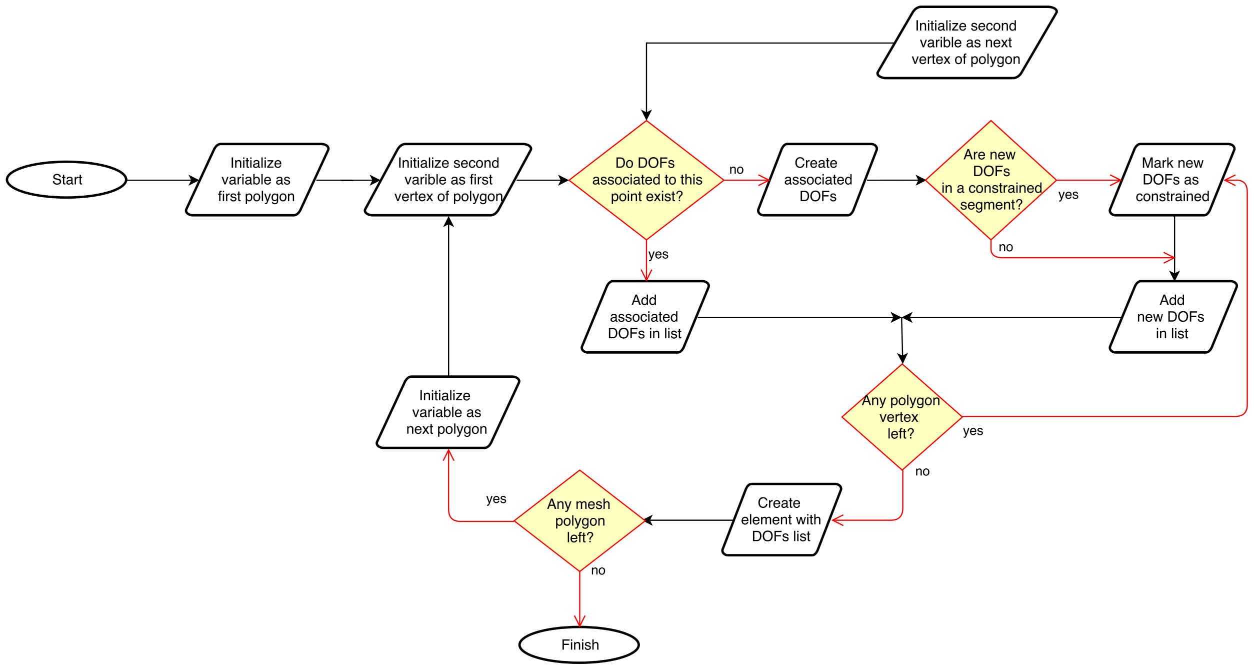

Start
Initialize variable as first polygon
Initialize second varible as first vertex of polygon
Do DOFs associated to this point exist?
no
yes
Create associated DOFs
Are new DOFs in a constrained segment?
yes
no
Mark new DOFs as constrained
Add new DOFs in list
Add associated DOFs in list
Initialize second varible as next vertex of polygon
Any polygon vertex left?
yes
no
Create element with DOFs list
Any mesh polygon left?
yes
no
Initialize variable as next polygon
Finish

Figure 4.2: *Flow diagram showing the initialization of the numerical system*

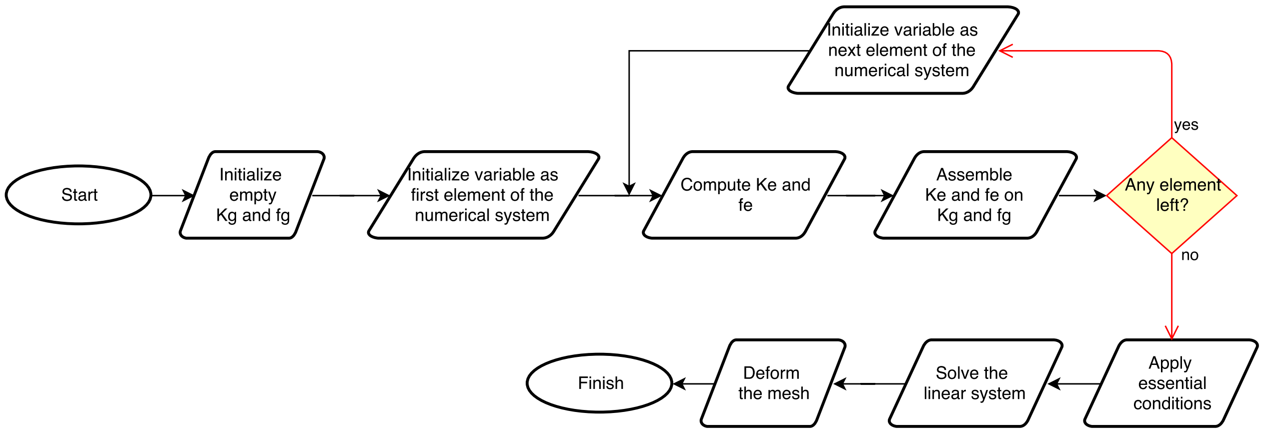

Start
Initialize empty Kg and fg
Initialize variable as first element of the numerical system
Compute Ke and fe
Assemble Ke and fe on Kg and fg
Any element left?
yes
Initialize variable as next element of the numerical system
no
Apply essential conditions
Solve the linear system
Deform the mesh
Finish

Figure 4.3: *Flow diagram of the computation of the displacements*

Modifiable meshes library

Finally, in this subsection we present the diagrams that present the two functionalities of the modifiable meshes library that are non-trivial to explain: breaking the mesh, and merging elements, which can be seen, respectively, in Figure 4.4 and Figure 4.5.

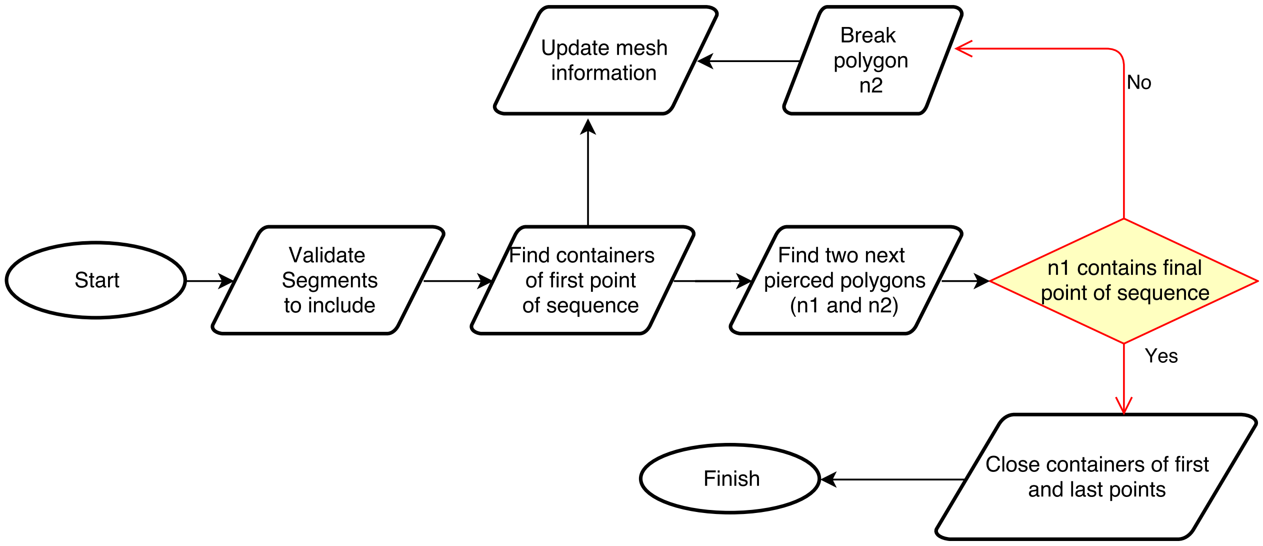

Start
Validate Segments to include
Find containers of first point of sequence
Find two next pierced polygons (n1 and n2)
n1 contains final point of sequence
No
Break polygon n2
Update mesh information
Yes
Close containers of first and last points
Finish

Figure 4.4: *Flow diagram of the breaking process*

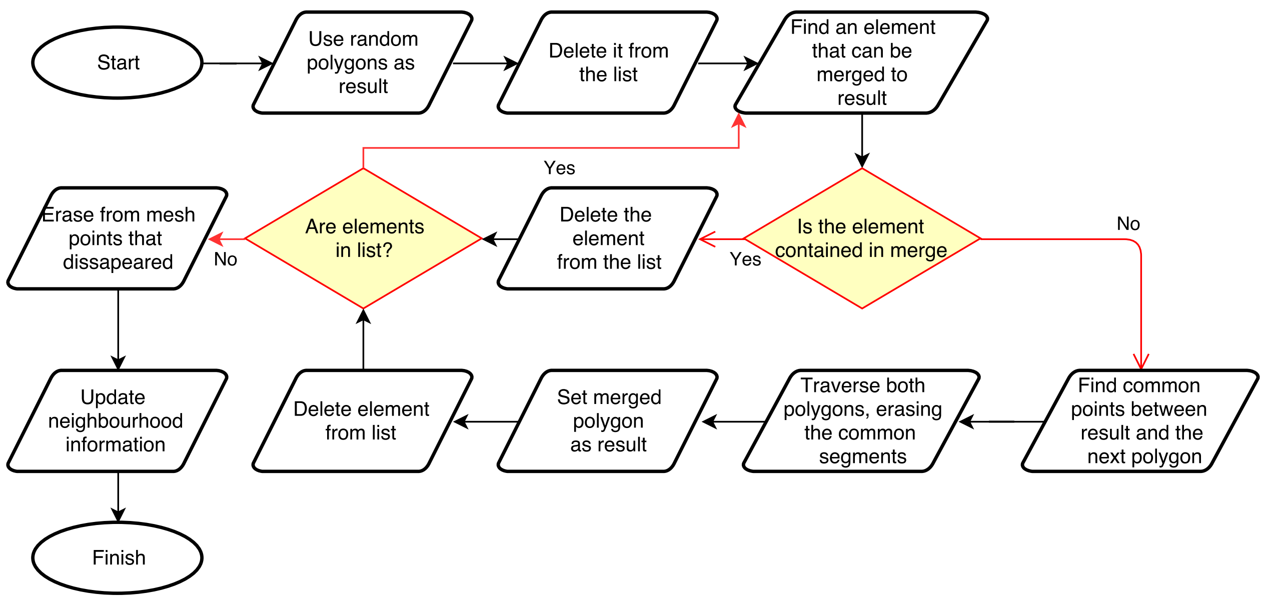

Start
Use random polygons as result
Delete it from the list
Find an element that can be merged to result
Is the element contained in merge
Yes
No
Delete the element from the list
Are elements in list?
Yes
No
Erase from mesh points that dissapeared
Update neighbourhood information
Finish
Find common points between result and the next polygon
Traverse both polygons, erasing the common segments
Set merged polygon as result
Delete element from list

Figure 4.5: *Flow diagram of the merging process*

4.3.5 Model design

Once we have defined all the important procedures that we will later implement, we can define how we will model the software, which is the main aim of this subsection; as it has been seen throughout the complete chapter, we will use the division of the three libraries for easier presentation of the concepts.

As we mentioned at the start of the design chapter, we require modularity and extensibility, therefore, we decide to implement our libraries using an object-oriented paradigm; this idea also allows us to use a UML class diagram as the main summary of our models and their relationships. This way, in the next three sections we will first present the UML diagram showing the structure of each library, followed by an explanation of the most relevant classes.

4.3.5.1 Voronoi generation library

In this subsection we will detail the model design of the Voronoi generation library, which, as mentioned before, is also in charge of giving a modeling base to represent geometric concepts necessary to model the fracture simulation problem; therefore, this subsection will first present the data structures chosen to represent the elements of a polygonal mesh, and, only with those ideas clear, we will show the UML class diagram with the concrete model design, ending with an explanation of the classes.

Data structure design There are many ways a mesh can be represented, depending on several factors, such as the restrictions in memory, complexity, operations desired, among many others. Our choice in this thesis is not any of the structures mentioned in Section 2.1.3, but a mix of several that fit our specific requirements (Section 4.1), which can be summarized as follows:

1. Points are represented as a pair of double precision numbers and a Boolean flag to indicate whether the point is in the boundary of the domain.
2. Polygons are a list of the indexes of the points that define them, in counterclockwise order. We decide to use indexes because of two reasons:
   - When meshes grow, using integers (32 bits) instead of pointers (64 bits in most machines nowadays) saves potentially fundamental memory space.
   - Since we must deal with precision errors, having access to a way of exactly comparing polygon vertices greatly simplifies several computations, including equality.
3. Any edge of the mesh, named segment, is represented using the indexes of its vertices; it is important to note that a segment has no concept of "inside" or "outside", so the segments (a, b) and (b, a) are equal.
4. Meshes are a collection of points and polygons, both kept in lists. As polygons depend on the indexes of the points, sorting, deleting or shuffling the list of points is not allowed; valid operations on the point list are searching and adding new points to the tail of the list. The polygon list has the same constraint, although deleting and merging are supported through clever manipulation of the indexes; in this case, this restriction is born from the chosen way of storing neighbourhood relationships.
5. Neighbourhood information can be represented in several ways; in our case, following the spirit of the Winged Edge structure, we choose two structures: the first we call a SegmentMap, and relates each segment of the mesh to the polygons that share it (without any order whatsoever); the second is the PointMap, which, as implied by its name, relates each point of the mesh to the polygons that contain it. Our SegmentMap is used so that searching for the neighbour of a polygon is constant time, although more operations are needed to keep the structure up to date when compared with the alternative of keeping the neighbours on the polygon itself. This choice is justified in the fact that segment information can be easily used to traverse the mesh in a given direction, an extremely important part of the classic algorithms to find the container of a point in the mesh; also, defining segment equality as order independent allows searching for the information related to both polygons, no matter how they consider the segment to be ordered. On the other side, our PointMap is mostly included to handle the great number of border cases that appear when adding extra points or traversing the mesh; in the cases one includes a point that coincides with a vertex, or, while advancing through the mesh, one finds a vertex, neighbourhood information related to the segments is completely useless, so, instead of trying to operate with non-optimal information, we choose to sacrifice some space and keep a PointMap, so that most algorithms are simpler and less prone to errors.

UML class diagram The UML class diagram with the most relevant classes of the Voronoi generation library (including their class members and method) is shown in Figure 4.6.

Classes details In the first place, Polygon is the class that models a polygon of the mesh; its parameters are its list of point indexes, diameter, centroid and area, the later three are kept at hand because they are small in space, but expensive to compute. Triangle is a subclass of Polygon that has only three points in its list, and is able to compute its circumcenter, which it keeps at hand since it is used several times in the Delaunay to Voronoi algorithm.

As a mesh is bounded to a domain, we need a class to model them, which we name Region, and define as a list of points (instead of indexes as mesh polygons). However, since we choose to define a Region using a single list of points, we need an extra way of including holes inside it. We choose to add a list of holes to Region, where the holes are represented a separate class. Hole is an interface class that contains the list of the points that define its shape, with two virtual functions: *getCenter* and *getSegments*. We include two concrete Hole implementations: PolygonalHole is a Hole that also inherits from Polygon, so it has all the methods of a Polygon, including getCentroid (that does all the work of getCenter) and an implementation of getSegment, that is used directly; and CircularHole, that inherits from Circle, using it for the implementation of the two mandatory methods. Circle is a small class in charge of modeling circles, which are continuous, in a discrete manner; it is contains the center and the radius of the circle, and discretizes the circle, creating points to represent its boundary.

Segments are an important concept in our library, but considering that we have two different ways of representing them (using their points directly or through their indexes), it should come as no surprise that we implement three classes to model them: Segment is an abstract template class that contains methods common to both kinds of segments; IndexSegment, a segment when referred by the indexes, is a concrete class that inherits from Segment<int>; while PointSegment, a segment containing the exact value of its end points, inherits from Segment<Point>.

SegmentMap is the structure that contains all neighbourhood information; it is a hash map with IndexSegment as key and a class called NeighboursBySegment as value. NeighboursBySegment is a container class that keeps two integers that represent the two polygons that are incident to a segment. A SegmentMap has several methods to add information and retrieve it, but it is also concerned with replacing and deleting neighbourhood information. In the exact same manner, PointMap is a map with Point as a key and a class named NeighboursByPoint as value; it is important to note that using the points directly as key has implications on precision tolerance that will be commented in the implementation chapters. The operations of the PointMap are mostly equal to those of SegmentMap, with the difference that the number of neighbours by point is not bounded (as it is the case with neighbours by segment, which, in most cases, are two), so PointMap requires that most of its methods work with lists of elements.

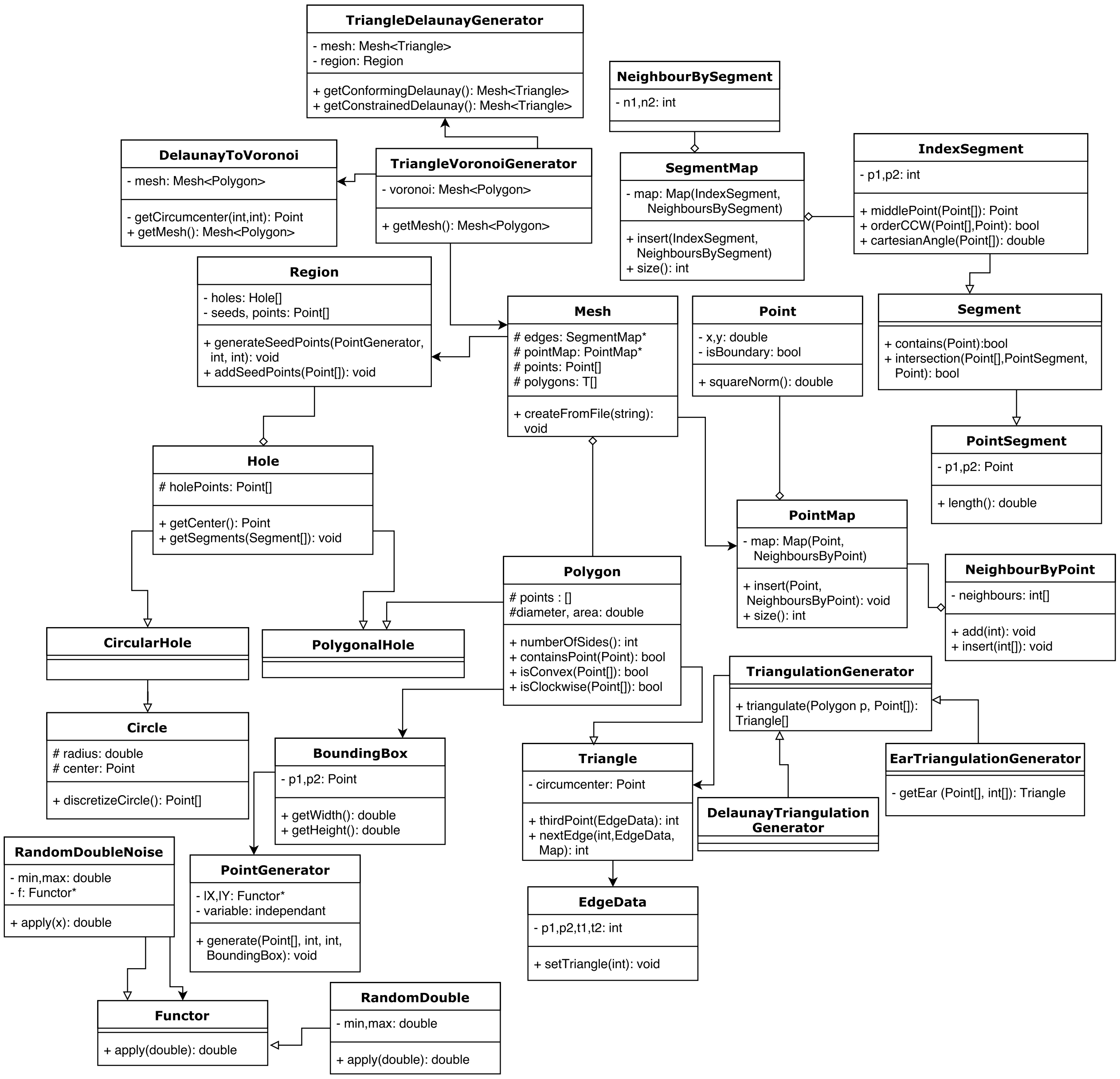


Figure 4.6: *UML class diagram of Voronoi generation library*

We have already defined mesh as the central structure of this library, but have not detailed its modeling. We have a template Mesh class that contains a list of instances of whatever mesh element one requires, in our case we implement either Triangles or Polygons; besides the list of mesh elements, Mesh contains the list of points (which need to be unique), the SegmentMap that includes both the segment and neighbourhood information, the PointMap with the neighourhood by point information, and a series of basic mesh operations, such as setters, getters and a method to print the mesh in a text file.

Finally, we create a utility class called BoundingBox, that models the axis-oriented bounding box of a polygon (defined in Section 2.1.1); it just contains two points, the corners of the rectangle.

4.3.5.2 Virtual Element Method library

Differently from the Voronoi generation library, the concepts found in numerical simulation applications, such as VEM and FEM, are readily translated to classes, hence, we do not need to detail any specific assumptions about the modeling. This way, we will first present the UML class diagram, followed by an explanation of the classes. It is important to note that, as we mentioned in Section 4.2.3, we include the implementation of FEM inside the same library, as a separate module.

UML class diagram The UML class diagram with the most relevant classes of the Virtual Element Method library (including their class members and method) is big enough to require partitioning so thtat the details can be better understood. We choose to divide the diagram into three parts:the classes related to the problem conditions, such as materials, constraints and body forces (shown in Figure 4.7); the modeling of Virtual Element and Finite Element Method (Figure 4.8); and the computation of the norms for both methods (Figure 4.9). It is important to note that, for the sake of clarity, we repeat some important classes in several diagrams.

Classes details As the Virtual Element Method library is divided in two modules, one that implements the Virtual Element Method itself, with all structures and algorithms both to create the system and compute the displacements that are required; the other oversees the Finite Element Method, implemented taking advantage of the tools included for VEM, plus some new ones that are specific.

Two abstract classes are central to this library, Calculator2D and Element. Calculator2D is designed in the spirit of the Controller design pattern, as it receives the conditions of the elasticity problem, creates the required structures, and runs the simulation itself, making the necessary changes to the matrix depending on the boundary conditions. Calculator2D, as an abstract class, has a number of methods that all inherited classes must implement; the two most important ones are the one in charge of creating the elements, and the one that computes the stiffness matrix and load vector. We implement two concrete Calculator2D classes, called Veamer and Feamer, the first representing the controller for the Virtual Element Method, and the second for Finite Element Method.

On the other side, Element, is the class that encapsulates the behavior of each element of the domain, in charge of keeping the degrees of freedom of the element, and its associated stiffness matrix and load vector; differently from Calculator2D, Element only has methods that are common to all concrete Element classes, and no virtual methods. Element contains methods to create and assign degrees of freedom, computing the load vector from all its components, and assembling the stiffness matrix and load vector; however, computing the elemental matrix and load vector is responsibility of each concrete Element implementation. An Element has a corresponding polygon that defines its boundary and gives it its characteristics, plus a number of degrees of freedom, which represent the variable to compute at the element's nodes. Element has two inherited classes, VeamyElement and FeamyElement, which, as it is the case with Veamer and Feamer, represent elements of the Virtual Element and Finite Element Methods; however, it is important to comment that, as mentioned in Section 2.2.4.1, there are different kinds of FEM elements, depending on the geometry and order of the element, so FeamyElement is also an abstract class. As implementing the Finite Element Method is done only as a means of comparison with VEM, we include the minimum necessary, and only implement the linear triangular element, which is called Tri3Element in our library. Finally, each FeamyElement concrete implementation has an equivalent ShapeFunctions concrete subclass, representing the shape functions that are used to interpolate the solution inside the element; in our case, we include a Tri3ShapeFunctions class, which represent the shape functions shown in Equation (2.17).

One of the structures related to all Element classes is called DOF (standard abbreviation of Degree Of Freedom), which, as it can be deduced from its name, describes a single degree of freedom of the problem; as our implementation is restricted to solving linear elasticity problems in two dimensions, each point of the mesh has two associated DOFs, one for each axis (x or y). The degrees of freedom are kept in a list (for convenience of implementation, each DOF also keeps its own index in this list) but as the list has functionalities beyond keeping the DOFs ordered, we create a wrapper that contains the vector, and implement a number of operations; accordingly, we name this container class DOFS. DOFS is the only class that keeps references to the DOF instances themselves, as all Elements use their corresponding indexes; therefore, most operations require DOFS as an input, which Calculator2D keeps as a private parameter that passes around as needed.

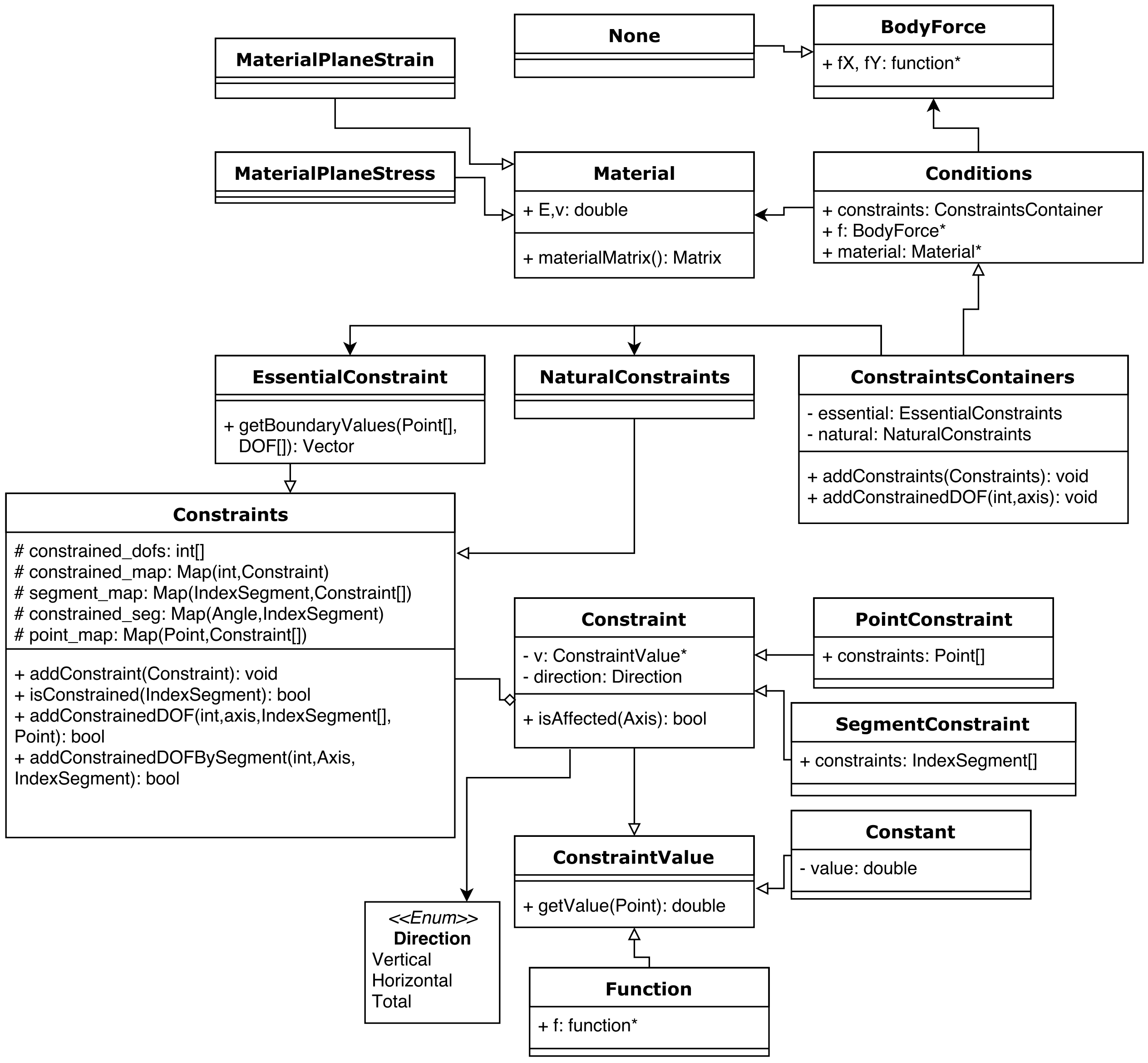

None
BodyForce
+ fX, fY: function*
MaterialPlaneStrain
MaterialPlaneStress
Material
+ E,v: double
+ materialMatrix(): Matrix
Conditions
+ constraints: ConstraintsContainer
+ f: BodyForce*
+ material: Material*
EssentialConstraint
+ getBoundaryValues(Point[], DOF[]): Vector
NaturalConstraints
ConstraintsContainers
- essential: EssentialConstraints
- natural: NaturalConstraints
+ addConstraints(Constraints): void
+ addConstrainedDOF(int,axis): void
Constraints
# constrained_dofs: int[]
# constrained_map: Map(int,Constraint)
# segment_map: Map(IndexSegment,Constraint[])
# constrained_seg: Map(Angle,IndexSegment)
# point_map: Map(Point,Constraint[])
+ addConstraint(Constraint): void
+ isConstrained(IndexSegment): bool
+ addConstrainedDOF(int,axis,IndexSegment[], Point): bool
+ addConstrainedDOFBySegment(int,Axis, IndexSegment): bool
Constraint
- v: ConstraintValue*
- direction: Direction
+ isAffected(Axis): bool
PointConstraint
+ constraints: Point[]
SegmentConstraint
+ constraints: IndexSegment[]
ConstraintValue
+ getValue(Point): double
Constant
- value: double
<<Enum>>
Direction
Vertical
Horizontal
Total
Function
+ f: function*

Figure 4.7: *UML class diagram of VEM library - Conditions of the elasticity problem*

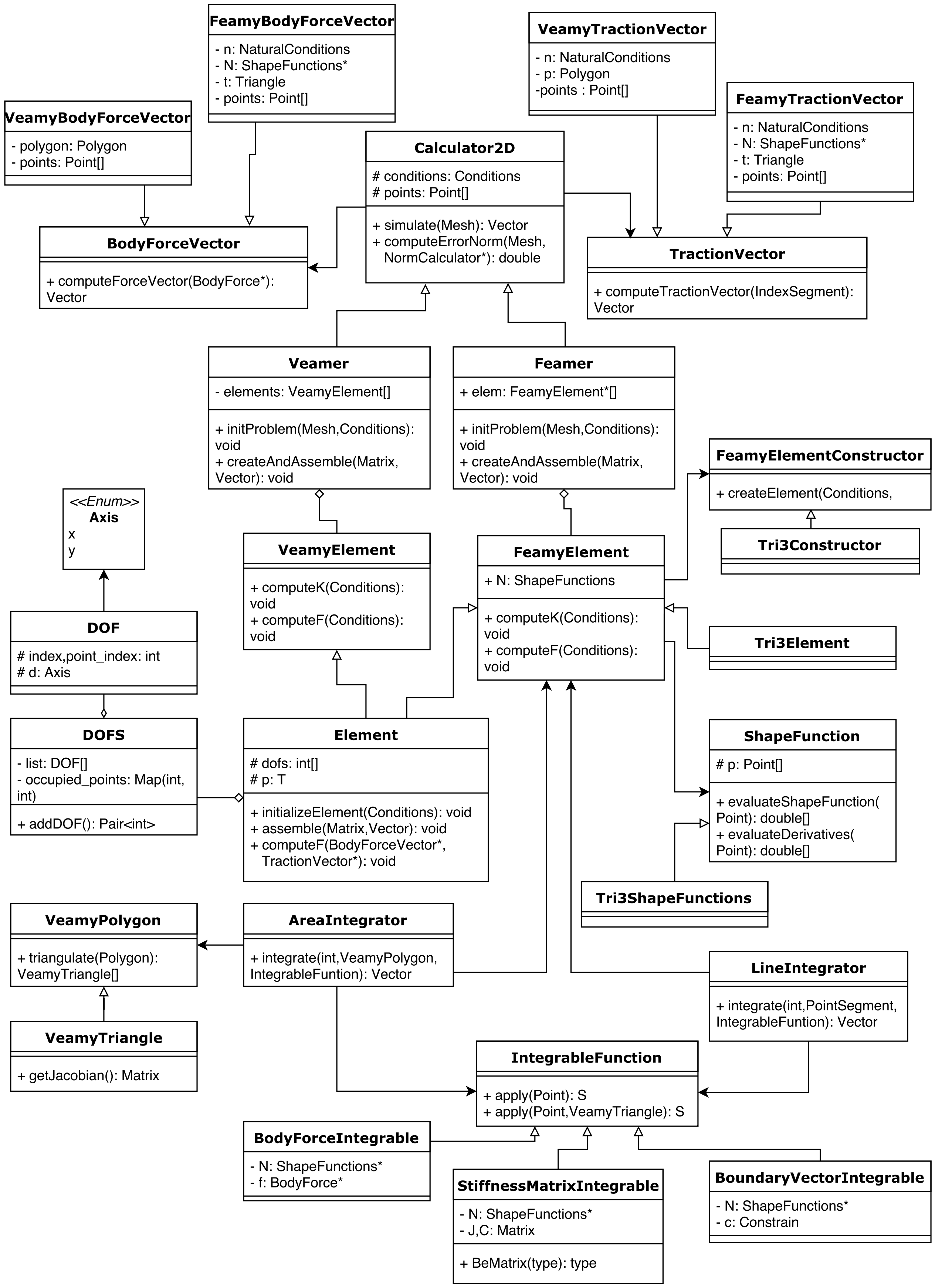


Figure 4.8: *UML class diagram of VEM library - VEM and FEM modeling*

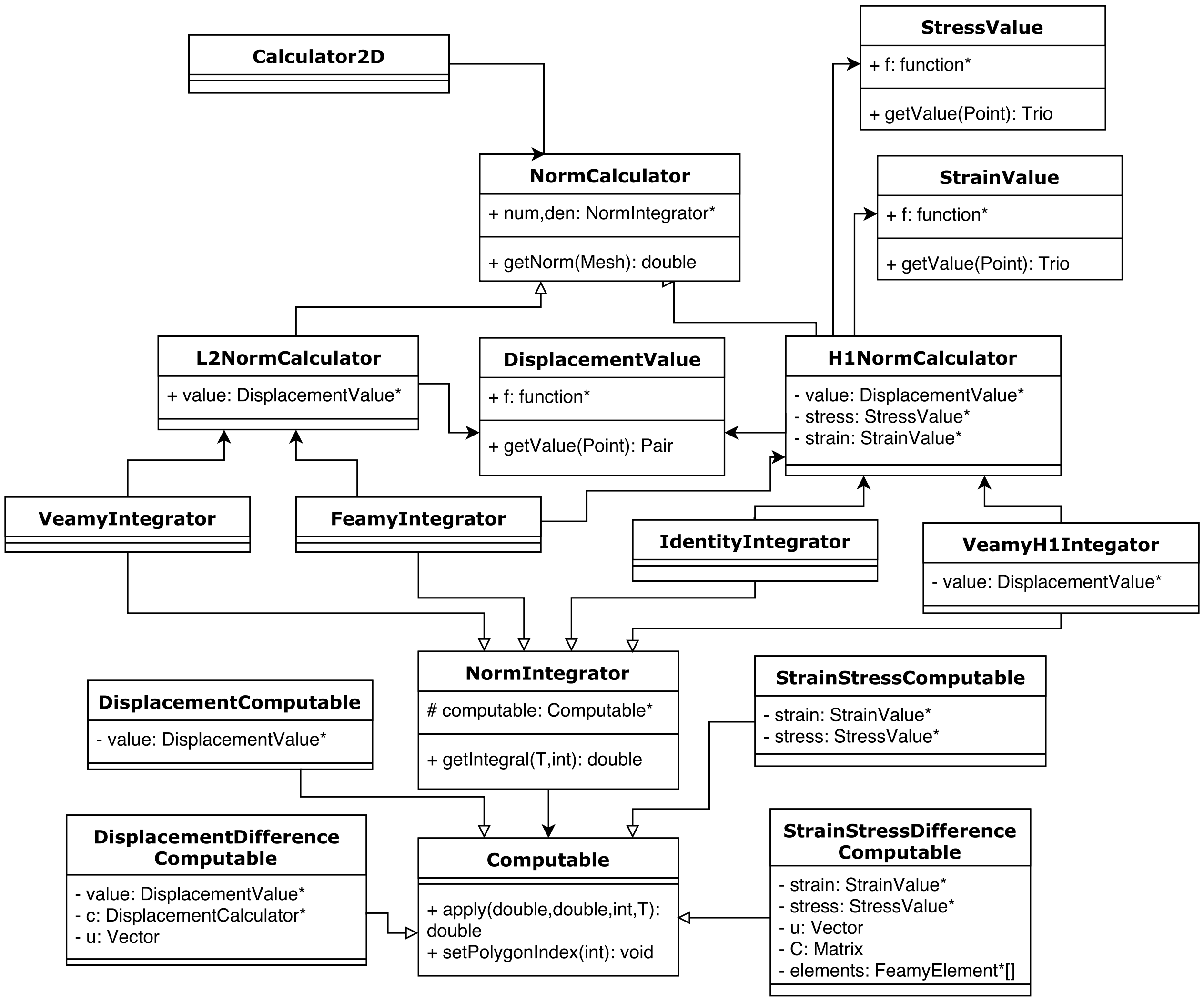


Figure 4.9: *UML class diagram of VEM library - Norm calculation*

The conditions of the linear elasticity problem are kept in a simple structure called Conditions: it includes the physical properties of the material (Material class), the boundary conditions and the body force (left hand side of the elasticity equation, modeled using the BodyForce class). BodyForce is a simple class that contains two functions pointers (each representing the body force in one axis), which must be instantiated by the user to include a body force in the problem; by default, Conditions creates an instance of the None class, a subclass of BodyForce that represents the nonexistence of body forces. Material is an abstract class that keeps the constants associated with the material properties (elastic module and Poisson coefficient) and has an abstract function that computes the material matrix C shown in Section 2.2.2; Material has two subclasses, MaterialPlaneStress and MaterialPlaneStrain, which return, respectively, the material matrix for plane stress and plane strain, and the trace of the same matrix. Finally, to help the user, we include a list of common materials and their material properties, and include a way to initialize any Material subclass instance just giving one of the names of this list.

To model the boundary conditions of the linear elasticity problem, we have created a number of classes: Constraint is an abstract class represents any unique constraint on the system, as we can have an arbitrary number of constraints, both natural and essential; PointConstraint and SegmentConstraint are concrete classes implementing Constraint, representing, respectively, a constraint in points and segments of the domain, which must be differenced because of how they are included in the load vector. Constraints is the class that manages all the constraints of the system, and the relationship between them and the DOFs; NaturalConditions and EssentialConditions both inherit from Constraints, with extra operations relevant to each kind; finally, ConstraintContainer is a utility class that contains a EssentialConstraints and NaturalConstraints instance, which is passed around as needed. Constraint keeps a list of domain segments subjected to the same condition, the value of this condition, and a certain direction (being vertical, horizontal or both). The interface called ConstraintValue is the way we have to control user input for constraints; to add any constraint, the user must choose between a constant value (Constant class), a function (Function class) or implement a new class, inheriting from ConstraintValue.

The computation of the load vector is not included inside the Element class or any of its subclasses because, although the different terms are computed differently in VEM and FEM, how the terms are added to obtain the final elemental load vector is exactly the same; this way, we decide to separate the computation of each part in abstract classes, which concrete subclasses depend on the method they implement. As commented in Section 2.2.4.1, the load vector has a component related to the body forces and other to the traction forces, which are represented, respectively, with the BodyForceVector and TractionVector classes; each one has a set of two concrete subclasses, one for VEM and the other for FEM, named, respectively, VeamyBodyForceVector and FeamyBodyForceVector, and VeamyTractionVector and FeamyTractionVector.

As we commented in detailed in the background chapter, the stiffness matrix in Virtual Element Method is easily computed from an analytic formula, but that is not the case in Finite Element Method. In FEM, it is necessary to numerically integrate both inside an element (area integration) and around it (line integration); to separate responsibilities and leave numerical integration available in case it is desired to use elsewhere, we create two classes, AreaIntegrator and LineIntegrator, which contain methods that integrate a given function inside and in the boundary of a given polygon. There are several classes related to the numerical integrations: first, IntegrableFunction is a simple template interface with one method that must be implemented, called *apply*, which receives a Point and the index of the point, and returns any type; we include several concrete IntegrableFunction implementations, used in the numerical integration of different elements of FEM, all with self-explanatory names, such as BodyForceIntegrable, TractionVectorIntegrable and StiffnessMatrixIntegrable. Second, numerically integrating polygons different from triangles is not direct, as it is first required triangulating the element, and then computing the required value in each obtained triangle; to leave numerical integration ready for use, we include two classes called VeamyTriangle and VeamyPolygon, that encapsulate this exact behavior.

One last aspect of the numerical computation of the displacements is how to validate if the results are reasonable; as we mentioned in Section 2.2.5, in this kind of method it is usual the calculation of norms to compare the numerical results with the analytical solution. We include methods to compute the L2 and H1 norms (which were explained in Section 2.2.5.2), in form of two classes named L2NormCalculator and H1NormCalculator; as the norms are computed similarly, both classes inherit from the abstract class NormCalculator, which contains all common methods.

Each NormCalculator instance contains two instances of what we call the NormIntegrator classes, in charge of numerically computing the integral value inside each sum; this class is necessary since VEM and FEM use different schemes for numerical integration. We implement three NormIntegrator: FeamyNormIntegrator, which uses gaussian integration; VeamyNormIntegrator, which computes the integral using the nodal scheme; and VeamyH1Integrator, which is does not compute any integral, but rather returns the approximate value for the H1 norm that was shown in Equation (2.39).

The NormIntegrator classes are in charge of computing integrals, with the function itself represented by what we call a Computable; this way, we can reuse the integration computation for several different norms. There are as many Computable classes as functions required for the norms: StrainStressComputable, StrainStressDifferenceComputable, DisplacementComputable, DisplacementeDifferenceComputable, among others. Finally, there is an important detail concerning how the norms are calculated in VEM and FEM: while VEM uses a strictly vertex-based approach, in FEM the norms are computed considering the interpolated value of the displacement, which is not direct to determine. Hence, we require to include ne final set of classes that represent this behavior; we call them the DisplacementCalculator, and implement two, the VeamyDisplacementCalculator (which returns the displacements by point index) and the FeamyDisplacementCalculator, which receives a point and computes its displacement considering the nodal values of the polygon that contains it.

4.3.5.3 Modifiable meshes library

In this section we detail the design of the models of the modifiable meshes library, starting with the UML class diagram to show the structure of the software, and continuing giving relevant details about the classes.

UML class diagram The UML class diagram with the most relevant classes of the modifiable meshes library (including their class members and method) is shown in Figure 4.10. It is important to comment that, as this library is an extension to the Voronoi generation one, there are several classes from the first library that appear in this diagram; to clarify which classes belong to each library, all elements from the generation library are marked as *external*.

Classes details The central class to this library is called xMesh, a subclass of the Mesh class we commented on previously. xMesh represents a mesh that can be modified in four different ways: it can be refined, de-refined, broken or fixed, and contains one or several methods for each of these functionalities; it also includes several other methods that are utilities for the main functions of the class. xMesh, as a Mesh subclass, contains a list of points, a SegmentMap, a PointMap and a list of elements; the elements, however, are neither Polygon nor Triangle instances, but rather a new class called xPolygon, which inherits from Polygon but contains a number of methods related to changes. It is important to note that we do not include an xTriangle class, as it is was considered useless because of how the library works. xMesh also contains additional members that are not part of Mesh, the most important being: xSegmentMap, an extension to SegmentMap with methods that allow changing the information; xPointMap, an extension to PointMap, with the exact same idea as xSegmentMap; xMeshElements, a structure that contains pointers to the points, elements and neighbourhood information of the mesh, used so that several structures can change the mesh but without having access to all xMesh methods; and several classes that we name "operators".

The "operator" classes are the ones in charge of modifying some part of the mesh; we have three abstract operators, all inheriting from an abstract class named MeshOperator: MeshMerger, which receives a list of elements and returns the result of merging them; MeshRefiner, which receives an element and return a list of elements, obtained after adding one or several points; and MeshFixer, which receives an element, or the mesh, and "fix" whatever characteristics it is desired. MeshOperator is in charge of receiving and setting the xMeshElements instance, an operation common to all operations that modify the mesh. Each operator has one or more concrete subclass, which represent a way of solving the problem; for example, MeshRefiner has two subclasses, TriangulateRefiner and VoronoiRefiner, which present two different ways of refining a given element, while MeshMerger and MeshFixer only have one, VertexIndexMerger (as we only implement one algorithm to merge polygons) and ConvexFixer (as we only wish to explore the consequences of non-convexity in VEM). This scheme of presenting an abstract interface and several concrete implementations was designed so to leave extension to the library easier. In Figure 4.10, we can see another subclass of MeshMerger, called NonAllowedMerger, which is not commented upon, as it is a class that represents an error when trying to merge, representing, for example, a triangulation that must keep its properties trying to delete a point and not re-triangulating.

The last important algorithm that our modifiable mesh implements is breaking the mesh with one or several segments; differently from the previous operations, breaking requires knowledge of all mesh structure and operations, so it is not implemented in a separate MeshOperator class, but instead inside xMesh itself. However, we do require a number of additional classes to model how breaking works. In the first place, how the broken polygons are processed depends heavily on user input, as one could wish to leave them as they are broken or process them, so we include an abstract class, named ElementReconstructor, which is used to model the different behaviors one could desire to include; in particular, we leave two concrete ElementReconstructor classes, IdentityReconstructor (or "leave polygons as they are broken") and PolygonToTriangleReconstructor (or "triangulate all broken polygons").

Second, breaking the mesh has an additional complication that is not direct; as we require that the polygons are not self-intersecting, we do not allow handing edges, so breaking the mesh requires a post processing in the likely case the user includes a segment that starts or ends inside an element of the mesh. We call the processing of the hanging edges "closing", and use an abstract class called ClosingRule to model the phenomenon; we include several ClosingRule concrete implementations: TruncateClosingRule simply removes the hanging edges; StraightLineClosingRule extends the hanging edge until it reaches a segment of the polygon, dividing the polygon in two parts; ClosestVertexClosingRule does the same, but instead of adding the intersection it searches the closest vertex to connect to the hanging edge; and TriangulateClosingRule, which triangulates the polygon that contains the hanging edge.

xMesh contains instances of all MeshOperators except for MeshFixer, which is given on demand, plus a ClosingRule and ElementReconstructor pointer; all these parameters are encapsulated in a class called Config, both so that the number of parameters that are required in xMesh is not overwhelming to the user, and so that we can implement a way of defining a set of rules that are left available to the user, in a similar manner to how materials are implemented in the VEM library.

Finally, to help the generator of points for polygon refinement, we include a class called PointCreator, similar to PointGenerator in the fact that both are used to fabricate points inside a given polygon; the difference is that PointCreator creates the points given a certain rule or property that depends on the polygon itself, while PointGenerator uses rules that only consider the coordinates of the points.

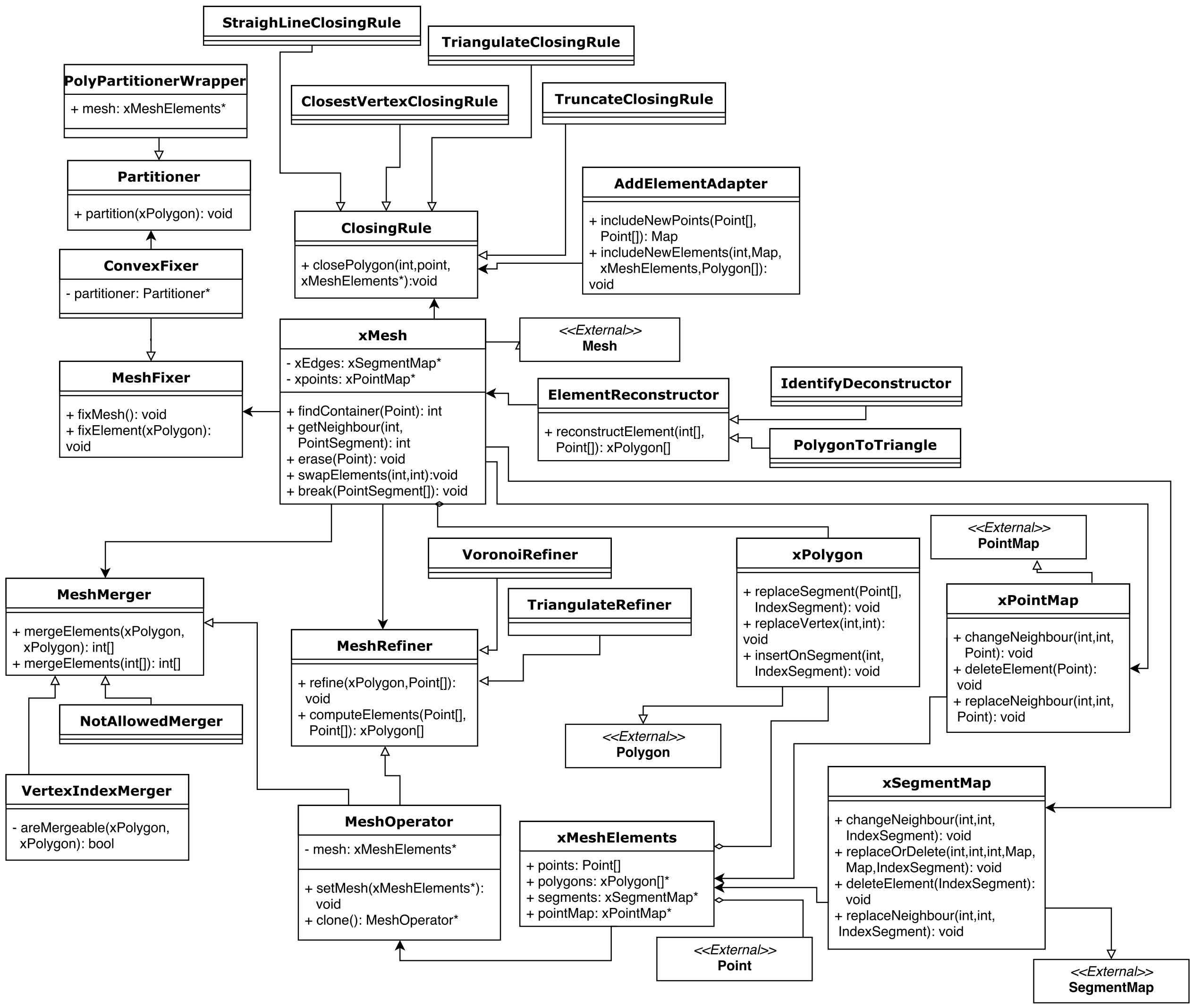

StraighLineClosingRule
TriangulateClosingRule
PolyPartitionerWrapper
+ mesh: xMeshElements*
ClosestVertexClosingRule
TruncateClosingRule
Partitioner
+ partition(xPolygon): void
AddElementAdapter
+ includeNewPoints(Point[], Point[]): Map
+ includeNewElements(int,Map, xMeshElements,Polygon[]): void
ClosingRule
+ closePolygon(int,point, xMeshElements*):void
ConvexFixer
- partitioner: Partitioner*
xMesh
- xEdges: xSegmentMap*
- xpoints: xPointMap*
+ findContainer(Point): int
+ getNeighbour(int, PointSegment): int
+ erase(Point): void
+ swapElements(int,int):void
+ break(PointSegment[]): void
<<External>>
Mesh
MeshFixer
+ fixMesh(): void
+ fixElement(xPolygon): void
ElementReconstructor
+ reconstructElement(int[], Point[]): xPolygon[]
IdentifyDeconstructor
PolygonToTriangle
VoronoiRefiner
xPolygon
+ replaceSegment(Point[], IndexSegment): void
+ replaceVertex(int,int): void
+ insertOnSegment(int, IndexSegment): void
<<External>>
PointMap
TriangulateRefiner
xPointMap
+ changeNeighbour(int,int, Point): void
+ deleteElement(Point): void
+ replaceNeighbour(int,int, Point): void
MeshMerger
+ mergeElements(xPolygon, xPolygon): int[]
+ mergeElements(int[]): int[]
MeshRefiner
+ refine(xPolygon,Point[]): void
+ computeElements(Point[], Point[]): xPolygon[]
NotAllowedMerger
<<External>>
Polygon
xSegmentMap
+ changeNeighbour(int,int, IndexSegment): void
+ replaceOrDelete(int,int,int,Map, Map,IndexSegment): void
+ deleteElement(IndexSegment): void
+ replaceNeighbour(int,int, IndexSegment): void
VertexIndexMerger
- areMergeable(xPolygon, xPolygon): bool
MeshOperator
- mesh: xMeshElements*
+ setMesh(xMeshElements*): void
+ clone(): MeshOperator*
xMeshElements
+ points: Point[]
+ polygons: xPolygon[]*
+ segments: xSegmentMap*
+ pointMap: xPointMap*
<<External>>
Point
<<External>>
SegmentMap

Figure 4.10: *UML class diagram of modifiable meshes library*

4.4 Experimental design

In this section we will present the test cases considered for the validation of the three previously defined libraries. We divide the experiments in two kinds: those concerned with the efficiency, and those with accuracy. The experiments in the first category can be applied on basically any library, while the second type is related exclusively to numerical results; hence, it is only relevant to the Virtual Element Method library. This way, in this chapter we will proceed as follows: first, we will mention the experiments that are common to all tested libraries, and then detail those that are specific to the Virtual Element Method one.

Before proceeding, we must comment on the fact that we decide not to test the extension to the polygonal mesh generation library, as it is a collection of algorithms that modify the mesh and are both dependent on user input and are not expected to be used intensely. However, although we do not test the modifiable meshes library itself, we do use its functionalities intensely in the numerical accuracy tests, as the main objective of this thesis is to prove that VEM is better suited for applications in which meshes can change.

4.4.1 Efficiency experiments

As mentioned in Section 4.1, we require our libraries to be efficient; however, that word is a complex one to use, as it could mean several things. In our case, we use it thinking of setting a good base for future work, meaning, we design the algorithms first thinking about their complexity, trying to minimize it, and then implement them, seeking to be as similar as possible to the design; it is important to note that, although we try to design to minimize complexity as much as possible, optimizing the algorithms after their implementation is out of the scope of this thesis, and is left as future work.

Therefore, we need to test the complexity of our implemented solution against its theoretical performance, both of which are unknown to us. This way, for each library, we first study its theoretical complexity using big-O analysis, which gives us a measure to compare against, and only then, we test the time it takes to process a rising number of randomly generated points; as generating tests on random domains could give unexpected results, all the tests will be done in a square domain of unitary side. Due to technical limitations, we restrict our tests to the hundreds of points, choosing a representative set of about ten different values starting from 10 and ending with around 1.000 points.

Finally, since beside the Voronoi generation library, which input are the points and domain, we will also test the VEM (and FEM), which require input beyond the previous, we need to define standard problem conditions to compare against; we decide to simplify as much as possible, and choose a situation commonly found in the literature, clamping one side of the domain and imposing a horizontal force in the opposite.

4.4.2 Numerical accuracy experiments

After detailing the experiments that will be used in all libraries indistinctly, we now present the tests cases for the validation of the Virtual Element Method library. We present the experiments with two different objectives in mind: first, we will validate that our implementations of VEM and FEM perform appropriately and numerically converge; second, knowing that our implementation gives appropriate results in the test cases, we present the experiments that compare the performance of VEM and FEM when meshes are modified in different ways, so to prove or disprove the hypothesis presented in this thesis. To keep the objectives of the experiments clear, we will separate the experiments in two subsections, and briefly explain each one; they will be presented in detail in the results chapter.

4.4.2.1 Convergence of the solutions

As it is usual with numerical method analysis, to probe that the implementation of the method converges we use as tests cases situations with known analytical solution, and compute some energy norms, that show that, as the size of the elements decrease, the accuracy of the solution increases. We use the L2 and H1 norms presented in Section 2.2.5.2, since their implementation and convergence rates are well known. We evaluate both implemented methods, calculating both selected norms in meshes of different size, in three different tests, named **Displacement patch test**, **Equilibrium patch test** and **Parabolic beam test**; the first two tests are simple enough to not require explanation beyond a few words, while the latter is more complex, so we show it schematically in Figure 4.11. Finally, each of these tests analytical solutions will be presented alongside the obtained results.

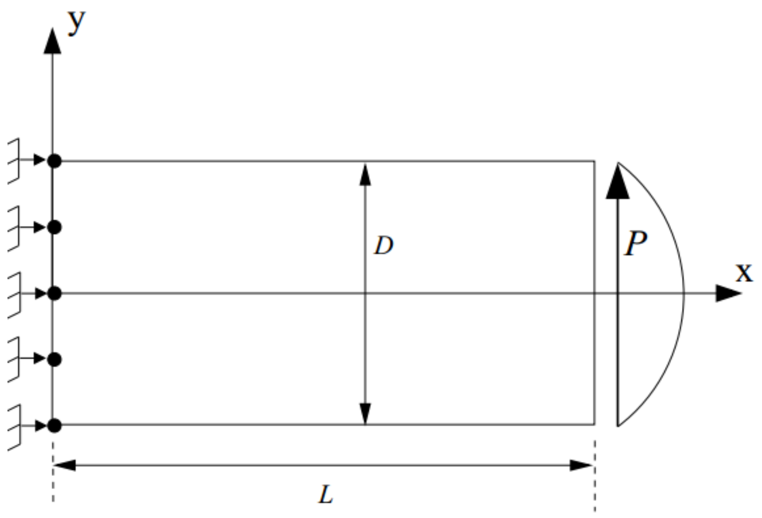


Figure 4.11: *Parabolic beam test*

4.4.2.2 Modifiable meshes quality experiments

Differently from the convergence experiments, which objective is to demonstrate that the method is correctly defined and implemented, this set assumes that the results are correct, and seek to compare the quality of the results when the meshes are changed. However, as comparing between two numerical methods is not recommended (since one does not have a benchmark), we rely again on examples which analytical solution is known, but with a different focus when compared with the experiments shown in the previous subsection. This time, we compare the nodal displacement error obtained, for VEM and FEM, in a mesh before and after a given modification, seeking to prove that VEM behaves better in most situations. We will work with two examples, one of which we shown in the previous subsection (in Figure 4.11), while the other, named **Pressurized cylinder**, can be seen in Figure 4.12; we will compute the displacements using VEM and FEM in a series of tests which include operations such as breaking, refining and fixing the convexity of the mesh, using comparable meshes (Voronoi-based for VEM and Delaunay-based for FEM, but computed using the exact same points).

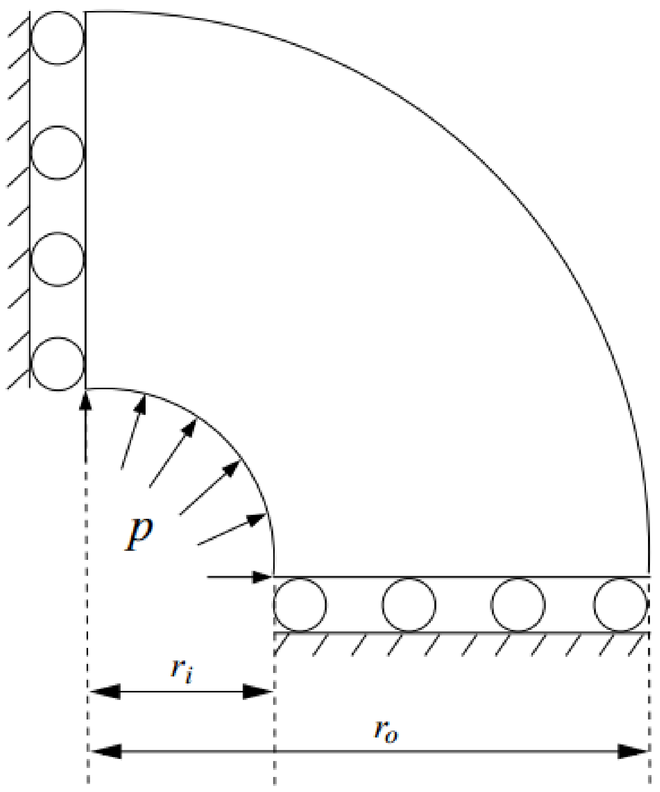


Figure 4.12: *Pressurized cylinder test*

Chapter 5

Implementation

In this chapter we present the concrete implementation of the three libraries, which design we detailed previously; as it is natural, we will keep the division presented before, starting with the Voronoi generation library, continuing with the Virtual Element Library, and finishing with the modifiable meshes library.

5.1 Voronoi generation library

This section is concerned with the concrete implementation of some of the aspects that we commented in the design chapter, in particular: the generation of the Voronoi diagram, and how the geometrical algorithms were implemented. However, it is important to note that, as mentioned in Section 4.3.3, we consider at this step of the work that the geometry is static, and include all methods related to mutability in the modifiable meshes library.

Before proceeding, however, we do need to stop on how we model the polygonal meshes; we showed the class design and explained most of the classes in Section 4.3.5.1, but there is one implementation detail that we do need to mention before proceeding. We mentioned that the points of a polygonal mesh have to be unique, but we have not made clear how: we decide that the cleanest way of doing this is implementing a custom structure, named *UniqueList*, that has the same functionalities of a list, but also makes sure that its elements are unique; we will comment on its concrete implementation in following a section. This way, the list of points of a mesh is not actually a list per se, but a special variety of list that checks for uniqueness.

5.1.1 Voronoi diagram generation

This section features the concrete implementation of all the steps shown in Figure 4.1; to ease understanding, we use the exact same names and proceed in the same order.

5.1.1.1 Domain/Hole processing

The first step in the generation of our bounded Voronoi diagram is to actually define the boundary region, which needs to be done by the user in the form of a Region instance; however, the Region needs to be processed, as our library allows almost arbitrary domains for our simulations, including holes both inside and in the boundary of our regions, and in the second case we need to clip the region to represent the presence of a hole. We can differentiate two cases:

1. The hole or holes are completely inside the domain. This case is simple to deal with since Triangle allows the triangulation of PSLG (which can include holes) and has an interface for the inclusion of holes. This way, Region keeps a list of holes, that only includes those completely contained in the region, as those that intersect it are processed.
2. A Hole or holes intersects the boundary of the domain; in this case we need to compute the intersection between the hole and the region. As mentioned in Section 4.3.2.2, we use Clipper to compute the difference between the domain and the desired hole.

Clipper uses the concept of Paths as a series of points that make up an arbitrary line; the difference between a polygon and hole completely contained has two Paths, while the same operation but now between two shapes that intersect has only one Path. Using this observation, we can easily determine when we need to change the region or include the hole in the list; this way, every time the user adds a hole, we use Clipper to compute the difference between the hole and the region and count the number of Paths of the solution to determine how to proceed:

- When the Clipper solution has size one we obtain the new region as the result of the difference, and recompute all the parameters of the domain instance without changing its direction in memory.
- On the other side, when the solution's size is more than one we have to check whether the hole is inside the region or not, which we do checking if the center of the region is inside the region. This works since it just takes for one point to be outside the region to either be completely outside or to intersect the region's boundary (and this second case was already discarded by our previous check); if the hole is inside the region we insert it in the hole list, the other case is considered an error, and throws an exception detailing the problem.

5.1.1.2 Generation of seed points

Once we have the boundary region processed, we need seed points to generate the Delaunay diagram. We choose to leave two options available: the user can simply enter a list of seed points externally generated, or, as it is usually the case with bigger regions, a certain function could be used as a creation rule. The first case is extremely simple, as a Region can receive a list of seed points, but the second requires further explaining.

We define an interface for all point generating functions (including some common cases such as constant, uniform growth, random, among others) called Functor; this class, being an interface, cannot override operators, which would be the optimal way of handling this problem, but instead has a simple virtual function, named *apply*, that receives a double and returns another double. Functor is later used by the PointGenerator class (shown in Code 5.1); this structure is created either using two pointers to Functor, as a user could want the rules for each axis to be independent, or one Functor pointer with a variable indicating that either x or y are a function of the other variable. As help to the user, we leave a list of available Functor classes called *functions*, from which one can readily obtain common examples; it is also possible to include custom Functor implementations and use them in the PointGenerator class.

Code 5.1: *PointGenerator class declaration*

```
class PointGenerator {
private:
    Functor* lX;
    Functor* lY;
    functions::independent_variable variable;

    Point result(double x, double y);
public:
    PointGenerator();
    PointGenerator(Functor* lX, Functor* lY);
    PointGenerator(Functor* l, functions::independent_variable variable);
    void generate(std::vector<Point>& vector, BoundingBox box, int nX, int
        nY);
};
```

Besides the Functor classes available to the user, we also include a way of adding noise, following a sort of decorator pattern. A Noise Functor class is simply a Functor class which contains a pointer to other Functor class, so that noise is applied on the result obtained from the original Functor class used; Noise Functor classes are listed in another list, similar to *functions*, called *noise*, and, again, custom implementations can also be used.

Before the generation of the Delaunay triangulation, the user must call one of the following Region methods or the processing would fail: it can either call the method *addSeedPoints*, with a vector of seed points generated externally as parameter; or the method *generateSeedPoints*, that creates the points automatically. The second method receives a PointGenerator instance and the number of points that the user wants in each axis; since the region can be of an arbitrary shape, to simplify the implementation, PointGenerator creates the points in the bounding box of the Region, and then the Region itself checks all the points, and keeps only those that are inside it. The code detailing the generation of the points themselves is shown in Code 5.2.

Code 5.2: *Generation of points in PointGenerator*

```
void PointGenerator::generate(std::vector<Point> &vector, BoundingBox box,
                              int nX, int nY) {
    double dX = box.getWidth()/(nX-1), dY = box.getHeight()/(nY-1);

    for(int i = 0; i<nY; i++){
        for(int j = 0; j<nX; j++){
            double x = box.xMin() + j*dX, y = box.yMin() + i*dY;

            vector.push_back(this->result(x, y));
        }
    }
}
```

5.1.1.3 Write input data to Triangle

In this step of the algorithm, we take the domain and seed points, and write them in the input format that Triangle requires to compute the constrained conforming Delaunay triangulation. The structure called *triangulateio* is part of the API of Triangle, both serving as a container of the input data and the output data; this structure has a great number of parameters that do not concern us, and those that do, are mostly self-explanatory. We present in Algorithm 10 how we write the relevant data in a *triangulateio* struct named *in*.

Algorithm 10: Writing input data for Delaunay computation

```
Data: Region $region$
Result: Struct triangulateio $in$ with data for Delaunay computation
$in$ ← struct triangulateio()
$points$ ← 0
for $p$ in $region.seedPoints$ do
    $in$.pointlist[$points$] ← $p$.X, $in$.pointlist[$points + 1$] ← $p$.Y
    $in$.pointattributelist ← 0
    $points$ ← $points + 2$
end
$segments$ ← $region$.getSegment
i ← $segments$.size
for $s$ in $segment$ do
    $in$.segmentlist[2i] ← $s$.getFirst, $in$.segmentlist[2i+1] ← $s$.getSecond
    i ← i + 1
end
$holes$ ← $region$.getHoles
i ← $holes$.size
for $h$ in $holes$ do
    $in$.holelist[2i] ← $h$.getCenter.X, $in$.holelist[2i+1] ← $h$.getCenter.Y
end
```

We use the previous information to call Triangle and obtain the constrained conforming Delaunay triangulation, which, at the same time, assures the quality of the triangles obtained, and prevents obtuse angles next to the border; the last point is important to avoid degenerate cases in the Voronoi diagram, since with well-formed triangles next to the boundary, we cannot have the case when the circumcenter of the triangles lies outside the domain. This way, just by adding an extra flag to Triangle, which transforms a simple constrained Delaunay triangulation to a conforming one, we prevent most of the problems that could arise when computing a constrained Voronoi diagram from a constrained Delaunay triangulation.

5.1.1.4 Processing of Triangle's output

Triangle's interface provides us with the Delaunay triangulation in a form similar to our data structure, plus some extra redundant data, that we are able to ask for using flags defined in the documentation of the library. This information is:

- Triangle's standard output includes the minimum amount of information needed to reconstruct the triangulation: the list of points and the list of triangles, using the indexes of the points as reference.
- The points in a Delaunay triangulation are the same as the ones provided by the user, so usually one would not bother returning them in a simple mesh generator, but, since we are dealing with a version of Delaunay that can add points, this information is relevant, even if we know the original points. Triangle returns the number of Delaunay points and an array of two times the number of points with the coordinates of the points (x, followed by y). Also, we use a list of point markers (one for each point) that indicate whether the point in question is in the boundary or not.
- We need neighbourhood information, but instead of working with the indexes of the neighbour polygons, for convenience of the Voronoi algorithm, we prefer listing points, and one edge incident to it. Triangle provides the list of edges of the triangulation when the user includes the flag -e; we use this information to fill a map with our desired information.

The information provided by Triangle is kept in several structures that will be used to compute the Voronoi diagram. In the next paragraphs we will go through the data structures created using Triangle's output, their structure, objectives and uses, in preparation to explain the Delaunay to Voronoi algorithm.

The points of the triangulation are kept in two vectors: the first one is the list provided by Triangle (named *meshPoints*), and the second one is a *UniqueList* of the indexes of the points that were actually used by the Delaunay triangulation (*realPoints*). We need the second list because it was discovered through testing that Triangle does not delete duplicated points, but rather keeps all copies in the point list, referring to only one; this led to several errors in the implementation, as we cannot delete duplicates, because we use indexes as reference, but also cannot allow them, as each Voronoi cell is associated with a point. Hence, we decided that the easiest, and cleanest, way of dealing with this was simply to keep all the points returned by Triangle as they come, but also keep a different list with the indexes of the points that are actually referenced by the triangulation, so duplicated points are still there, but are ignored. Finally, when points are created they have a flag set, obtained from Triangle's *pointmarker* list, that indicates whether the point is in the boundary.

The Delaunay triangles are kept as a vector of Triangle class instances, each created with the point indexes returned by Triangle; this list has relevant ordering, since the triangles will be referenced by their index later.

Neighbourhood information is represented indirectly, through the relation between points and edges, and edges and their incident polygons, both kept in vectors of custom structures. The first structure is called PointData, which relates the index of a point to the index of a segment that contains it, making sure that, if possible, each point is related to a boundary segment. The second data structure is EdgeData; it is formed by two non-negative indexes (representing a segment) and two other indexes that can be negative (representing the polygons incident to the segment, which can be negative if the segment is in the boundary).

However, for the Delaunay to Voronoi algorithm we need to find segments multiple times, so linearly searching through the EdgeData vector in not an opinion in the long run. We introduce an extra structure that allows us to quickly find the index of any EdgeData given its start and end point, called an *EdgeMap*, a hash map that uses a pair of points as a key, and the index of the related EdgeData as value; this way, finding an EdgeData from the pair of points that conform the segment is $O(1)$.

Using all the data structures mentioned before, we can show how the relevant input data for the Voronoi diagram computation is created; again, we present this part in pseudo code (shown in Algorithm 11) to abstract from unnecessary implementation details and leave the relevant points clear.

5.1.1.5 Summary of relevant data

Before continuing with the implementation of the step that actually computes the Voronoi diagram from the Delaunay triangulation, we present in this subsection a summary of all the data we mentioned before, to serve as reference when explaining the algorithm.

Input data:

- **MeshPoints**: Points (including duplicates) of the Delaunay triangulation.
- **RealPoints**: UniqueList of the indexes of points actually used by the Delaunay triangles.
- **Edges**: List of EdgeData with the edges of the triangulation and their incident polygons.
- **PointEdgeData**: List of PointData, relating a point to a segment that contains it.

Output data:

- **VoronoiPoints**: Points of the Voronoi diagram.
- **VoronoiEdges**: SegmentMap relating the segments of the diagram with the polygons incident to these segments.
- **VoronoiCells**: Polygons (referred also as cells) of the diagram.

Algorithm 11: Input data for Voronoi creation

```
Data: Struct triangulateio out
Result: List of Delaunay points meshPoints, list of point indexes used by Delaunay
        realPoints, list of Delaunay edges edges, list of structures relating points and
        segments points
for i in 0 to out.numberofpoints do
    data ← PointData(i)
    Insert data in points
    point ← Point(out.pointlist[2i], out.pointlist[2i + 1])
    if out.pointmarkerlist[i]==1 then
        point.setAsBoundary
    end
    Insert point in meshPoints
end
for i in 0 to out.numberofedges do
    data ← EdgeData(out.edgelist[2i], out.edgelist[2i + 1])
    Insert data in edges
    Insert in edgeMap the pair(out.edgelist[2i], out.edgelist[2i+1]),i)
    points[out.edgelist[2i]].setEdge(i, out.edgemarkerlist[i])
    points[out.edgelist[2i+1]].setEdge(i, out.edgemarkerlist[i])
end
for i in 0 to out.numberoftriangles do
    triangle_points ← out.trianglelist[3i], out.trianglelist[3i+1], out.trianglelist[3i+2]
    Insert out.trianglelist[3i] in realPoints
    Insert out.trianglelist[3i+1] in realPoints
    Insert out.trianglelist[3i+2] in realPoints
    triangle ← Triangle(triangle_points)
    Insert triangle in triangles
    i1 ← Index of pair (out.trianglelist[3i], out.trianglelist[3i+1]) in edgeMap
    i2 ← Index of pair (out.trianglelist[3i+1], out.trianglelist[3i+2]) in edgeMap
    i3 ← Index of pair (out.trianglelist[3i+2], out.trianglelist[3i]) in edgeMap
    edges[i1].setTriangle(i)
    edges[i2].setTriangle(i)
    edges[i3].setTriangle(i)
end
for i in 0 to edges.size() do
    data ← edges[i]
    segment ← Segment(data.p1, data.p2)
    neighbours ← Neighbours(data.t1, data.t2)
    Insert pair (segment, neighbours) in delaunayEdges
end
```

5.1.1.6 Voronoi diagram from Delaunay triangulation

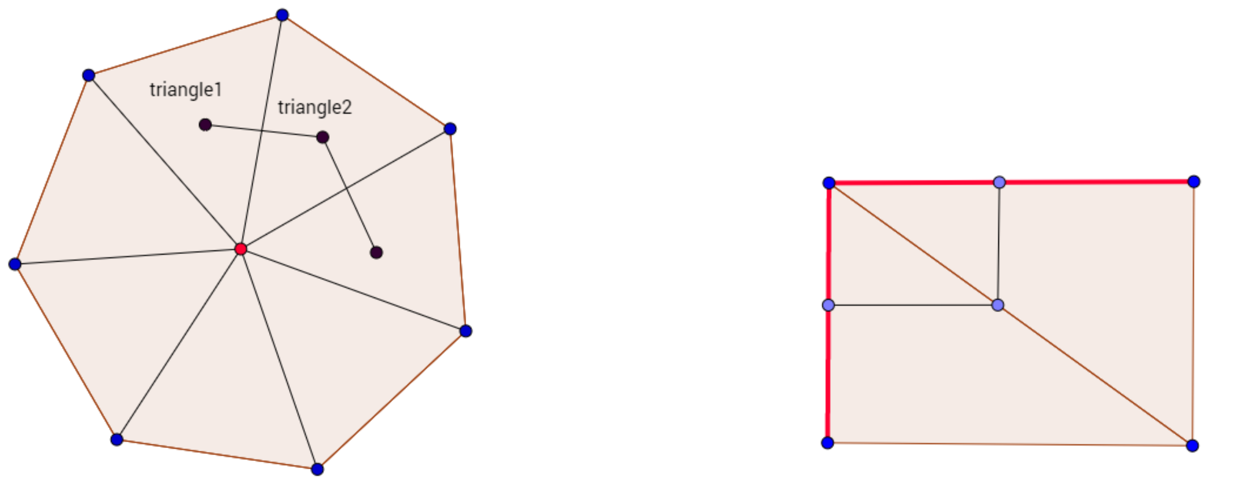
As most of the terms that we will use have already been explained before, we can proceed directly with the algorithm. Since the Delaunay to Voronoi algorithm is extremely long and complex, we have decided to present it by steps in Algorithm 12, separating the implementation to better explain each step.

Roughly, the algorithm consists on: for each point of the Delaunay triangulation, we first take the current edge and assign, as $triangle1$ and $triangle2$, the two triangles that will rotate around the *region_ center* point, creating the Voronoi cell; each step, we advance moving $triangle1$ one step forward by finding the edge that contains the center but is different from the current one, and assigning $triangle2$ as the former $triangle1$, then uniting their circumcenters (seen in Figure 5.1a).

Algorithm 12: Computation of the Voronoi diagram

```
Data: List of Delaunay points $meshPoints$, list of point indexes used by Delauny
      $realPoints$, list of Delaunay edges $edges$, list of structures relating points and
      segments $points$
Result: List of Voronoi points, list of Voronoi cells, map relating segments and
        incident polygons
for i in realPoints do
    Compute initial cell segment, shown in Algorithm 13
    Iterate until the cell is complete, shown in Algorithm 14
    Deal with boundary cases, shown in Algorithm 15
    Insert cell in mesh, shown in Algorithm 16
end
```

The first part of the algorithm consists on the declaration of parameters that will be used throughout the computation of the cell; this procedure, shown in Algorithm 13, is almost equal to the one done later in the iteration step, but it is needed to leave separate to properly declare all variables, and store the initial edge that later will be used to determine when to stop the iteration. The only special case that we need to check in Algorithm 13, which will be repeated throughout the entire algorithm, is the possibility that the circumcenters of both triangles are equal, as it happens when one considers a segment of the boundary, and the triangle is rectangle (Figure 5.1b), hence the check before inserting the second circumcenter and the segment in the mesh.



(a) Process of creation of a Voronoi cell **(b)** Circumcenters are equal border case

Figure 5.1: *Delaunay to Voronoi procedure*

Algorithm 13: Computation of the first segment and declaration of variables

```
cellPoints, cellEdges ← [ ]
region_center ← meshPoints[i]
edge_index ← pointEdgeData[i].getEdge
initial_edge ← edges[edge_index]
triangle1, triangle2 ← initial_edge.getTriangles
c1 ← getCircumcenter(triangle1, edge_index)
c2 ← getCircumcenter(triangle2, edge_index)
Insert c1 and c2 in VoronoiPoints
if c1 != c2 then
    Insert Segment(c2,c1) to cellEdges and c2 in cellPoints
end
Insert c1 in cellPoints
edge ← edges[triangle1.nextEdge]
```

Once we have declared and assigned all the variables that will be used, we can iterate around the *region_center* until the current edge is equal to the initial one, or we have reach a border, when *triangle*1 is equal to -1, as seen in Figure 5.2; the second case indicates the end of the computation for this cell, since, by construction, we make sure that each point keeps a reference to a boundary segment if possible, so reaching the second boundary segment naturally means that we have gone all the way around the point. The iteration procedure is shown in Algorithm 14.

After going around the center and determining that we are done with this cell, we still have to check if we need further operations; most of the time we can just stop, create the cell, and proceed with the next, but when dealing with regions on the boundary we need to manually close them, as the Voronoi diagram normally assumes that they are infinite. So,

Algorithm 14: Iteration step of the algorithm until the cell has been closed

```
while edge!=initial_ edge do
    triangle2 ← triangle1
    triangle1← Edge.triangle different from triangle2
    c1 ← getCircumcenter(triangle1, index of edge)
    c2 ← getCircumcenter(triangle2, index of edge)
    Insert c1 and c2 in VoronoiPoints
    if c1 != c2 then
        Insert Segment(c2,c1) to cellEdges and c2 in cellPoints
    end
    Insert c1 in cellPoints
    if triangle1!=-1 then
        edge ← edges[triangle1.nextEdge]
    else
        Break
    end
end
```

we check whether we finished in a boundary segment using the current edge: if the value of the second triangle is -1, we are on the boundary, since, as we commented before, we make sure that in the case the segment has only one incident polygon, it will be always assigned in the first position.

To properly close boundary regions, we need to consider that the region can be closed with one segment or two, depending on the disposition of the Voronoi points and the center of the region, as can be seen in Figure 5.3. When one segment is needed, the center is not actually part of the diagram, as it is contained in the boundary segment; on the other hand, when we need two segments, the center itself becomes a part of the diagram, so we mark it as part of the boundary, and include it in the mesh points. The easiest, and more efficient, way of checking which case we are dealing with is to verify whether the boundary points of the cell (named *firstPoint* and *lastPoint* as by construction they will have those positions when traversing the points of the cell) and the center are collinear; when they are, the center is a part of the segment, and we can close the cell including that edge between the first and last point; when they are not, we require two extra segments, one uniting *lastPoint* to the center, and one the center to the *firstPoint*. This procedure is presented in Algorithm 15.

After the closing the boundary cells in Algorithm 15, the only thing left to do is to actually create a polygon that represents the cell, and add it to the mesh, process which includes inserting it in the list of polygons, and add each of its segments to the SegmentMap, with

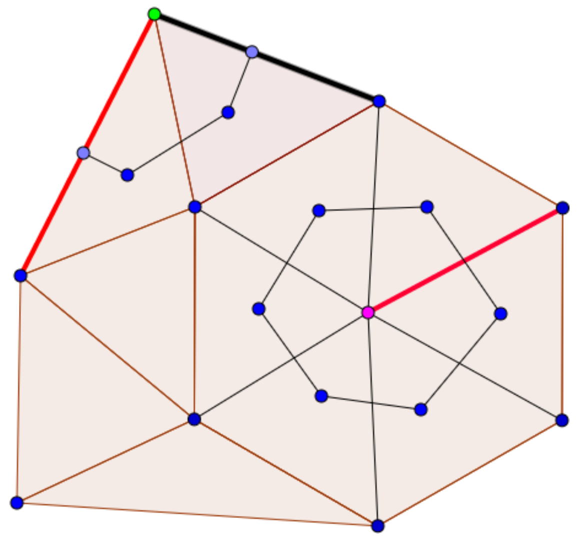


Figure 5.2: *Both stop conditions for Voronoi cells*

itself as incident polygon. This part of the Voronoi algorithm can be seen in Algorithm 16.

Algorithm 15: Boundary cases in Voronoi diagram generation

```
if edge.triangle2==-1 then
    firstPoint ← First element of cellPoints
    lastPoint ← Last element of cellPoints
    if firstPoint, lastPoint and region_center are collinear then
        Insert Segment (lastPoint, firstPoint) in cellEdges
    else
        Mark region_center as boundary point
        index ← Insert region_center in VoronoiPoints
        Insert index in cellPoint, and Segment(lastPoint, index) and Segment(index,
          firstPoint) in cellEdges
    end
end
```

Algorithm 16: Insert new polygon in mesh

```
Create Polygon cell with cellPoints
cellIndex ← Insert cell in VoronoiCell
for Segment s in cellEdges do
    Insert the pair (s, cellIndex) in VoronoiEdges
end
```

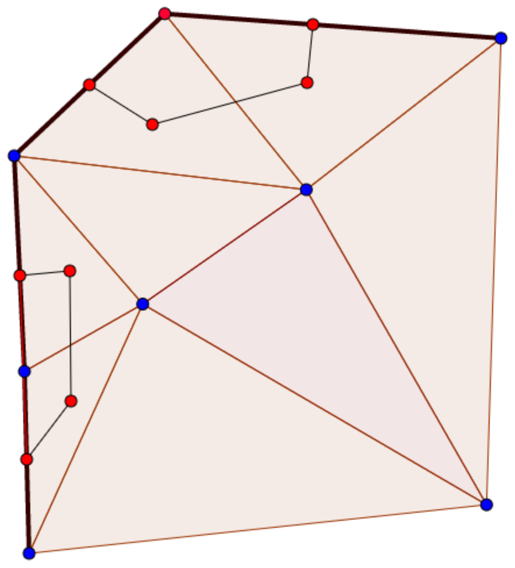


Figure 5.3: *Both configurations of boundary cells, with Voronoi points in red*

A final note of the algorithm is that, even considering that each Delaunay triangle keeps its own circumcenter, the function *getCircumcenter*, used in Algorithm 13 and Algorithm 14, is not as direct as one may think, so we present it in Algorithm 17, since we need to include the cases when the algorithm reaches the boundary; we consider that the circumcenter of a segment is its middle point. As we have commented before, these cases are easy to find, since it is just a matter of checking whether the triangle index given is different from -1.

Algorithm 17: Function *getCircumcenter*

```
Data: Triangle index $t$, Segment $s$
Result: Circumcenter $C$
if $t$ != -1 then
    return triangles[t].circumcenter
else
    middlePoint ← Compute middle point of $s$
    Mark middlePoint as a boundary point
    return middlePoint
end
```

5.1.2 Geometric algorithms

In this section we will go through several important geometric algorithms that are used to generate the Voronoi diagram or and left as utilities for another use, as they are generic enough to be considered part of the geometric modeling. We will use pseudo code to present the procedures, since their translation to our programming language of choice is more or less direct.

5.1.2.1 Self intersecting polygon test

One of the constraints of our polygons, as mentioned in Section 2.1.3.1, is that they cannot be self-intersecting, or some algorithms, such as the computation of the centroid, would give wrong results; differently from the other constraints that come from the chosen way of representing the polygons, this one needs to be checked every time a Polygon is created, and the program should end with a warning in the case the polygon being created is self-intersecting.

There are a good number of algorithms for the computation of the self-intersection points of a general polygon that work better than the naive approach, but as they are difficult to implement and prone to errors, we decide that they are impractical in our case; since the naive approach is $O(n^2)$ and the best algorithms are $O(nlogn)$, with n the number of vertices of the polygon, a value almost negligible when compared with the number of points of the mesh, we decide that the naive approach will suffice in our case.

The naive approach, shown in Algorithm 18, is, as it can be deduced from its name, extremely simple: we just check all the segments against each other for intersection, skipping those that are equal or connected; if we find two segments that intersect, this polygon is self-intersecting, but if we go through all possible pairs and none do, we can say that this polygon does not self-intersect.

Algorithm 18: Is a polygon self-intersecting

```
Data: Polygon $p$
Result: Boolean indicating if $p$ self-intersects
$segments \leftarrow p$.getSegments
$n \leftarrow segments$.size
for i in 0...$n$ do
    $s \leftarrow segments$[i]
    for $j$ in 0...$n$ do
        if i==$j$ or $j$==(i-1) or $j$==(i+1) then
            continue
        end
        $s_2 \leftarrow segments[j]$
        if $s.intersects(s_2)$ then
            return True
        end
    end
end
return False
```

5.1.2.2 Orientation test and fix

In computational geometry, the concept of orientation (intuitively known as the right-hand rule) is extremely common, since depending on whether a list of points is oriented counter-clockwise (positive) or clockwise (negative) the results of some algorithms can change. As some procedures create polygons, like the Voronoi diagram generation, for example, we have no guaranty that polygons are always correctly oriented; so, to make sure, we implement a method called $isClockwise$, in charge of determining the orientation of a list of points, and a second method, $fixCCW$, that uses the previous one, and reverses the order of the list of points if the polygon is found to be clockwise.

To determine whether a list of points is clockwise we use classic Shoelace formula, shown in Algorithm 19, a mathematical algorithm for compute the area of a non-self-intersecting polygon, but with sign depending on the orientation of the polygon. It is direct to see that this procedure has time complexity $O(n)$, since absolutely all operations are constant, and there is a single for-loop.

Finally, the $fixCCW$ method consists in a single call to Algorithm 19, and a reversing the order of the points of the polygon if they are found to be in clockwise order, which can be done with a standard C++ function,; however, we cannot forget to change the sign of the area, as it is the only pre-calculated polygon parameter that is orientation dependent. Hence, the complexity comes from the orientation test itself ($O(n)$) or the reverse function, which, according to the official documentation, is also linear, so the procedure is $O(n)$.

Algorithm 19: Clockwise orientation test

```
Data: Polygon $p$
Result: Boolean indicating if $p$ is oriented clockwise
$points \leftarrow p$.getPoints
$n \leftarrow points$.size
$p_{\mathrm{i}} \leftarrow points[0]$
$p_{\mathrm{i1}} \leftarrow points[n\text{-}1]$
$sum \leftarrow (p_{\mathrm{i}}.\mathrm{x}\text{-}p_{\mathrm{i1}}.\mathrm{x})(p_{\mathrm{i}}.\mathrm{y}+p_{\mathrm{i1}}.\mathrm{y})$
for i $in$ $0...n-1$ do
    $p_{\mathrm{i}} \leftarrow points[\mathrm{i}]$
    $p_{\mathrm{i1}} \leftarrow points[\mathrm{i}+1]$
    $sum \leftarrow sum + (p_{\mathrm{i}}.\mathrm{x}\text{-}p_{\mathrm{i1}}.\mathrm{x})(p_{\mathrm{i}}.\mathrm{y}+p_{\mathrm{i1}}.\mathrm{y})$
end
return $sum>0$
```

5.1.2.3 Ear clipping triangulation

Alongside the Delaunay triangulation scheme that we used to generate the Voronoi diagram, we include another, simpler one, called ear-clipping triangulation; it has the advantage that is simple to understand, quick to implement, and does not have the overhead of calling Triangle, so it is used in some applications in the VEM library. We create a class, called EarTriangulationGenerator, with one public method called *triangulate*, which receives a Polygon instances and returns a list of Triangles, its triangulation; the class has a private method called *getEar*, which does most of the work. We show the implementation of both *triangulate* and *getEar* in Algorithm 20 and Algorithm 21, respectively.

Algorithm 20: Ear clipping triangulation

```
Data: Polygon $p$
Result: List of triangles $triangles$
Create empty list of Triangles $triangles$
$pointList \leftarrow p$.getPoints
while Size of $pointList \geq 3$ do
    $ear \leftarrow$ getEar($pointList$)
    if $ear$ is Null then
        Break
    end
    Insert $ear$ in $triangles$
end
return $triangles$
```

Algorithm 21: Computing an ear for the ear clipping triangulation

```
Data: List of points defining a polygon $points$
Result: Triangle $ear$
if Size of $points < 3$ then
    return Null
end
if Size of $points$ is 3 then
    return Triangle($points$)
end
for i=0 to Size of $points$ do
    $test \leftarrow$ False
    Create $t$ as Triangle($points$[i-1], $points$[i], $points$[i+1])
    if $t$ is convex then
        for $j$=0 to Size of $points$ do
            if $points[j]$ is not vertex of $t$ and $t$ contains $point[j]$ then
                $test \leftarrow$ True
            end
        end
        if not $test$ then
            Erase point of index i from $points$
            return $t$
        end
    end
end
return Null
```

The theoretical cost of the ear clipping triangulation is $O(n^2)$. It is easy to check that our implementation also has that cost in the worst case, as the main operation (shown in Algorithm 20) has a single iteration on each point of the polygon, calling the ear computation operation each time; this last procedure iterates on all points of the polygon checking a condition on a triangle (which, as they have significantly fewer vertices than polygons, are considered constant time).

5.1.3 Utilities

There are a number of functions, structs and classes that are not considered an essential part of the modeling of the Voronoi generation library, but rather are used for single objectives. Such utilities include the implementation of the algorithm for the convex hull, that we have already gone through in detail in Algorithm 7, some functions that are easily classified as "geometric", and general functions, that are either limited to this library or are operations on parameters of the library.

First, the geometric functions are several versions (with different parameters) of the collinearity test, and polygon area computation, plus a simple function that returns the area of a triangle, all of them implemented following the pseudo codes shown in Section 2.1.4.

Finally, the general utilities are self-explanatory geometric concepts, shown in Code 5.3.

Code 5.3: *General utilities of Voronoi generation library*

```
double crossProduct(Point a, Point b){
    return a.getX()*b.getY() - a.getY()*b.getX();
}

double norm(Point p){
    return sqrt(pow(p.getX(),2) + pow(p.getY(),2));
}

double orientation(Point p, Point q, Point r){
    return crossProduct((q-p),(r-p));
}
```

5.1.4 Configurations

There are a number of parameters that are used in the library that we would like to leave open for the user to change if they want to, but also have a default value if not; to do this, we implement a configuration class. Since we will need to instantiate this class several times in different classes, we use the Singleton design pattern, that allows us to create a single instance, and save memory.

The class itself is simple, as it is a container of configuration constants; we leave setters to allow the users to change whatever they deem necessary, and getters to obtain those values. The configuration parameters are:

- Circles are continuous and need to be discretized with a certain number of points on its boundary. We call this number the *discretization grade*, and leave 12 as its default value.
- As we have commented several times throughout this thesis, precision problem, or how to compare real numbers considering machine errors, are incredibly common in our area of work; when we compare two numbers we do not use equality, but instead do the following:

Code 5.4: *Comparison of doubles using precision tolerance*

```
    Config* config = Config::instance();
    std::abs(a - b)<config->getTolerance();
```

- As we detailed in Section 5.1.1.1, we use Clipper, for which we need a certain parameter to scale our double precision numbers to integers; by testing, we notice that 100.000 is around the upper limit that allows us to keep most of the precision, and not have problems with Clipper.
- As we commented several times throughout the design and implementation of this library, we leave the possibility of printing any generated mesh in a text file, which can later be used as seen fit; it is not direct, however, to decide the number of decimals, as C++ as certain number by default. This way, we include a configuration parameter called precision, which determines the number of decimals that will be shown when printing any element of the library in a file.

5.2 Virtual Element Method Library

This section features the implementation of the Virtual Element Method for elasticity library, referred as VEM library; we organize the information as follows: we first show how we initialize the numerical system (creating elements and degrees of freedom to represent the problem), and later, using the classes created previously, we present the computation of the displacements. We finish the chapter explaining how, using all tools included for VEM, we also implement the Finite Element Method for linear elasticity.

5.2.1 Initialization of the system

Veamer is the class in charge of taking the mesh and creating the elements; it takes each mesh polygon, and starts the process of creating its associated element, shown in Figure 4.2. As the previous process goes on, DOFs are created, and assigned to each point of the mesh, and Constraints are filled with the DOFs that are inside constrained segments.

Code 5.5: *Creation of the VEM elements in Veamer*

```
for(Polygon p: mesh.getPolygons()){
    this->elements.push_back(VeamyElement(conditions, p, points, DOFs));
}
```

When we create an element, it first keeps the related polygon as a class parameter, and then iterates through each vertex of the polygon, creating two DOFs; VeamyElement delegates all responsibility of this operation to the DOFS structure, using the method *addDOF*, taking the result of the operation, the indexes of the DOFs associated to the vertex, and adds it to the list of DOFs of the Element. The *addDOF* method takes the vertex index, the ConstraintsContainer and the pair of segments incident to the given vertex, and checks whether it is necessary to create DOFs or not, depending on the already existent information; DOFS keeps both a list of the DOFs instances of the system and a map to ease the cost of searching; this map relates a point index to the index of one of its associated DOF (the one with x axis, since the one with y is the one with the consecutive index); this way, when we want to add a DOF related to a certain point, we just check in the map for the index, and if a match is found, we return directly. We show part of the code of the *addDOF* method in Code 5.6.

Code 5.6: *Creation of DOF associated to point_index*

```
1   std::unordered_map<int,int>::iterator it =
2                      occupied_point_indexes.find(point_index);
3
4  if(it!=occupied_point_indexes.end()){
5      int DOF_index = it->second;
6
7      return Pair<int>(DOF_index, DOF_index+1);
8  }
9
10 DOF newDOF = DOF(list.size(),point_index, DOF::Axis::x);
11 DOF newDOF2 = DOF(list.size()+1,point_index, DOF::Axis::y);
12
13 int newIndex = list.push_back(newDOF);
14 list.push_back(newDOF2);
15
16 constraints.addConstrainedDOF(points, newIndex, DOF::Axis::x, pair);
17 constraints.addConstrainedDOF(points, newIndex+1, DOF::Axis::y, pair);
18
19 occupied_point_indexes[point_index] = newIndex;
20 Pair<int> dofPair(newIndex, newIndex+1);
21
22 return dofPair;
```

The code from line 10 forward of Code 5.6 refers to the fact that, when a new DOF is created, it is necessary to check whether it is constrained or not, and act accordingly. This procedure is done calling the ConstraintsContainer class method *addConstrainedDOF*, simply calls the same method of both EssentialConstraints and NaturalConstraints, which is exactly the same as they both inherit from Constraints, which implements the method. Finally, *addConstrainedDOF* does several checks, as the degree of freedom could be constrained because they define a constrained segment, are a point contained in a constrained segment, or coincide with a constrained point. This way, *addConstrainedDOF* calls three different methods: first it calls *addConstrainedDOFBySegment* twice, once for each incident segment to the point that is used to create this DOFs, which does the required task in case the DOF is constrained by segment; this method returns a Boolean indicating whether the DOF was or not constrained. In case both tests return false, the method *checkIfContainedInConstraint* is called, in charge of checking whether the point the DOF represents is contained in a constrained segment; this method, in case it finds a constrained segment containing the point, also constrained the DOF. Finally, the method *addConstrainedDOFByPoint* is called, which does the exact same operation as *addConstrainedDOFBySegment* but with the PointConstraints.

The code for *addConstrainedDOFBySegment* has several aspects that make it worthwhile to be shown; we will first present it in Code 5.7, and then comment on those points.

Code 5.7: *Adding DOF to constraints*

```
isConstrainedInfo info = isConstrainedBySegment(points, s);

if(info.isConstrained){
    std::vector<SegmentConstraint> constraints = segment_map[info.
       container];

    for (SegmentConstraint constraint: constraints) {
        Constraint::Direction = constraint.getDirection();

        bool insert;

        switch(direction){
            case(Constraint::Direction::Vertical):
                insert = axis==DOF::Axis::y;
                break;
            case (Constraint::Direction::Horizontal):
                insert = axis==DOF::Axis::x;
                break;
            default:
                insert = true;
        }

        if(insert){
            constrained_dofs.push_back(DOF_index);
            segment_constraints_map[DOF_index] = constraint;
        }
    }
}
```

The method *isConstrainedBySegment*, seen in Code 5.7, checks if the segment s, which contains the point associated to the new DOFs, is constrained, meaning, that it is contained (or equal) to a constrained domain segment; if that is the case, this method returns a struct that has the relevant constrained segment and a flag to indicate if a segment was found. When the flag is true, we look for the SegmentConstraint related to the constrained segment, and decide if the DOF is indeed constrained or not (by checking whether its direction is affected by the constraint); finally, the DOF index is inserted in the constrained DOFs list, and also included as a key, with the corresponding SegmentConstraint as the value, in the map that relates DOFs and SegmentConstraint. The concrete implementation of the *isConstrainedBySegment* method can be seen in Code 5.8.

Code 5.8: *Determining if a segment is constrained*

```
Angle angle(s.cartesianAngle(points));
auto iter = constrained_segments.find(angle);
```

```

if(iter == constrained_segments.end())
    return isConstrainedInfo();

std::vector<IndexSegment> segments = iter->second;

for (int i = 0; i < segments.size(); ++i) {
    if(segments[i].contains(points,s)){
        return isConstrainedInfo(segments[i]);
    }
}

return isConstrainedInfo();
```

To improve performance of *isConstrainedBySegment* in the case there are many constrained segments, we decide to think beyond a simple linear search, and create a data structure that diminishes the complexity. To this effect, when the Constraints are created, we do not keep the constrained segments on a list, but fill a map that has the angle of the segment as the key, and a list of all the constrained segments with this angle as a value; this way, our linear search is done in a much smaller set of segments. Nonetheless, the angle of the segment is a sensitive matter, as Cartesian angles equality requires to compute equivalent angles (for example, 90° is equal to 270°); to simplify the implementation, and handle precision errors, we use a structure called Angle that computes equivalent angles when created.

5.2.2 Computation of the displacements

In this section we will go through the steps required for the computation of the nodal displacements given certain conditions, following the steps shown in Section 2.2.4.1, but starting from the second, as the first one refers to mesh generation, aspect completely handled by the Voronoi generation library.

Although not completely intuitive, it is decided that each element must oversee both computing and assembling the stiffness matrix and load vector; this choice is justified on object oriented programming and the fact that each element has all the information necessary for the computations, and are completely independent from each other. Hence, Veamer just asks each element to compute the required values and assemble them in the global ones, without knowing any details of the elements. It is important to note that we have mentioned the force vector in this section because of its parallel to the elemental matrix in when it is computed and saved, but the actual process of computing it is completely different, so we will detail it in its own subsection.

5.2.2.1 Calculation of the elemental stiffness matrix

The elemental stiffness matrix in VEM is easily computed using the analytic expressions that were presented in Section 2.2.4.2. Hence, the implementation of this step is relatively simple, as it just requires to replace values on a formula.

To calculate the relevant matrices (W_R, W_C, N_R and N_C) we need the centroid of the polygon, already available from the geometric library, and $q_{\rm i}$, obtained as shown in Code 5.9.

Code 5.9: *Calculation of the relevant terms for* $K_{\rm e}$

```
    Point vertex = points[polyPoints[vertex_id]];

    Edge prev (polyPoints[(n+vertex_id-1)%n], polyPoints[vertex_id]);
    Edge next (polyPoints[vertex_id], polyPoints[(n+vertex_id+1)%n]);

    Pair<double> pN = utilities::normalize(prev.getNormal(points));
    Pair<double> nN = utilities::normalize(next.getNormal(points));

    double pL = prev.getLength(points);
    double nL = next.getLength(points);

    double xDiff = vertex.getX() - average.getX();
    double yDiff = vertex.getY() - average.getY();

    double qi_x = (pN.first*pL + nN.first*nL)/(4*area);
    double qi_y = (pN.second*pL + nN.second*nL)/(4*area);
```

One important detail that we have not mentioned until now, and is extremely important in Code 5.9, is that all norms for matrix computations must be normalized, otherwise, the results are completely distorted.

Finally, using the values obtained from Code 5.9, calculating the elemental matrix is straightforward, using the formulas presented in Equation (2.29) and Equation (2.33).

5.2.2.2 Calculation of the elemental load vector

The elemental load vector has two different parts: the surface integral from the body forces, and the line integral from all the natural boundary conditions on the system, calculated separately, and then added to obtain the required value. We use a BodyForceVector to compute the surface integral, and a TractionVector for the line integrals related to the traction loads, which respective methods, *computeForceVector* and *computeTractionVector*, we show the two following subsections. The computation of the load vector for a given Element using the previous methods is presented in Code 5.10; this method receives as parameters the BodyForceVector and TractionVector instances used to compute the vector, which, in the case of VeamyElement, are VeamyBodyForceVector and VeamyTractionVector.

Code 5.10: *Computation of the elemental load vector in Element*

```
int n = this->p.numberOfSides(), m = this->dofs.size();
std::vector<IndexSegment> segments;
this->p.getSegments(segments);

this->f = Eigen::VectorXd::Zero(m);
Eigen::VectorXd bF = bFVector->computeForceVector(conditions.f);

for (int i = 0; i < n; ++i) {
    Eigen::VectorXd natural = tVector->computeTractionVector(segments[i]);

    this->f(2*i) = this->f(2*i) + bF(2*i) + natural(0);
    this->f((2*i + 1)%m) = this->f((2*i + 1)%m) + bF(2*i+1) + natural(1);
    this->f((2*(i+1))%m) =  this->f((2*(i+1))%m) + natural(2);
    this->f((2*(i+1) + 1)%m) =  this->f((2*(i+1) + 1)%m) + natural(3);
}
```

We compute the body force vector outside the for-loop since its value is dependent on the Element, not each DOF, so it is always the same.

Body Force Vector For the numerical integration of the body force, we choose to follow Gain et.al's approach[1] and use a simple approximation, that only uses geometrical information but has a first order accuracy (it is only able to integrate exactly constant and linear fields); nonetheless, the fact that it does not require any information whatsoever about the shape functions inside the elements is extremely valuable, since it preserves the spirit of VEM. Moreover, the authors cited [1] explain that, for fields sufficiently smooth, the error in replacing the exact form of the load vector for this first order approximation is linear, conserving the convergence rate.

The procedure *computeForceVector* of the VeamyBodyForceVector class is the one in charge of receiving a BodyForce and computing the vector; however, we consider the integration using Gain et. al's approach to be valuable enough to be implemented as a separated function. This way, *computeForceVector* (shown in Code 5.11) is simple, just having to call the nodal integration function twice (which can be seen in Code 5.12), and then assembling the result in an Eigen vector.

Code 5.11: *Computation of the elemental body force vector*

```
int n = this->polygon.numberOfSides();
double integralX = nodal_quadrature(polygon, points, f->getX());
double integralY = nodal_quadrature(polygon, points, f->getY());;

Eigen::VectorXd result;
result = Eigen::VectorXd::Zero(2*n);

for (int j = 0; j < n; ++j) {
    result(2*j) = integralX;
    result(2*j+1) = integralY;
}

return result;
```

Code 5.12: *Implementation of the nodal integration function*

```
double nodal_quadrature(Polygon poly, std::vector<Point> points,
   Computable* f) {
    double result = 0;
    std::vector<int> polygonPoints = poly.getPoints();
    int n = (int) polygonPoints.size();

    IndexSegment prev, next;

    for (int i = 0; i < n; ++i) {
        prev = IndexSegment(polygonPoints[(n+i-1)%n], polygonPoints[i]);
        next = IndexSegment(polygonPoints[i], polygonPoints[(i+1)%n]);

        Point = points[polygonPoints[i]];
        std::vector<Point> small = {prev.middlePoint(points),
                                    points[polygonPoints[i]],
                                    next.middlePoint(points),
                                    poly.getCentroid()};

        double area = geometry_functions::area(small);
        result += area*f->(point.getX(), point.getY(), polygonPoints[i]);
    }

    return result;
}
```

Traction Vector As can be seen in Code 5.10, differently from the body force vector, the traction vector related to the natural boundary conditions needs to be computed for every degree of freedom separately. The task, in VEM, is done by VeamyTractionVector; however, it shares a great part of the responsibilities with NaturalConditions, as it is the class responsible to determine whether the DOF in question is constrained or not, and then check, for each incident edge to the DOF point, if they are constrained as well, since, by construction, at least one must be.

Determining if a DOF is constrained is direct from Constraints, but the check for an edge is not; we use the same method as shown in Section 5.2.1, using the angle of the edge to narrow down the search. It is important to note that the search is more important as one would think, as we also need it search to determine the ConstraintValue related to the segment, so we can obtain the value to integrate. Code 5.13 shows the procedure of computing the traction vector related to the natural boundary conditions, for a given DOF, with index i, associated Point of index p and Polygon *poly*.

Code 5.13: *Computation of the line integral*

```
Eigen::VectorXd result(4);
isConstrainedInfo constrainedInfo = natural.isConstrainedBySegment(points,
    segment);

if(constrainedInfo.isConstrained){
    std::vector<int> polygonPoints = p.getPoints();
    int n = (int) polygonPoints.size();

    std::vector<SegmentConstraint> constraints = natural.
       getConstraintInformation(constrainedInfo.container);

    Eigen::MatrixXd Nbar;
    Nbar = Eigen::MatrixXd::Zero(2,2);
    Nbar(0,0) = 1.0/2;
    Nbar(1,1) = 1.0/2;

    Eigen::VectorXd hFirst, hSecond;
    hFirst = Eigen::VectorXd::Zero(2), hSecond = Eigen::VectorXd::Zero(2);

    for(Constraint c: constraints){
        hFirst(0) += c.getValue(points[segment.getFirst()])*
                     c.isAffected(DOF::Axis::x);
        hFirst(1) += c.getValue(points[segment.getFirst()])*
                     c.isAffected(DOF::Axis::y);

        hSecond(0) += c.getValue(points[segment.getSecond()])*
                      c.isAffected(DOF::Axis::x);
        hSecond(1) += c.getValue(points[segment.getSecond()])*
                      c.isAffected(DOF::Axis::y);
    }

    double length = segment.length(points);
    Eigen::VectorXd resultFirst = length*(Nbar.transpose()*hFirst);
    Eigen::VectorXd resultSecond = length*(Nbar.transpose()*hSecond);

    result << resultFirst, resultSecond;
} else {
    result = Eigen::VectorXd::Zero(4);
}

point_forces::addPointForces(result, natural, points[segment.getFirst()],
   points[segment.getSecond()]);

return result;
```

The method *isAffected* of class Constraint, that can be seen in Code 5.13, is required because of how we choose to model constraints; as the constraints given by the user are associated to a given axis (x, y or *both*), when we compute the traction we are as general as possible and, instead of checking one by one if each constraint affects a given DOF, we choose to always assume that is the case, and use *isAffected* to handle the different cases. This way, *isAffected* is a simple function that checks if the given DOF coincides with the DOF associated to the Constraint, and returns one when that is the case, or zero if it is not.

5.2.2.3 Assembly

The matrix assembly algorithm shown in Algorithm 9 can be greatly simplified in our case, using a combination of our data structures, and the fact that we are strictly bounded to 2D problems. The most relevant observation that allows this simplifications is that each position of the elemental matrix (and load vector) represents a degree of freedom of the element, changing vertex every two degrees (the first row is the degree associated with the x axis and the second, the y axis), in the exactly same order in which we created the DOFs previously; this idea allows us to iterate through the rows and columns of the elemental matrix and know exactly which degree of freedom they represent. With the DOF instance at hand it is simply a matter of asking its global index to assembly the matrix.

It is important to note that, differently from most finite element implementations, we do not assemble the matrix and vector in the central class (Veamer in our case), but rather choose to delegate the assembly procedure to the elements themselves, as they have the information of their own degrees of freedom; this way, in Code 5.14, we can see how the Element class implements the assembly procedure, using its own DOFs index list, and the DOFS instance, appropriately named *DOFS*.

Code 5.14: *Assembly of global matrix and vector in Element*

```
 for (int i = 0; i < this->K.rows(); i++) {
    int gI = DOFS.get(this->dofs[i]).globalIndex();

    for (int j = 0; j < this->K.cols(); j++) {
        int gJ = DOFS.get(this->dofs[j]).globalIndex();

        Kg(gI, gJ) = Kg(gI, gJ) + this->K(i, j);
    }

    Fg(gI) = Fg(gI) + this->f(i);
}
```

Finally, as shown in Code 5.14, we also assemble the load vector at the same time, saving computation time.

5.2.2.4 Boundary conditions

We have already talked about how the DOFs are created and each assigned to its corresponding point, and then related to a Constraint instance if necessary, and how natural boundary conditions are included directly in the computation of the load vector; hence, before solving the linear system itself, the only aspect left is the imposition of the essential boundary conditions.

Essential boundary conditions give literal solutions to certain degrees of freedom, which values that need to be manually imposed, with the DOFs taken out of the problem; as we saw before, each row and column of the stiffness matrix represents a DOF, and, as we have the list of the constrained ones, it is simply a matter of iterating through them and obtaining the related constraint value. Finally, to impose a solution to our system, for each essentially constrained degree of freedom, we change all the values of the related row and column for zeros, except for the one located in the diagonal; to keep the symmetry of the stiffness matrix, which makes the system easier to solve, an extra operation is needed: to each element of the load vector we subtract the value changed of the stiffness matrix times the boundary value imposed.

Since this operation does not require information about the Elements, is done after computing the global stiffness matrix, and does not require knowledge of the method implemented, it is decided that Calculator2D should be in charge of this procedure, so that both FEM and VEM can take advantage of this implementation. The code showing this operation is shown in Code 5.15.

Code 5.15: Imposition of the essential conditions

```
EssentialConstraints e = conditions.constraints.getEssentialConstraints();
std::vector<int> c = e.getConstrainedDOF();

Eigen::VectorXd boundary = e.getBoundaryValues(points,DOFs.getDOFS());

for (int j = 0; j < c.size(); ++j) {
    for (int i = 0; i < K.rows(); ++i) {
        f(i) = f(i) - (K(i,c[j])*boundary(j));
        K(c[j],i) = 0, K(i,c[j]) = 0
    }

    K(c[j], c[j]) = 1;
    f(c[j]) = boundary(j);
}
```

5.2.2.5 Linear system solving

The last important step in the VEM simulation is to solve the linear system; as the range of the global stiffness matrix is in no way assured, using its inverse can lead to failure in a lot of cases, which was actually encountered several times during the development. To solve this problem, we first investigate how MATLAB solves linear problems, as it is usually the language of choice in most FEA, and it is known to solve almost any system with good stability. We discover that MATLAB's backslash operator makes a series of checks to decide which method would most likely work for each problem; funnily, the first check is whether the matrix is square or not, and, as by construction our matrix is always square, we can use the first option given by MATLAB, a QR decomposition solver.

Finally, after further investigating, we discover that Eigen provide us with two tools related to this process, a full pivoting or column pivoting QR solver, the first being more stable but slower; we choose the first one after testing the results, and concluding that the stability helps to better fulfill the essential boundary conditions.

5.2.3 Implementing the Finite Element Method

As we have already visited all the required elements for the implementation of the Virtual Element Method, explaining also all abstract interfaces that are involved, detailing the implementation of the Finite Element Method is relatively simple. We follow the same steps, first showing how the numerical system is initialized, and then how the displacements are computed.

5.2.3.1 Initialization of the system

Feamer is the class analogue to Veamer, so its implementation is extremely similar, with the only difference that FEM does not create the elements directly, as there are many different kinds. Hence, we add an extra degree of indirection and instead of creating the elements in Feamer, we use an additional class, called a FeamyElementConstructor, which the user provides when creating a Feamer instances; this class instance completely defines the type of element that will be used, so it is expected to be provided externally. This way, the creating of the FeamyElement is implemented as shown in Code 5.16; besides this small difference, all other processes detailed in Section 5.2.1 are the exact same, and will not be repeated.

Code 5.16: *Creation of the elements in Feamer*

```
for(Triangle t: mesh.getPolygons()){
    this->elements.push_back(constructor->createElement(conditions, t,
                                                         points, DOFs));
}
```

5.2.3.2 Numerical integration

We require to implement a Gaussian numerical integration for the computation of both the stiffness matrix and load vector, but decide to leave it generic so it can be reused for other ends; even more, this choice was proven valuable during the development, as numerical integration is required for the calculation of the norms in FEM. We implement numerical line integration, necessary for the load vector, and area integrator, for the stiffness matrix, in two template classes named AreaIntegrator and LineIntegrator; each one has a method named *integrate*, which receives an IntegrableFunction pointer, representing the function one needs to integrate, and an instance of either a VeamyTriangle or a VeamyPolygon. It is important to remark on the fact that leaving both classes as template give us flexibility in the signature of the functions that can be given to the relevant methods, so that our integration scheme can integrate both multivariable or single variable functions.

We show the procedure that integrates a given function inside an area in Code 5.17; the line integration method is relatively simple, and similar enough to what we show to not be relevant to presented in detail.

Code 5.17: *Numerical area integration*

```
std::vector<VeamyTriangle> triangles = element.triangulate(points);

for(VeamyTriangle t: triangles){
    std::vector<Point> p = t.getPoints(points);
    Eigen::MatrixXd J = t.getJacobian(points);

    std::vector<Point> gaussPoints;
    std::vector<double> weights;
    gauss_quadrature::gauss_triangle(nGauss, gaussPoints, weights);

    for (int i = 0; i < gaussPoints.size(); ++i) {
        double g1 = p[0].getX() +
            (p[1].getX() - p[0].getX())*gaussPoints[i].getX()  +
            (p[2].getX() - p[0].getX())*gaussPoints[i].getY();
        double g2 = p[0].getY() +
            (p[1].getY() - p[0].getY())*gaussPoints[i].getX()  +
            (p[2].getY() - p[0].getY())*gaussPoints[i].getY();

        result += weights[i] * integrable->apply(Point(g1, g2), t) *
            J.determinant();
    }
}
```

Most of the implementation is self-explanatory, but there are a number of details that require explanation. First, we allow numerical integration in all types of polygons, not only triangles, but there are no known integration rules for generic shapes; instead, we triangulate every polygon we receive (or return the same polygon in the case it was already a triangle) and calculate the integral value inside each triangle. Second, we include inside the library a list of the quadrature points and weights for the iso-parametric triangle, but it is highly likely that our triangles have a different shape; hence, before evaluating the function we need to convert the iso-parametric gauss points to their equivalent inside our real triangle.

5.2.3.3 Computation of the displacements

The flow line of the displacement computation for VEM is exactly the same for FEM, which explains why we did not commented further on the subject in the design chapter; concretely, the process is exactly equal except for how the elemental stiffness matrix and load vector are computed. Hence, in this subsection we will explain how both relevant values are computed, and assume that the rest of the steps of the FEM process are known.

Calculation of the elemental stiffness matrix Parallel to VeamyElement, FeamyElement is in charge of computing the elemental stiffness matrix; however, differently from VEM, in FEM numerical integration is required, so FeamyElement mostly delegates responsibility to AreaIntegrator and a concrete implementation of IntegrableFunction, named StiffnessMatrixIntegrable, to compute the integral and calculate the matrix Be, respectively. We show the implementation of the method *computeK* in Code 5.18.

Code 5.18: *Computation of the stiffness matrix in FeamyElement*

```
int n = this->N->numberOfShapeFunctions();
K = Eigen::MatrixXd::Zero(2*n, 2*n);

VeamyTriangle t(this->p);
IntegrableFunction<Eigen::MatrixXd>* i =
  new StiffnessMatrixIntegrable(t, points.getList(),
                                conditions.material->getMaterialMatrix(),
                                this->N);

AreaIntegrator<VeamyTriangle,Eigen::MatrixXd>::integrate(K, nGauss, t,
                                                          points.getList(),
                                                              i);

delete integrable;
```

We show the implementation of the *apply* method of StiffnessMatrixIntegrable, which is used inside *integrate* and does most of the actual computation of the stiffness matrix, in Code 5.19.

Code 5.19: *Computation of the term inside the stiffness matrix integral*

```
Eigen::MatrixXd Be;
Be = Eigen::MatrixXd::Zero(3, N->numberOfShapeFunctions()*2);

std::vector<Pair<double>> dN = N->evaluateDerivatives(p);
Eigen::MatrixXd invJ = J.inverse();

for (int v = 0; v < N->numberOfShapeFunctions(); ++v) {
    Be(0,2*v) = dN[v].first*invJ(0,0) + dN[v].second*invJ(0,1);
    Be(1,2*v+1) = dN[v].first*invJ(1,0) + dN[v].second*invJ(1,1);
    Be(2,2*v) = dN[v].first*invJ(1,0) + dN[v].second*invJ(1,1);
    Be(2,2*v+1) = dN[v].first*invJ(0,0) + dN[v].second*invJ(0,1);
}

return Be.transpose()*D*Be;
```

Calculation of the elemental load vector Again, in parallel with FeamyElement, the calculation of the load vector is handled by the concrete implementation of two abstract classes, named FeamyBodyVector and FeamyTractionVector, which are created inside the FeamyElement and used in Code 5.10; hence, we will just show how these classes are implemented, and not repeat the aspects that are equal to what we shown for VEM. We first present the implementation of the method *apply* of class FeamyBodyVector, in Code 5.20; the code is sufficiently self-explanatory to not require further comment.

Code 5.20: *Computation of term inside the body force vector integral*

```
int n = N->numberOfShapeFunctions();
Eigen::MatrixXd Ne;
Ne = Eigen::MatrixXd::Zero(2,2*n);

std::vector<double> Ni = N->evaluateShapeFunction(p);
for (int i = 0; i < n; ++i) {
    Ne(0,2*i) = Ni[2*i];
    Ne(1,2*i+1) = Ni[2*i+1];
}

Point real = this->N->evaluateRealPoint(p);

Eigen::VectorXd b;
b = Eigen::VectorXd::Zero(2);
b(0) = f->getX()->apply(real.getX(), real.getY());
b(1) = f->getY()->apply(real.getX(), real.getY());

return Ne.transpose()*b;
```

The implementation of the *apply* method of FeamyTractionVector is shown in Code 5.21; it is important to note that, although the code is similar, the traction vector is computed in the boundary of the elements (so it must be integrated on a line), while the body vector is calculated inside the complete element.

Code 5.21: *Computation of term inside the traction vector integral*

```
std::vector<double> Ni = this->N->evaluateShapeFunctionCartesian(p);
Eigen::MatrixXd N;
N = Eigen::MatrixXd::Zero(2,2*this->indexes.size());

for (int i = 0; i < 2; ++i) {
    N(0,2*i) = Ni[indexes[i]];
    N(1,2*i+1) = Ni[indexes[i]];
}

Eigen::VectorXd t;
t = Eigen::VectorXd::Zero(2);
t(0) = c.getValue(p)*c.isAffected(DOF::Axis::x);
t(1) += c.getValue(p)*c.isAffected(DOF::Axis::y);

return N.transpose()*t;
```

As it was the case with the computation of the traction vector on VEM, we make use of the method *isAffected* of the Constraint class to be able to distinguish loads that affect only one axis associated to a given point.

5.2.4 Post processing

As commented in the background and design chapters, we calculate two measures of the error of the solution, which is considered an operation common enough to be included inside the library itself. For both VEM and FEM, we compute the L2 and H1 norms using the formula and information presented in Section 2.2.5.2; although the methods have their difference in this implementation, they are similar enough for most of the procedure to be shared. Moreover, the procedure to compute the norms is actually simple after the analysis we show next.

In the first place, we note that the calculation of the norm itself, assuming that the integrals are known, is extremely direct, consisting in an iteration over each element of the mesh, with some simple sums; this procedure is implemented in NormCalculator and shown in Code 5.22, where *num* and *den* are pointers to NormIntegrator instances.

Code 5.22: *Generic computation of the norms*

```
double numerator = 0, denominator = 0;
std::vector<T> meshElements = mesh.getPolygons();
std::vector<Point> points = mesh.getPoints().getList();

for(int i = 0; i<meshElements.size(); i++){
    numerator += num->getIntegral(meshElements[i], i, points);
    denominator += den->getIntegral(meshElements[i], i, points);
}

return std::sqrt(numerator/denominator);
```

Second, the integral computation, for both VEM and FEM, can be reused from their respective integration methods, so both VeamyIntegrator and FeamyIntegrator consist in a number of arithmetic calculations and a call to the relevant integration method. The only relevant NormIntegrator class to show is VeamyH1Integrator (Code 5.23), since it does not actually compute an integral, but rather follows the formula shown in Equation (2.39); we note from the same formula that the VEM version of the H1 norm is not a fraction (as it is the case with all the rest), hence, to follow the pattern shown in NormIntegrator, we include the IdentityIntegrator class, which *getIntegral* method always returns one.

Code 5.23: *Computation of the H1 norm for VEM*

```
std::vector<IndexSegment> segments;
poly.getSegments(segments);

double result = 0;

for(IndexSegment s: segments){
    Pair<double> u1 = value->getValue(points[s.getFirst()]);
    Pair<double> u2 = value->getValue(points[s.getSecond()]);

    Pair<int> point1_dofs = dofs.pointToDOFS(s.getFirst());
    Pair<int> point2_dofs = dofs.pointToDOFS(s.getFirst());
    Pair<double> uH1 = Pair<double>(nodalValues[point1_dofs.first],
        nodalValues[point1_dofs.second]);
    Pair<double> uH2 = Pair<double>(nodalValues[point2_dofs.first],
        nodalValues[point2_dofs.second]);

    Pair<double> e1 = u1 - uH1;
    Pair<double> e2 = u2 - uH2;

    result += (std::pow(e1.first - e2.first, 2) +
        std::pow(e1.second - e2.second, 2))/s.length(points);
}

return poly.getDiameter()*result;
```

For the rest of the norms (L2 for both methods and H1 for FEM), the actual work of the NormIntegrator is to call the *apply* method of each Computable instance, which calculate, for a given point, the value inside the integral in Equation (2.37) and Equation (2.38). We start with the Computable classes related to the L2 norm: in Code 5.24 we show the norm of the difference of the displacements, while the norm of the displacements is shown in Code 5.25.

Code 5.24: *Computation of the norm of the displacement difference*

```
Pair<double> u = value->getValue(Point(x,y));
calculator->setPolygonIndex(this->polygonIndex);
Pair<double> uH = calculator->getDisplacement(x, y, index, container);

return std::pow(u.first - uH.first, 2) + std::pow(u.second - uH.second, 2)
    ;
```

The most relevant step of Code 5.24 is the call to the *getDisplacement* method; this method belongs to the DisplacementCalculator class, and its two concrete implementations Veamy-DisplacementCalculator and FeamyDisplacementCalculator, and is in charge of taking the point information and compute the related displacement. For VEM, VeamyDisplacementCalculator takes the index of a mesh point and returns its displacements directly; for FEM, FeamyDisplacementCalculator takes the point and the container polygon, and calculates the interpolated displacement (procedure shown in Code 5.26).

Code 5.25: *Computation of the norm of the displacement*

```
Pair<double> u = value->getValue(Point(x,y));
return std::pow(u.first,2) + std::pow(u.second,2);
```

Code 5.26: *Computation of the approximated displacement in FEM*

```
double uHx = 0, uHy = 0;
std::vector<double> N = this->elements[this->polygonIndex]->
    getShapeFunctions()->evaluateShapeFunctionCartesian(Point(x,y));

std::vector<int> containerPoints = container.getPoints();

for (int i = 0; i < containerPoints.size(); ++i) {
    Pair<int> pointDOFS = this->dofs.pointToDOFS(containerPoints[i]);

    uHx += this->nodalValues[pointDOFS.first]*N[i];
    uHy += this->nodalValues[pointDOFS.second]*N[i];
}

return Pair<double>(uHx, uHy);
```

The call to *getValue* in Code 5.25 is the only remarkable aspect of the procedure. Similarly to how we implemented the BodyForce class, the analytical solutions to the displacement, strain and stress are represented as classes (called DisplacementValue, StrainValue and StressValue, respectively) which contain pointers to the functions that compute the solution; this way, *getValue* is simply a call to the function that represents the analytical solution.

Finally, the calculation of the terms for the H1 norm for FEM are shown in Code 5.27 (norm of the difference between analytical and approximate solutions) and Code 5.28 (norm of the analytical solution); the first requires further comment, but the second is completely parallel to what we shown in Code 5.25, hence, it is considered self-explanatory.

Code 5.27: *Computation of the norm of the difference in stress and strain*

```
Eigen::VectorXd uH;
uH = Eigen::VectorXd::Zero(2*container.numberOfSides());

std::vector<int> containerPoints = container.getPoints();
for (int i = 0; i < container.numberOfSides(); ++i) {
    Pair<int> point_dofs = dofs.pointToDOFS(containerPoints[i]);

    uH(2*i) = nodalValues[point_dofs.first];
    uH(2*i + 1) = nodalValues[point_dofs.second];
}

Eigen::MatrixXd Be =
    StiffnessMatrixIntegrable::BeMatrix(Point(x,y),
        VeamyTriangle(container).getJacobian(this->points),
        this->elements[this->polygonIndex]->getShapeFunctions());
```

```

Eigen::VectorXd strain = Be*uH, stress = this->C*strain;

Trio<double> eH = veamy_functions::to_trio<double>(strain);
Trio<double> sH = veamy_functions::to_trio<double>(stress);
Trio<double> e = this->strainValue->getValue(Point(x,y));
Trio<double> s = this->stressValue->getValue(Point(x,y));

return (e - eH).dot(s - sH);
```

The formulas used to compute the strain and the stress in Code 5.27 are considered known, as they are inputted by the user; it is important to note that, although we refer to the shape functions in the procedure, we could use this same method for VEM if we chose some shape function to interpolate the value of the displacement inside the elements.

Code 5.28: *Computation of the norm of the strain and stress vectors*

```
Trio<double> stress = this->stressValue->getValue(Point(x,y));
Trio<double> strain = this->strainValue->getValue(Point(x,y));

return stress.dot(strain);
```

5.2.5 Configurations

As it was the case with the geometric library, this one also has a Singleton configuration class, that allows us to abstract from certain values, which are used throughout the library; in the VEM library case, there are three relevant values:

- The first one is the tolerance for the comparison of double precision numbers, exactly the same as the one presented in the implementation of the Voronoi generation, but left independent as the Virtual Element Method library can be used without the user knowing about the Voronoi generation one.
- The second is the value γ shown in Equation (2.35), related to the influence of the stability term in the stiffness matrix. We leave this value as a configuration parameter, since it is not expected for the user to change it, but it could be depending on the conditions of the problem, or to experiment about its relevance in the results.
- As it is the case with the Voronoi generation library, we include a parameter, called precision, used to control the number of decimals printed when results are written in a text file. It is important to remark that it is expected that a user interested in VEM would most likely not concern itself with the mesh generation library, but rather expect to configure precision just once. This way, to help the user, a change to the precision of the VEM library affects the precision of the geometrical library also, so that, when printing the mesh and the displacement results, the precision coincides.

Finally, since the VEM library includes a module to compute the displacements using FEM, it is natural that some configuration parameters are exclusive to FEM; in particular, we can name:

- The number of Gauss points, used in the calculation of all numerical integrals in the library, is left as a global configuration parameter, that the user can modify when they first create the system or when asking for the displacements.

5.3 Modifiable meshes library

This section features the implementation of the modifiable meshes simulation library. The model design of the class was explained in detail in Section 4.3.5.3, so, since this library is a collection of algorithms and models, this section will mostly deal with the algorithms

5.3.1 Modifiable algorithms

In this section we present the implementation of the four main algorithms of the modifiable mesh library; we explain most of the algorithms and tools used, except for some that are considered generic enough to be used for several ends, so they are left apart at the end of this chapter.

5.3.1.1 Eliminating non-convex mesh elements

We call "fix" the operation which changes an undesirable property of the mesh to another, more desirable; we leave the operation open to further implementations, but only include one fixing operation: taking a polygonal mesh, with elements both convex and non-convex, and partitioning the non-convex ones to obtain a completely convex mesh. We implement this operation in a concrete operator class named ConvexFixer.

The partitioning of a non-convex polygon in convex sub parts is a complex problem which has many possible solutions, all with different advantages and disadvantages; we choose not to use a triangulation scheme, which are the simplest to implement, as they defeat the idea of polygonal meshes, and instead evaluate algorithms that return convex polygons. We decide to use an external library, called PolyPartition[19], which implements the Hertel-Mehlhorn algorithm, a convex partition algorithm that has a good time performance and gives at most four times the minimal number of convex polygons. This way, to implement the convex fix operation we create two concrete classes, ConvexFixer and PolyPartitionWrapper, the first in charge of selecting the polygons to be fixed, asking the second (which is in charge of computing the convex partition itself) for the new elements, and including the elements in the mesh; ConvexFixer implements two methods, *fixMesh* and *fixElement*, which have the exact same functionality, but leave the possibility of applying the same operation on a single element or the complete mesh. We show the implementation of the convex fixer operation in

Algorithm 22.

Algorithm 22: Convex fixing a non-convex mesh polygon

```
Data: Non-convex polygon $poly$, List of mesh polygons
element_index ← Index of $poly$ in the list of polygons
partitioned_element ← Result of partitioning $poly$
pointMap ← Call the method $includeNewPoints$ of class AddElementsAdapter with
  partitioned_element as argument; Call method $includeNewElements$ of class
  AddElementsAdapter
```

We do not include an explanation of the procedure that partitions the polygon, as it is simple a wrapper call to the PolyPartition library, nor the calls to AddElementsAdapter methods, which, as it is considered a utility, is shown at the end of the chapter.

5.3.1.2 Refining a mesh element

The process of refining a mesh element consists in the inclusion of new mesh points inside it, and the generation of new elements which replaces the original; there are various way in which the new elements can be generated, ranging from simple triangulation schemes to more complex polygonal generation, so, as it is the case with all modifiable functions, we create an abstract interface called MeshRefiner that gives the base algorithm of refinement, that all concrete refinement classes must complete, which we show in Algorithm 23.

Algorithm 23: Refining a given polygon

```
Data: Polygon to refine $poly$, List of points inside $poly$ to include
element_index ← Index of $poly$ in the list of polygons
newElements, newPoints ← Call the $computeElements$ method of MeshRefiner
pointMap ← Call the method $includeNewPoints$ of class AddElementsAdapter with
  partitioned_element as argument; Call method $includeNewElements$ of class
  AddElementsAdapter
```

The class AddElementsAdapter and its methods, which were also seen in the previous section, will be touched in its own section at the end of this chapter. There is, however, one point in Algorithm 23 that is neither part of AddElementsAdapter nor have been touched before: *computeElements*. *ComputeElements* is a virtual function that is not implemented on MeshRefiner, but rather all concrete classes that inherit from it must implement; this way, following the Strategy design pattern, we have a way of reusing all algorithm, which is exactly the same no matter how the refinement scheme, and only make the difference in what is relevant. We include two concrete MeshRefiner classes, TriangulateRefiner and VoronoiRefiner, which both make use, respectively, of the Delaunay triangulation and Voronoi diagram generators that we included in the mesh generation library; this way, neither class is complex enough to be shown.

Creating points for refinement In this section so far, we have explained how the refining procedure is done, assuming that the points to add, that actually define the elements resulting of this process, are known beforehand, but we have not touched at all how these points are obtained. We include four different ways of including refining points when calling the xMesh method *refine*:

1. Giving a single point: We receive a Point from the user and use it as a basis for refinement; however, there is an important number of border cases that need to be handled. In the first place, if the Point is a vertex or outside the domain the method returns immediately; next, if the Point is in the boundary of the domain, it is included in the container segment; next, if the Point is a non-boundary edge, we merge both polygons that contain the edge and the refine the resulting one; finally, if none of the previous conditions is true, it means that the Point is contained inside the polygon, hence, we refine directly.
2. Giving a list of points: We simply use the previous method on each of the points to add.
3. Giving a PointGenerator: We reuse the functionality of the PointGenerator (shown in Section 5.1.1.2) to create the points inside the polygon to refine; we create a Region instance from the given polygon and use the same methods presented in the previous chapter.
4. Giving a PointCreator: As we commented in Section 4.3.5.3, the PointCreator class is created as a tool that, given a polygon, returns the points corresponding to a given property; it is different from the PointGenerator class as it is actually considers the polygon in which the points will be created. We include three concrete PointCreator classes: CentroidPointCreator, which return the centroid of the polygon; DelaunayPointCreator, which returns the centroids of each element of the Delaunay triangulation of the polygon; and EarPointCreator, which does the same as the previous one, but using the ear clipping triangulation.

5.3.1.3 Deleting a mesh element

We include methods to delete either a point of the mesh (which merges all polygons containing said point into one) or a polygon (which does the same, but with all polygon neighbours), but choose not to implement erasing of segments as it implies the possibility of leaving a nonconforming mesh. We present the algorithms to erase a point and a polygon in Algorithm 24 and Algorithm 25, respectively, noting that most of the algorithm is merging polygons, subject on which we will stop at the end of this chapter.

There are several border cases that can appear from these procedures, all of which are ignored, since they depend on user input and defeat the propose of the algorithms, among them, we can name: erasing the last polygon of the mesh; deleting a boundary polygon or a point, both changing the shape of the domain; deleting a boundary point which do not change the shape of the domain but do not change at all the structure of the mesh.

Algorithm 24: Refining a given polygon

```
Data: Polygon to refine $poly$, List of points inside $poly$ to include
element_index ← Index of $poly$ in the list of polygons
newElements, newPoints ← Call the $computeElements$ method of MeshRefiner
pointMap ← Call the method $includeNewPoints$ of class AddElementsAdapter with
  partitioned_element as argument; Call method $includeNewElements$ of class
  AddElementsAdapter
```

Algorithm 25: Deleting a polygon from the mesh

```
Data: Polygon to refine $poly$, List of points inside $poly$ to include
element_index ← Index of $poly$ in the list of polygons
newElements, newPoints ← Call the $computeElements$ method of MeshRefiner
pointMap ← Call $includeNewPoints$ of AddElementsAdapter with $newPoints$ as
  argument
Call $includeNewElements$ of AddElementsAdapter
```

5.3.1.4 Breaking the mesh

We call "breaking" the procedure that forcibly adds new lines segments to a mesh; it is a complex algorithm, as it has a number of border cases and must keep the mesh consistent. Moreover, differently from the three algorithms we presented before, breaking requires knowledge of the structure of the mesh, hence, it is implemented by xMesh directly and not by an operator class. We include two methods that break the mesh, both named *breakMesh*, but one receiving a single segment to include and the other a sequence of segments; in the second case, we force the user to give a consecutive sequence (which do not leave any hanging edges) or return with an error.

The mesh breaking procedure that receives a sequence of segments is presented in Algorithm 26; we do not show the version that receives one as it simply calls the more complex one with a list of one element. Finally, it is important to note that it calls a method called *breakPolygons*, which computes the new elements generated each step of the algorithm, and is also in charge of keeping mesh information updated. The only responsibilities of the *breakMesh* procedure are iterating, and determining when to stop.

BreakMesh initializes all the data that will be used when iterating, such as *poly1*, *poly2* and *init*, and decides when to stop iterating, by checking each step if *poly1* contains the last point of the crack; it repeats the same steps for each segment to include in the mesh, calling the method *BreakPolygons* with all the relevant information each time, and then updates the structures to advance one step. It is important to note that the *breakMesh* procedure is more complex that what we show in the algorithm, as there are a number of border cases to consider, such as, for example, breaking starting from the boundary, from a vertex or from an edge, or traversing the mesh through an already existent segment, among other; we decide not to include those details as they are not common and do not change the core of how the method works.

Algorithm 26: Breaking the mesh

```
Data: List of segments segments that will be included in the mesh
Check that segments is a consecutive sequence and return and error if it is not
for Segment s in segments do
    Find the polygon that contains the first point of s, name it init
    Determine the neighbour of init in the direction s, name it poly1
    while true do
        if poly1 contains the second point of crack then
            return
        end
        Determine neighbour of poly1 in the direction of s, name it poly2
        Compute the new polygons created dividing poly1 in two, using method
          BreakPolygons
        Replace poly1 with one of the new polygons, insert the other at the end of the
          mesh
        if This is the first iteration on s and s is not the first segment of the sequence
          then
            Connect the last point of the previous segment to the first of s
        end
        Update neighbourhood information of all neighbours of the old poly1
        Assign init as poly1 and poly1 as poly2
    end
end
Close the first and last polygon of the sequence using the required ClosingRule
```

There are two relevant sub steps in the *breakMesh* procedure: the previously mentioned *breakPolygons*, and determining the neighbour in a given direction, named *getNeighbour*; however, the second procedure is used to determine the polygon that contains a point, an algorithm that is generic enough to have its own subsection at the end of this section, so we will not comment further on it now.

BreakPolygons is the method is charge of computing the two polygons that will replace the segmented polygon. We show the most relevant points of Algorithm 27, making special emphasis in how to correctly assign the neighbourhood information; we complement the algorithm with Figure 5.4, for further clarification of its operation.

The *BreakPolygons* method makes use of yet another method, named *computeNewPolygons*, which finally determines the new polygons. The separation of responsibilities, seemingly redundant, is justified by the fact that we can reuse the computation of the polygons to several other situations, such as closing the polygons at the end of the breaking algorithm or to handle the border cases of the same algorithm.

Finally, the only operation left to analyze is *computeNewPolygons*, shown in Algorithm 28. The procedure takes advantage of our chosen way of representing the polygons, using the indexes of their vertices, transforming a complex problem into a two-step process: searching for an index to start, and iterating around the polygon until finding a certain index. To

Algorithm 27: Breaking a polygon

```
Data: NeighbourInfo $n1$ and $n2$, new points $p1$, $p2$, $p3$ and $p4$, initial polygon index
      $init$
$poly \leftarrow$ Get polygon $n1$.neighbour
Insert $n1$.intersection and $n2$.intersection on the mesh, returning their indexes $p1$ and
 $p2$
$new1 \leftarrow p1, p2$
$new2 \leftarrow p2, p1$
$newElements \leftarrow$ Compute the new polygons, using $new1$ and $new2$
Insert $p1$ on segment $n1$.edge of polygon with index $init$
Insert $p2$ on segment $n2$.edge of polygon with index $n2$.neighbour
Delete segments $n1$.edge and $n2$.edge from $mesh$.segments
Insert pair Segment($p1$,$p2$), NeighboursBySegment(Index of $poly1$, Index of $poly1$) in
 $mesh$.segments
Insert pairs Point of index $p1$, NeighboursByPoint($init$, Index of $poly1$, Index of $poly1$)
Insert pairs Point of index $p2$, NeighboursByPoint(Index of $poly2$, Index of $poly1$, Index
 of $poly1$)
for $elem$ in $newElements$ do
    for $s$ in $elem.segments$ do
        Replace neighbour related to $s$ from $n1$.neighbour to index of $elem$
    end
    for $p$ in $elem.points$ do
        Replace neighbour related to $p$ from $n1$.neighbour to index of $elem$
    end
end
Insert $init$ in Neighbours associated to Segment($p1$, $n1$.edge.first)
Insert $init$ in Neighbours associated to Segment($p1$, $n1$.edge.second)
Insert $n2$.neighbour in Neighbours associated to Segment($p2$, $n2$.edge.first)
Insert $n2$.neighbour in Neighbours associated to Segment($p2$, $n2$.edge.second)
Set $n2$.extraPoint as $p4$
```

further clarify the algorithm, we include in Figure 5.4 an example showing the positioning of the variables referenced in Algorithm 28.

The only relevant aspect of Algorithm 28 is the existence and use of the ElementReconstructor class. We create this library to handle both triangulations and polygonal meshes, so we include a mechanism to post process the result of the breaking algorithm to preserve a desired property; for example, we can use the reconstructor to take the result of partitioning the polygon, two new elements, and triangulate each one, so that the mesh contains only triangles after the process.

Finally, it is important to note that, as it was the case with break, *computeNewPolygons* has a number of border cases that one needs to handle (related to breaking the polygon through a vertex mostly), on which we did not stop since they do not change the main behavior of the algorithm.

Algorithm 28: Computing the polygons when breaking

```
Data: NeighbourInfo $n1$ and $n2$, new points $p1$, $p2$, polygon to break $poly1$, list of
      points to fill $new1$ and $new2$, ElementReconstructor $constructor$
Order $n1$.edge and $n2$.edge in CCW with respect to the centroid of $poly1$
if Orientation of $p1$, $p2$ and $n2.edge.first$ is positive then
    $indexOfStart \leftarrow$ Index of $n2$.edge.first in $poly1$.points, $point \leftarrow n2$.edge.first
else
    $indexOfStart \leftarrow$ Index of $n2$.edge.second in $poly1$.points, $point \leftarrow n2$.edge.second
end
$edgePointPassed \leftarrow$ False
while True do
    if $point$==$n1.edge.first$ or $point$==$n1.edge.second$ or $point$==$n1.extraPoint$ then
        if $edgePointPassed$ then
            Break
        else
            $edgePointPassed \leftarrow$ True
        end
        if $point$ is not $p1$ or $p2$ then
            Insert $point$ in $new1$
        end
        $point \leftarrow poly1$.points[$indexOfStart + 1$], $indexOfStart \leftarrow indexOfStart + 1$
    end
end
$edgePointPassed \leftarrow$ False
while True do
    if $point$==$n2.edge.first$ or $point$==$n2.edge.second$ then
        if $edgePointPassed$ then
            Break
        else
            $edgePointPassed \leftarrow$ True
        end
        if $point$ is not $p1$ or $p2$ then
            Insert $point$ in $new2$
        end
        $point \leftarrow poly1$.points[$indexOfStart + 1$], $indexOfStart \leftarrow indexOfStart + 1$
    end
end
Create a list of new elements called $newPolygons1$ and $newPolygons2$ calling
  $reconstructor$ with the lists $new1$ and $new2$
Create an empty list $indexes$
Replace the polygon of index $n1$.neighbour with the first element of $newPolygon$ and
  insert the index in $indexes$
Insert the rest of $newPolygons1$ and all $newPolygons2$ at the end of $mesh$.polygons,
  inserting each index in $indexes$
return $indexes$
```

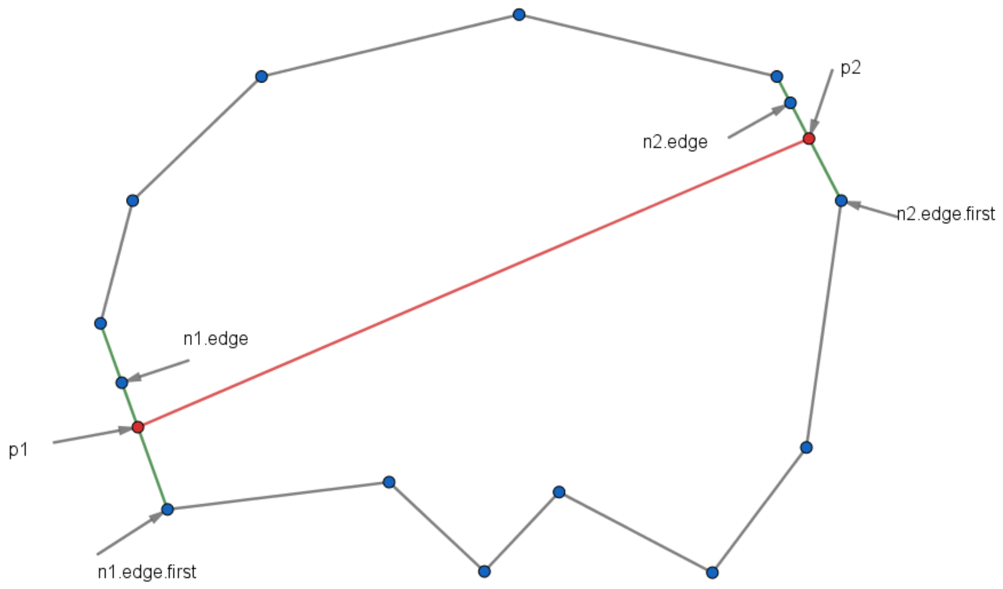
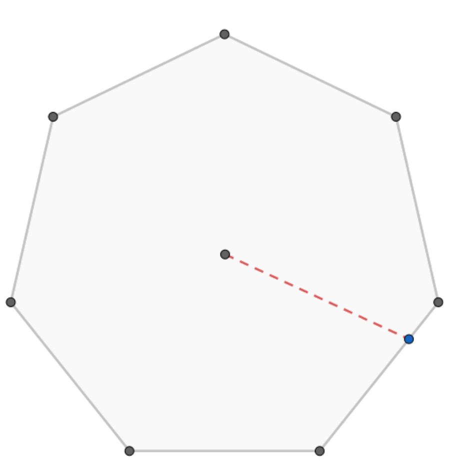


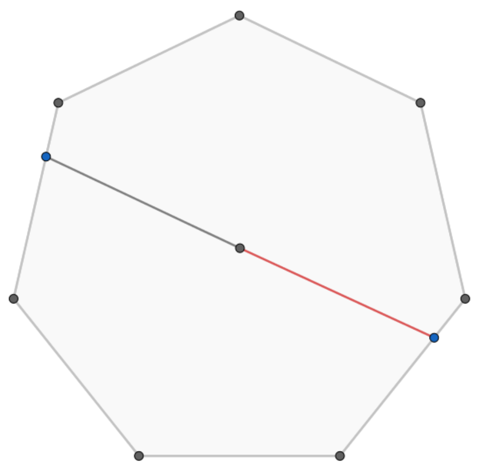
Figure 5.4: *Computation of the new polygons when breaking*

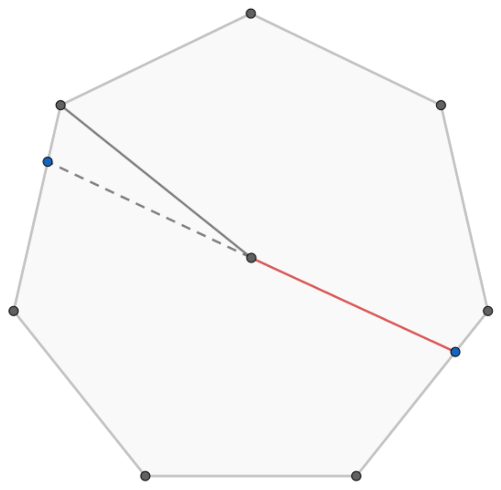
Closing rules The closing rules that we reference in Algorithm 26 are the last point that we need to mention to fully understand how the mesh is broken. When a user includes a segment on the mesh, it is highly likely that the endpoints of the segment are inside mesh elements and not in segments or vertex, so including them directly could leave hanging edges; hence, we must include some way of "closing" those polygons when the mesh is left non-conforming. We create a set of classes named ClosingRule that represent different ways in which the polygons can be closed; these classes all implement a method called *closePolygon*, that receive an xMeshElement pointer (so that the mesh can be modified) and the polygon that must be closed, plus some of the information computed during the breaking process. Four different ClosingRule implementations are included: TruncateClosingRule, which simply deletes the hanging edge; StraightLineClosingRule, which extends the hanging edge until it reaches a mesh segment; ClosestVertexClosingRule also extends the hanging edge until it reaches a mesh segment, does not adds an extra point; and finally, TriangulateClosingRule, which triangulates the polygon, forcing the inclusion of the hanging edge. Figure 5.5 shows a schematic representation of the four closing rules.

5.3.2 Utility algorithms

In this section we will detail all those algorithms related to the modifiable meshes algorithms, that we have used but not explained, such as searching for the container of a given point, polygon merging, among others; we decided to leave them in their own separate section for easier reference, and to avoid overly complicating the steps of the algorithms shown before.


(a) Truncate closing rule


(b) Straight line closing rule


(c) Closest vertex closing rule

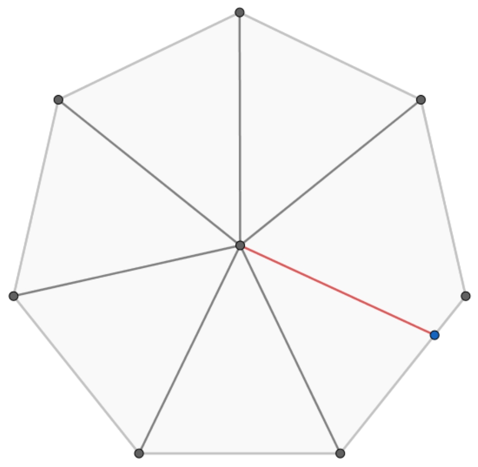

(d) Triangulate closing rule

Figure 5.5: *Available closing rules*

5.3.2.1 Search for container polygon

This problem consists in finding the polygon P of the mesh that contains any point given p, which can be either inside or in its boundary. First, it is important to remember that our meshes are represented as a list of polygons, so the simplest solution to this problem is to linearly search through the list, testing for containment, until one finds the correct polygon or reaches the end of the list (in this case, the point is not on the mesh and should be handled without the program failing); this approach is simple but inefficient when one needs to use this operation constantly, hence the need for a more complex, but more efficient algorithm.

We propose using a technique called Jump-and-walk, originally presented to decrease the cost in Delaunay triangulation generation[30]; this algorithm works by randomly selecting a starting polygon, and traversing the mesh in the direction defined by the point p, until we either find the polygon that contains p, or end outside the mesh. In the worst case this algorithm is no better that linearly searching through the list, but in most normal cases it is extremely fast, to the point that it is amortized $O(1)$; this low cost is explained the by fact that the number of neighbours that each polygon has is at least three, and on average quite a few more, so each time we choose a direction to traverse to, we discard an important fraction of the whole mesh. However, it is important to note that this approach has the important restriction of only working on convex domains, as in non-convex ones exists the possibility of incorrectly determining that p is not in the mesh. As we allow for non-convex domains, we need to complement jump-and-walk to work in our conditions. We decide to use a mixed approach: we search for the container polygon using jump-and walk, and use the linear approach in those cases jump-and-walk determines that p in not in the mesh, corroborating that the previous result is correct; this approach gives us an average complexity still $O(1)$, but making sure we always return correct results.

Besides the points mentioned before, we also take an extra consideration to further optimize this algorithm; we consider that if we have some sort of idea of where the point is contained, as it is the case in the fracture simulation library, we could start from there instead of using a randomly selected polygon.. Hence, our implementation of jump-and-walk includes a version that receives the polygon from where to start searching. The latter version of the algorithm that does most of the computation is in Algorithm 29; the standard one simply uses Algorithm 29, after selecting a random polygon index.

```
Algorithm 29: Find container polygon
  Data: Point $p$, mesh with polygons with index $0, 1 ... n-1$, starting polygon i
  Result: Index of the container polygon, or -1 if point is not inside the mesh
  while True do
      found ← false
      poly ← Get polygon i from mesh
      if $p$ is inside poly then
          return i
      else
          centroid ← Centroid of polygon poly
          path ← Segment between centroid and $p$
          neighbour ← getNeighbour(i, path)
          if neighbour > -1 then
              i ← neighbour
              found ← true
          end
      end
      if not found then
          Search for container polygon linearly
      end
  end
```

The only aspect of Algorithm 29 left unexplained is the *getNeighbour* algorithm, in charge of deciding which of the neighbours of the polygon intersects a given direction; this computation is more or less straightforward, but needs to be examined nonetheless, because of its relevance. We show the implementation of the *getNeighbour* method in Algorithm 30.

The polygon indexes list *previous* given in Algorithm 30 as input data, is used to avoid a common issue in this algorithm: if we do not keep information of the polygons already visited by the algorithm, it is possible to select as next polygon exactly the same as one that we came from, leaving us in an infinite loop; therefore, we keep all the indexes of the polygons that come before, so that we can be always sure that we are advancing.

Finally, as C++ does not allow for multiple return parameters, we encapsulate the results of Algorithm 30 in a structure named *NeighbourInfo*, which only objective is to contain the information computed in the *GetNeighbour* method. *NeighbourInfo* contains the point of intersection computed by the algorithm, the segment the point is contained in, and the neighbour in the given direction.

Algorithm 30: Find neighbour that intersects segment

```
Data: Polygon $p$, mesh with polygons with index $0, 1...n-1$, direction of advance d,
      list of indexes of previous polygons $previous$
Result: Neighbour $next\_polygon$ that intersects d, common segment between $p$ and
        $next\_polygon$, intersection point $intersection$
segments ← Get segments of $p$
for $Segment\ s\ in\ segments$ do
    intersects, intersection ← Compute intersection between $s$ and d
    if $intersects$ then
        neighbours ← Get incident polygons to $s$
        $next\_polygon$ ← Get polygon from neighbour different to $p$
        if $next\_polygon\ is\ not\ in\ previous$ then
            return $next\_polygon, s, intersection$
        end
    end
end
return $-1, Segment(-1,-1), Point()$
```

5.3.2.2 Polygon merging

As we have seen before, it is necessary to merge polygons to erase elements (either polygons or points) of the mesh. However, polygon merging is a complex problem in its own, especially considering numeric precision problems. Hence, we decide to implement a custom polygon merging algorithm, taking advantage of the fact that our polygons are defined by the indexes of their points, so we can avoid comparing doubles.

We implement our merging algorithm considering two variants: one that takes two polygons and merge them directly, and one that takes a list with several; we will show both versions next, however, the second one uses the first several times, instead of implementing a more efficient schema, which is justified in the fact that implementing the merging of several polygons simultaneously is too complex. Hence, the merging of several polygons simply iterates over the list of elements to merge, looking from two that can be joined together, until we have included them all. The relevant methods are both named *mergeElements*, but have different signatures to tell them apart, and belong to our xMesh class; neither method returns the new polygon, but includes it in the mesh, updating the neighbourhood information. However, xMesh does not actually computes the desired result, being only in change of keeping consistency; we call the operator interface in charge of merging polygons MeshMerger, and implement a variant called VertexIndexMerger, so that later in the development it is easier to include more efficient ways of merging.

Merging of two polygons We first present the implementation of the first version of the *mergePolygons* method of xMesh, the one that takes two polygons, in Algorithm 31.

Algorithm 31: Merging of two polygons in xMesh

```
Data: Mesh mesh, Polygon indexes to merge poly1 and poly2, deleted points list
      deleted
Initialize p1 as polygons[poly1] and p2 as polygons[poly2]
if poly1 and poly2 are not neighbours then
    return
end
Create Map[Neighbours,int] merged_pairs and insert (Neighbour(poly1,poly2),0)
Swap polygon of index poly2 with the last of polygons
Set merged as the polygon created from the result of merging polygons p1 and p2
Insert merged in index poly1
for Segment s in poly2.getSegments do
    edges.replace_or_delete(s, last element of polygons, poly2, poly1)
end
for Point d in deleted do
    Swap point of index d with last element of points
    Delete last element of points
end
Delete last element of polygons
```

There are three procedures mentioned in Algorithm 31 that require explanation: swapping polygons, swapping points and the method *replace_or_delete*; which we will go through next.

Swapping elements The procedures that swaps mesh elements (either points and polygons), that inverts their positions in the list, is important considering that both elements are kept in a list, and we need to preserve the ordering, so deleting a polygon directly from the list is not possible. The only way one is able to delete an element in this kind of data structure is to erase the last element of the list; hence, to delete an element in an arbitrary position, we first swap the last element with the one we wish to erase, keeping all neighbourhood information consistent, and only then we erase the desired one. The operation of swapping the position of two given polygons is shown in Algorithm 32, while the one that swaps point is shown in Algorithm 33.

Algorithm 32: Swapping of two polygons

```
Data: List of polygons $polygons$, polygon indexes $poly1\_index$ and $poly2\_index$, map
      $edges$ relating segments and incident polygons
Initialize poly1 as polygons[$poly1\_index$] and poly2 as polygons[$poly2\_index$]
Set firstSegments as the segments of poly1
Set secondSegments as the segments of poly2
for Segment s in firstSegments do
    In edges[s], swap the value $poly1\_index$ for $poly2\_index$
end
for Segment s in secondSegments do
    In edges[s], swap the value $poly2\_index$ for $poly1\_index$
end
polygons[$poly1\_index$] ← poly2
polygons[$poly2\_index$] ← poly1
```

Algorithm 33: Swapping of two points

```
Data: List of points $points$, point indexes i1 and i2, map $edges$ relating segments and
      incident polygons
$p1$, $p2$ ← Points of index i1 and i2
$Containers1$, $Containers2$ ← Containers of $p1$ and $p2$
for Index i in $Containers1$ do
    Call $replace\_vertex$ of polygon of index i with arguments $p1$, $p2$ and $edges$
end
for Index i in $Containers2$ do
    Call $replace\_vertex$ of polygon of index i with arguments $p2$, $p1$ and $edges$
end
points[i1] ← $p2$
points[i2] ← $p1$
```

$Replace\_vertex$ is a simple method of xPolygon that seeks the position of a given vertex inside the polygon, replace its value, and changes the value of the neighbourhood information inside $edges$.

Replace and delete $Replace\_or\_delete$ is a utility method of xSegmentMap that helps us delete segments when polygons are merged, and update the segments to reflect the new state, taking into consideration two cases:

1. When merging polygons, the segments shares by the polygons to merge disappear from the mesh. When merging several polygons, this problem gets complex, as there are several segments, belonging to different polygons, that should be eliminated. We handle this issue in *replace_or_delete* using a hash map called $merged\_pairs$, that contains all possible pairs of merged polygons, which is used to determine which Neighbours combinations should be eliminated from the mesh.

2. As we delete segments, after some time of processing we inevitably end trying to delete a segment more than once, which leads to errors. To handle this, we use another hash map, named *erased_segments*, initially empty, that is filled as segments, and their associated NeighboursBySegment, are deleted from the xSegmentMap.

Finally, we show the pseudo code detailing the most relevant aspects of the implementation of the *replace_or_delete* method in Algorithm 34.

Algorithm 34: Replace_or_delete method

```
Data: Segment s; polygon indexes oldCurrent, oldOld and new; hash maps
      merged_pairs and erased_segments
if s in erased_segments then
    return
end
n ← Get value associated with s in the SegmentMap
Change neighbour in n from oldCcurrent to oldOld
if n in merged_pairs then
    Delete n from the SegmentMap
    Insert pair (s,0) in erased_segments
else
    Change the neighbour in n from oldOld to oldCurrent
    Change the neighbour associated to s from oldCurrent to new, or insert it if not
     found
end
```

In Algorithm 34 we need to use the two polygon indexes *oldOld* and *oldCurrent* because of the order of the operations in Algorithm 31; as we first create the map *merged_pairs*, and then swap the polygons, *replace_or_delete* needs the information of the indexes before the swapping.

Computing the merged polygon The only aspect left to detail about the procedure shown in Algorithm 31 is the actual computation of the result of merging the two polygons, which is done by VertexIndexMerger in its method *mergePolygons* (Algorithm 35). This operation mostly relies in finding the indexes of the vertices common to both polygons, but that are further apart, in one of the polygons, and replace the value of those two for all the non-common indexes of the other polygon; this methodology only works when representing the polygons using point indexes, and has the advantage that it does not depend on the convexity of the polygon or, most specifically, the fact that in non-convex polygons there are sections of points in clockwise order even if the complete polygon is in counter clockwise order.

The method *commonEdgeBorder* is the one in charge of determining the end points of the common set of segments of two polygons, again using indexes to determine equality. This procedure finds the index in the second polygon ($p2$) of the first common vertex (checking an important border case: we do not consider as the start of the common edge a lone common

Algorithm 35: Merging of two polygons in VertexIndexMerger

```
Data: Polygons $p1$ and $p2$
Result: Polygon obtained from merging $p1$ and $p2$
if $p1 == p2$ then
    return $p1$
end
Call CommonEdgeBorder to obtain $endPoints$, the common vertices of p1 and p2
Initialize mergedPolygon as an empty list
Set as first_1 and second_1 the index of both endPoints in the points of p1
if first_1>second_1 or first_1+1 is not vertex of p2 then
    Swap the values of first_1 and second_1
end
Set as first_2 and second_2 the index of both endPoints in the points of p2
if first_2>second_2 or first_2+1 is not vertex of p1 then
    Swap the values of first_2 and second_2
end
i ← second_1 + 1
while i is not first_1 do
    Insert $p1$.points[i] in mergedPolygon, i ← i + 1
end
i ← second_2
while i is not first_2 do
    Insert $p2$.points[i] in mergedPolygon, i ← i + 1
end
Insert $p2$.points[first_2] in mergedPolygon
return mergedPolygon
```

vertex, so we need to make sure that either the next, or the previous, point is also a common one, or continue searching) and traverse the border of $p2$ to find the two jumps, from common vertex to non-common vertex, that define the common border. Hence, it is actually pretty simple to determine that the cost of the procedure is $O(m)$, as it first searches for the index of a vertex in a polygon, and then iterates through the same list of vertices.

Merging a list of polygons When we desire to merge more than two polygons, we call the procedure to merge two polygons several times in a row, taking the first polygon of the list as a start, and then include one at a time until we have merged them all; however, this operation is not as simple as it seems, since we require for each pair of polygons to be neighbours, otherwise, the two-polygon merging method would fail. This way, we select the first polygon of the list and set it as the result, and continue iterating through the remaining list searching for elements that can be merged to the result; however, being neighbours is not enough to be defined as "mergeable", since there are a number of border cases that we need to handle carefully:

- Polygons that are neighbours by more than two separate set of segments (as shown in Figure 5.6a) are not mergeable, as they have the possibility of leaving a merged polygon with a hole.
- Polygons that are neighbours by one set of segments but touch (have at least one common point) in any vertex not inside that set (as shown in Figure 5.6b) are not mergeable, since this also has the potential of leaving a hole.

We summarize the previous behavior in a method called $areMergeable$ in VertexIndexMerger, which cost is trivially $O(n)$, as it iterates through the vertices of one polygon checking all the mentioned conditions.

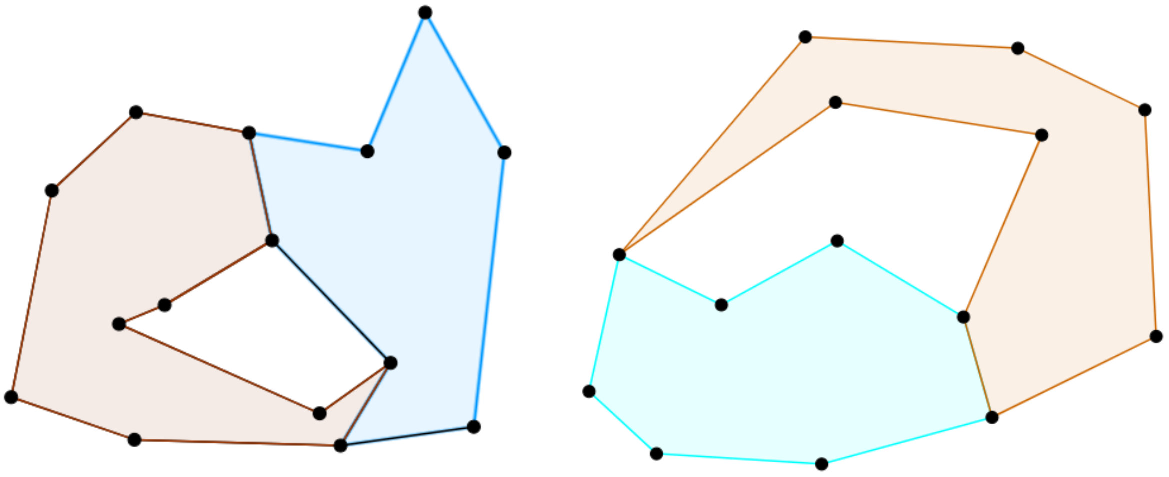

(a) More than one set of common segments **(b)** Neighbours but also touching

Figure 5.6: *Unmergeable polygon dispositions*

Using the *areMergeable* method, we implement a version of $mergePolygons$ in VertexIndex that receives a list of polygons and returns a new one, which pseudo code can be seen in Algorithm 36. It is important to note that the method will fail in the cases one gives it a list of polygons that do not define a single closed one.

As we have gone through all the methods required for the previous implementation, we can present directly the method $mergePolygons$ of xMesh, the one that is actually called by the user, in Algorithm 37.

5.3.2.3 Including new elements on the mesh

Including new elements on the mesh while keeping all neighbourhood information updated is a relevant, and complex, operation that is used by at least two different modifiable mesh algorithms; moreover, it is actually quite simple to separate from the implementation of both algorithms, which further justify why it was preferred to leave it as a separated algorithm. The inclusion of new elements in the mesh is handled by a class named AddElementsAdapter, which has two methods, *includeNewPoints*, in charge of adding extra points to the mesh, and *includeNewElements*, which takes the elements and adds them to the mesh, creating

Algorithm 36: Merging of several polygons in VertexIndexMerger

```
Data: List of polygons indexes $polys$, xMesh $mesh$
Result: Polygon obtained from merging $polys$
Initialize merged as first element of $polys$
Erase last element of $polys$
Initialize as $j$ ($polys$.size - 1)
while $polys.size$ is not zero do
    while not $mesh.areMergeable(merged, polys[j])$ do
        $j \leftarrow j - 1$
    end
    Initialize as $polyJ$ the polygon of index $polys[j]$) in the mesh
    merged $\leftarrow$ mergePolygons(merged, $polyJ$)
    Erase $polys[j]$ from $polys$, $j \leftarrow polys$.size - 1
end
return merged
```

neighbourhood relationships as necessary. In this subsection, we will first comment on the implementation of *includeNewPoints*, and then of *includeNewElements*.

Inclusion of new points We show the code for the *includeNewPoints* method in Code 5.29, which is in charge of receiving a list of Point instances, and include them in the mesh. The most relevant aspect of this procedure is that, as it appends points, it fills a hash map relating the indexes of the points in the original list to the indexes in the mesh, so that later we can convert polygons that were created considering the original indexes.

Code 5.29: *Include new points on the mesh*

```
std::unordered_map<int,int> pointMap;

for (int j = 0; j < points.size() ; ++j) {
    int pointIndex = meshPoints.push_back(points[j]);
    pointMap.insert(std::make_pair(j,pointIndex));
}

return pointMap;
```

Inclusion of new elements We show the procedure that takes a list of polygons and points, which represent new elements to add to the mesh, and includes them, updating all mesh information as necessary, in Algorithm 38. One important detail to comment is that the method, although it modifies the mesh, it does not require access to any xMesh method; this way, it does not require the xMesh instance, but rather an XMeshElement one, which gives it access to the elements, points and neighbourhood maps.

The algorithm shown in Algorithm 38 has one sub-procedure which purpose, and implementation, is not self-explanatory: the method *replace_segment*, which we will comment briefly.

Algorithm 37: Merging of a list of polygons in xMesh

```
Data: List of polygons indexes $polygons$
Merge $polys$ and set them in $merged$
Sort $polys$ descendantly
Set $polys[polys$.last] as $merged$
Initialize map with all possible pairs of $polys$ elements, called $neighbourPairs$
Initialize empty map $toIgnore$
for i in $polys$ do
    Swap polygons i and $polys$.size - 1
    Initialize as $polyI$ the polygon of index $polys[j]$) in the mesh
    Initialize $polyISegs$ as the segments of $pI$
    for $s$ in $polyISegs$ do
        Replace_or_delete($s$, $polys$.size - 1, i, $polys$.last, $neighbourPairs$, $toIgnore$)
    end
end
return merged
```

Replace_segment is a necessary operation to keep the mesh consistent, as it informs the neighbours of the changed polygon that points were added to one, or several, of its segments. The method is complex, and relevant, enough for its implementation to be shown in Algorithm 39. Its functionality is seemingly simple, since it receives a single segment and a list of segments, replacing the first for the second; however, the list of segments need to be consecutive, and start, and end, with the exact same vertices as the initial segment. Hence, this method does the computation necessary to order the segments counter clockwise, and determine how the old segment should be replaced.

The first check seen in Algorithm 39 is used as an optimization measure, as processing the replacement of a segment for another exactly equal is wasteful, and that case is extremely common in the adaptation of the triangles.

This algorithm has the potential of entering an infinite loop in the case the input is not correctly given, as it assumes that the list of segments is continuous and that it is possible to order it counter clockwise; when given a list of segments that define two or more disjoint sets, this algorithm iterates infinitely trying to order them. We assume this risk and do not check for the condition that could make the algorithm loop, considering that this procedure is only called by *adaptTriangulationToMesh* and, by construction, we correctly define the list of segments; hence, it is not recommended for the user to invoke this method.

5.3.3 Configurations

Differently from the Singleton configuration classes seen in the previous libraries, which were in charge of keeping certain configuration parameters that were mostly expected not to be changed, in this library we include a configuration class, but that will always (or at least most of the time) created directly by the user. The configuration class, named this time Config,

Algorithm 38: Including new elements on the mesh

```
Data: List of elements to add new, xMeshElements e, Hash map pointMap
Initialize a Map changesMap, relating polygon index, old segment and new segments
for i in 0 to number of elements of new do
    Initialize newPoints as an empty list
    for k in 0 to new[i].points do
        Insert e.pointMap[k] in newPoints
    end
    Create polygon newPolygon using newPoints
    Insert newPolygon at the end of the list of polygons of e
    Insert segments in e.segmentMap and update information
    if Any segment of the new polygon is in the boundary of the old polygon then
        Find the current container and insert all information in changesMap
    end
end
for p in the keys of the map changes do
    polygon ← Get polygon p
    for Pair segment, segments in the map associated to p do
        Call the method replace_segment of polygon with parameters(segment,
          segments)
    end
end
```

is in charge of keeping the different settings given by the user, meaning, the MeshRefiner, MeshMerger, ElementReconstructor and ClosingRule pointer that completely define three of the four modifiable mesh operations (MeshFixer is given by the user directly). We show the header of the Config class in Code 5.30, which is enough to understand how it works.

Code 5.30: *Configuration class of the modifiable mesh library*

```
class Config{
public:
    MeshRefiner* refiner;
    MeshMerger* merger;
    ElementReconstructor* reconstructor;
    ClosingRule* closingRule;

    Config(MeshRefiner* refiner, MeshMerger* merger,
        ElementReconstructor* reconstructor, ClosingRule* closingRule);
    Config(Configurations::config config);
    void setMesh(xMeshElements* mesh);
};
```

The *Configurations* namespace that can be seen in Code 5.30 is a list with common configurations that we leave available to the user both as examples and as a utility.

Algorithm 39: Replace_segment method implementation

```
Data: Polygon $poly$, Segment $seg$, List of segments $segs$
if $segs.size$ is one and $seg$ is equal to $segs[0]$ then
    return
end
Initialize $orderedPoints$ as an empty list
$segment$ ← Last element of $segs$
Insert points of $segment$ in $orderedPoints$, in CCW with respect to $poly$.centroid
Delete $segment$ from $segs$ and set i as $segs$.size - 1
while $segs.size$ is not zero do
    if i < 0 then
        i ← $segs$.size - 1
    end
    Order $segs$[i] in CCW with respect to $poly$.centroid
    if The first element of $orderedPoints$ is vertex of $segs$[i] then
        Insert $segs$[i].first at the start of $orderedPoints$ and delete $segs$[i] from $segs$
    end
    if The last element of $orderedPoints$ is vertex of $segs$[i] then
        Insert $segs$[i].second at the end of $orderedPoints$ and delete $segs$[i] from $segs$
    end
    i ← i - 1
end
$indexOfStart$ ← Index of $seg$.first in $poly$.points
Insert all the elements of $orderedPoints$ between the second and the second to last in
  $poly$.points, starting from the index $indexOfStart$
Recompute $poly$ with the new points included
```

5.4 Utilities module

After reviewing the implementation of the three libraries that make up this work, conveniently separated in three chapters, we need to stop here to present tools that were used but not explained, such as the UniqueList and Pair classes, fundamental to the work, but not inherent to any aspect of the modeling itself. As these tools were designed to be generic, they were programmed using the template abilities of C++, so they can be reused for several ends.

5.4.1 Unique list

As mentioned several times during both the background and implementation chapters, duplicated elements tend to cause problems both in the geometrical and numerical algorithms, hence, we do need to make sure to have unique elements in several of the list we manage; to make sure that one does not have duplicated elements on a list, there are usually two alternatives:

1. One can insert with no check and clean up duplicates at the end.
2. Check for existence before inserting, so that naturally one ends with a list of unique elements.

We choose the second alternative as we dynamically add points throughout the fracture simulation process, so the choice of checking for duplicates at the end is not feasible. For our convenience and ease of programming, we decide to implement a structure that encapsulates this behavior, first searching if the element is already in the structure, and only including it if it is not found, which we call *UniqueList*.

Our UniqueList is a template class that can contain any kind of element, from integers to objects of custom classes, with minimal restrictions: among them, the most important is the need for the ability to compare objects, as we have to decide whether two objects are the same. UniqueList is a simple wrapper around the standard template library of the C++ vector implementation, reusing most of its operations:

1. We use, as it is the case with vector and most array-like structures, [ ] to access elements.
2. We use *push_back* as the standard way of adding new elements to a UniqueList. This operation inserts an element at the end of the list, but making sure it is not already contained. Differently from the vector version, UniqueList's *push_back* returns the index where the element was inserted, or, in the case it was already contained, the position where it can be found.
3. We include a *push_back* version that receives a vector, and returns a vector with the indexes where each element was inserted.
4. To know the number of elements inside the UniqueList, one can call the *size* method.

As it can be concluded from the previous paragraphs, the most important responsibility of a UniqueList is making sure that the elements in its list are unique; the naivest way of implementing this is searching in the vector every time an element is added, but, since search is linear time in vectors, intensely using this method is prohibitively expensive. Hence the idea of using a classic trick: including a hash map (or similar structure) to be able to search for containment in constant time.

We have a number of classic data structures to choose from, however, we have to deal with machine precision, which bounds what we can actually use; in this particular case, floating point precision forbids us to use a standard hash map, as the hash calculation does not take in consideration that two incredibly close points are actually the same, and modifying how the hash is computed leads to inefficiencies in both its implementation and the actual performance of the hash map. Therefore, we sacrifice some efficiency and choose to use another variety of map, provided directly by C++, called simply a *map*; this structure is based on ordering instead of hashing (fundamentally, it is a binary tree), giving us a way to include comparison up to a certain degree of precision. The disadvantage of this structure is that look-up is not constant in time (as it is the case with a hash map), but logarithmic; this is not considered catastrophic since the gain in time from the linear naive method is still relevant. Finally, it is important to mention that, when one desires to use this map with a custom structure as key, it is needed to override the *lesser than* operator, which is used both for ordering and comparison (since a == b is true when both a>b and a<b are false); this detail is important, since it allows us to define that two extremely similar elements are equal by imposing that *lesser than* is always false when two elements are closer than a certain value.

Using the previous idea, we keep two structures inside our UniqueList, a standard vector and a map; we use the first as a means of ordering, but with no information whatsoever about duplicates. On the other hand, the map contains pairs relating elements to their index on the vector, so it is used to keep tabs on already present elements, and avoid linearly searching on the vector; when we insert a new element on the list, we first search for it in the map, and if it is found, we simply return the value associated to this key, and when it is not found, we insert it both in the vector and in the map.

5.4.2 Pair and Trio

Another relevant template class implemented is called *Pair*, used to represent a container of two equal classes. Pair is extensively used since we work in 2D, and, therefore, most physical parameters and results have two values (one for each axis), such as, for example: any vector value to compute area or orientation; the displacement, associated degrees of freedom and constraints of a point; common edges, among many others. Because of how common it is, we decide to include Pair as a utility instead of inside a specific library.

Even though Pair is left as generic, with no restriction whatsoever on the type of class it can contain, almost all of its operations imply that, whatever the Pair contains, it at least need to be numeric, as Pair overrides sum and multiplication, and calculates dot product; it is important to note that using Pair as a container of another type of class is not wrong, but trying to use its operations will most likely generate an error.

Trio is another template class, exactly equal to Pair, except that it contains three instances of the same class instead of two; it is created to contain the two physical parameters that have three values: strain and stress value for a given point.

5.4.3 Miscellaneous functions

There are a number of small functions needed for different reasons that do not belong in the modeling of any of the implemented libraries, and are generic enough to be left as utilities. It is important to note that this section is concerned with general utilities, as those belonging to specific aspects of our libraries were commented in their own subsections. These functions are self-explanatory, so just showing their code is enough to understand their uses; we show the namespace with all these functions in Code 5.31.

Code 5.31: *Namespace with the utilities functions*

```
namespace utilities {
    template <typename T> int indexOf(std::vector<T> v, T element){
        int pos = std::find(v.begin(), v.end(), element) - v.begin();
        return pos < (int) v.size()? pos : -1;
    }

    template <typename T> int sign(T x){
        return (x > 0) - (x < 0);
    }

    int random_integer(int min, int max){
        static std::random_device rd; static std::mt19937 rng (rd());
        std::uniform_int_distribution<int> uni(min,max);

        return uni(rng);
    }

    Pair<double> normalize(Pair<double> vector){
        double n = std::sqrt(std::pow(vector.first,2) +
                             std::pow(vector.second,2));
        return Pair<double>(vector.first/n, vector.second/n);
    }

    double radian(double angle){
        return angle*M_PI/180;
    }

    double degrees(double angle){
        return (angle > 0 ? angle : (2*M_PI + angle)) * 360 / (2*M_PI);
    }
}
```

The last function of this namespace, not shown in Code 5.31, is called *hash32*, which, as its name indicates, calculates the hashing function of a 32-bit element; we need this function because we rely on computing unique hashing values for small and medium sized integers to save on performance in a lot of structures, and the standard hashing function in C++ for integers is the identity. The identity function is useful to uniquely identify a single integer value, but for pairs it can be extremely problematic, as it is easy to find a pair with the same hashing value. This problem is further intensified by the fact that we want the hash value of the pair (a,b) to be equal to the one of (b,a) (for example, in SegmentMap), but different from all other possible pairs; for example, using the identity function makes that the pairs (2,3), (3,2), (1,4) and (4,1) all equal, which is not the desired behavior. Because of all these issues, we decide to implement a hashing function of our own, show in Code 5.32, that uses a big number, and several bit operations, to compute a value that uniquely identifies each integer.

Code 5.32: *Computation of the hash value for a 32-bit value*

```
int hash32(int key) {
    int c2=0x27d4eb2d;

    key = (key ^ 61) ^ (key >> 16) + (key << 3);
    key = key ^ (key >> 4)*c2;
    key = key ^ (key >> 15);

    return key;
}
```

Chapter 6

Results

In this chapter we will present both forms of results, theoretical and experimental, of the three libraries implemented. We start showing the big-O analysis of the algorithms, followed by the results obtained from the time experiments; we finish the chapter showing the numerical results that arose from this thesis' objectives, guided by the specific experiments presented in Section 4.4.

6.1 Theoretical results

As mentioned in Section 4.4, we will dedicate this section to the study of the theoretical complexities of the implemented algorithms, so to have a manner of comparison when doing the experiments on the performance of the libraries. This analysis is important as it allows us to have a general notion of how the system would work as the number of points, or elements, considered grow, which is extremely useful in our case, as our resources are limited, and it is hard, or downright impossible, to test extreme cases. However, as it is out of the scope of this thesis, we will not use mathematical proof for this study, but instead rely in analysis, known orders and literature for our deductions.

First, we have studied the complexity of several smaller algorithms as they were presented, in both the background chapter and the implementation of the Voronoi generation library, which we will now gather and summarize on a single section. After that summary, we will proceed to study the two most relevant algorithms of this thesis: the generation of Voronoi diagrams and the setup and calculation of the displacements through VEM and FEM. Finally, it is important to note that some of the procedures used by these algorithms are from external libraries, so we will refer to their authors for approximate complexity, or use theoretical values in case those are not available.

Finally, in this complete section, otherwise stated, we use N as the number of points of the mesh, and n the number of vertices of a mesh polygon.

6.1.1 Utilities

Before starting with the analysis of the complexities of the important algorithms of this thesis, we will first refer to the time order of the different implementations presented as part of the utilities module, as they will be referenced several times throughout this section.

In the first place, we can easily see that all Pair methods are constant, since they are either accesses, assignments or constant operations, such as sums and multiplications; hence, we will not remark further on this.

UniqueList methods are a little harder to analyze, since they include operations on two data structures at the same time; as we have explained in Section 5.4.1, UniqueList uses both a vector and map, the later only to ensure uniqueness of the elements. The map that we choose is based on binary trees, so both insertion and search are $O(logN)$, with N the size of the map; on the other hand, the vector operations chosen are strictly constant, such as access by index, and size. This way, the most commonly used methods of UniqueList, insertion and access by index, are $O(logN)$ and $O(1)$, respectively. Finally, the remaining operations implemented in the utilities are either obviously constant (such as the computation of the sign, radian and degrees) or simple to determine, like $indexOf$, which is a simple linear search through a vector. The only function which order could lead to doubt is $hash32$, but as it can be seen in its implementation, it consists of arithmetic and bitwise operations, so it is also constant.

6.1.2 Complexities summary

We will first present, in Table 6.1, a summary of the time orders of the basic functions that were shown as part of the geometric background; the methods that are particular to our implementation can be seen in a separate section next.

6.1.2.1 Basic functions

In this subsection, we denote n, as usual, as the number of vertices of a given polygon.

Table 6.1: *Basic geometric functions orders*

	Order
Segment intersection(2.1.4.1)	O(1)
Point on Segment(2.1.4.2)	O(1)
Polygon area(2.1.4.3)	O(n)
Polygon centroid(2.1.4.3)	O(1)
Polygon diameter(2.1.4.5)	O(nlogn)
Point in polygon(2.1.4.6)	O(n)

6.1.2.2 Custom methods

By custom methods we refer to those functions that are used by the generation of the Voronoi diagram, or the VEM algorithm, and were not presented as theoretical background, since they are specific to our needs. All these algorithms have been analyzed as they were presented, so in this section we will just summarize in a table all the complexities; again, all algorithms denote n as the number of vertices of the polygon in which the procedure is ran. We present the general order of the methods presented at the end of the Voronoi generation library in Table 6.2.

Table 6.2: *Orders of the custom geometric methods*

	Order
Self-intersecting polygon test(5.1.2.1)	O(n^2)
Check (and fix) orientation(5.1.2.2)	O(n)
Ear clipping triangulation(5.1.2.3)	O(n^2)

6.1.3 Main algorithms analysis

After we have summarized the complexities of all methods that are included as part of the libraries, we can analyze the two most important algorithms that make up most of our work, the Voronoi generation diagram and the computation of the displacements using VEM, each in its own subsection next; we include the analysis of the cost of FEM in those steps where it diverges from VEM at the end of the relevant subsections.

6.1.3.1 Voronoi diagram generation

We start with the Voronoi generation algorithm, which will be analyzed following the same steps seen both in Figure 4.1 and Section 5.1.1; we will go through each sub procedure referring to its implementation, and using the tools presented in Table 6.1 and Table 6.2, concluding with the complexity of the complete algorithm.

Domain processing As we mentioned in Section 5.1.1.1, before starting the computation of the Voronoi diagram, we need to process the domain, and, to that end, we use polygon intersection in the form of the Clipper library; however, since this processing is done on a single Region, which most likely will have few vertices when compared with the number of points of the mesh, it is considered negligible and not included on the complexity analysis. For further reference, the computation of the intersection of two polygons using Vatti's algorithm (implemented by Clipper) is $O(nm)$[47], with n and m the number of vertices of the two polygons.

Seed point generation The generation of the seed points for Delaunay is completely dependent on the user input, so it is harder to analyze than most aspects of this algorithm. For example, if the user inputs the seed points directly, we do not need to process them at all, so it is $O(1)$; on the other hand, when the points are generated by the library, the complexity is mostly dependent on the number of points desired on each axis, and the generation functions, both parameters given by the user. The number of points is not negligible, since they define the minimum number of triangles of the Delaunay triangulation, which, in turn, determine the number of polygons of the mesh; but, as their generation is done only once, it could be considered part of the start-up cost of the algorithm, and will be ignored unless it really slows down the whole process.

Generation of Delaunay input data We use an external library, Triangle, for the computation of the constrained conforming Delaunay triangulation of our domain; to call Triangle we need to transform the information entered by the user to the form required by the library. This procedure is shown in Algorithm 10; we can see that it is just three for-loops with no complex operations inside (all of them are trivially $O(1)$), so the cost depends completely on the number of iterations, given by: the number of seed points, the number of segments of the boundary region and the number of holes, respectively; the latter two are most likely small, and the former depends completely on user input, so we just leave it as a parameter, giving us a total cost of $O(N)$, with N the number of seed points (which is similar to the number of points of the mesh).

Delaunay triangulation computation The actual computation of the Delaunay triangulation is done by Triangle, which has time complexity $O(NlogN)$, with N the number of seed points, as it is declared by its author.

Processing of Delaunay output The raw data returned by Triangle is not useful for the Voronoi algorithm, hence the need of processing it; this process is shown in Algorithm 11. It is easy to note that this part of the algorithm mostly consists on iterations through different structures; inside this for-loops there are different operations: access to both vectors and maps (constant time), insertions on the same structures ($O(1)$ on average), and some custom ones, such as $setAsBoundary$ (switch the value of a flag), and $setEdge$ and $setTriangle$ (assign a value), that are obviously $O(1)$. Hence, the cost of this procedure is determined completely by the loops: one depending on the number of pointsof the mesh, two on the number of edges, and one on the number of triangles; we note that the number of points, segments and edges are related, and all $O(N)$, following Euler's formula, so the final cost is $O(N)$.

Voronoi diagram from Delaunay triangulation The Voronoi diagram generation algorithm was divided in several steps to ease of analysis, as shown in Algorithm 12; we readily note that the for-loop of the algorithm iterates over all the points of the Delaunay triangulation, giving us a base $O(N)$ complexity, so, depending on the cost of each step, we could preserve this order, or increase it.

Initial cell computation We note in Algorithm 13 that almost all the steps in this algorithm are variable initialization from constant time access to data structures: both accessing by index on a vector, or by key on hash maps.

The only operations that could give doubts are $getCircumcenter$ and $nextEdge$. The first procedure (Algorithm 17) includes an if condition with a constant time check, and each branch is also constant: the first gets a previously calculated value from Triangle; while the second calls the $middlePoint$ method from Segment, that computes averages, so it is still $O(1)$. The second method uses three comparisons to determine the point that is not on the currentEdge, creates the key representing the edge, finally getting it from a hash map, all $O(1)$ operations.

Finally, all these operations are done only once (to start the algorithm), so this sub procedure is $O(1)$.

Iteration procedure The procedures inside this part of the algorithm are the same as the previous section, so the order only depends on the loop. The number of iterations is determined by the number of edges that share a given point, which varies on different meshes; however, from the literature[32] it is known that the average number of segments incident to a point in a Delaunay triangulation is six, so the iteration can be considered $O(1)$.

Boundary cases First, and most importantly, this part of the algorithm is not even done most of the times, since it involves closing the cells that are on the boundary of the domain; we first check the final edge of the Voronoi cell (on $O(1)$), and then do several operations when we detect that we are working on a boundary cell, all of them already studied. and determined to be $O(1)$.

Creation of cell This section is simple, as it consists only in the creation of a new Polygon instance that represents the cell, and the insertion of the corresponding segments in the SegmentMap. Both operations depend on the number of vertices of the Voronoi cell, but we can use the same idea of Section 6.1.3.1; since the number of segments incident to a point is six, and considering that the generation of a Voronoi cell depends on the number of segments that share a Delaunay point, we can conclude that the number of vertices of said cell is also around six, hence $O(1)$.

The creation of a Polygon involves the computation of its diameter, its centroid and area, of which only the first one is $O(nlogn)$ (the rest are $O(1)$), but since the number of vertices of the polygon is constant on average, this cost is considered irrelevant; however, the check for self-intersection is quadratic on the number of vertices, so creating a polygon is $O(n^2)$. On the other side, the insertion of the segments has cost $O(1)$ for each segment, and since we concluded that there are $O(1)$ segments, the complete operation is $O(n^2)$, defined by creating the cell.

Complete algorithm Using the previous analysis, we can conclude that the generation of the Voronoi diagram from the Delaunay triangulation is $O(N)$, as it is an $O(N)$ for-loop with $O(1)$ operations each step. Therefore, the complexity of our algorithm is equal to the maximum value between the computation of the Delaunay triangulation (reported by Triangle's author to be $O(N)$) and the cost of computing the Voronoi diagram from the Delaunay triangulation ($O(N)$ from our analysis). In conclusion, the complete algorithm that takes a domain with seed points and computes the Voronoi diagram is $O(NlogN)$, determined by the cost of computing the Delaunay triangulation.

6.1.3.2 Linear elasticity solving

We will now analyze the time complexity of the algorithms shown in Figure 4.2 and Figure 4.3, and implemented in Section 5.2.1 and Section 5.2.2, again following the steps used to present the implementation. We go procedure by procedure analyzing the algorithms, ending with the conclusion of the complexity of the complete algorithm.

Initialization of the numerical system To analyze the cost of the process that initializes all the necessary variables used to represent the numerical problem, we use the implementations shown in Code 5.6, Code 5.7 and Code 5.8. We analyze for one element, and simply assume that all the rest are more or less similar.

For an Element with associated Polygon p, the process starts with a simple iteration through all the vertices of p, calling the method $addDOF$ of DOFS, and inserting the result on a vector; both the iteration and the vector insertion have costs that are trivial to compute ($O(n)$ and $O(1)$, respectively), but we do need to analyze $addDOF$ in more detail to determine its complexity.

We presented the code for the $addDOF$ method in Code 5.6. The cost depends on whether the vertex of interest has already associated degrees of freedom or not; for simplicity of the analysis, we will assume that the better case, where the creation is not necessary, never happens, and always work in the worst scenario, creating a degree of freedom from scratch each time. The procedure starts by searching in a hash map ($O(1)$), then inserting in a UniqueList ($O(logN)$ as seen in Section 6.1.1), finally calling yet another method that we will analyze next, called $addConstrainedDOF$.

$addConstrainedDOF$ calls the function *addConstrainedDOFBySegment*(Code 5.7) twice, once for each incident segment to the point that the DOF represent and *addConstrainedDOFByPoint* once; this way, the cost of *addConstrainedDOF* is twice the cost of the *addConstrainedDOFBySegment* plus the cost of *addConstrainedDOFByPoint*. As we commented in Section 5.2.1, the procedures for PointConstraints and SegmentConstraints are equal, so we can just analyze *addConstraintDOFBySegment* to conclude about the cost of *addConstrainedDOF*. We can see on Code 5.7, we have three main procedures on this algorithm: the call to $isConstrained$, an if with a series of accesses and a switch, and an insertion in a *UniqueList*; for the last two it is direct to see their costs, $O(1)$ and $O(logN)$, respectively.

i*sConstrain*ed was shown upon in detail in Code 5.8, and will be analyzed here using its implementation as reference. i*sConstrain*ed receives a Segment, computes its angle ($O(1)$, as it is simply a formula), searches in the map for a vector of Segments related to the angle; if it is not found, we immediately return false. The average case, which defines the cost of the method, occurs when we find a match in the map, and we need to linearly search through the vector of Segments; the analysis in this case depends highly on the conditions on the problem, since, in the worst possible case, all the constrained segments have the same angle, and the map optimization is completely useless, giving us a cost of $O(M)$, where M is the number of constrained segments of the system. However, on most cases, the number of constrained segments is either small, or has varied angles, so the total cost of this method would be $O(1)$ most of the time, since the search in the map would discard most segments, making the linear search only on a small portion of the total number.

This way, the total cost of creating an Element is $O(nlogE) = O(nlogN)$ (where n is the number of vertices of the polygon related to the Element, and E is the total number of degrees of freedom of the system, two times the number of points of the mesh), determined by the iteration over all vertices of the polygon, and the search and insertion in the *UniqueList*; as the number of vertices of any polygon is negligible when compared with the number of degrees of freedom, we can consider this operation to be $O(logN)$. Finally, as we have as many elements as polygons on the mesh, and, as will be analyzed in detail in the post processing step, the number of polygons is assumed to be $O(N)$, the cost of initializing the numerical system is $O(NlogN)$ (obtained adding $O(logN)$ N times).

Computation of the displacements After the analysis of how the numerical system set ups, we can now, using this information, analyze the displacements computation algorithm itself; as we saw in previous occasions, we take advantage of the fact that the algorithm was already divided in clear steps during its design, and use this same division to analyze its complexity.

Elemental matrix Each Element computes its elemental matrix each time it is needed for the numerical computation, since, after applying the loads, the polygon the Element represents is deformed, changing its geometrical properties; as the elemental matrix in VEM depends entirely on the shape of the polygon, keeping the matrix after the polygon changes is useless. It is important to note that we recompute the matrix even if the polygon does not change, so this cost is always present.

The actual cost of the elemental matrix is easily determined from its definition, as it is divided in two steps: first, it does several constant time operations (simple multiplications and additions) for each vertex of the polygon associated to the element (a $O(n)$ loop), obtaining four matrices; next, we need to multiply these matrices, which size depends on the number of vertices of the polygon (all of them are $2n \times 3$). The cost of matrix multiplication depends on Eigen's implementation, and it is not informed in their documentation, so we assume the naive cost of $O(npm)$ when multiplying two matrices of sizes $n \times p$ and $p \times m$.

To obtain the projection matrix, the actual matrix multiplication needed is shown in Equation (2.27), which cost is $O(2n \times 3 \times 2n) = O(12n^2)$; finally, using the matrices P_r and P_c we obtain, by simple sum, P_p of size $2n \times 2n$, so the cost of computing the elemental matrix is $O(2n \times 2n \times 2n) = O(n^3)$, determined by the multiplication of P_p and its transpose.

In FEM, the stiffness matrix must be computed using numerical integration, process explained in Section 5.2.3. The process of numerically integrating a function consists in one call to the function for each of the Gauss point desired; by default, we use three Gauss points for the integration, hence, as each element is a triangle, the iteration for integration is $O(n)$. The function that is integrated consists on the computation of the B_e matrix, and then a multiplication with its transpose and the material matrix. The first operation is shown in Code 5.19, and is obviously $O(n)$, as both the number of iterations and the computation of the shape functions are equal to the number of vertices; the cost of the second operation is equally as simple to determine: B_e is size $3 \times 2n$ and C is 3×3, so, using the formula presented in the previous paragraph, we conclude it is $O(2n \times 3 \times 2n) = O(n^2)$.

This way, the total cost of computing the stiffness matrix is $O(n^3)$, where n is because of the iteration over the Gauss points, while n^2 is the cost of computing the term inside the integral ($O(n^2)$).

Load vector The cost of computing the load vector for a given Element is determined by the computation of the surface integral and the line integral; the first one is done only once, as it is equal for all degrees of freedom of the element, but the second is done inside a for-loop. The surface integral computation is an iteration over all vertices of the polygon, computing the area of a triangle ($O(3)$, since a triangle always has three points and the computation of the area of any polygon is $O(n)$) and doing sums and multiplications; therefore, the total cost is $O(n)$.

On the other side, the line integral must be calculated once for each degree of freedom of the element; the computation itself is shown in Code 5.13. We can see that it consists on: a call to the version of i$sConstrain$ed that receives a DOF index, which cost is constant, since it consists on a lookup on a hash map; two calls to the version of the method that receives a segment, which was analyzed before, and determined to be $O(1)$ in average case; plus other constant operations, such as ifs and access to map elements. The only aspect which cost we have not analyzed is the $getValu$e method of Constraint, that calls the $getValu$e method of an implementation of the ConstraintValue interface, completely dependent on user input; however, it is constant in the two implementations of ConstraintValue given to the user (Constant and Function), so we assume that order, as those two are the most common cases.

In conclusion, the cost of computing the line integral is $O(1)$; as we need to do this procedure once for each degree of freedom, this part of the load vector has cost $O(2n) = O(n)$, as the number of degrees of freedom of an element is twice the number of vertices of its related polygon.

Finally, we can conclude that the complexity of the load vector calculation is $O(n)$; this cost is determined by the cost of the surface integral and the for-loop of $O(1)$ operations.

As it was the case with the stiffness matrix, in FEM the load vector is computed using numerical integration; however, the separation between body force vector (computed once for each element) and traction vector (computed for each degree of freedom) is the same as what was shown with VEM, so our analysis will proceed in the same order.

The body force vector is computed using area integration, analyzed in the stiffness matrix calculation, and determined to be $O(n)$ times the cost of the function inside the integral. The function, shown in Code 5.20, contains a call to the shape functions ($O(n)$) and an iteration over the polygon vertices, plus a few constant time operations. This way, the body force vector is calculated in time $O(n^2)$.

On the other side, the traction vector requires line integration, which we have not analyzed until now; the line integration procedure iterates over each Gauss point (by default we use 3, so it is not negligible when compared with the number of vertices of the elements), calling the internal function each time. The function inside the line integral, shown in Code 5.21, has a call to the shape functions ($O(n)$) and all constant operations. This way, the computation of the traction vector is $O(n^3)$, n for the computation of each degree of freedom, n for the line integration and n for the function inside the integral.

Assembly The assembly procedure (shown theoretically in Algorithm 9, while our implementation can be seen in Section 5.2.2.3) consists on accesses to matrix and vector values, and assignments with sums, all of them trivially constant. That way, the cost of the assembly is defined by the nested for-loops, and the number of elements that do this procedure, giving us a cost of $O(E \times N^2)$, with E the number of elements and N the number of vertices of the mesh; this is easily computed changing the order of the for-loops in Algorithm 9, iterating first on all nodes twice, and only then considering the relevant elements for each one (the number of elements that share a given degree of freedom depends on the mesh, but it is small in most non degenerate cases).

Imposition of boundary conditions This part of the algorithm involves only the essential boundary conditions, since the natural ones are included in the computation of the load vector; the procedure is shown in Code 5.15. Its analysis is extremely simple, as there is a single operation outside the for-loops that is not constant, the call to $getBoundaryValues$; this method consists on successive calls to the $getValue$ method of ConstraintValue instances, which, as mentioned before, is assumed to be constant. This way, we consider the total cost of $getBoundaryValues$ to be $O(1 \times c) = O(c)$, where c is the number of constrained degrees of freedom of the system.

On the other side, the for-loops costs are easily calculated from their ranges: the first one depends on the number of constrained DOF, and the second one depends on the number of DOF (twice the number of vertices of the mesh); the former is determined by the problem conditions, so the total cost is $O(2N \times c) = O(Nc)$.

Finally, we can conclude that the total cost of imposing the boundary conditions on the system is determined by the cost of the for-loops, and it is equal to $O(nc)$.

Linear system solving An important operation in this algorithm is to obtain the displacements, which means we need to solve a linear system that is not necessarily stable; hence, as mentioned in Section 5.2.2.5, simply inverting the stiffness matrix (which is an already expensive operation, $O(N^3)$ in worst case, $O(N^{2.373})$ when optimized) is not an option, and instead need to use a full pivoting QR solver. Since matrix algebra is a complex matter that is considered given in this project, we use Eigen's QR solver, which time complexity is not known, but assumed to be equal to the theoretical one, $O(N^3)$.

Complete displacement calculation algorithm The total cost of the displacement calculation for both VEM and FEM is given by the following points. We use: E as the number of elements of the mesh, n as the number of vertices of an element and N, the number of points of the mesh.

1. Computing E elemental matrices, of cost $O(n^3)$ each for both methods.
2. Computing E load vectors, of cost $O(n)$ each in VEM, $O(n^3)$ in FEM.
3. Assembling the elemental and load vectors, with cost $O(N \times n^2)$ for both methods.
4. Solving the linear system, with cost $O(N^3)$ for both methods.

Hence, the cost of the complete algorithm, for both methods, is dominated by the resolution of the linear system, giving a total order of $O(N^3)$ for the complete displacement calculation algorithm.

6.2 Experimental results

As commented in the beginning of this chapter, after studying the theoretical performance of the implemented algorithms, we now show the experimental results, generated following the guidelines presented in Section 4.4.1.

The results of the two libraries that could be analyzed completely are presented next, each in their own subsection, starting with the Voronoi generation library, and followed by the VEM library.

6.2.1 Voronoi generation

We present the results of the time experiments on the Voronoi generation library in Figure 6.1, including two fitting functions, linear and quadratic, that help us determine the order of the algorithm from the experimental data.

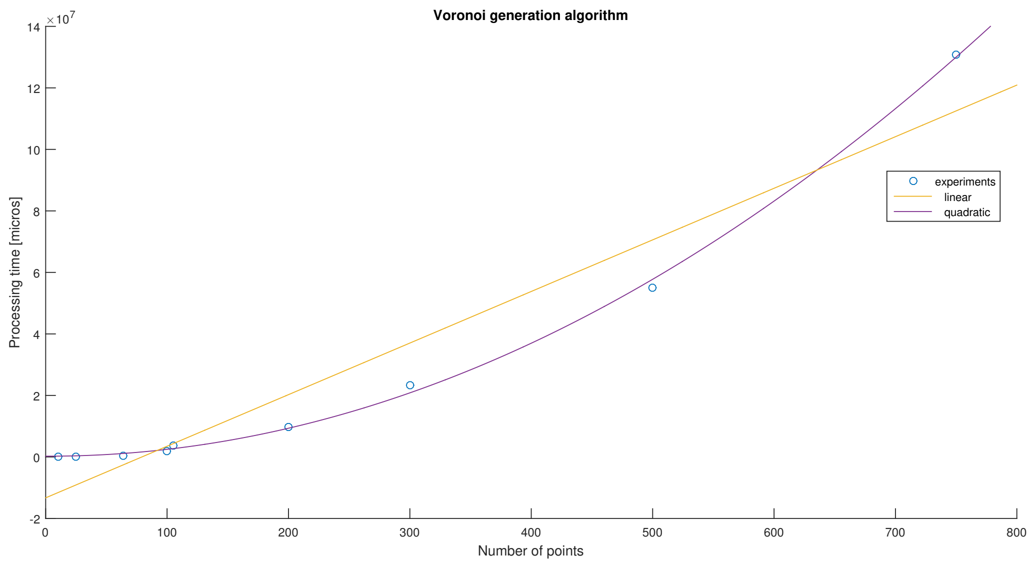


Figure 6.1: *Results of time experiments on the Voronoi generation library*

From the results shown in Figure 6.1, we can see that the time complexity of the Voronoi generation algorithm seems to fit better to a quadratic curve, than it does to a linear one; as we expect a $O(nlogn)$ complexity, which should fit better to a linear function, these results hint that our implementation is not as efficient as we would like. We remark, however, on the fact that our algorithm is not exactly quadratic, since, as the number of points grow, the data does not fit as well as it does when n is smaller; this way, we can say that we are probably closer to a complexity of $O(n^m)$, with $m \leq 2$. Finally, using the time tests, we can conclude that our implementation of the Voronoi generation algorithm needs optimizing to reach the theoretical order determined in the theoretical analysis stage; however, the actual process of optimizing after the implementation is out of the scope of this thesis, and left as future work.

6.2.2 Virtual Element Method

We present the results of the time experiments on the Virtual Element Method library in Figure 6.2, including a cubic fitting function so that is easier to determine the trend of the data. We present the results of the exact same experiments but using the Finite Element Method in Figure 6.3.

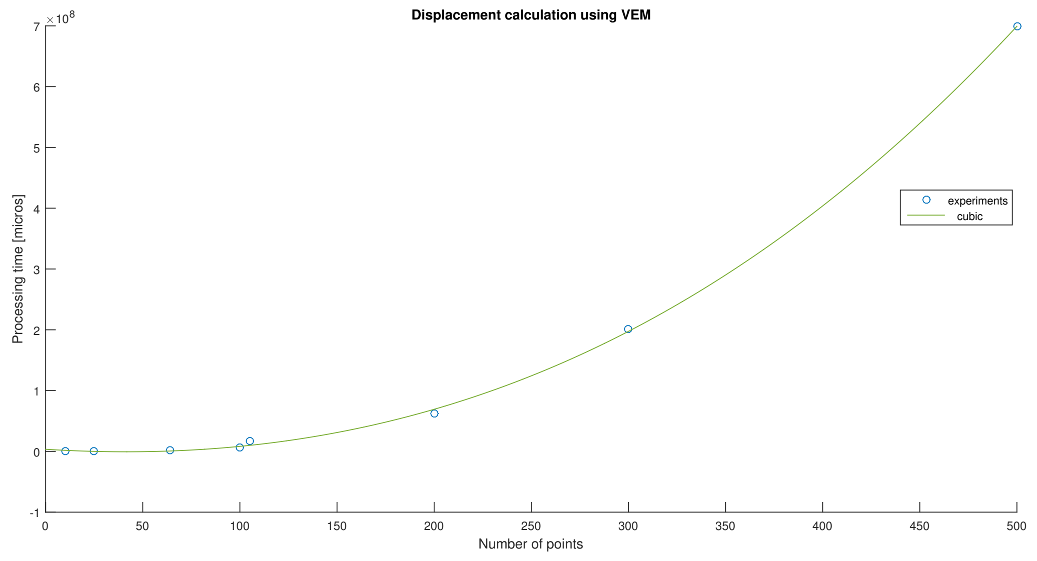


Figure 6.2: *Results of time experiments on the Virtual Element Method library*

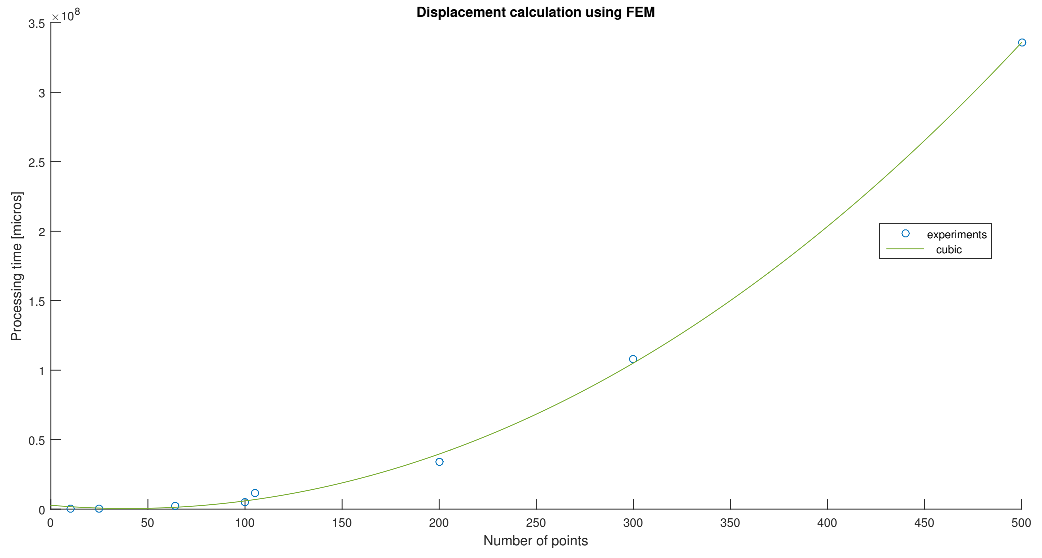


Figure 6.3: *Results of time experiments on the Finite Element Method module*

From Figure 6.2 and Figure 6.3, we can readily conclude that our analysis, which told us that the cost of both VEM and FEM is $O(n^3)$, was spot on, to the point that the data obtained from the experiments fit almost exactly to the cubic curve shown; this way, we can conclude that the complexity of the VEM library is indeed determined by the cost of solving the linear system, which, as mentioned before, is a common issue to all numerical simulation software.

6.3 Numerical results

In this section we present the results of the numerical experiments first explained in Section 4.4.2; we divide this section following the same separation used in the design chapter, first showing the numerical convergence test, and then the modifiable mesh tests. We divide the results by experiments, first presenting the test case, including the analytical solutions against which the numerical results will be compared, followed by the results and a small analysis.

6.3.1 Convergence tests

We include here the three tests cases that are used to validate the numerical solution obtained in both VEM and FEM; the error in the solution is calculated using both the L2 and H1 norms, calculated as presented in the implementation chapter.

One important aspect that is required for the convergence tests is to compute, from the known displacements functions, the analytical strain and stress; as we commented in Section 5.2.4, our implementation of the norms calculation assumes those functions to be known, so here we require to compute them. Here we include the general formula for the strain and stress assuming the displacements are known; this way, each test can obtain the required values just replacing the displacement: the strain is computed from the displacements following its definition (formula shown in Equation (6.1)), while the stress is obtained from the stress using the material matrix (we include the formula for plane strain in Equation (6.2), and for plane stress in Equation (6.3)).

$$\boldsymbol{\varepsilon} = \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_{xy} \end{Bmatrix} = \begin{Bmatrix} \frac{\partial u_x}{\partial x} \\ \frac{\partial u_y}{\partial y} \\ \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} \end{Bmatrix} \tag{6.1}$$

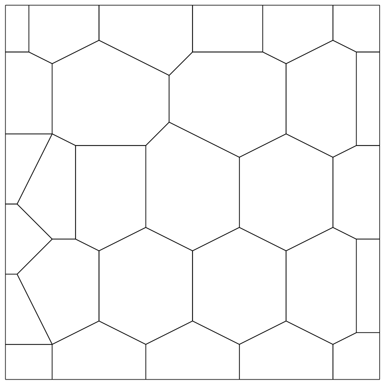
$$\boldsymbol{\sigma} = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_{xy} \end{Bmatrix} = \frac{E}{(1+\nu)(1-2\nu)} \begin{Bmatrix} (1-\nu)\varepsilon_x + \nu\varepsilon_y \\ \nu\varepsilon_x + (1-\nu)\varepsilon_y \\ \frac{1-2\nu}{2}\varepsilon_{xy} \end{Bmatrix} \tag{6.2}$$

$$\boldsymbol{\sigma} = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_{xy} \end{Bmatrix} = \frac{E}{1-\nu^2} \begin{Bmatrix} \varepsilon_x + \nu\varepsilon_y \\ \nu\varepsilon_x + \varepsilon_y \\ \frac{1-\nu}{2}\varepsilon_{xy} \end{Bmatrix} \tag{6.3}$$

6.3.1.1 Patch test

The patch test is a common, and simple to calculate, indicator of the quality of a numerical solution. The patch test consists in a solution of a partial differential equation that has an exact solution that can, in principle, be exactly reproduced by the numerical approximation.

In our case, we choose a unitary square domain in which we impose the Dirichlet boundary condition $\boldsymbol{g} = \{x \quad x+y\}^T$ on the entire boundary, with no body forces; we use plane strain conditions with material constants $E = 1^7$ and $\nu = 0.3$. The meshes are made of polygons in the case of VEM, and of triangles in the case of FEM; examples of the meshes used for the patch test can be seen in Figure 6.4.


(a) Mesh for patch test - VEM

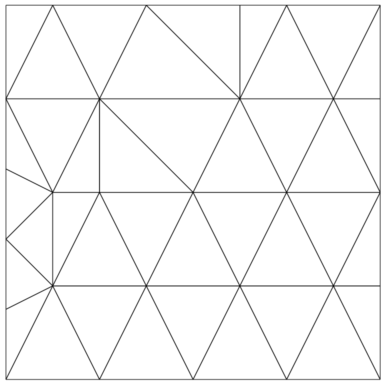

(b) Mesh for patch test - FEM

Figure 6.4: *Example of meshes used for the patch test*

As the imposed conditions are linear, the solution obtained must reproduce the Dirichlet conditions imposed; to obtain the analytical strain and stress functions, we apply the formulas shown in Equation (6.1) and Equation (6.2). We summarize the analytical solutions for displacement, strain and stress in Equation (6.4).

$$\begin{aligned} \boldsymbol{u} &= \{x \quad x+y\}^T \\ \boldsymbol{\varepsilon} &= \{1 \quad 1 \quad 1\}^T \\ \boldsymbol{\sigma} &= \frac{E}{(1+\nu)(1-2\nu)} \left\{1 \quad 1 \quad \frac{1-2\nu}{2}\right\}^T \end{aligned} \tag{6.4}$$

We present the results of the calculation of both norm for the patch test in Table 6.3, where we can readily note that both FEM and VEM present results that coincide with the expected up to machine precision. The only remarkable point to comment about the results is that the norms tend to grow with the number of elements (the opposite to what is expected), but that can be explained considering that is simply because it is the accumulation of the machine precision errors.

Table 6.3: *Norms calculated for the patch test*

Number of elements	Norm L2 - VEM	Norm L2 - FEM	Norm H1 - VEM	Norm H1 - FEM
25	2.56406×10^{-16}	8.34987×10^{-17}	1.30573×10^{-15}	1.81808×10^{-16}
49	5.64144×10^{-16}	1.2132×10^{-15}	1.61415×10^{-15}	1.51918×10^{-15}
100	7.98875×10^{-16}	2.84987×10^{-15}	2.36541×10^{-15}	3.41229×10^{-15}
225	2.19786×10^{-15}	6.89787×10^{-15}	3.61921×10^{-15}	8.90478×10^{-15}

6.3.1.2 Equilibrium patch test

The equilibrium patch test is another version of the patch test, but that imposes both Neumann and Dirichlet boundary conditions, so it is useful to test conditions that are out of the scope of the standard patch test. Again, we use the unitary square, in which the lower segment is constrained vertically, with the lower right point also constrained horizontally; a unitary vertical force is applied on the upper segment. We do not show the meshes used for the equilibrium patch test as they are the same as the ones shown in Figure 6.4. The analytical solution to the problem are shown in Equation (6.5), while the norms are tabulated in Table 6.4.

$$\begin{aligned} \boldsymbol{u} &= \{x \quad x+y\}^T \\ \boldsymbol{\varepsilon} &= \{1 \quad 1 \quad 1\}^T \\ \boldsymbol{\sigma} &= \frac{E}{(1+\nu)(1-2\nu)} \left\{1 \quad 1 \quad \frac{1-2\nu}{2}\right\}^T \end{aligned} \tag{6.5}$$

Table 6.4: *Norms calculated for the equilibrium patch test*

Number of elements	Norm L2 - VEM	Norm L2 - FEM	Norm H1 - VEM	Norm H1 - FEM
25	3.88022×10^{-15}	1.08431×10^{-14}	5.8988×10^{-23}	1.03126×10^{-14}
49	9.99186×10^{-15}	1.7586×10^{-14}	9.37374×10^{-23}	1.18819×10^{-14}
100	1.22194×10^{-14}	3.67711×10^{-14}	8.91267×10^{-23}	1.37925×10^{-14}
225	1.43946×10^{-13}	2.37885×10^{-14}	3.71054×10^{-23}	1.69996×10^{-14}

Again, we note that the errors in the norms, although greater than what we obtained for the Patch test, fall in the category of machine precision errors, validating both implementations.

6.3.1.3 Parabolic beam test

Differently from both previous tests, this one will be presented in the spirit of the numerical experiments commonly found in the literature; we will plot the obtained norms (for both methods) as the size of the mesh elements decrease, so that we can check on the rate of convergence of the method. The graphic for the L2 norm can be seen in Figure 6.5, while the one for the H1 norm is presented in Figure 6.6 for VEM and Figure 6.7 for FEM (this separation will be made obvious when the results are analyzed). As reference, we include a table with the norms calculated in Table 6.5 and Table 6.6, for the L2 norm and H1 norm.

The analytical solution for the **parabolic beam test** is presented in Equation (6.6); note that the stress solution is not included as it is easy to derive from the strain, and much more complex than the previous examples.

$$\begin{aligned}\boldsymbol{u} &= \frac{P}{6\bar{E}I}\begin{bmatrix} -y\left[(6L-3x)x+(2+\bar{v})y^2-\frac{3D^2}{2}(1+\bar{v})\right] \\ [3\bar{v}y^2*(L-x)+(3L-x)x^2] \end{bmatrix} \\ \boldsymbol{\varepsilon} &= \frac{P}{6\bar{E}I}\begin{bmatrix} -y(6L-6x) \\ 6\bar{v}y(L-x) \\ (\frac{3D^2}{2}-6y^2)(1+\bar{v}) \end{bmatrix}\end{aligned} \tag{6.6}$$

Where $E = 1{\times}10^7$, $\nu = 0.3$, $P = -1000$, $D = 4$, $L = 8$, $\bar{E} = ^{E}/_{(1-\nu^2)}$, $\bar{\nu} = ^{\nu}/_{(1-\nu)}$ and I is the second moment of area, which, as we assume the depth is unitary, is $I = ^{D^3}/_{12}$.

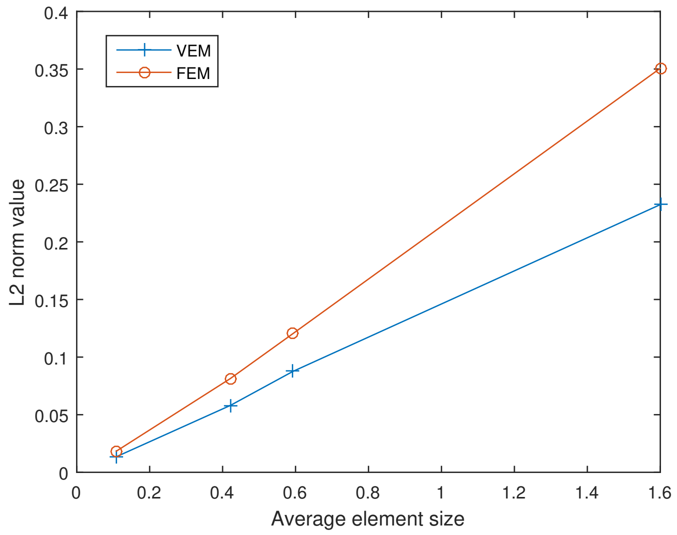


Figure 6.5: *L2 Norm for both methods*

Table 6.5: *L2 norm for the parabolic beam test*

Number of elements	Average element size	Norm L2 - VEM	Norm L2 - FEM
18 (6 × 3)	1.6	0.232334	0.350652
50 (10 × 5)	0.592593	0.0875514	0.120318
72 (12 × 6)	0.421053	0.0580081	0.0813466
288 (24 × 12)	0.108475	0.0134931	0.0183208

From the L2 norm graph and table, we can readily note that both VEM and FEM present lesser errors as the size of the element decreases, which coincides with the theory; VEM, however, consistently presents lower error, which further reinforces our hypothesis that VEM is better suited for most linear elasticity applications. With these results we can validate the convergence of the method in situations more complex than a simple patch test, which means that we can continue with the modifiable meshes study.

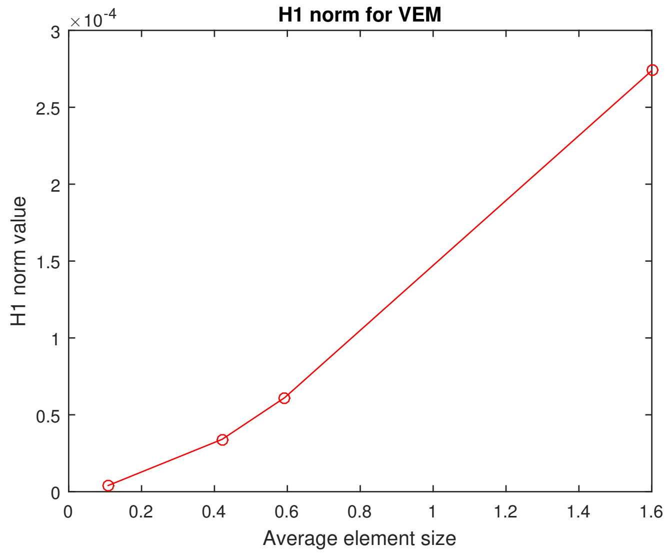


Figure 6.6: *H1 Norm for VEM*

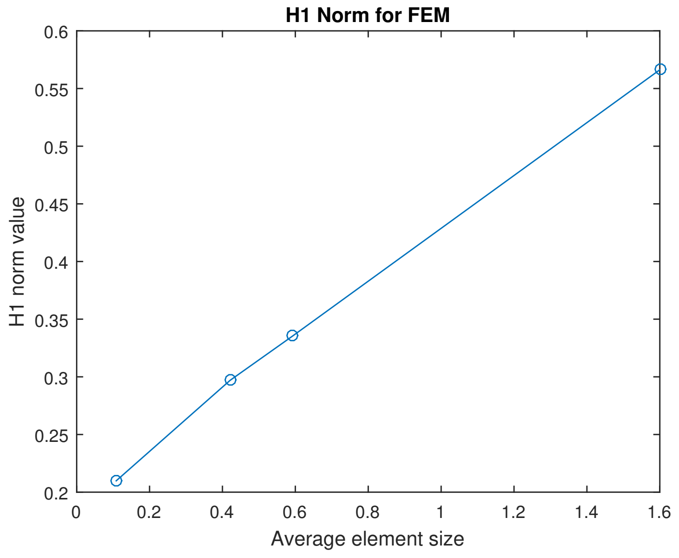


Figure 6.7: *H1 Norm for FEM*

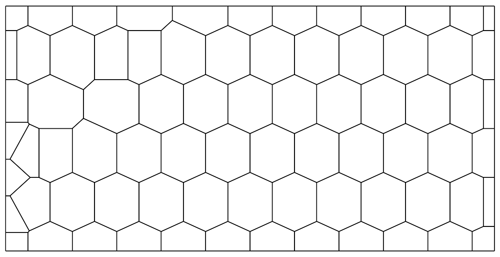
Table 6.6: *H1 norm for the parabolic beam test*

Number of elements	Average element size	Norm H1 - VEM	Norm H1 - FEM
18 (6 × 3)	1.6	0.000273841	0.566256
50 (10 × 5)	0.592593	6.11406×10^{-5}	0.335411
72 (12 × 6)	0.421053	3.40451×10^{-5}	0.296976
288 (24 × 12)	0.108475	4.02832×10^{-6}	0.209824

The first remarkable detail about the results presented in Table 6.6 are the seemingly incoherent scale of the VEM results; we explain this issue simply by admitting that the chosen approximation of the H1 norm (as we do not have access to shape functions) is probably not useful in our applications. We leave the implementation and evaluation of better approximation of the H1 norm for VEM as future work, and instead focus on the tendency of the data, which is meaningful. To better analyze the data, we plot the norms separately in Figure 6.6 and Figure 6.7, and note that, again, that both VEM and FEM error again tends to decrease with the size of the elements, which corresponds with the theory.

6.3.2 Modifiable meshes tests

In this section we will present the results to the experiments design to confirm the hypothesis presented by this thesis, which is, in short words, that VEM should give better results, numerically speaking in meshes with changes that are included after their creation; meaning, that including non-staged changes and not remeshing is less catastrophic in VEM than in FEM. To prove our hypothesis, we select two meshes for each test (one made of polygons, for VEM, and another made of triangles, for FEM), shown in in Figure 6.8 for the parabolic beam test and in Figure 6.9 for the pressurized cylinder test, and prepare a series of tests in which a change is applied to each mesh, and compare the nodal errors obtained solving the problem using both FEM and VEM; each of this tests will be explained in their own separate subsection, first presenting the geometrical changes and then the numerical errors. As a benchmark, we include an initial subsection which presents the nodal errors in both methods in a mesh without changes.


(a) Mesh for the parabolic beam tests - VEM

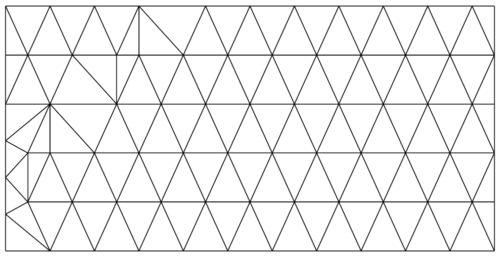
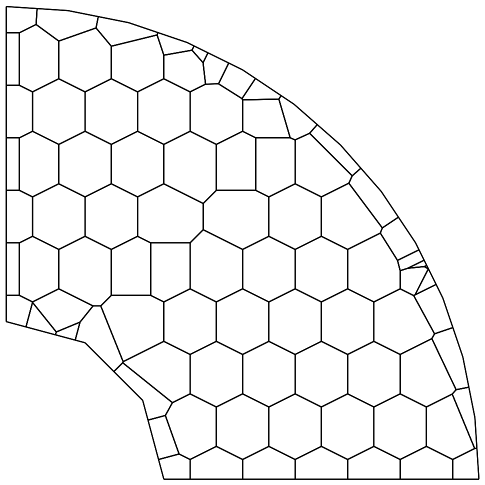

(b) Mesh for the parabolic beam tests - FEM

Figure 6.8: *Meshes for the parabolic beam tests, for both methods*


(a) Mesh for the pressurized cylinder tests - VEM

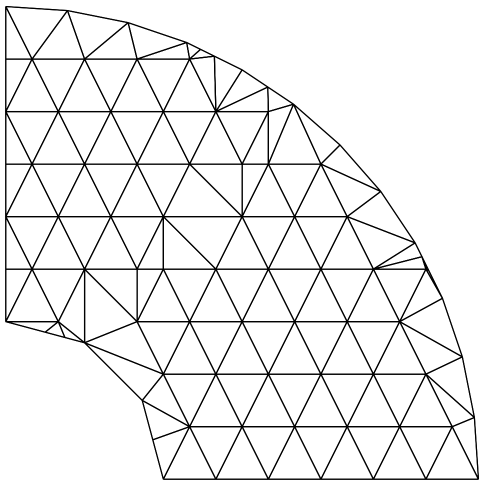

(b) Mesh for the pressurized cylinder tests - FEM

Figure 6.9: *Meshes for the pressurized cylinder tests, for both methods*

Finally, as it was commented in the design chapter, we use two different test cases: the parabolic beam test that was detailed in Section 6.3.1.3, and a new one, called pressurized cylinder test, which was shown in Section 4.4.2.2; the parabolic beam analytical displacement, necessary for our tests, were shown in Equation (6.6), so it will not be repeated here. The analytical solution of the pressurized cylinder, shown in Equation (6.7), is given in radial coordinates, but as we do know the Cartesian coordinates of each points, we can obtain the analytical displacement value for each point simply replacing $r = \sqrt{x^2 + y^2}$, with the approximated radial displacement $u_r = \sqrt{u_x^2 + u_y^2}$.

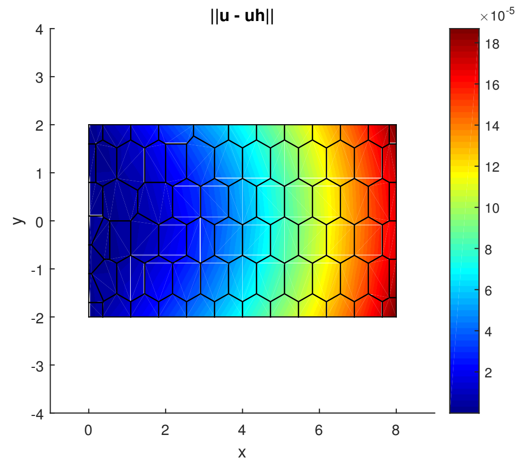
$$u_r = \frac{(1+\nu) r_{\mathrm{i}}^2 p}{E(r_o^2 - r_{\mathrm{i}}^2)} \left(\frac{r_o^2}{r} + r(1 - 2\nu) \right) \tag{6.7}$$

6.3.2.1 Original results

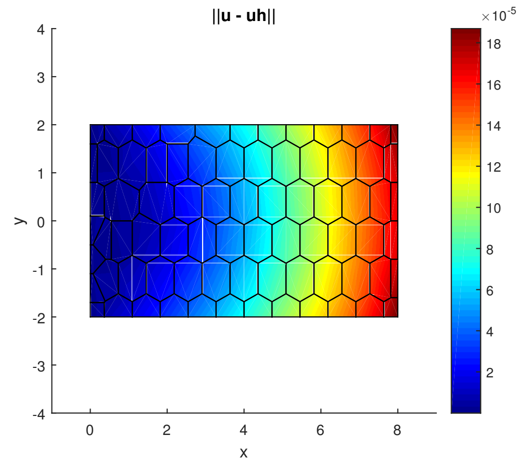
In this subsection we show the results obtained for the original meshes, so that all tests can be compared against the error value obtained for their respective unmodified mesh; as further help in the comparison, we include Table 6.7, that summarizes the minimum, maximum and average nodal error for the four different tests. The norm of the nodal error for the four meshes that we will use through these tests, can be seen in Figure 6.10.

Table 6.7: *Original error for the modifiable meshes tests*

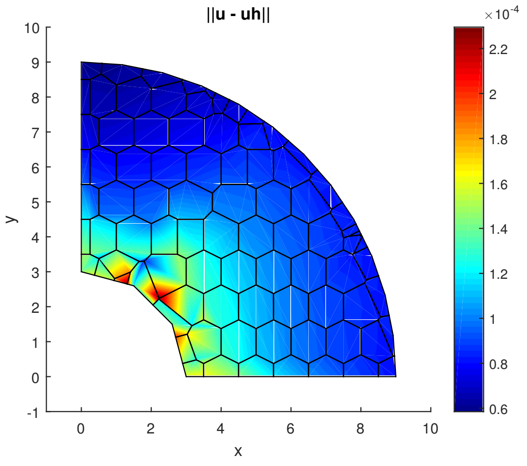
Experiments		Minimum nodal error	Maximum nodal error	Average nodal error
Parabolic Beam	VEM	2.8948×10^{-20}	1.8690×10^{-4}	7.0227×10^{-5}
	FEM	7.2371×10^{-21}	2.5466×10^{-4}	1.0177×10^{-4}
Pressurized cylinder	VEM	5.8539×10^{-5}	2.2953×10^{-4}	9.7145×10^{-5}
	FEM	8.4440×10^{-5}	4.2781×10^{-4}	1.8835×10^{-4}



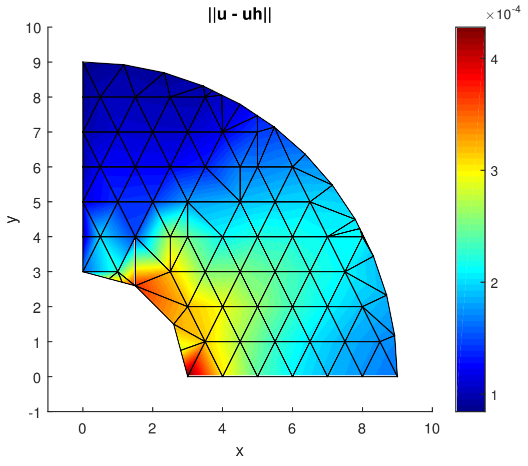
(a) Parabolic beam - VEM



(b) Parabolic beam - FEM



(c) Pressurized cylinder - VEM



(d) Pressurized cylinder - FEM

Figure 6.10: *Original results of the modifiable meshes tests*

We note that both tests present satisfactory results for the two studied methods, so we can safely use both for our studies. We will mostly rely on the parabolic beam because the shape of the domain makes it is easier to implement the different tests; we do, however, have a special test which was designed considering the shape of the pressurized cylinder, which explains why it was included in this thesis.

Table 6.8: *Error statistics for the convexity tests*

	Minimum nodal error	Maximum nodal error	Average nodal error
Original results	5.8539×10^{-5}	2.2953×10^{-4}	9.7145×10^{-5}
All convex mesh	6.6309×10^{-5}	2.4360×10^{-4}	1.0114×10^{-4}

6.3.2.2 Convexity tests

One of the first claims we made when justifying our choice of the Virtual Element Method was the fact that it does not have much restriction in the shape or quality of the elements, even allowing non-convex elements, which is usually disastrous for other methods. Hence, our first test will be to prove this claim comparing the performance of VEM in a mesh with non-convex elements, and the same mesh, but with the nonconvex elements partitioned, so that no nonconvex elements remain; we use the mesh shown in Section 6.3.2, which was chosen exactly because of the two big nonconvex elements that can be seen in the inner part of the cylinder. We use the convex fixer functionality of the modifiable mesh library to modify our mesh, and then apply the conditions of the problem, obtaining the results shown in Figure 6.11; again, we complement the results with statistics, presented in Table 6.8, as additional help, we include the original results obtained for the tested condition.

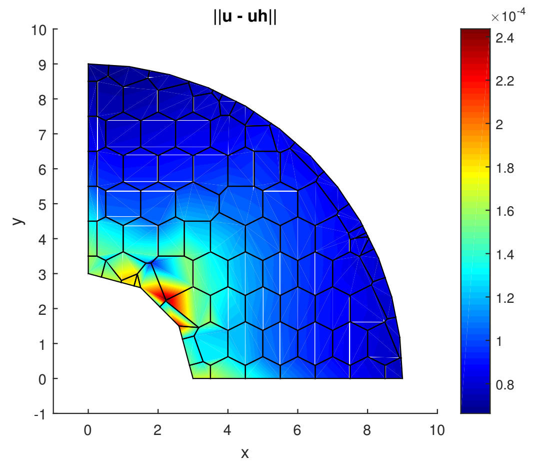


Figure 6.11: *Nodal error result for the pressurized cylinder mesh after partitioning non-convex elements*

The most remarkable point about the results shown in Figure 6.11 and Table 6.8 is the fact that the results actually worsen when the convex fix is applied, when it should have no effect whatsoever or improve the accuracy of the solution. The explanation to this effect relies on two important facts: first, as can be seen in the colormap shown in Figure 6.10 and Figure 6.11, the most sensitive part of the problem is the inner wall of the cylinder, therefore, any small error in this zone can easily increase the total error; second, and more importantly, we had to apply an approximation to be able to implement the problem.

As we have commented throughout the entire design and implementation chapters, we assumed, almost immediately, that the problems that we will solve are in Cartesian coordinates, and implemented accordingly; as seen in Equation (6.7), that is not the case with the pressurized cylinder test. Hence, instead of re-implementing a good part of the library, we choose to adapt both the conditions and the solution to Cartesian coordinates, which is correct from a theoretical point of view, but greatly increases machine precision errors; for example, just including the inner pressure was a great challenge, as we had to decompose the force in two components (using trigonometric functions, known for their imprecision).

This way, we conclude that is highly likely that our library must be restructured to accept problem defined in polar coordinates to obtain better results to the **pressurized cylinder**; this improvement is considered out of the scope of the work, and left as future work.

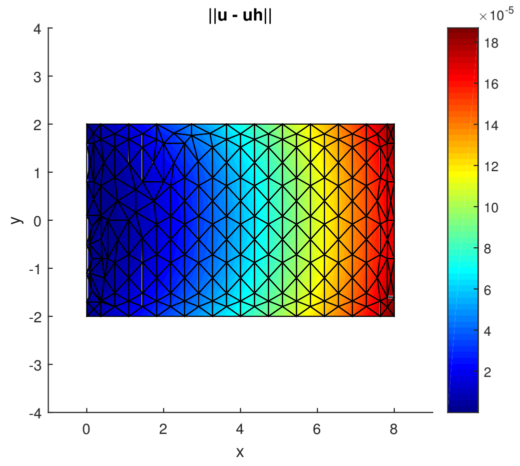
6.3.2.3 Refining the mesh

In this subsection we study the effect that refining elements of the mesh has on the accuracy on the results; as refining has long been known as the more effective way of decreasing error in numerical simulations, we expect that this operation decreases the error of the solution. We propose two experiments, the first consisting in refining the entire mesh, while the second just selects specific elements; concretely, the first experiment refines each element of the mesh including its centroid (a refining method that usually has good results), and the second just randomly generates 10 points (to be just in the comparison, we use the same random points for both VEM and FEM) and includes them in the mesh, without concern about the quality of the elements that will be generated. We apply this operation on the parabolic beam test for both methods, showing the visual results in Figure 6.12, and the error statistics in Table 6.9.

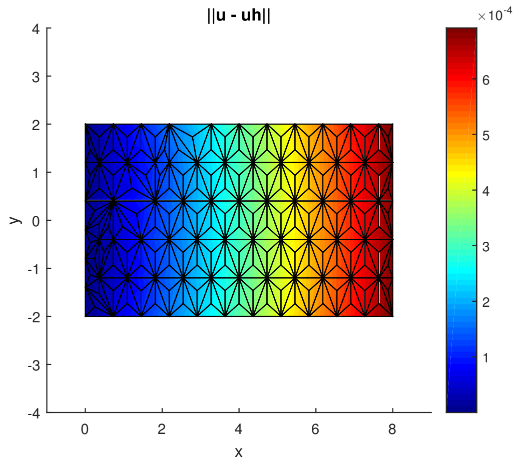
Table 6.9: *Error results of refining tests*

Experiments		Minimum nodal error	Maximum nodal error	Average nodal error
Original results	VEM	2.8948×10^{-20}	1.8690×10^{-4}	7.0227×10^{-5}
	FEM	7.2371×10^{-21}	2.5466×10^{-4}	1.0177×10^{-4}
All refined	VEM	2.8948×10^{-20}	1.8683×10^{-4}	7.1399×10^{-5}
	FEM	7.2371×10^{-21}	6.9147×10^{-4}	3.0923×10^{-4}
Random refine	VEM	2.8948×10^{-20}	1.9862×10^{-4}	7.3110×10^{-5}
	FEM	7.2371×10^{-21}	6.9147×10^{-4}	2.9885×10^{-4}

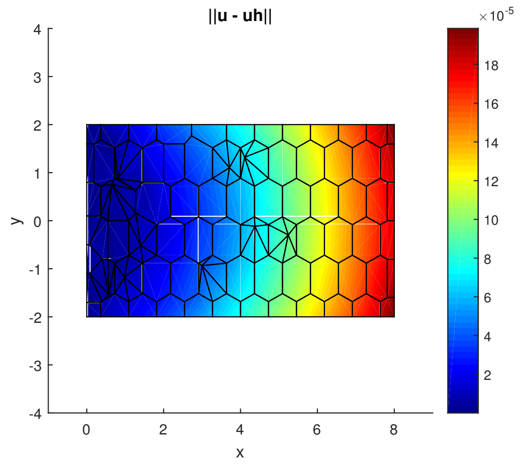
There are several important points to make in Figure 6.12 and Table 6.9. The first is that, contrary to what we expected, the maximum and average nodal errors in FEM does not decrease when refining, but rather increase; we explain this phenomenon by noting that, although including the centroid tends to give good quality meshes, doing so in all the mesh creates some long and thin triangles ("bad quality ones") that make the average error of the method grow. This hypothesis is further reinforced when noting that the average error of FEM when randomly selecting points is lower than refining all the mesh. Hence, we conclude that a few bad quality triangles isolated from one another seems to have a lesser



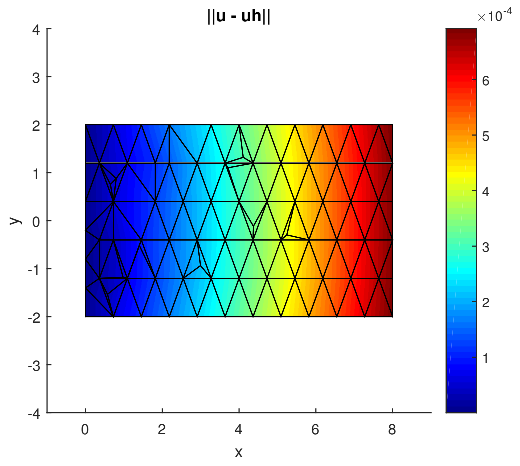
(a) Refine all elements - VEM



(b) Refine all elements - FEM



(c) Refine 10 random elements - VEM



(d) Refine 10 random elements - FEM

Figure 6.12: *Original results of the modifiable meshes tests*

effect when compared with bad triangles together; this seemingly incoherent results require further testing, which is out of the scope of this thesis.

In the side of VEM, it is remarkable the fact that the method seems completely indifferent to the change on the mesh, keeping the error rate almost constant; we theorize that this effect is probably is due to using triangles to refine the mesh, which are known for the stiff behavior, hence, we think that using another refinement scheme would probably yield better results for VEM. It is important to note, however, that, differently from FEM, the errors do not grow when the included triangles are long and thin, which in itself demonstrate that VEM responds better to mesh changes.

Geometrically speaking, we include the most relevant characteristics of the meshes after being refined in Table 6.10; this table summarizes enough information so that we can conclude about the performance of the polygonal mesh versus the triangulation for the refine operation. There are several observations that can be made from the data: first, randomly refining a polygonal mesh does not have as catastrophic effects as doing the same on a triangulation;

Table 6.10: *Relevant properties of the meshes after being refined*

		Element Area			Edge length	
		Min	Max	Average	Min	Max
All refined	VEM	0.00137898	0.138957	0.0760095	0.0359989	0.90909
	FEM	0.036363	0.145456	0.0911681	0.242424	1.09091
Random refined	VEM	0.00137898	0.682193	0.25	0.0359989	0.913338
	FEM	0.000454211	0.436364	0.233577	0.115033	1.09091

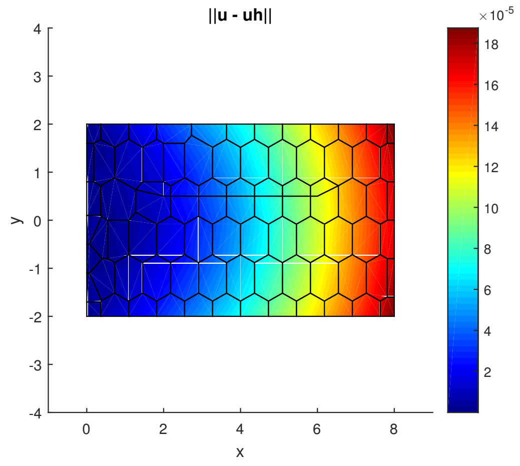
in particular, we can note that the minimum element area for a polygonal mesh does not change between both types of refinement, while for the triangular one it decreases sharply. Second, the minimum area and edge length in the polygonal mesh is extremely small, but as the same value appears for both cases, it is not related to the refinement itself, but rather to the mesh generation. Third, refining with the centroid, as we use a triangulation scheme, gives similar results for both meshes, so we do not comment further on the subject and rather use this case as comparison for the random refinement.

Hence, we conclude that polygonal meshes give substantially better results when including random points on the mesh, but it is no different to the triangular mesh when using a scheme that we know gives good quality elements, such as including the centroid of each element.

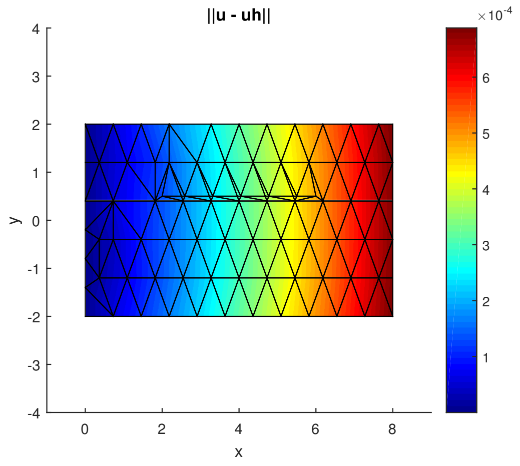
6.3.2.4 Breaking the mesh

In this subsection we will study how including segments in the mesh after its creation could lead to much worse results in FEM; these tests could be considered the central ones to this thesis, as they are expected to be the clearest in demonstrating that VEM is better suited for applications in which the mesh changes online. We present four tests, varying simply in the position and complexity of the segments to be added: first, we include a horizontal line that crosses half of the domain; second, a vertical line starting from the boundary; third, a diagonal line inside the boundary; and fourth, a sequence of four segments zigzagging inside the mesh. Again, we solve the parabolic beam problem using both VEM and FEM, presenting the results visually, in the same order as we define them, in Figure 6.13, Figure 6.14, Figure 6.15 and Figure 6.16; a table summarizing the nodal error maximum, minimum and average value for each of the test cases in included in Table 6.13.

We immediately note from the information presented in Table 6.13 that the error in FEM steadily increases as the breaking tests involve more elements and leave behind long and thin triangles; on the other hand, VEM consistently performs no matter the circumstance. This very fact gives us a good hint that VEM is indeed better suited for situations in which segments or points are included in the mesh after it is generated. The effect that breaking the mesh and not allowing to remesh is particularly important in the last example, in which the error, on average, in FEM is two orders of magnitude greater than in VEM.

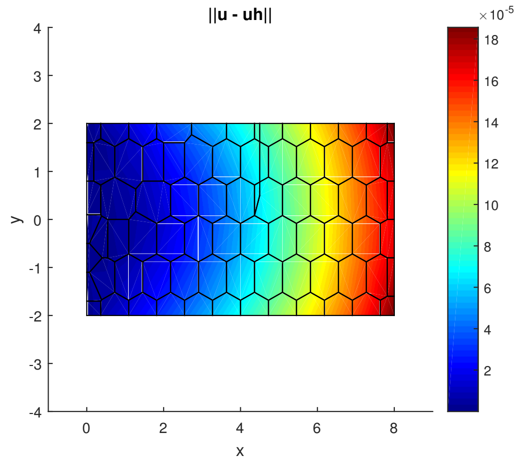


(a) Breaking with a straight line - VEM

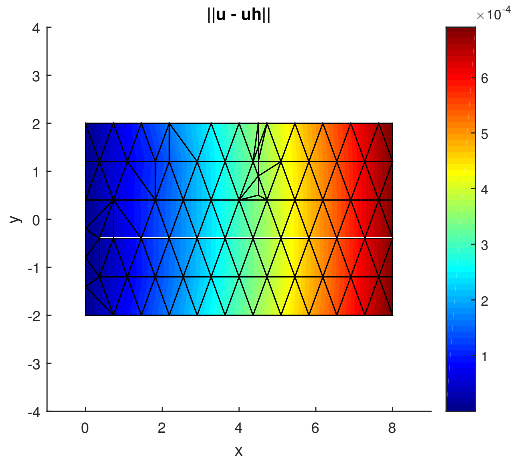


(b) Breaking with a straight line - FEM

Figure 6.13: *Error for the breaking with a straight-line test*

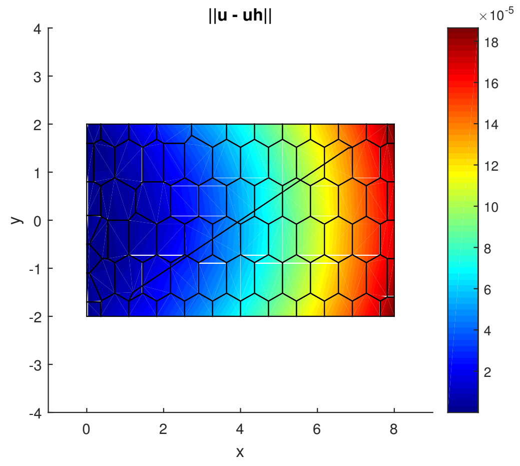


(a) Breaking with a vertical line - VEM

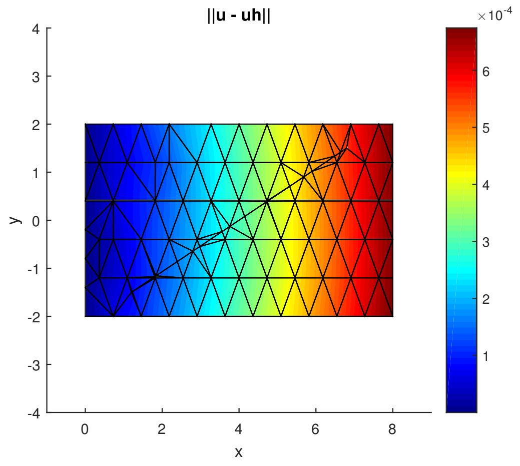


(b) Breaking with a vertical line - FEM

Figure 6.14: *Error for the breaking with a vertical line test*

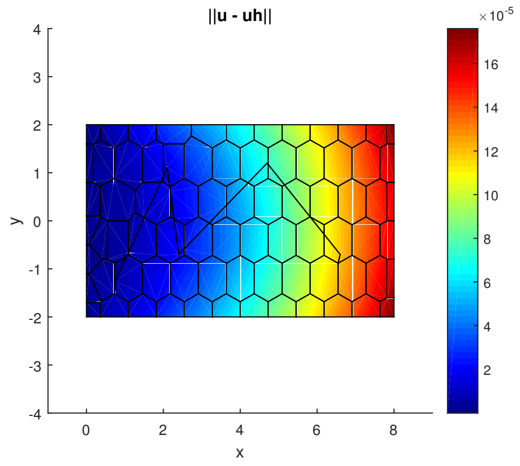


(a) Breaking with a diagonal line - VEM

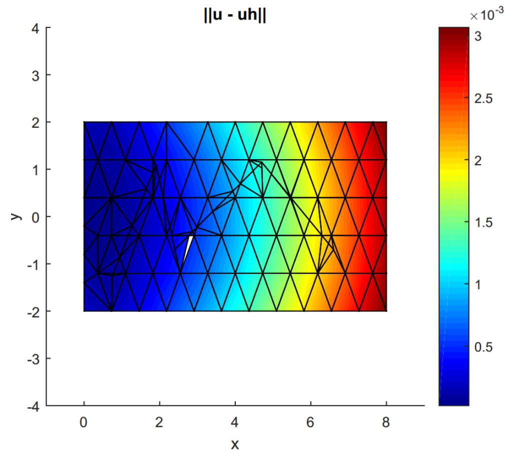


(b) Breaking with a diagonal line - FEM

Figure 6.15: *Error for the breaking with a diagonal line test*



(a) Breaking with a zigzagging sequence - VEM



(b) Breaking with a zigzagging sequence - FEM

Figure 6.16: *Error for the breaking with a zigzagging sequence test*

Beside the fact that VEM gives better results in these breaking scenarios, it also has an advantage geometrically speaking. As VEM allows polygons of arbitrary shape, the inclusion of a segment, or a sequence of them, is not as catastrophic as in FEM, as the broken polygons can be left behind with no post processing whatsoever; in FEM, however, the broken polygons must be triangulated, which leads to situations in which long and thin triangles (with area close to zero) appear. Moreover, the very fact that the broken polygons are left as they appear (and not triangulated), makes VEM be able to represent the same change with fewer elements; this way, VEM solves faster in this kind of applications. We summarize the meshes properties before and after applying the breaking in Table 6.12; as the number of experiments is high, we only study the last two, as they are more interesting geometrically (they intersect a bigger number of mesh elements).

Table 6.11: *Comparison of meshes before and after being broken*

		Number of points		Number of elements	
		Original	After breaking	Original	After breaking
Diagonal inside	VEM	154	165	76	86
	FEM	76	98	117	161
Zigzag	VEM	154	175	76	93
	FEM	76	107	117	179

Table 6.12: *Relevant properties of the meshes after being broken*

		Element Area			Edge length	
		Min	Max	Average	Min	Max
Diagonal inside	VEM	0.029796	0.682193	0.372093	0.0359989	0.90909
	FEM	5.21622e-005	0.436364	0.198758	0.00917856	1.09091
Zigzag	VEM	0.00105666	0.681997	0.344086	0.000625753	0.938519
	FEM	0.0000	0.436364	0.179893	0.0348284	1.09091

Besides the numerical analysis presented before, again, we use the tests to prove our first geometric hypothesis: that polygonal meshes are better suited for online changes on meshes, leaving better quality meshes with less elements. From Table 6.11 and Table 6.12 we note several points: first, the flexibility that gives us the polygonal meshes is specially obvious when one analyses the number of elements and points created when broken; in the most demanding case (zigzagging segments), the polygonal mesh increases its number of elements in a 22%, while the triangulation, as it needs to keep broken elements triangular, does so in 53%, phenomenon that ultimately leads to FEM (that uses triangulations) being slower in this kind of applications. Second, that polygonal meshes consistently leaves better quality elements (with bigger area and smaller maximum edge length) when being broken when compared with a triangulation, due to the flexibility of arbitrary polygons; on the other hand, forcing a triangulation and not remeshing leaves meshes with extremely small elements, with small area and large maximum edge size. Third, the minimum area of the triangular mesh in the zigzag case is reported as zero, which means that the smallest element of the mesh is so small that its area is considered zero up to machine precision; this phenomenon explains easily why the nodal error for FEM in this particular test is high, as the area of the elements is used as denominator in several computations.

Table 6.13: *Error statistics for the breaking tests*

Experiments		Minimum nodal error	Maximum nodal error	Average nodal error
Break Straight	VEM	2.8948×10^{-20}	1.8749×10^{-4}	6.9599×10^{-5}
	FEM	7.2371×10^{-21}	6.8783×10^{-4}	3.1161×10^{-4}
Vertical from boundary	VEM	2.8948×10^{-20}	1.8553×10^{-4}	6.9963×10^{-5}
	FEM	7.2371×10^{-21}	6.9000×10^{-4}	3.1621×10^{-4}
Diagonal inside	VEM	2.8948×10^{-20}	1.8641×10^{-4}	6.9122×10^{-5}
	FEM	7.2371×10^{-21}	6.7410×10^{-4}	3.0654×10^{-4}
Zigzag	VEM	2.8948×10^{-20}	1.7602×10^{-4}	6.2741×10^{-5}
	FEM	1.9480×10^{-5}	3.100×10^{-3}	1.100×10^{-3}

Finally, using the information given by the breaking and refining experiments, we can conclude that, as we theorized at the beginning of this thesis, polygonal meshes indeed present better results (both in the number of elements created and their quality) when both modifiable operations are applied.

Chapter 7

Geometric examples

In this chapter we will show some of the functionality achieved by the geometric libraries. We do not include this examples in the results chapter as the theoretical study of the mesh generator, besides proving the two hypothesis presented in Section 1.2, is out of the scope of this thesis. As the first investigation question was answered during Section 6.3.2, here we refer to the second: is it possible to create, from known tools, a robust and flexible polygonal mesh generator?

Hence, in this chapter we will focus on showing examples of mesh generation in different circumstances, so to make clear that our mesh generator is robust (solving a great number of cases) and flexible (allowing several different ways of creating the meshes). We also include code snippets showing how each example is generated, so to better illustrate on all different capabilities of the implemented software, and further demonstrating that simplicity in the user interface of the libraries was at the core of their design.

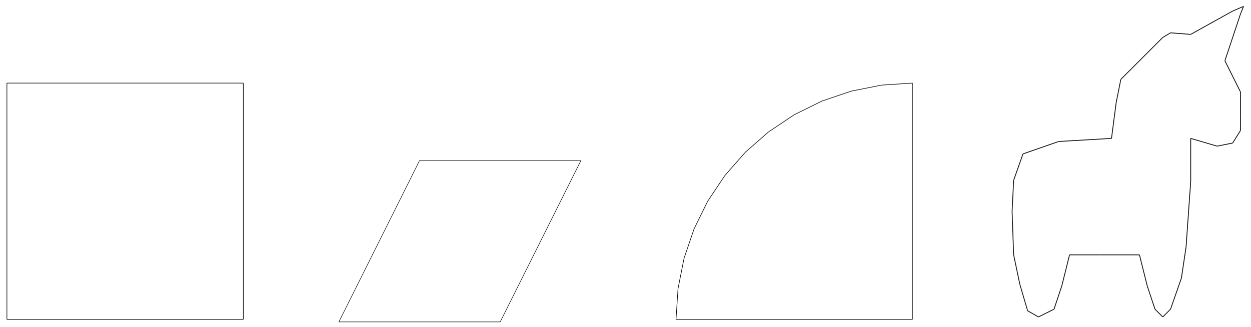
We divide this chapter in two sections, the first concerning mesh generation, and including point generation, inclusion of holes, triangulation and polygonal mesh generation; the second section will include varied examples of operations on modifiable meshes, including showing border cases and restrictions not made obvious by the numerical tests.

7.1 Mesh generation

In this section we present some examples of meshes generated using our Voronoi generation library, featuring the different capabilities achieved. At the end of each subsection, we include a small code snippet showing how the examples were generated.

7.1.1 Defining domains

Our domains, the area that define the boundary of the mesh, are created from a counter-clockwise list of points, which gives great flexibility in what can be represented. We show examples of domains of varying complexity in Figure 7.1. There is, however, an important limitation to our approach: using a single list of points one can only define polygons without holes. As we are interested in providing as much flexibility to the domains, we include the possibility of adding holes, both internal or intersecting, to the domain as an external object; when one includes an internal hole, it is stored inside the domain, while an external hole is processed and becomes a part of the domain. We show examples of domains with one or several intersecting holes in Figure 7.2.



(a) Simple square **(b)** Rhomboid shape **(c)** Quarter circle domain **(d)** Unicorn-shaped domain

Figure 7.1: *Examples of domains*

We show the code to generate the unitary square and the quarter circle domain in Code 7.1; we do not include the rhomboid domain as it is similar to the square, nor the unicorn, as each point must be created manually, giving a long list that is not interesting to show.

Code 7.1: *Creation of an square and quarter circle domain*

```
std::vector<Point> square_points = {Point(0,0), Point(10,0),
                                    Point(10,10), Point(0,10)};
Region square(square_points);

std::vector<Point> qc_points = {Point(0,0), Point(10,0),
                                Point(10,10)};
std::vector<Point> quarter =
    delynoi_utilities::generateArcPoints(Point(10,0), 10, 90.0, 180.0);

qc_points.insert(quarter_circle_points.end(), quarter.begin(), quarter.end
   ());
Region quarter_circle(qc_points);
```

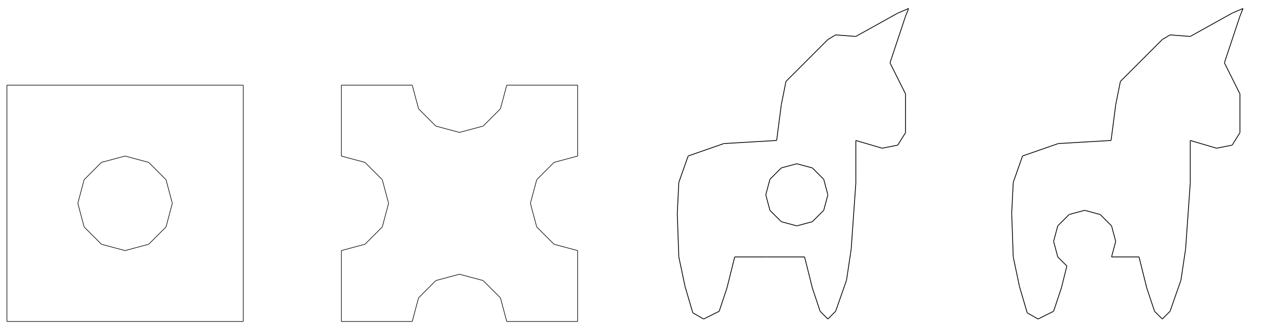

(a) Square with an inner hole **(b)** Square with several intersecting holes **(c)** Unicorn domain with an inner hole **(d)** Unicorn domain with intersecting hole

Figure 7.2: *Examples of domains with holes*

To include a hole to a given domain, it is simply a matter of creating the required hole (that can be either circular or polygonal), and adding to the Region, as seen in Code 7.2.

Code 7.2: *Including a hole in the center of square*

```
Hole circular = CircularHole(Point(5,5), 2);
square.addHole(circular);
```

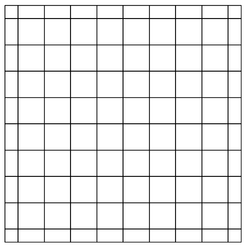
7.1.2 Generating with different rules

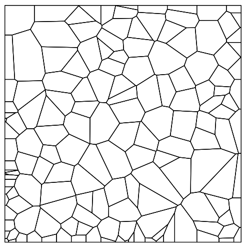
We include a number of different rules for the generation of the seeds points for the Voronoi diagram, ranging from simple ones such as uniform or random, to more complex ones such as sine or cosine; in Figure 7.3 we show examples of meshes generated inside a unitary square, using different rules. We show how to generate uniform or random points for a given domain in Code 7.3.

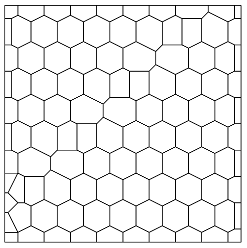
Code 7.3: *Generation of uniform and random points*

```
dom1.generateSeedPoints(PointGenerator(functions::constant(),
                                       functions::constant()), nX, nY);
dom2.generateSeedPoints(PointGenerator(functions::random_double(0,maxX),
                                       functions::random_double(0,maxY)),
                        nX, nY);
```

We also include the possibility of adding noise to the generation rules; we implement a random noise function, which adds a random variation to each seed point. In Figure 7.4 we show the results of using the same generation rules as shown in Figure 7.3, but including random noise to each point. We present the code necessary to include noise to a uniform generating rule in Code 7.4.


(a) Uniform points


(b) Random points


(c) Displacing alternating rows

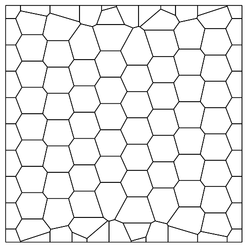
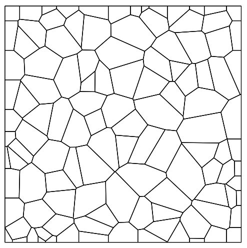
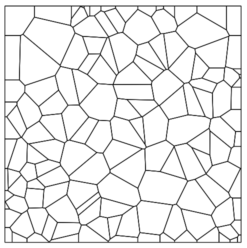

(d) Sine

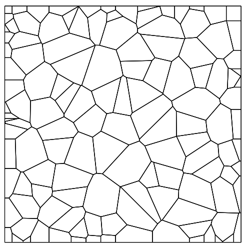
Figure 7.3: *Generation using different rules*

Code 7.4: *Creating uniform points with random noise*

```
Functor* n = noise::random_double_noise(functions::constant(),
                                        minNoise,maxNoise);
dom.generateSeedPoints(PointGenerator(n,n,nX, nY);
```


(a) Uniform points with noise


(b) Random points with noise


(c) Displacing alternating rows with noise

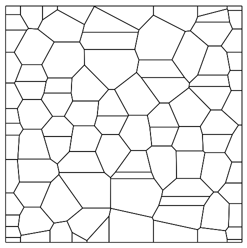
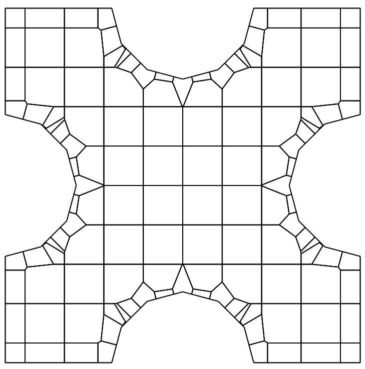

(d) Sine with noise

Figure 7.4: *Generation using different rules with random noise added*

7.1.3 Generation in complex domains

Finally, we show examples of meshes generated using both uniform and random rules, inside some complex domains in Figure 7.6 and Figure 7.5; in particular, we will use our unicorn-shaped domain with both inner and intersecting holes, and the square will multiple holes. On the other hand, in Figure 7.8 and Figure 7.7 we show the exact same conditions, but generating triangular meshes. This examples clearly show that our library is capable of generating meshes in complex domains.


(a) Square with multiple holes and uniform points

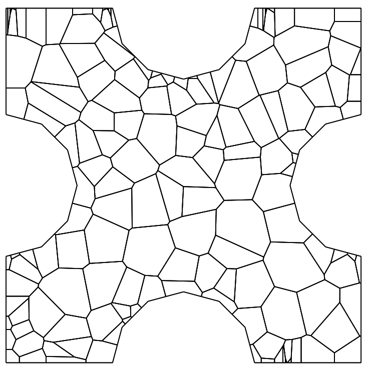
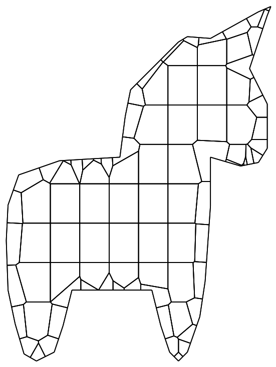
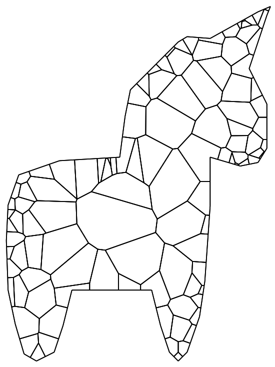
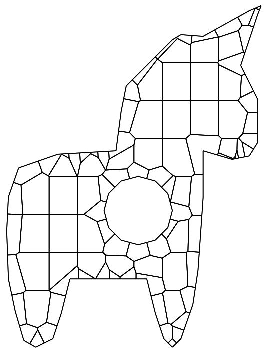
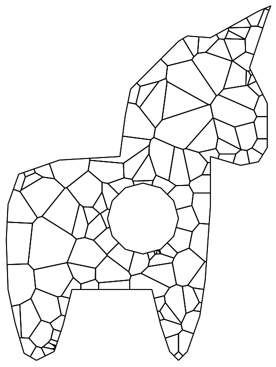

(b) Square with multiple holes and random points

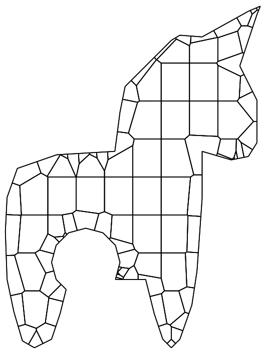
Figure 7.5: *Examples of polygonal meshes in complex domains*


(a) Unicorn domain with uniform points


(b) Unicorn domain with random points


(c) Unicorn with inner hole and uniform points


(d) Unicorn with inner hole and random points


(e) Unicorn with intersecting hole and uniform points

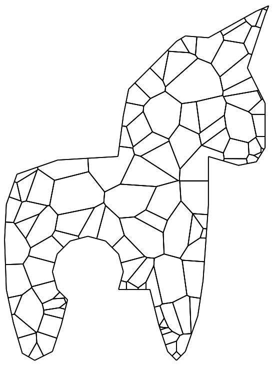
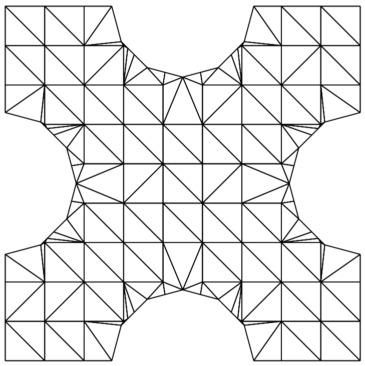

(f) Unicorn with intersecting hole and random points

Figure 7.6: *Examples of polygonal meshes in complex domains*


(a) Square with multiple holes and uniform points

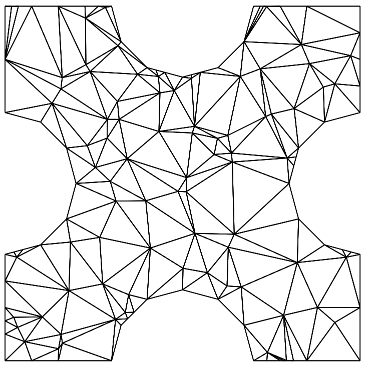
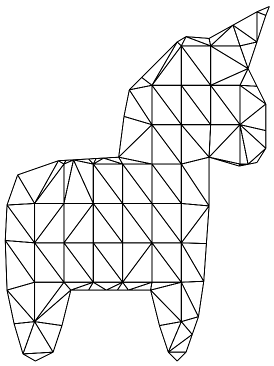
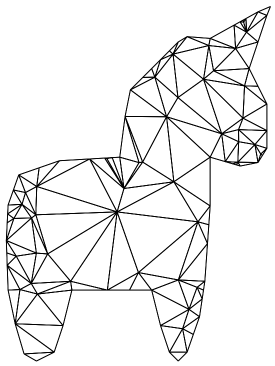
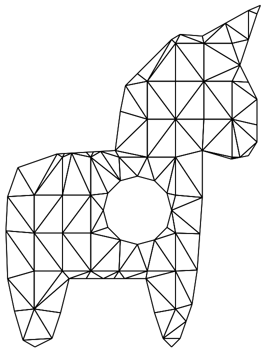
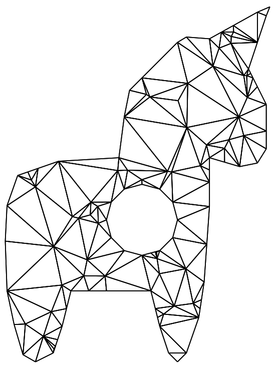

(b) Square with multiple holes and random points

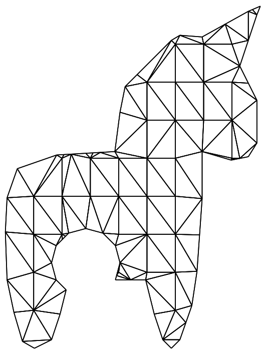
Figure 7.7: *Examples of triangular meshes in complex domains*


(a) Unicorn domain with uniform points


(b) Unicorn domain with random points


(c) Unicorn with inner hole and uniform points


(d) Unicorn with inner hole and random points


(e) Unicorn with intersecting hole and uniform points

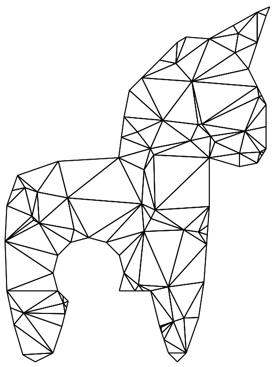

(f) Unicorn with intersecting hole and random points

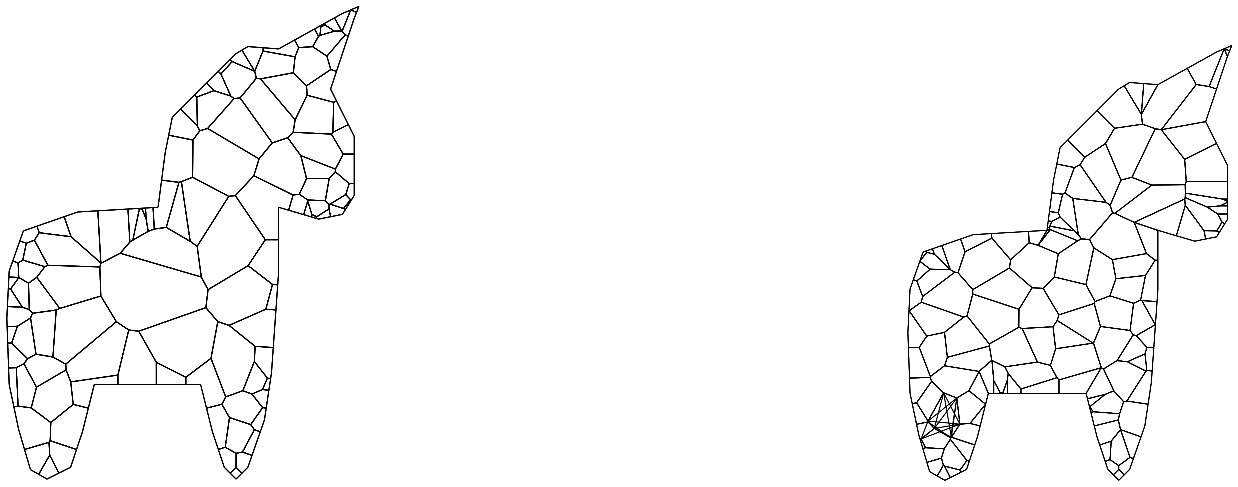
Figure 7.8: *Examples of triangular meshes in complex domains*

7.2 Modifiable meshes operations

In this section we show examples of the operations on modifiable meshes, such as breaking, refining, deleting elements and fixing convexity. For simplicity, we use as base some of the examples presented in the section before, in particular, the unicorn shaped domain and the square with a single intersecting hole, with two different generating rules (uniform and random). In the following subsections, we will show of applying each operation on the two selected meshes, in different circumstances.

7.2.1 Fixing convexity

We show the result of partitioning all nonconvex elements in the unicorn shaped domain in Figure 7.9; so, to be able to compare, we include the original mesh alongside the obtained one. The code lines required to apply this operation on the mesh are presented in Code 7.5.



(a) Original **(b)** Only convex elements

Figure 7.9: *Partitioning nonconvex elements in the unicorn mesh*

Code 7.5: *Fixing nonconvex elements*

```
//Create a modifiable mesh from a simple mesh
xMesh modifiable(mesh, Config(Configurations::config::PolygonalDefault));

MeshFixer* convexFixer = new ConvexFixer();
modifiable.fix(convexFixer);
```

7.2.2 Deleting elements

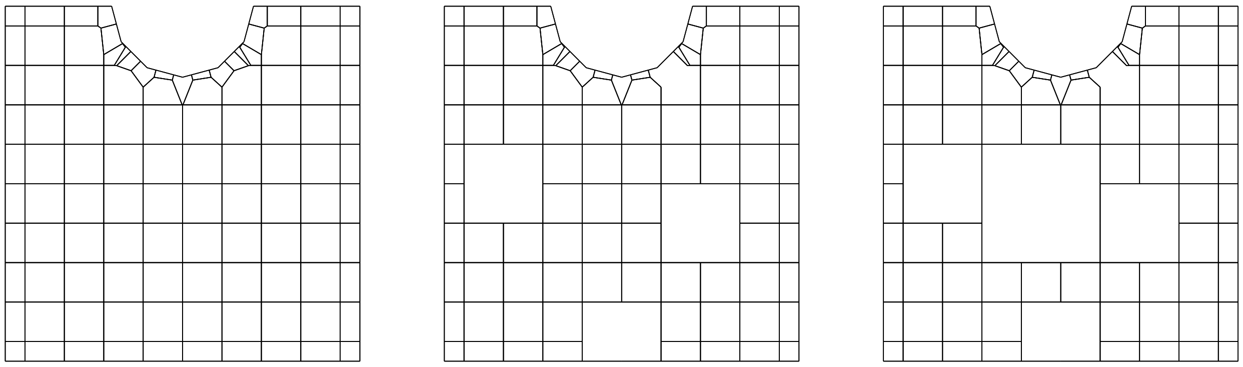
We present the result of deleting 4 points in the square with a hole mesh in Figure 7.10, for both types of generation rules, again, we include the original mesh alongside it. To further note that the mesh keeps its consistency even after deleting points, we include the result of deleting an element on the mesh without the 4 points. The code lines that implement both operations (deleting points and deleting elements) are included in Code 7.6.

Code 7.6: *Deleting points or elements from a mesh*

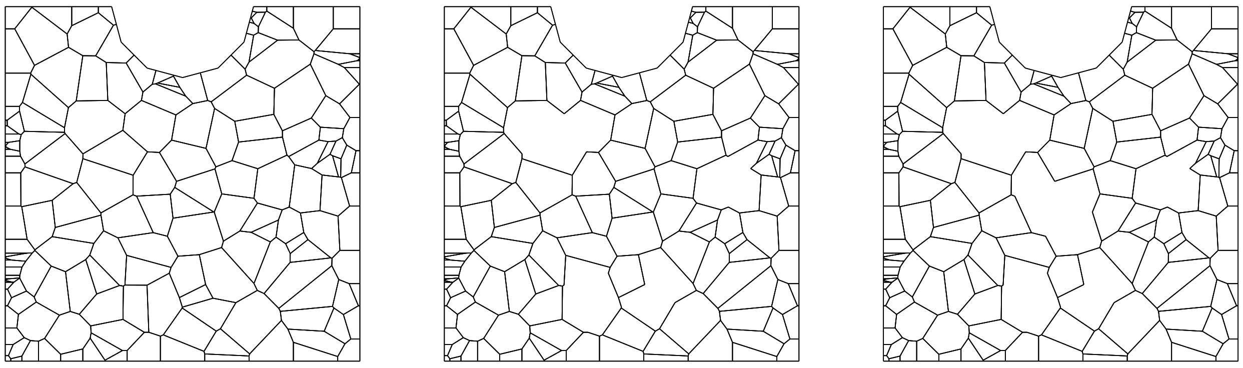
```
xMesh m(mesh, Config(Configurations::config::PolygonalDefault));

Point pointToErase = destructable.getPoint(index_point);
m.erase(pointToErase);

Polygon elemToErase = destructable.getPolygon(index_poly);
m.erase(elemToErase);
```



(a) Original **(b)** Deleting 4 points **(c)** Deleting a polygon



(d) Original **(e)** Deleting 4 points **(f)** Deleting a polygon

Figure 7.10: *Deleting points and elements*

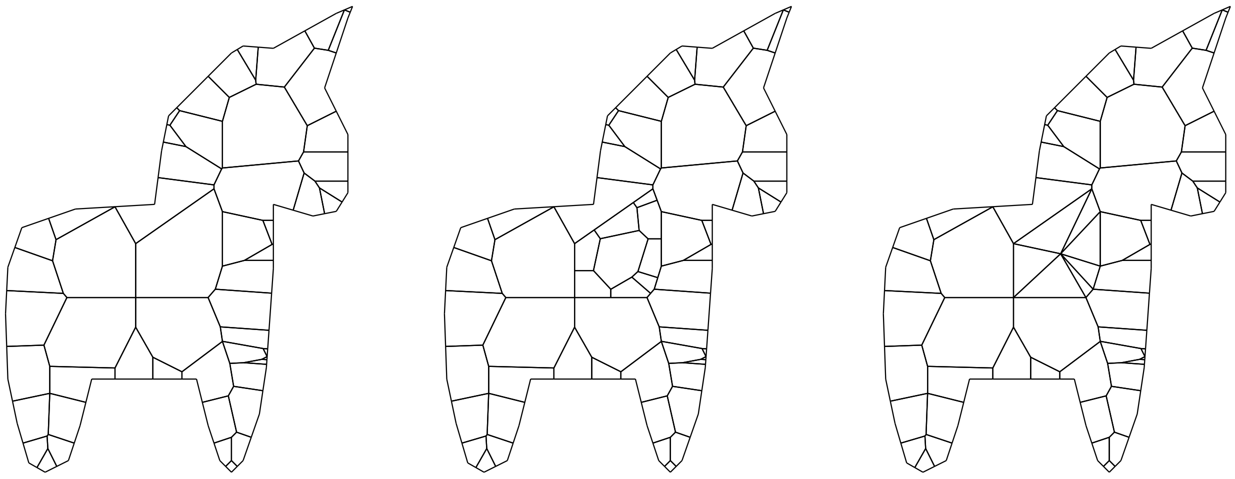
7.2.3 Refining elements

We show the result of refining an element (with a triangular scheme and a polygonal scheme) of the unicorn mesh in two different scenarios: including the centroid of the element (Figure 7.11), and using a PointGenerator instance which generates uniform displaced rows (in Figure 7.12) ; we use only the uniform mesh since it allows to better see the effects of the operation. Finally, we show the code required for the example in Code 7.7.

Code 7.7: *Refining an element*

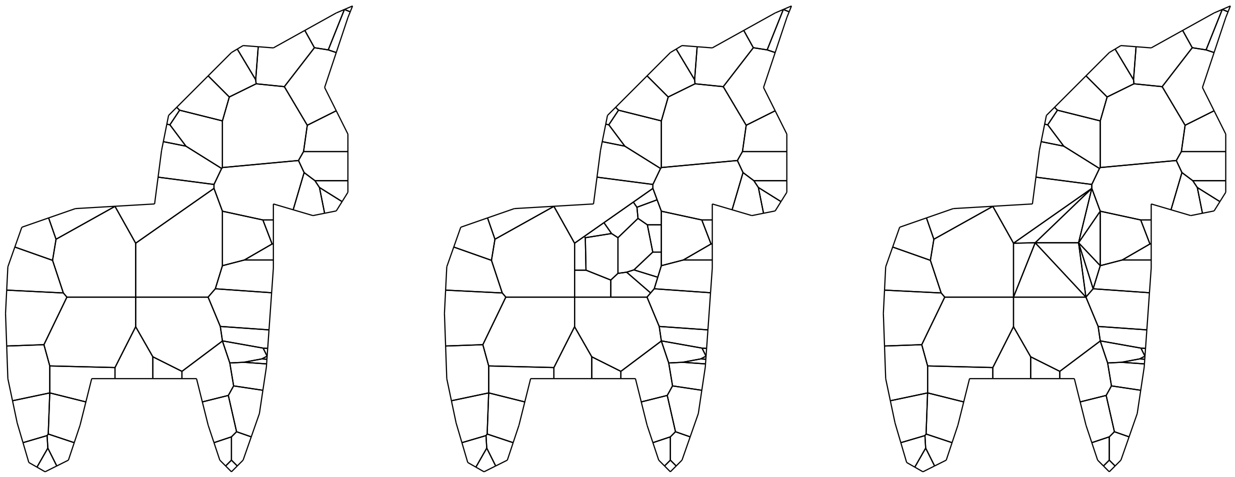
```
//Creates a modifiable mesh that refines with polygons
xMesh m_poly(mesh, Config(Configurations::config::PolygonalDefault));
//Creates a modifiable mesh that refines with triangles
xMesh m_triangle(mesh, Config(Configurations::config::
   PolygonalTriangulateRefine))

m_poly.refine(m_poly.getPolygon(poly_index),
              PointGenerator(functions::constantAlternating(),
                             functions::constant()), n, m);
m_triangle.refine(m_triangle.getPolygon(poly_index),
                  new CentroidPointCreator);
```



(a) Original **(b)** Polygonal refinement **(c)** Triangular refinement

Figure 7.11: *Refining an element using its centroid*

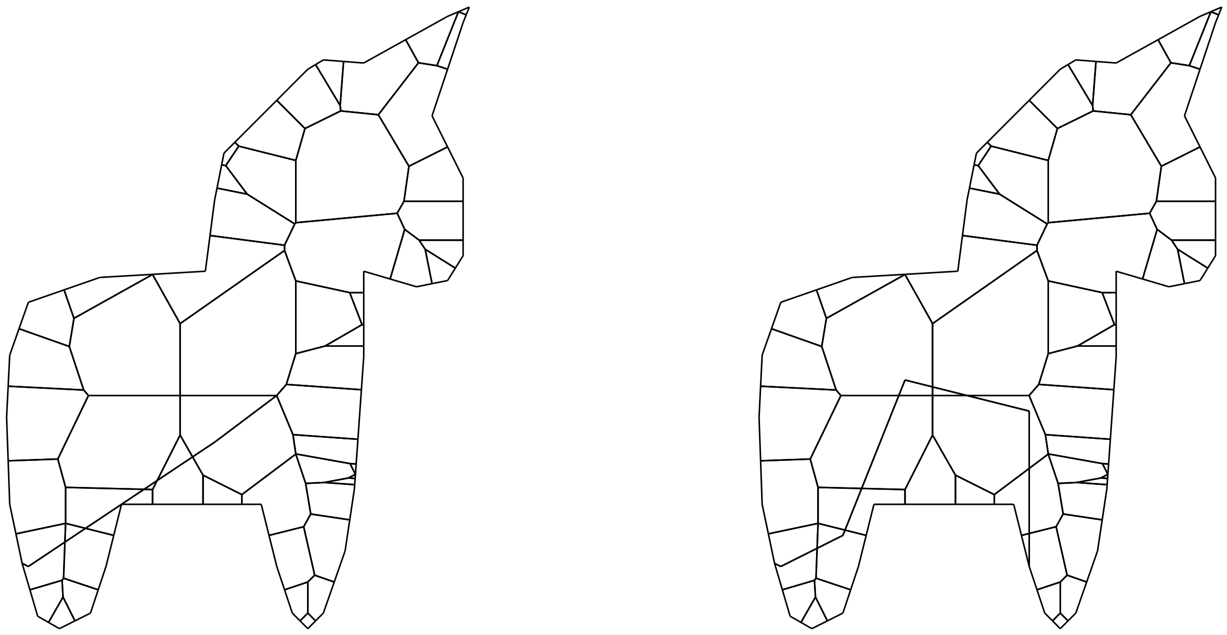


(a) Original **(b)** Polygonal refinement **(c)** Triangular refinement

Figure 7.12: *Refining an element using a PointGenerator*

7.2.4 Breaking the mesh

We present the result of breaking the unicorn polygonal mesh (generated from random points) in different configurations in Figure 7.13; we consider that, since the numerical examples were mostly focused on breaking the mesh, that this functionality was illustrated enough and, hence, we only we include two examples: the first using a single segment, and a second, using a sequence of them. Code 7.8 illustrates how the examples were created.



(a) Single segment **(b)** Multiple segments

Figure 7.13: *Breaking the mesh*

Code 7.8: *Breaking the mesh*

```
xMesh m_poly(m, Config(Configurations::config::PolygonalDefault));
m_poly.break(segment_list)
```

Chapter 8

Conclusions

We presented a hypothesis which we wanted to prove: that the Virtual Element Method (VEM) is better suited, numerically speaking, for the solution of linear elasticity problem in meshes with non-staged changes, when compared with the standard, and commonly used, Finite Element Method (FEM). To prove the previous idea, we use constrained Voronoi diagrams generated from different distributions of point as our original polygonal meshes for VEM, and their associated Delaunay triangulation for FEM.

We presented the minimum background information to understand the problem of linear elasticity and numerical methods for its solutions, explaining all relevant terminology. We present the related work which was studied as an alternative to solve the problem of linear elasticity with modifiable meshes, explaining the advantages and disadvantages that lead us to choosing the Virtual Element Method.

We designed a series of experiments that will help us conclude about the accuracy of the proposed hypothesis, comparing the numerical accuracy of VEM and FEM in good quality meshes that will be modified online, without remeshing to preserve their original properties. We designed and implemented the three libraries as required tools for the experimentation, each in charge of one aspect of the problem: the generation of the Delaunay triangulations and constrained Voronoi diagrams as initial meshes, the solution of the linear elasticity problem using VEM and FEM, and the operations to modify polygonal or triangular meshes. We use the same division in three parts throughout the entire work, first presenting the high-level design, and then the concrete implementation.

We validate our implementation through the use of theoretical order analysis and numerical convergence experiments, giving satisfactory results. Then. we ran our tests, and compared the increase or decrease of the error rates in all different circumstances, finally concluding the VEM behaves better in all circumstances, keeping a steady error while FEM tends to give worse results as the quality of the mesh decreases, being specially sensitive to elements of small area and large maximum edge length. We also analyze the properties of the generated meshes, demonstrating that polygonal meshes behave better (in quality and number of elements) when modifiable operations, such as including extra points or segments, are applied.

Future work

As we have commented throughout this entire document, there are several improvements that can be made to the system to both optimize it, and extends its functionalities; we leave these ideas listed as a form of guideline for future extensions of the project.

In the first place, we had several issues with the performance of the library, which made impossible for the efficiency experiments to go beyond about 500 elements; trying to run any experiment beyond that filled the C++ heap and made all programs fail. Hence, we propose a study in the memory usage of the library, so that they can be optimize. For example, we do not for certain that most implementations of the Finite Element Method take advantage of the fact that the global stiffness matrix is sparse, which decreases memory usage, as all elements equal to zero are not stored; due to time constraints, we were not able to include this important optimization, which, at the same time, should help with the errors related to memory consumption, and make simulations run faster.

Second, in the complexity analysis we concluded that the Voronoi generation algorithm is bounded by $O(NlogN)$, but our experiments gave rates closer to $O(N^2)$, which means that there are several points that require optimization there. In particular, we remark on one point: we mentioned in the implementation chapter that keeping the mesh points unique had cost $O(logN)$, which means that just processing the Voronoi points has cost $O(NlogN)$, so it could be significant when the mesh grows. Hence, it is advisable to design some way of keeping the points unique in $O(1)$, but considering the machine precision problems.

Third, as we saw in the experimentation, in particular with the pressurized cylinder test, our VEM/FEM library is not suited for problems not presented in Cartesian coordinates, which could end up being an important restriction. This way, we propose the modification of the library so that the user can decide the coordinate system it will use.

Now, concerning the experiments themselves, there are a number of other alternatives that need to be studied to make sure that VEM is indeed the best suited numerical method for our particular problem. In the first place, we could compare the results, both in time and accuracy, obtained when using a second order triangle as element; we theorize that the FEM results should improve, but not reaching those of VEM, but with the added cost of twice as many points. Second, we could compare the performance of Polygonal Finite Elements in our problem, which could use the exact same meshes as VEM, so it does not present the problem of long and thin triangles; we do think, however, that the restriction of PFEM on the quality and properties of the polygons will make it less suitable for modifiable meshes than VEM.

Glossary

N	Number of points in a mesh
P	Number of polygons in a mesh
n	Number of points of a polygon
FEM	Finite Element Method
PFEM	Polygonal Finite Element Method
VEM	Virtual Element Method
u	Displacement
ε	Strain
σ	Stress
E	Young module
ν	Poisson ratio
C	Material matrix, also called Stress/Strain matrix
$\boldsymbol{u}^h$	Approximated displacement, also $\boldsymbol{\varepsilon}^h$ is the approximated strain and $\boldsymbol{\sigma}^h$
K	Stiffness matrix, $\boldsymbol{K}_{\mathrm{e}}$ is the element-wise matrix, while $\boldsymbol{K}_g$ is the global one
b	Body force vector
f	Load vector, $\boldsymbol{f}_{\mathrm{e}}$ is the element-wise vector, $\boldsymbol{f}_g$ is the global vector

Bibliography

[1] C Talischi A Gain and G Paulino. On the virtual element method for three-dimensional linear elasticity problems on arbitrary polyhedral meshes. *Computer Methods in Applied Mechanics and Engineering*, 282:132–160, 2014.

[2] G Wells A Logg and J Hake. Dolfin: A c++/python finite element library. *Automated Solution of Differential Equations by the Finite Element Method*, pages 173–225, 2012.

[3] F Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[4] C Lovadina B da Veiga, L Beirão and D Mora. A virtual element method for elastic and inelastic problems on polytope meshes. *Computer Methods in Applied Mechanics and Engineering*, 295:327–346, 2015.

[5] G Guennebaud et al. B Jacob. Eigen. `http://www.http://eigen.tuxfamily.org`, March 2017.

[6] R Stogner B Kirk, J Peterson and G Carey. libmesh: a c++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3-4):237–254, 2006.

[7] C Barber and H Huhdanpaa. Qhull. `http://www.qhull.org`, 1995.

[8] M Berger and P Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of computational Physics*, 82(1):64–84, 1989.

[9] Boost. Boost c++ libraries. `http://www.boost.org/`, 2015.

[10] P Bourke. Calculating the area and centroid of a polygon. *Swinburne Univ. of Technology*, 1988.

[11] D Dobkin C Barber and H Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

[12] A Pereira C Talischi, G Paulino and I Menezes. Polygonal finite elements for topology optimization: A unifying paradigm. *International journal for numerical methods in engineering*, 82(6):671–698, 2010.

[13] A Pereira C Talischi, G Paulino and I Menezes. Polymesher: a general-purpose mesh generator for polygonal elements written in matlab. *Structural and Multidisciplinary Optimization*, 45(3):309–328, 2012.

[14] P Alliez D Sieger and M Botsch. Optimizing voronoi diagrams for polygonal finite element computations. In *IMR*, pages 335–350. Springer, 2010.

[15] B Lévy D Yan, W Wang and Y Liu. Efficient computation of clipped voronoi diagram for mesh generation. *Computer-Aided Design*, 45(4):843–852, 2013.

[16] G Dasgupta. Integration within polygonal finite elements. *Journal of Aerospace Engineering*, 16(1):9–18, 2003.

[17] R Wein G Zucker B Zukerman E Fogel, O Setter and D Halperin. 2D regularized boolean set-operations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.10 edition, 2017.

[18] M Floater. Mean value coordinates. *Computer aided geometric design*, 20(1):19–27, 2003.

[19] I Fratric. Polypartition - tiny polygon partitioning and triangulation library. `https://github.com/ivanfratric/polypartition`, nov 2017.

[20] A Russo G Manzini and N Sukumar. New perspectives on polygonal and polyhedral finite element methods. *Mathematical Models and Methods in Applied Sciences*, 24(08):1665–1699, 2014.

[21] G Golub and C Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[22] B da Veiga H Chi and G Paulino. Some basic formulations of the virtual element method (vem) for finite deformations. *Computer Methods in Applied Mechanics and Engineering*, 318:148–192, 2017.

[23] O Lopez-Pamies H Chi, C Talischi and G Paulino. Polygonal finite elements for finite elasticity. *International Journal for Numerical Methods in Engineering*, 101(4):305–328, 2015.

[24] S Herring and B Jean. Voronoi meshes with prescribed boundaries. pp.research note, 24th International Meshing Roundtable, October 2015.

[25] N itschfeld and M Rivara. Automatic construction of non-obtuse boundary and/or interface delaunay triangulations for control volume methods. *International Journal for Numerical Methods in Engineering*, 55(7):803–816, 2002.

[26] M Kenamond J Pouderoux, M Berndt and M Shashkov. Shapo: A framework for generating 2d voronoi meshes. In *The seventh International Conference on Numerical Methods for Multi-Material Fluid Flow*, 2015.

[27] A Johnson. Clipper - an open source freeware library for clipping and offsetting lines and polygons. `http://www.angusj.com/delphi/clipper.php`, dic 2016.

[28] G Manzini K Lipnikov and M Shashkov. Mimetic finite difference method. *Journal of Computational Physics*, 257:1163–1227, 2014.

[29] P Knupp. Remarks on mesh quality. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2007.

[30] C Lawson. Software for c1 surface interpolation. *Mathematical Software III*, pages 161–194, 1977.

[31] S Pieraccini M Benedetto, S Berrone and S Scialò. The virtual element method for discrete fracture network simulations. *Computer Methods in Applied Mechanics and Engineering*, 280:135–156, 2014.

[32] M van Kreveld M de Berg, O Cheong and M Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag Berlin Heidelberg, 2008.

[33] H Nilsen O Andersen and X Raynaud. On the use of the virtual element method for geomechanics on reservoir grids. *arXiv preprint arXiv:1606.09508*, 2016.

[34] C Mobarry R De Fainchtein P MacNeice, K Olson and C Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer physics communications*, 126(3):330–354, 2000.

[35] Maria Q Du, M Emelianenko and L Ju. Convergence of the lloyd algorithm for computing centroidal voronoi tessellations. *SIAM journal on numerical analysis*, 44(1):102–119, 2006.

[36] V Faber Q Du and M Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM review*, 41(4):637–676, 1999.

[37] J Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148, pages 203–222. Springer-Verlag, 1996.

[38] J Shewchuk. What is a good linear finite element? interpolation, conditioning, anisotropy, and quality measures. *University of California at Berkeley*, 73, 2002.

[39] S Sloan and G Houlsby. An implementation of watson's algorithm for computing 2-dimensional delaunay triangulations. *Advances in Engineering Software (1978)*, 6(4):192–197, 1984.

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

[41] I Sutherland and G Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[42] O Sutton. The virtual element method in 50 lines of matlab. *arXiv preprint arXiv:1604.06021*, 2016.

[43] A Tabarraei and N Sukumar. Application of polygonal finite elements in linear elasticity. *International Journal of Computational Methods*, 3(04):503–520, 2006.

[44] T Taniguchi. Automatic mesh generation for fem: use of delaunay triangulation. *Morikita Publ*, 1992.

[45] G Toussaint. Solving geometric problems with the rotating calipers. In *Proc. IEEE Melecon*, volume 83, page A10, 1983.

[46] E Cáceres V. Mixed Virtual Element Methods. Applications in Fluid Mechanics. B.S. Thesis, Universidad de Concepción, Concepción, Chile, 2015.

[47] B Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63, 1992.

[48] R Hartmann W Bangerth and G Kanschat. deal. ii—a general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):24, 2007.

[49] E Wachspress. *A rational finite element basis.* Elsevier, 1975.

[50] H Zimmer. Voronoi and delaunay techniques. *Proceedings of Lecture Notes, Computer Sciences*, 8, 2005.