Université catholique de Louvain
École polytechnique de Louvain
Institut de Mécanique, Matériaux et génie Civil



Hexahedral-dominant mesh generation

Doctoral dissertation presented by

Tristan Carrier Baudouin

in partial fulfillment of the requirements for the degree of

Docteur en Sciences de l'Ingénieur

Thesis committee :

Pr Christophe Geuzaine, Université de Liège
Pr Bruno Lévy, Institut national de recherche en informatique et en automatique
Pr Emilie Marchandise, Université catholique de Louvain (Advisor)
Pr Jean-François Remacle, Université catholique de Louvain (Advisor)
Pr Grégoire Winckelmans, Université catholique de Louvain (President)

Louvain-la-Neuve, November 15, 2013

I would like to thank my advisors, Prof. Jean-François Remacle and Prof. Emilie Marchandise, for their essential support throughout this thesis. I am also grateful to the members of the DOMHEX project, notably Prof. Christophe Geuzaine, Prof. Eric Béchet, Thibaud Mouton, Amaury Johnen and Gaëtan Compère. I am very appreciative of the efforts undertaken by the jury members Prof. Grégoire Winckelmans and Prof. Bruno Lévy. Finally, I would like to thank Gmsh contributors Jonathan Lambrechts and François Henrotte as well as my colleagues at the Euler building and my family in Montreal.

Contents

1 Introduction **1**
- 1.1 Previous work . . . 4
- 1.2 Outline . . . 9
- 1.3 Contributions . . . 11
- 1.4 Publications . . . 11

2 Triangles and tetrahedra merging **13**
- 2.1 Borouchaki's and Blossom-quad algorithms . . . 14
- 2.2 Yamakawa-Shimada's algorithm . . . 15
- 2.3 Finite element conformity . . . 16
- 2.4 Results analysis . . . 20

3 Cardinal direction fields **27**
- 3.1 Two-dimensional space . . . 28
- 3.2 Three-dimensional surfaces . . . 32
- 3.3 Three-dimensional space . . . 40

4 Lloyd's energy minimization in two dimensions **43**
- 4.1 Lévy-Liu's algorithm . . . 48
  - 4.1.1 Clipped Voronoi diagrams . . . 48
  - 4.1.2 Energy and gradient evaluation . . . 54
  - 4.1.3 LBFGS optimization . . . 64
- 4.2 Results . . . 65

5 Lloyd's energy minimization in three dimensions **73**
- 5.1 Voronoi diagrams . . . 75

5.2 Energy and gradient evaluation 77
5.3 LBFGS optimization 82
5.4 Results 82

6 The frontal approach in three dimensions 93
6.1 Vertices creation 94
6.2 Distance measurements 98
6.3 Spatial searches with R-trees 100
6.4 Cardinal direction field smoothing 103
6.5 Results 106

7 The frontal approach for surfaces 117
7.1 Vertices creation 118
7.2 Distance measurements 120
7.3 Spatial searches with R-trees 123
7.4 Results 125

8 Conclusion 129

9 Appendix 137
9.1 The frontal algorithm in three dimensions 137
9.2 The computation of the L_p Lloyd's energy and its gradient in two dimensions 141

Bibliography 149

Chapter 1

Introduction

Comparing the performance of tetrahedral and hexahedral meshes is an area of research within the field of finite element analysis. The same simulation is performed on various types of meshes and the results are verified against experimental or numerical data. Hexahedral meshes have been found to possess many advantages over tetrahedral meshes.

Linear tetrahedra are known to produce inaccurate results in elasto-plastic problems [8]. Quadratic tetrahedra, quadratic hexahedra or linear hexahedra are much better alternatives [86, 8]. However, linear hexahedra can be affected by shear locking when used to simulate bending [86, 8]. In other words, the bending is not as pronounced as it should be because the energy is lost in artificial shear strain [19]. This issue can be resolved by employing selective reduced integration [41, 8]. Linear hexahedra can also be affected by locking in torsion problems. Even in this situation, they are still more accurate than linear tetrahedra [8].

In computational fluid dynamics, meshes are commonly refined near the boundary layer in order to increase the precision. Anisotropic refinement is preferred because it avoids creating an excess of elements, which would slow down the calculations [9]. However, the anisotropic refinement of tetrahedra can progressively deteriorate their quality [9]. Hexahedra are not affected by this issue. An hexahedron can be subdivided anisotropically and still retains an appropriate quality [9]. Some researchers have compared tetrahedral meshes to hexahedral meshes in

the context of computational bio-fluids. More specifically, the flow of air in the respiratory tract was simulated [48, 84]. In the first experiment, four types of meshes were employed: structured hexahedral, tetrahedral, isotropic adaptative tetrahedral and hybrid [48]. The hybrid mesh was composed of pyramids at the boundary and tetrahedra elsewhere. The structured hexahedral mesh style was found to have the best grid convergence [48]. In the second experiment, four types of meshes were again tested: structured hexahedral, unstructured hexahedral, prismatic and hybrid [84]. This time, the hybrid mesh was composed of hexahedra near the boundary and tetrahedra elsewhere. The hybrid mesh exhibited the worse grid convergence and leaded to a different velocity solution [84]. Aligning hexahedra and prisms in the direction of the flow was hypothesized to lower the numerical diffusion [84].

Linear tetrahedra have been reported to be inaccurate in metal forming modeling [24, 3]. Refining the tetrahedral mesh may improve the accuracy at the expense of the execution time [24, 3]. Quadratic tetrahedra cannot be employed because they lead to very complex workpiece-die contact conditions [3]. Some researchers therefore consider the linear hexahedron as the ideal element for metal forming simulation [24]. Linear hexahedra, contrary to linear tetrahedra, are also less prone to generate errors when strongly distorted [43, 18].

The simulation of flow in porous materials often involves complex domains with fractures and strata [65, 66]. By using high aspect-ratio hexahedra, these thin structures can be represented accurately with a minimum number of elements [65, 66]. Moreover, the hexahedral quality is not compromised by the thinness [65, 66]. These meshes have been used in conjunction with finite volume and finite element-finite volume solvers [65, 66].

Hexahedral-dominant meshes contain mixes of hexahedra, prisms, pyramids and tetrahedra. They are usually easier to generate than pure hexahedral meshes [63]. A recent article comparing hexahedral-dominant, pure hexahedral and tetrahedral meshes can be found in the literature [83]. Elasto-plastic beam bending and metallic foam compression were used as test problems. In terms of accuracy, the hexahedral-dominant meshes were superior to the linear tetrahedral ones [83]. Surprisingly, the hexahedral-dominant meshes could even outperform the pure hexahedral ones despite the presence of prisms and tetrahedra [83]. These results indicate the usefulness and practicality of mixed hexahedral meshes.

Many European and North American companies offer hexahedral-dominant mesh generators, either as stand alone software or embedded in larger suites. Table 1.1 shows a few of them. This array of commercial products indicates an interest from the industry.

Company	Software	Head office
Sharc Ltd	Harpoon	Manchester
Computing Objects	CM2 HexaMesh	Versailles
Ciespace Corporation	Ciespace Preprocessing	Santa Clara
Adina R&D	Adina 8.7	Massachusetts
Coventor	CoventorWare	North Carolina
ANSYS	Workbench Platform	Pennsylvania
BETA CAE Systems	ANSA	Thessaloniki

Table 1.1: Nonexclusive list of commercial software with mixed hexahedral mesh generation functionalities

The automatic generation of non-uniform triangular and tetrahedral meshes inside arbitrary domains is now thoroughly mastered [79, 76]. Common strategies include Delaunay insertion and advancing front methods [27]. The same level of performance has been sought regarding quadrilateral and hexahedral mesh generation. The requirements are well known in the literature:

1. The procedure should be automatic. A certain level of human intervention is always necessary, for example to specify the mesh size. However, complex geometries should not be more difficult to mesh than simpler ones [10].

2. The elements should have a high quality [42]. In the isotropic case, quadrilaterals should be as close as possible to perfect squares and hexahedra should be as close as possible to perfect cubes.

3. The edges of the quadrilaterals and hexahedra near the boundary should be either tangent or normal to the boundary [10, 42]. Fulfilling this requirement in conjunction with the previous one is possible if the elements orientation is made to vary smoothly throughout the domain.

4. The user should be able to specify a variable, non-uniform mesh size [10, 42].

The quadrilateral mesh generation algorithm Q-Morph published in 1999 meets all of these requirements [64]. It first constructs a triangulation of the domain using standard methods. It then recombines the triangles in order to create quadrilaterals. The algorithm proceeds in a frontal manner, starting from the boundaries and gradually filling the interior with quadrilaterals. The quadrilaterals are added layer by layer. Q-Morph has notably been implemented as part of commercial software. However, the situation is very different regarding hexahedral mesh generation. Research is still ongoing without clear solutions [76].

Hexahedral-dominant meshes contain prisms, pyramids and tetrahedra in addition to hexahedra. It may be easier to develop methods satisfying the four requirements discussed above if non-hexahedral elements are allowed. Evidently, the goal would be to maximize as much as possible the proportion of hexahedra in order to benefit from their advantages.

1.1 Previous work

A large variety of procedures for hexahedral meshing have been proposed over the years. This section reviews a small number of them, focusing on the automatic ones.

Several methods extract hexahedral meshes from octree data structures [75]. These methods are capable of generating non-uniform full hexahedral meshes in arbitrary geometries. However, the hexahedra orientation is defined by the octree root, which is a cube englobing the domain. Boundary hexahedra parallel to the boundary normals can be obtained by projecting buffer layers [52, 37]. Fig. 1.1 shows a quadtree subdividing the space around a loop. Quadtrees are the two-dimensional versions of octrees. The quadtree cells are all oriented in the same direction independently from the loop geometry.

A two-dimensional parametrization technique has shown very promising results [42]. It starts by computing a smooth orientation field aligned with the boundaries of the domain. It then identifies all the singularities of the orientation field. Singularities are points where the orientation is undefined. For example, the black arrows on Fig. 1.2 illustrate the orientation field inside a triangle. The red point is the only singularity. The parametrization technique traces special paths following the orientation field and originating from the singularities. These paths automatically decompose the domain into four-sided blocks. Each

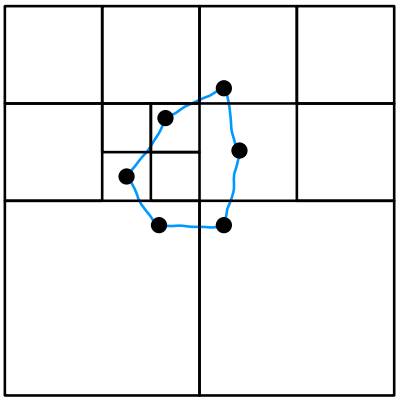


Figure 1.1: A quadtree subdividing the space around a loop composed of six vertices. Each quadtree cell cannot contain more than one vertex.

of these blocks can be easily meshed in a structured manner. Even if Fig. 1.2 only shows a triangle, very complex domains can be meshed with this parametrization technique. The resulting quadrilaterals have a very high quality and are well oriented. However, the user has only a limited control over the mesh size [42].

Certain parametrization methods have been able to mesh complex three-dimensional domains [36, 46]. Again, these methods start by computing a three-dimensional orientation field. The singularities of the orientation field are then identified. Later on, the domain is cut in order to put all singularities on the boundary and to reduce the genus to 0 [35]. A parametrization minimizing the difference between the hexahedra orientation and the three-dimensional orientation field is selected. The supplementary cutting required to put all singularities on the boundary is necessary because it leads to a better parametrization [36]. Full hexahedral meshes of very good quality can be obtained for complicated domains. However, a variable mesh size cannot be prescribed. Fig. 1.3 shows a two-dimensional triangular domain after the cutting. The red point represents the only singularity.

Graph theory can be applied to the problem of creating hexahedra by recombining tetrahedra [54]. The starting point of the method is a tetrahedral mesh, which can be viewed as a graph. In the region-face-graph representation, each tetrahedron corresponds to a node [54]. Two nodes are connected by an edge if their respective tetrahedra share a triangular face. The method consists of searching through the mesh to

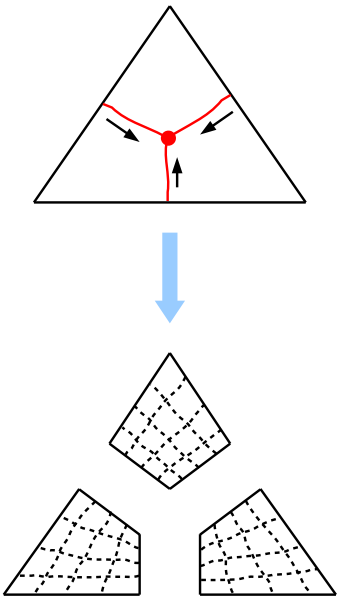


Figure 1.2: A triangular domain decomposed into three blocks

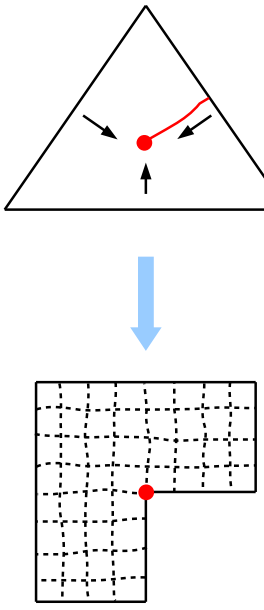


Figure 1.3: A triangular domain cut in order to put the singularity on the boundary (inspired from [59])

identify subgraphs yielding hexahedra. There are six particular subgraphs to look for [54]. One of them is composed of five nodes and the other ones are composed of six nodes. The number of nodes in these subgraphs corresponds to the number of tetrahedra required to construct an hexahedron. Once a potential hexahedron is identified, it can either be created immediately or stored in a data structure. Storing potential hexahedra has the advantage of allowing their classification along various criterions, such as element quality. The algorithm then iterates through the data structure and attempts to create the next best potential hexahedron. This graph theoritical approach is also applicable to prisms and pyramids [54].

The purpose of Lloyd's algorithm is to improve the spatial distribution of a set of vertices. It repeatedly moves the vertices to the centroid of their Voronoi cell [21]. As the number of iterations increases, the Voronoi cells take the shape of perfect hexagons. In fact, it has been proven that Lloyd's algorithm minimizes an energy function equal to the sum of the moments of inertia of the Voronoi cells [47]. Lloyd's algorithm corresponds to a fixed-point iteration process, but the convergence can be accelerated by employing the limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) optimization procedure [47]. Fig. 1.4 illustrates the optimization of a Voronoi diagram. Lévy-Liu's algorithm consists of minimizing higher order moments of inertia [45]. The Voronoi cells then become squarely instead of hexagonal, which has the effect of aligning the vertices in precise directions. Tetrahedral meshes treated by Lévy-Liu's algorithm make excellent candidates for recombination. When used in conjunction with the graph recombination method discussed above, Lévy-Liu's algorithm can generate mixed meshes with high hexahedra percentages [45].

The authors Yamakawa and Shimada have designed an iterative procedure capable of aligning vertices in three dimensions [88]. It is inspired from the field of newtonian mechanics. The first step of the procedure consists of filling the domain with crystal cells. Each crystal cell has the shape of a cube: there is one particle at the center and eight particles in the corners. Crystal cells exert force on each other via their particles. A set of equations of motion govern the cells positions. Throughout the process, cells can be added or removed depending on the local density. Once cells velocities have sufficiently decrease, the center of each crystal cell becomes a mesh vertex. The vertices are then tetrahedralized. Finally, the tetrahedra are recombined into hexahedra, prisms and pyramids using an algorithm based on recombination patterns.

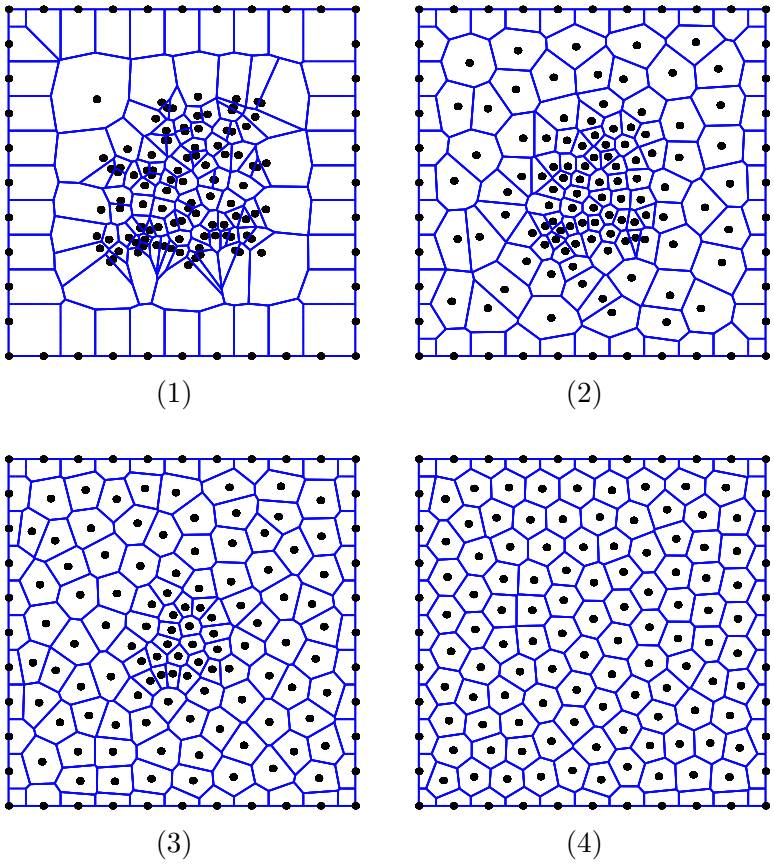


Figure 1.4: The mesh vertices and their Voronoi cells after 0 iterations (1), 7 iterations (2), 15 iterations (3) and 80 iterations (4)

Full hexahedral meshes can be created by subdividing tetrahedra [82]. However, the hexahedra generated this way are randomly oriented and have a low quality. These meshes have been used with success in certain applications [82].

Hausner's method was not initially developed for mesh generation [33], but could still be useful. Its purpose is to align vertices in two dimensions. It is based on the classical Lloyd's algorithm, which iteratively moves vertices to the centroid of their Voronoi cells. However, this time the Voronoi diagrams are computed with the L_1 norm instead of the Euclidean norm. The method is therefore able to align points in precise directions instead of simply distributing them uniformly. The

Voronoi diagrams have also the particularity of being discrete and not continuous. They are created by projecting pyramids on the graphic card Z-buffer. The overall procedure is very simple, which is an important advantage. Unfortunately, it may not be practical for meshes with large variations in element size, because of the limited number of pixels. By recombining triangles [70], this method could lead to well aligned quadrilateral meshes.

Some hexahedral mesh generation methods have been developed for specific geometries. For example, blood flow simulations require three-dimensional meshes of the vascular system. Zhang et al. have published an automatic procedure capable of creating hexahedral meshes inside arteries [91]. Each artery is swept along its length by a quadrilateral mesh, creating a semi-structured hexahedral mesh [91]. The quadrilateral mesh is always perpendicular to the median axis line of the artery. However, arteries can also contain junctions of three or more branches. These junctions are treated by special templates [91].

1.2 Outline

Blossom-quad is an algorithm capable of creating full quadrilateral meshes by recombining the triangles of an initial triangular mesh [70]. Fig. 1.5 shows a mesh generated with Blossom-quad. This type of indirect approach is also applicable in three dimensions. However, five, six or seven tetrahedra are required to construct a single hexahedron [54]. Yamakawa-Shimada's algorithm can generate mixed hexahedral meshes by recombining tetrahedra [88]. A certain number of tetrahedra usually remain in the final mesh. Blossom-quad is able to achieve a recombination rate of 100% because it employs Edmund's Blossom algorithm, but no such strategy exists in three dimensions [70, 54]. The indirect approach has the advantage of being completely automatic and applicable to complex domains. It also allows the user to prescribe a variable mesh size. In three dimensions, the main drawback is the presence of tetrahedra in the final mesh.

Recombination algorithms based on the indirect approach do not modify the initial vertices distribution. However, the vertices distribution strongly influences the quality and alignment of the final elements, whether they are quadrilaterals or hexahedra [7, 6]. In order to obtain high quality elements parallel to the boundary and to avoid random orientations, an initial distribution composed of aligned vertices

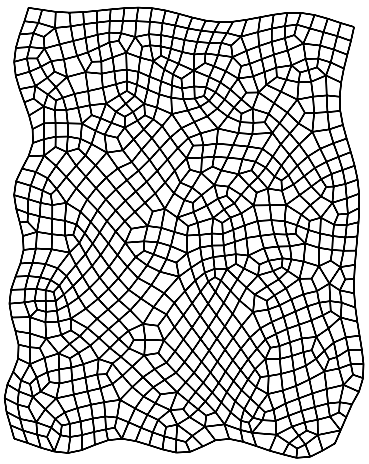


Figure 1.5: A quadrilateral mesh of a two-dimensional domain

is necessary [7, 6]. In three dimensions, aligned vertices also result in significantly higher hexahedra percentages [6]. Nevertheless, standard meshing algorithms usually attempt to generate equilateral triangles in two dimensions and regular tetrahedra in three dimensions. Therefore, the corresponding vertices distributions do not have a particular orientation. Methods capable of aligning vertices are going to be necessary. These methods can be divided into two classes: the iterative ones and the frontal ones. Iterative methods start from an initial distribution and repeatedly move the vertices. For example, Lévy-Liu's algorithm is an iterative method. Frontal methods do not displace vertices but create them layer by layer, usually starting from the boundary. Frontal methods are very recent and not particularly numerous. The Delquad algorithm is one of them [69, 68], along with the procedures explained in this text.

Chapter 2 reviews Borouchaki's, Blossom-quad and Yamakawa-Shimada's recombination algorithms. Chapter 3 describes certain methods for generating orientation fields inside two and three-dimensional domains. These orientation fields will determine the alignment directions of the future quadrilaterals and hexahedra. Chapter 4 discusses Lloyd's energy minimization in two dimensions. Chapter 5 presents an attempt to minimize Lloyd's energy in three dimensions. However, the execution time is too large. Consequently, chapter 6 introduces a frontal algorithm for three-dimensional domains. Chapter 7 proceeds

with a surfacic version of the frontal algorithm. Chapter 8 concludes this work. Chapter 9 is an appendix that contains pseudo-code relevant to the frontal algorithm and Lloyd's energy minimization.

1.3 Contributions

The author of the present document has devised and implemented the frontal algorithm in three dimensions. He has also implemented the algorithms for minimizing Lloyd's energy in two and three dimensions. These algorithms are modified versions of the original procedure described in Lévy and Liu's article [45]. Finally, the present author has implemented the hexahedra recombination algorithm conceived by Yamakawa and Shimada [88].

1.4 Publications

This work has led to the following publications:

Tristan Carrier Baudouin, Jean-François Remacle, Emilie Marchandise, Jonathan Lambrechts and François Henrotte. Lloyd's energy minimization in the L_p norm for quadrilateral surface mesh generation. *Eng Comput*, DOI 10.1007/s00366-012-0290-x, 2012.

Tristan Carrier Baudouin, Jean-François Remacle, Emilie Marchandise, François Henrotte and Christophe Geuzaine. A frontal approach to hex-dominant mesh generation. *Adv Model Sim Eng Sci*, submitted.

Tristan Carrier Baudouin, Jean-François Remacle, Emilie Marchandise and Jonathan Lambrechts. L_p Lloyd's energy minimization for quadrilateral surface mesh generation. In W. R. Quadros, editor, *Proceedings of the 20th International Meshing Roundtable*, Paris, 2011.

Jean-François Remacle, François Henrotte, Tristan Carrier Baudouin, Eric Béchet, Emilie Marchandise, Christophe Geuzaine and Thibaud Mouton. A Frontal Delaunay quad mesh generator using the L_∞ norm. *Int J Numer Meth Eng*, 94:494-512, 2013.

Jean-François Remacle, François Henrotte, Tristan Carrier Baudouin, Christophe Geuzaine, Eric Béchet, Thibaud Mouton and Emilie Marchandise. A Frontal Delaunay quad mesh generator using the L_∞ norm. In W. R. Quadros, editor, *Proceedings of the 20th International Meshing Roundtable*, Paris, 2011.

Jean-François Remacle, Amaury Johnen, Jonathan Lambrechts, Thomas Toulorge, Tristan Carrier Baudouin, Emilie Marchandise and Christophe Geuzaine. New Mesh generation developments in Gmsh. In *Actes du 11e Colloque National en Calcul des Structures*, Giens, 2013.

Acknowledgement. This work was partially supported by the Belgian Walloon Region under WIST grants ONELAB 1017086 and DOMHEX 1017074.

Chapter 2

Triangles and tetrahedra merging

The direct/indirect terminology was introduced by the author Steve Owen in a 1998 article in order to classify unstructured quadrilateral and hexahedral mesh generation techniques [61]. Following this terminology, indirect methods create quadrilaterals and hexahedra by merging the simplices of an initial mesh. On the contrary, direct methods do not rely on recombination schemes. The paving algorithm published in 1991 is a direct method [61, 12]. The quadrilaterals are created row by row and the colliding fronts are seamed together. The paving algorithm has been implemented as part of commercial software [61]. A three-dimensional version of the paving algorithm, called plastering, has also been published [61, 11]. However, treating collisions of hexahedra is very complex. Direct methods are much diverse. The octree-based procedure presented in chapter 1 can be considered to be direct.

Indirect methods can be applied to triangular or tetrahedral meshes with large variations in element size. They are therefore capable of producing non-uniform quadrilateral and mixed hexahedral meshes. In two dimensions, two triangles are necessary to construct a single quadrilateral. In three dimensions, five or six tetrahedra are necessary to construct a single hexahedron [88, 54]. In certain cases, up to seven tetrahedra can be required.

Borouchaki's and Blossom-quad algorithms are both two-dimensional indirect methods [15, 70]. Yamakawa-Shimada's algorithm is a three-dimensional indirect method [88]. These three procedures are briefly reviewed in the present chapter. A few examples are presented and ideas of improvements are discussed.

2.1 Borouchaki's and Blossom-quad algorithms

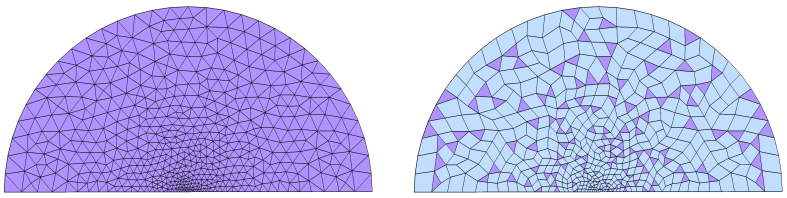
A quadrilateral can be created by merging two triangles sharing one edge. The Borouchaki's algorithm starts by identifying all possible triangle pairs leading to quadrilaterals. These pairs are sorted by quadrilateral quality. Starting from the highest quality triangle pair, the ones composed of unmarked triangles are merged in order. When a triangle pair is merged, both triangles are marked for deletion. A certain number of triangles will remain in the final mesh.

The quality of a quadrilateral $\eta(q)$ is defined by the values of its four angles $\alpha_k, k = 1, 2, 3, 4$ [70]:

$$\eta(q) = \max\left(1 - \frac{2}{\pi}\max_k\left(\left|\frac{\pi}{2} - \alpha_k\right|\right), 0\right) \tag{2.1}$$

The quality goes from zero to one. If the quadrilateral is a perfect square, it is equal to one.

Fig. 2.1 shows a quadrilateral mesh created by Borouchaki's algorithm. It contains 427 quadrilaterals and 136 triangles. The elements have an average quality of 0.69.

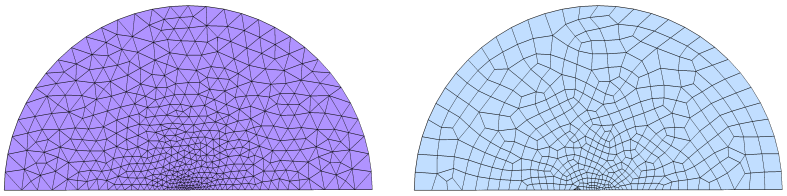


(1) (2)

Figure 2.1: Mesh (2) has been created by combining the triangles of mesh (1) using Borouchaki's algorithm.

The Blossom-quad algorithm also identifies all possible triangle pairs. However, it then employs Edmond's Blossom algorithm to find a perfect matching maximizing the mean quadrilateral quality [70]. Usually, no triangles remain in the final mesh. Blossom-quad has a polynomial (quadratic) complexity.

Fig. 2.2 shows a quadrilateral mesh created by the Blossom-quad algorithm. The quadrilaterals have an average quality of 0.75. The meshes labelled (1) on Fig. 2.1 and Fig. 2.2 are identical.



(1) (2)

Figure 2.2: Mesh (2) has been created by combining the triangles of mesh (1) using the Blossom-quad algorithm.

2.2 Yamakawa-Shimada's algorithm

The Blossom-quad algorithm can determine the perfect matching of triangles that maximize the average quadrilateral quality [70]. Its execution time is bounded by a polynomial function of the number of vertices and edges [70]. However, finding a matching of tetrahedra that optimizes a certain criterion is an NP-complete problem [54]. There is currently no algorithm capable of solving an NP-complete problem in polynomial time [73]. In fact, if such an algorithm was identified, then all NP-complete problems could be solved polynomially [73, 38]. In order to avoid impractical execution times, a common strategy is to use heuristics [73]. Heuristics are not expected to find the perfect solution. Yamakawa-Shimada's algorithm is one of the available options for recombining tetrahedra [88, 54]. It is the three-dimensional version of Borouchaki's algorithm.

Yamakawa-Shimada's algorithm begins by iterating through the tetrahedra of the initial mesh. For each tetrahedron, it attempts to find neighboring tetrahedra with which to construct an hexahedron. Three patterns of assembly are considered. Two of these three patterns are

described in [88]. When a potential hexahedron is found, it is added to an array. However, the hexahedron will not necessarily be part of the final mesh. Once all tetrahedra have been visited, the array is sorted by hexahedral quality.

Starting from the highest quality hexahedron, the algorithm then iterates through the array. Potential hexahedra composed of tetrahedra not yet marked for deletion are added to the mesh. It is to be noted that only a small percentage of all potential hexahedra appear in the final mesh.

Prisms can later be added by following a similar procedure [88]. All prisms are composed of three tetrahedra. There is only one pattern of construction for prisms [88].

Surface meshes can be extracted from mixed volumic meshes created by Yamakawa-Shimada's algorithm. Surfacic quadrilateral faces belonging to hexahedra, prisms and pyramids become quadrilaterals. Surfacic triangular faces belonging to prisms, pyramids and tetrahedra become triangles.

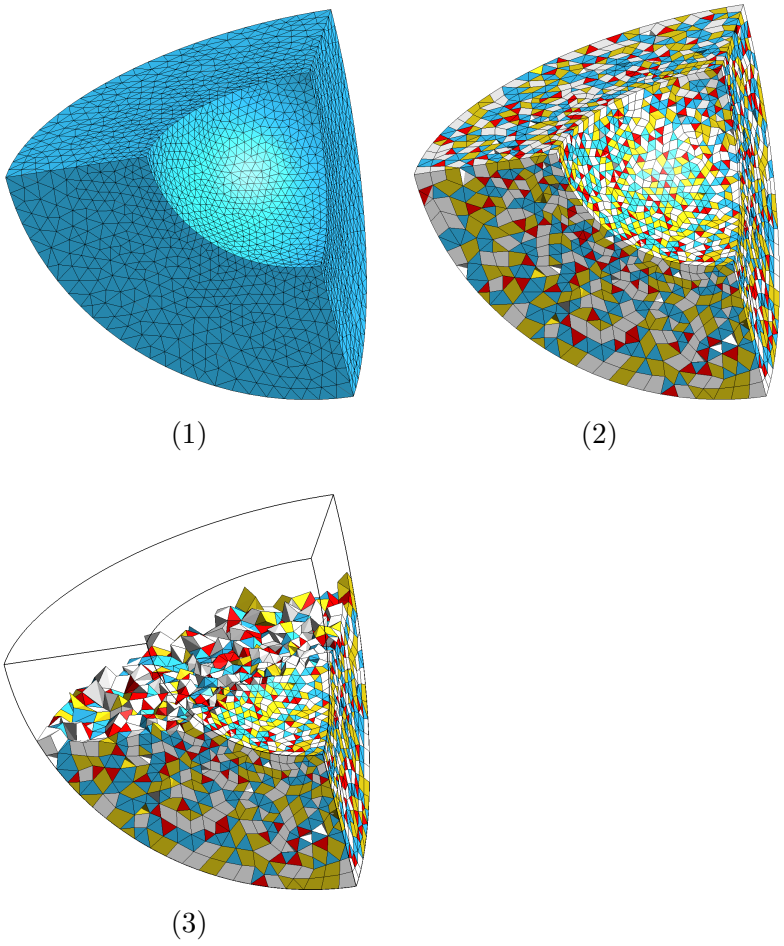
The quality Q of an hexahedron is defined in Eq. 2.2. i is the vertex number: it goes from 1 to 8. $\boldsymbol{v_{i1}}$, $\boldsymbol{v_{i2}}$ and $\boldsymbol{v_{i3}}$ are the three vectors parallel to the three edges connected to vertex i. Their orientations have no impact on the final value of Q. Q is in fact the modulus of the minimum scaled Jacobian [88]. Evidently, Q is meaningless for invalid hexahedra. Invalid hexahedra are characterized by a null or negative Jacobian determinant, which renders the mesh improper for calculations.

$$Q = \min_{i=1..8} \left| \frac{\boldsymbol{v_{i1}} \cdot (\boldsymbol{v_{i2}} \times \boldsymbol{v_{i3}})}{||\boldsymbol{v_{i1}}||\ ||\boldsymbol{v_{i2}}||\ ||\boldsymbol{v_{i3}}||} \right| \tag{2.2}$$

Fig. 2.3 shows a mesh created with Yamakawa-Shimada's algorithm. The colors indicate the element type. The mesh contains 11.28% of hexahedra by number and 37.11% by volume. The hexahedra have an average quality of 0.64.

2.3 Finite element conformity

Let's assume that finite elements of the first order are employed. The tetrahedral shape functions are therefore linear, while the hexahedral shape functions are trilinear [26]. On triangular faces, the interpolation is linear and takes into account three degrees of liberty. On quadrilateral



(1) (2)

(3)

Figure 2.3: Mesh (2) has been created by combining the tetrahedra of mesh (1) using Yamakawa-Shimada's algorithm. Mesh (3) is a cutaway view. The hexahedra are white, the prisms are yellow, the pyramids are red and the tetrahedra are light blue.

faces, the interpolation is bilinear and takes into account four degrees of liberty [88, 17]. If a nonplanar quadrilateral face is adjacent to a triangular face, there will be a gap or overlap [88]. The elements are not going to be a perfect partition of the domain anymore, which goes against the basic assumptions of the finite element method. Gaps or overlaps can also be created by several configurations of neighboring hexahedra or prisms. Figs. 2.4 and 2.5 show four cases of non-conformities between hexahedra [88].

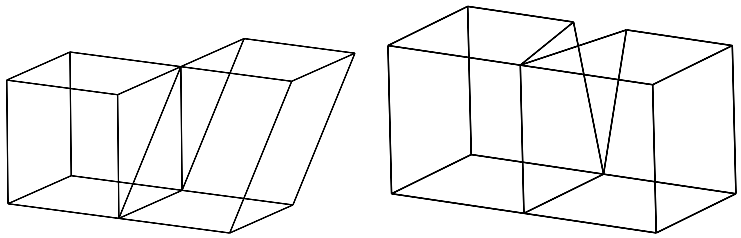


Figure 2.4: Non-conformities between two hexahedra sharing three vertices

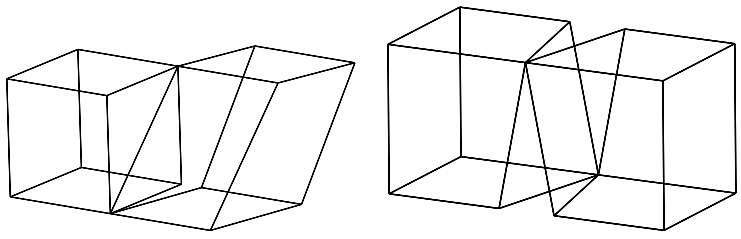


Figure 2.5: Non-conformities between two hexahedra sharing two vertices

Figs. 2.6 and 2.7 show six cases of non-conformities between one hexahedron and one prism. Non-conformities resulting from neighboring prisms can be deduced from these six cases.

Yamakawa-Shimada's algorithm should therefore avoid creating the configurations illustrated on Fig. 2.4 to 2.7 while iterating through the sorted arrays of potential hexahedra and prisms. Non-conformities can be efficiently identified by employing hashing techniques.

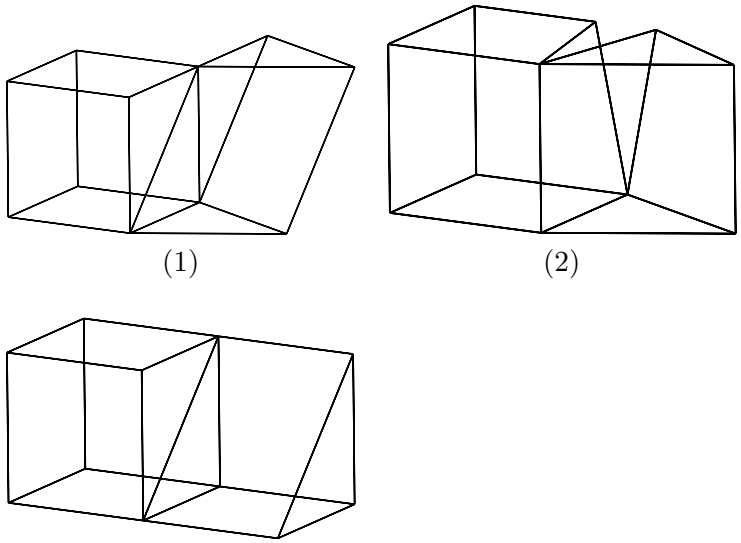


Figure 2.6: Non-conformities between one hexahedra and one prism sharing three vertices

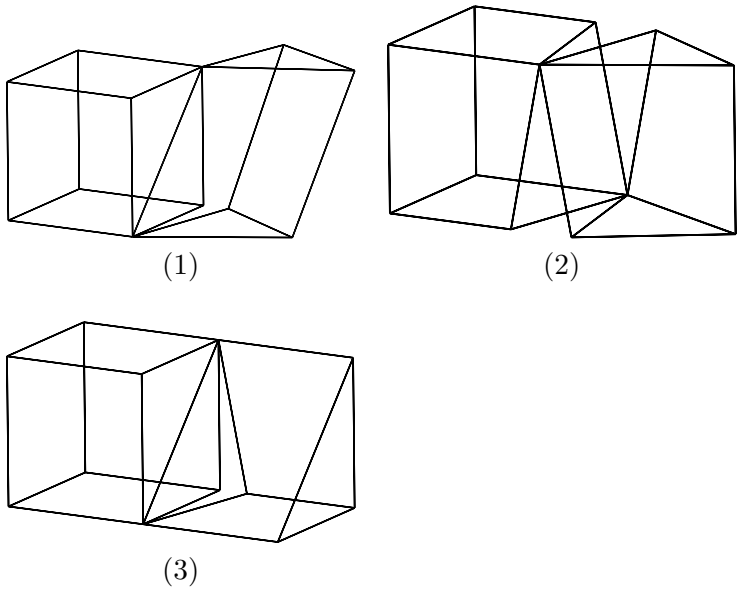


Figure 2.7: Non-conformities between one hexahedra and one prism sharing two vertices

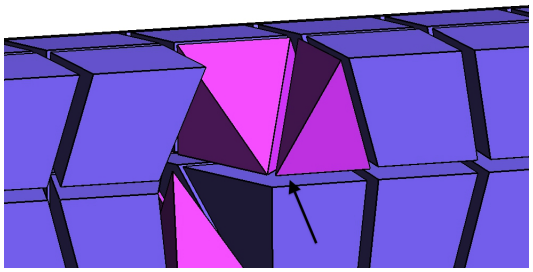
After the creation of hexahedra and prisms, some tetrahedra are recombined into pyramids. Every pair of tetrahedra resting on a quadrilateral face is merged to form a pyramid. This step can fix many non-conformities. However, it does not resolve them all. As shown on Fig. 2.8, many quadrilateral faces can still be adjacent to triangular faces belonging to either tetrahedra or pyramids.

These non-conformities can be repaired by Owen-Canann-Saigal's algorithm [62]. Owen-Canann-Saigal's algorithm first creates a flat pyramid on each non-conformal quadrilateral face. The apex of the pyramid was not initially present in the mesh, it was added. Surrounding tetrahedra and pyramids need to be subdivided to accommodate this new vertex. The pyramid is then raised in order to have a non-null volume.

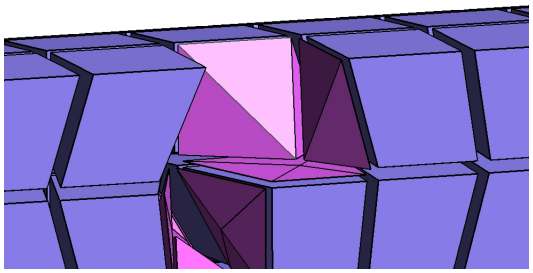
Owen-Canann-Saigal's algorithm can render a mixed hexahedral mesh completely conformal. However, it has a drawback. It increases the number of tetrahedra and pyramids, which lowers the percentage of hexahedra by number. As a consequence, it was not used in this work. Finite element solvers capable of handling a limited number of non-conformal faces are required. Quadratic tetrahedra are often preferred to linear tetrahedra because of the better accuracy they provide [17]. Quadratic tetrahedra can be used in conjunction with linear or quadratic hexahedra. Some of the nodes of the tetrahedra adjacent to hexahedra will have to be ignored in order to avoid hanging nodes. For example, if a 10-node quadratic tetrahedron is adjacent to a 20-node quadratic hexahedron, there will be an hanging node located at the middle of the edge crossing the quadrilateral face diagonally [17]. This node needs to be ignored. As a result, the tetrahedron will only have nine nodes. If a quadratic tetrahedron is adjacent to a linear hexahedron, three of the tetrahedral nodes will have to be ignored [17]. Off course, removing hanging nodes does not solve the non-conformities. Despite this fact, non-conformal mixed hexahedral meshes have been tested in the literature [17]. Tension, torsion and bending were some of the problems considered. Acceptable error has been achieved in some cases [17].

2.4 Results analysis

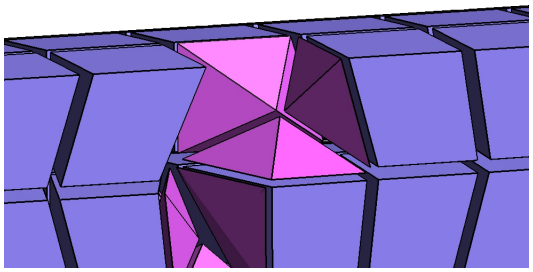
Contrary to triangles and tetrahedra, quadrilaterals and hexahedra have an orientation. In general, the edges of the quadrilateral and hexahedral elements sufficiently close to the boundary should be either tangent or



(1)



(2)

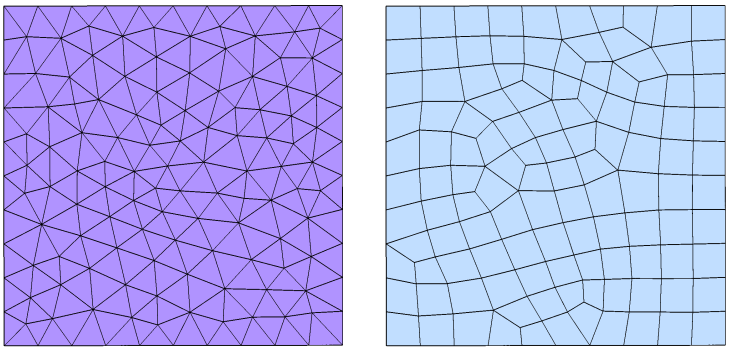


(3)

Figure 2.8: As indicated by the arrow, there is a non-conformity between the two pyramids and the lower hexahedra on mesh (1). Meshes (2) and (3) illustrate the different steps taken by Owen-Canann-Saigal's algorithm to fix the non-conformity. Tetrahedra are not shown to facilitate visibility.

normal to the boundary. Moreover, the orientation variations inside the domain should be as smooth as possible. Quadrilateral and hexahedral meshes can be evaluated in terms of this orientation criterion.

Fig. 2.9 shows a mesh created with the Blossom-quad algorithm. The quadrilaterals have an average quality of 0.82 and there are no remaining triangles.



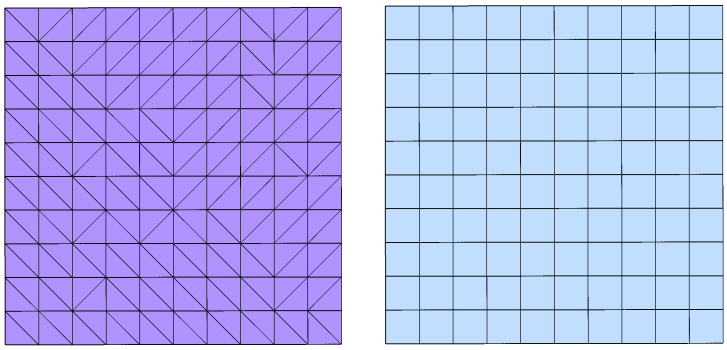
(1) (2)

Figure 2.9: Mesh (2) was created by applying the Blossom-quad algorithm to mesh (1).

Fig. 2.10 shows another mesh created with the Blossom-quad algorithm. However, this time the vertices of the initial triangular mesh were aligned in a rectangular manner. Contrary to Fig. 2.9, a standard unstructured triangular mesh was not used. The quadrilaterals all have a quality of 1.0.

The quadrilaterals of Fig. 2.10 are far superior to those of Fig. 2.9 in terms of quality. In addition, they are oriented in the same direction. As a consequence, the quadrilateral mesh illustrated on Fig. 2.10 is much more preferable.

This example shows the importance of the initial vertices distribution. Well positioned initial vertices can improve both the average quadrilateral quality and the orientation smoothness. Several methods could be devised to increase the alignment of the initial vertices prior to the recombination step. For example, the mesh vertices could be moved iteratively. The displacements would become smaller and smaller after each iteration until convergence. Alternatively, the aligned vertices could be constructed all at once, using a frontal approach. These methods will be discussed in the next chapters. Of course, for more complex



(1) (2)

Figure 2.10: Mesh (2) was created by applying the Blossom-quad algorithm to mesh (1).

geometries, the average quadrilateral quality will not be perfect, contrary to Fig. 2.10.

Fig. 2.11 shows a mesh created with Yamakawa-Shimada's algorithm. It contains 11.83% of hexahedra by number and 38.04% by volume. The hexahedra have an average quality of 0.64. The amount of hexahedra in the final mesh is very low. The initial tetrahedral mesh was constructed with a Delaunay refinement algorithm [77, 28].

Fig. 2.12 shows another mesh created with Yamakawa-Shimada's algorithm. It contains 100% of hexahedra. The hexahedra all have a quality of 1.0. This time, the vertices of the initial tetrahedral mesh were aligned in a rectangular manner.

Fig. 2.12 contains much more hexahedra than Fig. 2.11. The initial vertices distribution is thus also of great importance in three dimensions. If the vertices are not well aligned prior to the recombination step, the hexahedral elements will be in minority. In addition, aligned vertices can lead to higher hexahedra quality and smoother orientations. The iterative and frontal methods mentioned earlier can also be applied in three dimensions. However, for most geometries, a certain number of tetrahedra will remain in the final mesh. Edges, surfaces and volumes are assumed to be meshed sequentially. Therefore, the surfacic vertices distribution has a large influence on the volumic vertices distribution.

To summarize, algorithms capable of aligning vertices are important in two dimensions because they lead to higher quadrilateral quality and smoother orientations. These algorithms are also important in three

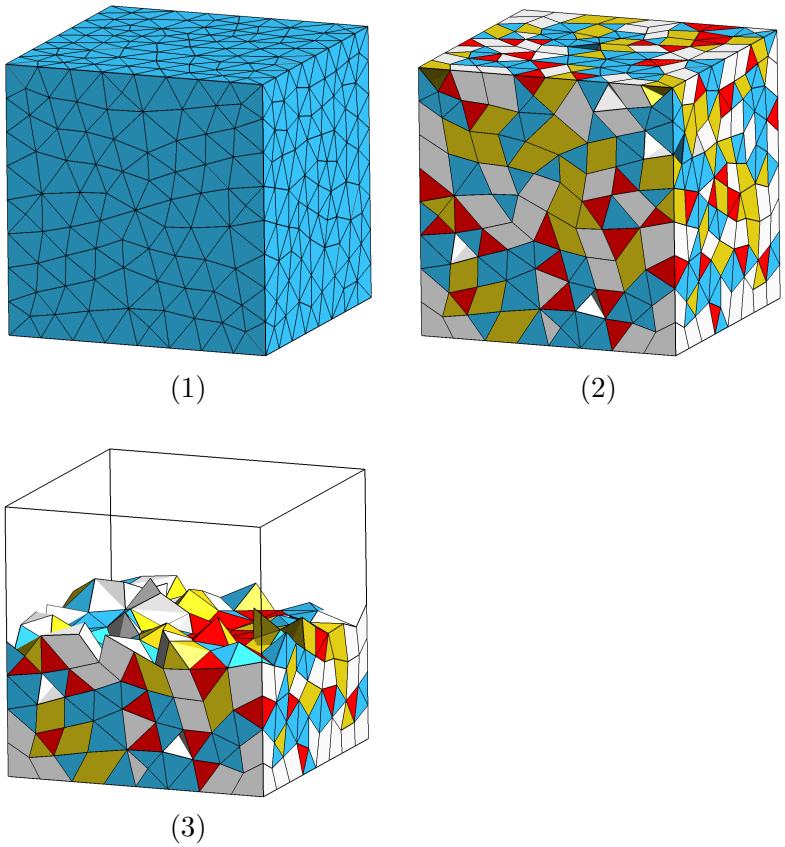
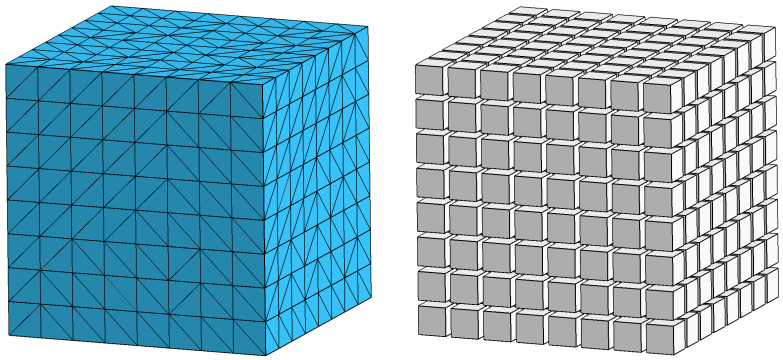


Figure 2.11: Mesh (2) was created by applying Yamakawa-Shimada's algorithm to mesh (1). Mesh (3) is a cutaway view.

dimensions mainly because they result in higher percentages of hexahedra. The alignment directions are determined by pre-defined orientation fields. The next chapter will discuss methods capable of generating such orientation fields.



(1) (2)

Figure 2.12: Mesh (2) was created by applying Yamakawa-Shimada's algorithm to mesh (1). The hexahedra have been shrank for better visibility.

Chapter 3

Cardinal direction fields

As said in chapter 2, quadrilaterals and hexahedra are characterized by an orientation. A quadrilateral orientation is specified by four cardinal directions (left, right, up and down). An hexahedron orientation is specified by six cardinal directions (left, right, up, down, front and back). These cardinal directions need to be defined at every point of the domain so they can form a cardinal direction field. Cardinal direction fields are necessary in order to guide the construction of aligned vertices distributions. Near the boundaries of the domain, the orientation is already known: the edges of the quadrilaterals and hexahedra should be either tangent or normal to the boundary. Inside the domain, the orientation should vary smoothly, or at least as smoothly as possible. This chapter discusses automatic methods capable of obtaining such orientations for planar domain, three-dimensional surfaces and volumes.

There are several techniques available for computing non-uniform size fields in an automatic manner. For example, the user can define the mesh size at the geometrical vertices of the CAD model. A Laplace solver can then be employed to obtain the size field over the surfaces and inside the volumes. There are no differences between tetrahedra and hexahedra regarding size fields generation.

3.1 Two-dimensional space

Fig. 3.1 shows an example of a cardinal direction field inside a planar domain. The cardinal direction field is smooth and parallel to the boundary normals. An automatic method capable of obtaining such fields for arbitrary geometries would be very useful.

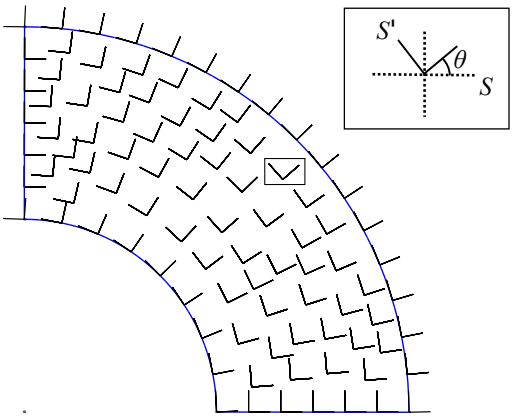


Figure 3.1: A smooth cardinal direction field inside a planar domain

θ is the angle between the local coordinate system S' and the global coordinate system S. It is a function of the position $\boldsymbol{x}$. θ can be used to compute the four cardinal directions ($\boldsymbol{e_1}$ and $\boldsymbol{e_2}$ are the standard basis vectors of S):

$$\boldsymbol{d_1} = \cos(\theta)\boldsymbol{e_1} + \sin(\theta)\boldsymbol{e_2} \tag{3.1}$$

$$\boldsymbol{d_2} = -\sin(\theta)\boldsymbol{e_1} + \cos(\theta)\boldsymbol{e_2}$$

$$\boldsymbol{d_3} = -\boldsymbol{d_1}$$

$$\boldsymbol{d_4} = -\boldsymbol{d_2}$$

The quadrilaterals are expected to be aligned with the boundaries. θ is therefore already known at the boundary. The Laplace equation could prove useful to obtain a value of θ inside the domain [78]. Nevertheless, the Laplace equation cannot be directly applied to θ, because all discrete values $\theta + k\frac{\pi}{2}$, $k \in \mathbb{Z}$ indicate the same orientation. For example, let's

consider a square domain aligned with S. At the boundary, the angle θ could be equal to 0, $\frac{\pi}{2}$, π, $\frac{3\pi}{2}$, etc. The smoothing process would lead to an incorrect orientation.

Let z be a complex function of θ.

$$z(\boldsymbol{x}) = a(\boldsymbol{x}) + ib(\boldsymbol{x}) = e^{i4\theta(\boldsymbol{x})} \quad \text{with} \tag{3.2}$$

$$a(\boldsymbol{x}) = \cos(4\theta(\boldsymbol{x})) \quad \text{and} \quad b(\boldsymbol{x}) = \sin(4\theta(\boldsymbol{x}))$$

z is invariant for all discrete values $\theta + k\frac{\pi}{2}$, $k \in \mathbb{Z}$.

$$\begin{aligned} e^{i4(\theta(\boldsymbol{x})+k\frac{\pi}{2})} &= e^{i4\theta(\boldsymbol{x})+ik2\pi} \\ &= e^{i4\theta(\boldsymbol{x})}e^{ik2\pi} \\ &= e^{i4\theta(\boldsymbol{x})}(e^{i2\pi})^k \\ &= e^{i4\theta(\boldsymbol{x})} \end{aligned} \tag{3.3}$$

It is the components of z that are smoothed with the Laplace equation [7, 69, 68]. The values of a and b are already known at the boundary.

$$\nabla^2 a = 0 \quad \text{and} \quad \nabla^2 b = 0 \quad \text{inside the domain} \tag{3.4}$$

$$a = \cos(4\theta(\boldsymbol{x})) \quad \text{and} \quad b = \sin(4\theta(\boldsymbol{x})) \quad \text{on the boundaries}$$

Let P be a boundary mesh vertex located at position $\boldsymbol{x}$. P is connected to one boundary edge at its left and one boundary edge at its right. $\theta(\boldsymbol{x})$ is equal to the angle of one of the two boundary edges with respect to the x-axis of S.

Following the maximum-minimum principle, the components of z inside the domain will never escape the range of -1 to 1.

The modulus of z is equal to one on the boundaries, i.e. $\sqrt{a^2 + b^2} = 1$. However, there are no guarantees that the modulus of z will remain equal to one inside the domain. This is why both values a and b are necessary in order to retrieve the angle θ:

$$\theta(\boldsymbol{x}) = \frac{1}{4}\text{atan2}(b(\boldsymbol{x}), a(\boldsymbol{x})) \tag{3.5}$$

The image of the function atan2 ranges from $-\pi$ to π. The value of the angle θ inside the domain will therefore oscillate between $-\frac{\pi}{4}$ and $\frac{\pi}{4}$.

Continuity is an important requirement. Small variations in θ should lead to small variations in a and b. Conversely, small variations in a and b should lead to small variations in θ. Fortunately, a and b are continuous functions of θ. The function atan2, which is used to retrieve θ from a and b, is mostly continuous. However, it has a discontinuity along one axis and can jump from π to $-\pi$. As shown on equation 3.5, these values are further divided by four. $\frac{\pi}{4}$ and $-\frac{\pi}{4}$ indicate the same orientation.

The modulus of z is not constant for the geometry illustrated on Fig. 3.2. It becomes smaller as the distance with the boundary increases.



Figure 3.2: A color map of the modulus of $z(\boldsymbol{x}) = e^{i4\theta(\boldsymbol{x})}$. The modulus is not constant over the domain.

The variability of the modulus of z is a drawback. The modulus becomes smaller because the components of z, a and b, also become smaller. However, the angle θ is extracted from a and b. The numerical error will therefore gain a larger influence on the final value of θ. If a and b approach zero, the orientation could even become random. The Laplace equations are solved with continuous finite elements. Formula (3.6) shows the relation between the numerical error ϵ in the L_2 norm and the mesh size h, with k being the order of the finite element method

[26]. A smaller mesh size h leads to a smaller numerical error.

$$\epsilon \simeq Ch^{k+1} \tag{3.6}$$

Figs. 3.3, 3.4 and 3.5 show examples of cardinal direction fields over various two-dimensional domains. Again, the cardinal direction fields are smooth and parallel to the boundary normals.

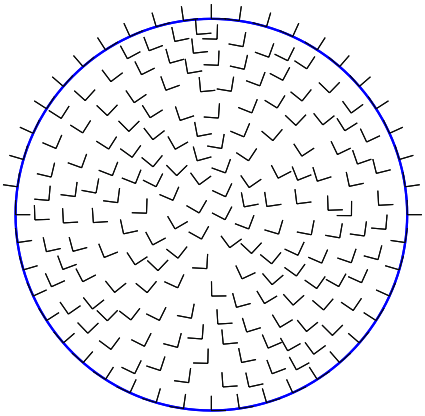


Figure 3.3: The cardinal direction field inside a disc

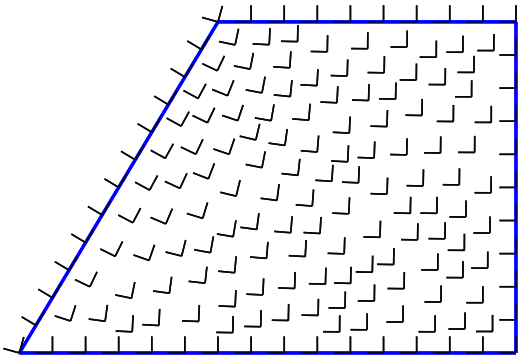


Figure 3.4: The cardinal direction field inside a trapezoid

Recently, a supplementary scheme capable of keeping the modulus of z constant has been developed [42].

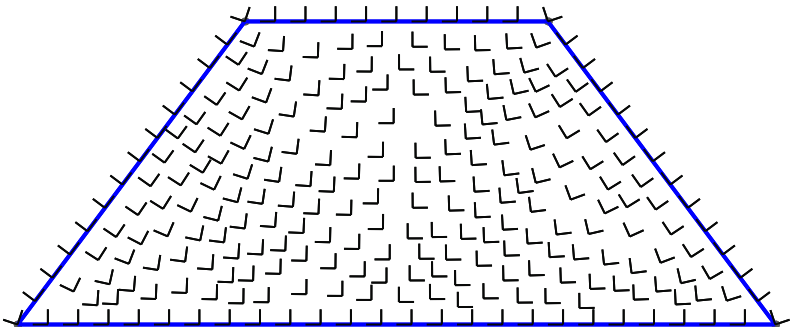


Figure 3.5: The cardinal direction field inside an isosceles trapezoid

3.2 Three-dimensional surfaces

In two-dimensional domains, a single angle is sufficient to indicate the local orientation. This is not the case for three-dimensional surfaces. The method developed earlier would be very difficult to extend to multiple angles.

However, reducing to one the number of parameters necessary to specify the orientation on three-dimensional surfaces may be possible with parametrization techniques. The method presented in the last section would then become applicable.

Certain authors have used the local curvature to define the orientation on three-dimensional surfaces [45]. This approach is not going to be followed here, because it may not give correct results for near-planar surfaces.

Let M be a mapping between a two-dimensional parametric domain and a three-dimensional surface [67, 50, 51]. Fig. 3.6 shows a quarter sphere parametrized by a planar domain.

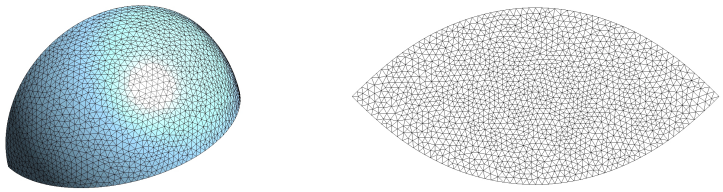
$$\mathrm{M} : (u, v) \mapsto (x, y, z) \tag{3.7}$$

$\boldsymbol{t_1}$ and $\boldsymbol{t_2}$ are the first derivatives of the mapping M. They are tangent to the surface.

$$\boldsymbol{t_1} = (\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}), \quad \boldsymbol{t_2} = (\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}) \tag{3.8}$$

The normal to the surface is equal to the cross product of $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$.

$$\boldsymbol{n} = \frac{\boldsymbol{t_1} \times \boldsymbol{t_2}}{||\boldsymbol{t_1} \times \boldsymbol{t_2}||} \tag{3.9}$$



(1) (2)

Figure 3.6: A three-dimensional surface (1) parametrized by a two-dimensional domain (2)

$\boldsymbol{s_1}$ and $\boldsymbol{s_2}$ are two tangent orthonormal vectors constructed from $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$.

$$\boldsymbol{s_1} = \frac{\boldsymbol{t_1}}{||\boldsymbol{t_1}||}, \quad \boldsymbol{s_2} = \frac{\boldsymbol{t_1} \times \boldsymbol{n}}{||\boldsymbol{t_1} \times \boldsymbol{n}||} \tag{3.10}$$

The local cardinal directions are linear combinations of $\boldsymbol{s_1}$ and $\boldsymbol{s_2}$:

$$\boldsymbol{d_1} = c_1\boldsymbol{s_1} + c_2\boldsymbol{s_2} \tag{3.11}$$

$$\boldsymbol{d_2} = -c_2\boldsymbol{s_1} + c_1\boldsymbol{s_2}$$

$$\boldsymbol{d_3} = -\boldsymbol{d_1}$$

$$\boldsymbol{d_4} = -\boldsymbol{d_2}$$

Adding the constraint $\sqrt{c_1^2 + c_2^2} = 1$ normalizes the vectors $\boldsymbol{d_1}$, $\boldsymbol{d_2}$, $\boldsymbol{d_3}$ and $\boldsymbol{d_4}$. c_1 and c_2 can then be expressed in terms of the angle θ.

$$c_1 = \cos(\theta) \text{ and } c_2 = \sin(\theta) \tag{3.12}$$

The local cardinal directions now depend on only one parameter: θ. All discrete values $\theta + k\frac{\pi}{2}$, $k \in \mathbb{Z}$ indicate the same orientation.

$$\boldsymbol{d_1} = \cos(\theta)\boldsymbol{s_1} + \sin(\theta)\boldsymbol{s_2} \tag{3.13}$$

$$\boldsymbol{d_2} = -\sin(\theta)\boldsymbol{s_1} + \cos(\theta)\boldsymbol{s_2}$$

$$\boldsymbol{d_3} = -\boldsymbol{d_1}$$

$$\boldsymbol{d_4} = -\boldsymbol{d_2}$$

At the boundary of the three-dimensional surface, the vector $\boldsymbol{d_1}$ is chosen to be parallel to the boundary edge. This choice fixes the value of θ. θ can be obtained by solving the following linear system of equations. $\boldsymbol{v}$ is a vector parallel to the boundary edge.

$$\begin{bmatrix}\boldsymbol{s_1} & \boldsymbol{s_2}\end{bmatrix}\begin{bmatrix}c_1\\c_2\end{bmatrix} = \begin{bmatrix}v_x\\v_y\end{bmatrix} \tag{3.14}$$

$$\theta = \operatorname{atan2}(c_2, c_1) \tag{3.15}$$

Once the boundary value of θ is known, the method developed in the previous section can be employed. Solving the two Laplace equations determines the value of θ inside the whole domain.

Figs. 3.7 and 3.8 show examples of cardinal direction fields over three-dimensional surfaces. The cardinal direction fields are smooth and parallel to the boundary normals.

However, as seen from Fig. 3.9, the cardinal direction field can sometimes differ from what is expected. In Fig. 3.9, the local directions should have been either parallel or perpendicular to the cylinder axis. The cylinder is composed of a single face making a complete loop.

The complex number z was defined in equation 3.2. As shown on Fig. 3.10, the modulus of z is not constant over the cylinder. It becomes especially close to zero near the middle region. When the components of z, a and b, are very small, the finite element error has a large influence on the final value of θ.

Local cardinal directions are composed of four vectors. At each point of the cylinder surface, one of these four vectors has a minimum angle with respect to the cylinder axis. The average of these angles over the

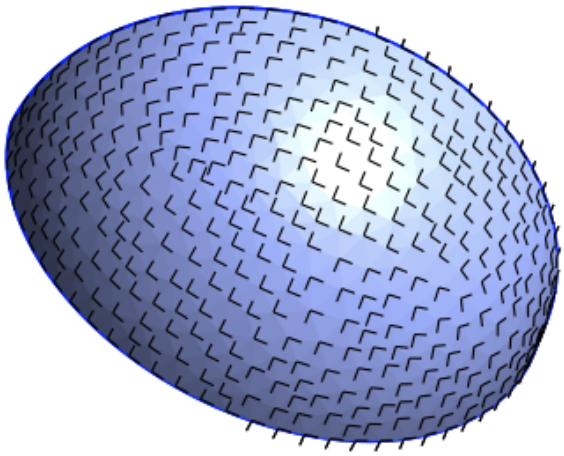


Figure 3.7: The cardinal direction field over a quarter sphere

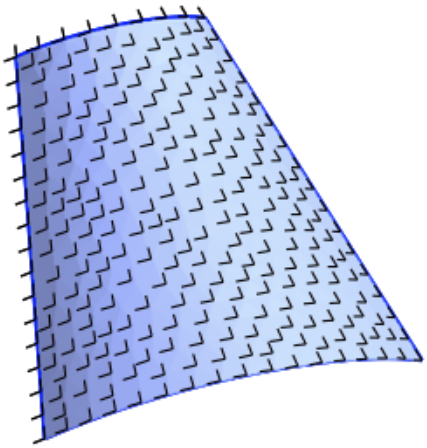


Figure 3.8: The cardinal direction field over a sheet

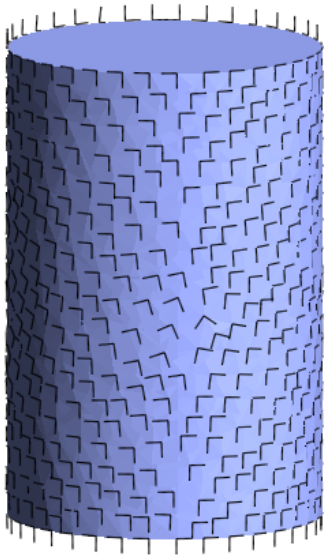


Figure 3.9: The cardinal direction field over a cylinder. The local directions are not always parallel or perpendicular to the cylinder axis.

whole surface can be computed. Table 3.1 contains the value of the average angle along with the mesh size. Diminishing the mesh size leads to a decrease in the finite element error, which in turn often leads to a decrease in the average angle. A small average angle indicates an almost correct orientation. A mesh size of 0.2 was employed for Fig. 3.9.

Mesh size	Angle (◦)
0.2	11.03
0.175	7.14
0.15	6.12
0.125	6.57
0.1	2.85
0.075	2.75
0.05	1.69

Table 3.1: Average angle to the cylinder axis for various mesh sizes

Fig. 3.11 shows the cardinal direction field on a mechanical part composed of several faces.

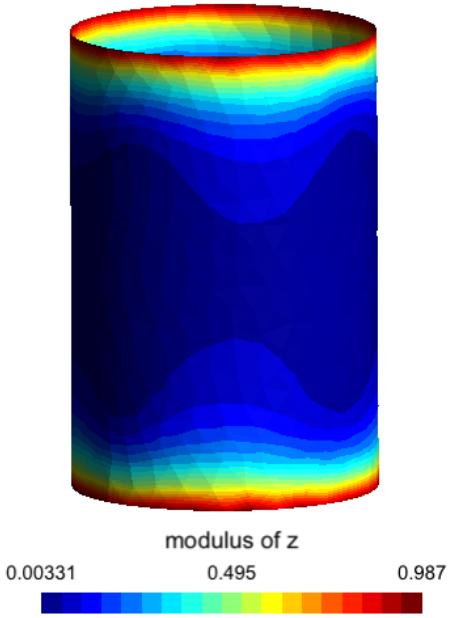


Figure 3.10: An isosurface plot of the modulus of $z(\boldsymbol{x}) = e^{i4\theta(\boldsymbol{x})}$. The modulus is much smaller than 1 at mid-height.

A singularity is a point where the cardinal directions are not well defined. Fig. 3.12 (1) and (2) shows two examples of singularities. The singularity of the first picture has an index of $\frac{1}{4}$, because the local cardinal directions rotate of $+\frac{\pi}{2}$ when a complete turn is performed around the red circle in the counterclockwise direction [60]. $+\frac{\pi}{2}$ corresponds to a quarter of a full rotation. One the contrary, the singularity of the second picture has an index of $-\frac{1}{4}$, because the local cardinal directions rotate of $-\frac{\pi}{2}$ when a complete turn is performed around the red circle in the counterclockwise direction [60].

Let g be the number of handles possessed by a three-dimensional closed surface [60]. g is also known as the genus [71]. For example, a sphere has zero handles, but a torus has one [71]. The Euler characteristic χ is equal to $2 - 2g$ [60]. A sphere has a χ of 2 and a torus has a χ of 0. By definition, all topologically equivalent closed surfaces have the same number of handles [81]. Consequently, they have all the same Euler characteristic [29, 34]. The Poincaré-Hopf theorem states that for any cardinal direction field defined on a closed surface, the sum of the

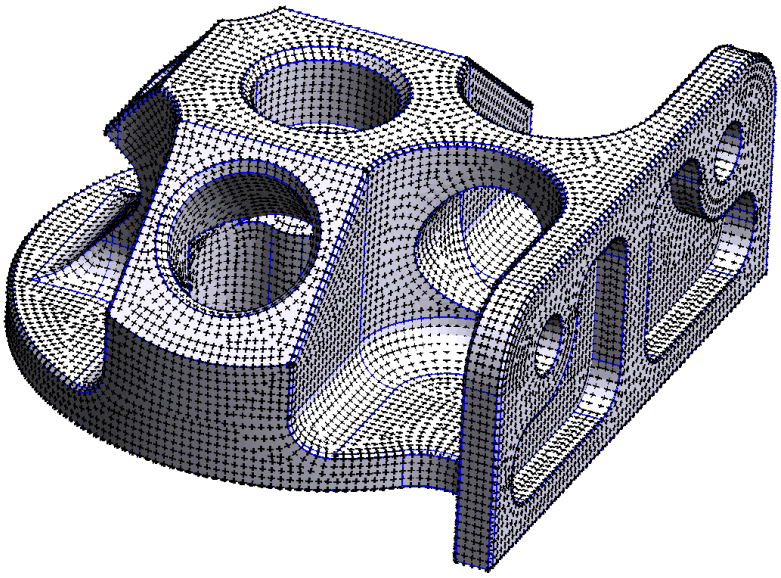
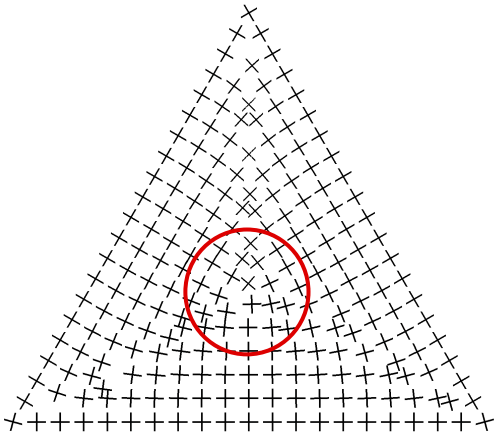


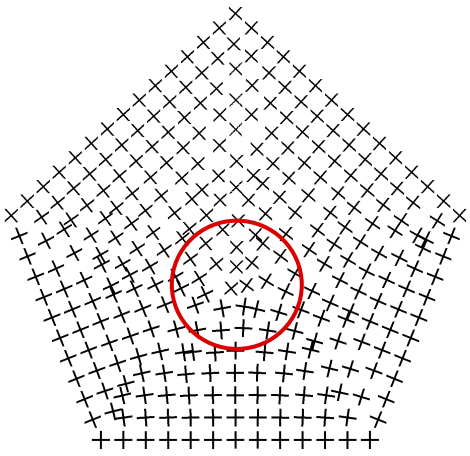
Figure 3.11: The cardinal direction field on a mechanical part (credits: Jean-François Remacle)

indices of the singularities must be equal to the Euler characteristic [60]. As said before, a closed surface topologically equivalent to a sphere has a Euler characteristic of 2. Therefore, a cardinal direction field defined on this surface will absolutely have to contain singularities, such as four $\frac{1}{4}$ singularities.

In a perfect quadrilateral mesh, each vertex is connected to four other vertices. In other words, it has a valence of four. A vertex is said to be irregular when its valence is different than four. It has been demonstrated in the literature that a perfect quadrilateral mesh cannot exist on closed surfaces topologically equivalent to spheres [60]. Such quadrilateral meshes will have to contain irregular vertices. However, toruses allow perfect quadrilateral meshes. This result is intuitively consistent with the Poincaré-Hopf theorem discussed above.



(1)



(2)

Figure 3.12: The singularity of each cardinal direction field is enclosed by a red circle

3.3 Three-dimensional space

The cardinal directions at a precise point inside a given volume are chosen to be equal to the cardinal directions of the nearest surfacic mesh vertex [45]. ANN nearest neighbor library is employed for the queries [45, 55]. By using this method, the cardinal directions near the boundaries are guaranteed to be parallel to the surface normals. However, for many geometries, the cardinal direction field is also going to be discontinuous. Evidently, a smooth cardinal direction field would have been preferable. Nevertheless, extending the method based on Laplace equations to the three dimensions would be very difficult.

Figs. 3.13 and 3.14 present two examples of cardinal direction fields inside volumes. The cardinal direction fields are not smooth and contain discontinuities. Discontinuities are region where the orientation changes abruptly.

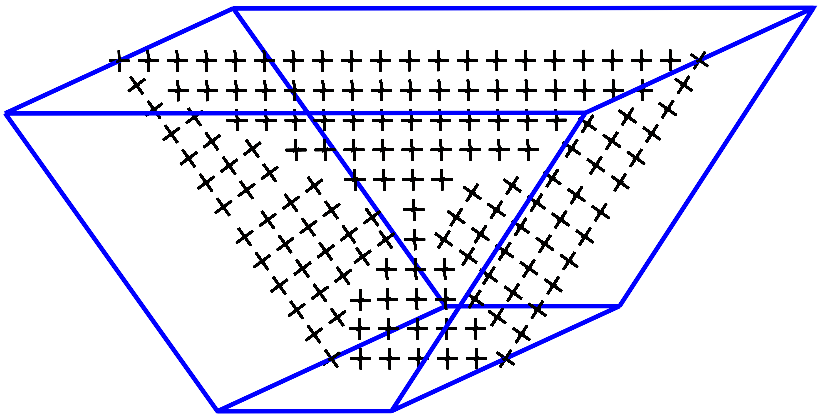


Figure 3.13: A slice of the cardinal direction field inside a trapezoidal prism

Recently, a technique capable of generating smooth cardinal direction fields in three dimensions has been developed [35]. Initial cardinal directions are first assigned to the volumic points. The cardinal directions are expressed as rotation matrices. A rotation matrix can be combined with another rotation of $k\frac{\pi}{2}$, $k \in \mathbb{Z}$ around one of its columns without altering the local cardinal directions. For each volumic point, all of these rotations are grouped into a set. A function $f = h(R^T \boldsymbol{s})$ is then introduced, with $\boldsymbol{s}$ being a vector [35]. When $\boldsymbol{s}$ is kept constant, this function returns the same value for all of the matrices belonging to

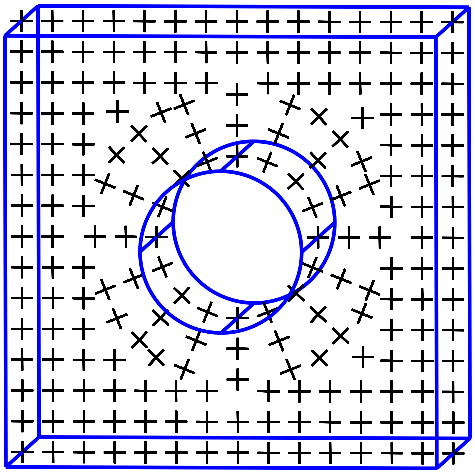


Figure 3.14: A slice of the cardinal direction field inside an hollowed cube

a given set. A sphere can be described by keeping the norm of the vector $\boldsymbol{s}$ constant and allowing one of its extremities to move freely. Over this sphere, the function f reaches its maximum value at the eight corner of a circumscribed cube whose orientation is determined by the matrix R. The function f is subsequently expressed as a sum of spherical harmonics [35]. The coefficients of the spherical harmonics depend on the matrix R. The last step consists of minimizing the Dirichlet's energy of the spherical harmonic coefficients with a limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) optimization procedure. The overall technique results in a smoother cardinal direction field.

A few solutions have been proposed regarding the problem of generating cardinal direction fields over surfaces and inside volumes. The next chapter will present a two-dimensional iterative algorithm capable of aligning the mesh vertices in accordance with these cardinal direction fields.

Chapter 4

Lloyd's energy minimization in two dimensions

Chapter 2 previously discussed the importance of aligned vertices distributions for quadrilateral mesh generation. Aligned vertices were shown to lead to higher quality quadrilaterals. Two potential approaches were proposed: the iterative one and the frontal one. This chapter describes a two-dimensional iterative method. By employing conformal parametrization techniques, three-dimensional surfaces can also be treated [50].

The Voronoi cell of a vertex v is the area composed of points closer to v than to v', with v' being any other vertex. Fig. 4.1 shows an example of Voronoi cell.

Lloyd's algorithm comprises two steps. The first step consists of moving each vertex to the centroid of its Voronoi cell [21]. The second step is to reconstruct the Voronoi diagram, which is necessary because the vertices positions have changed. These two steps are repeated a certain number of times. The whole process converges toward a centroidal Voronoi tessellation (CVT) [21]. By definition, the vertices of a CVT coincide with the centroids. The final result is a more uniform vertices distribution with near hexagonal Voronoi cells [21].

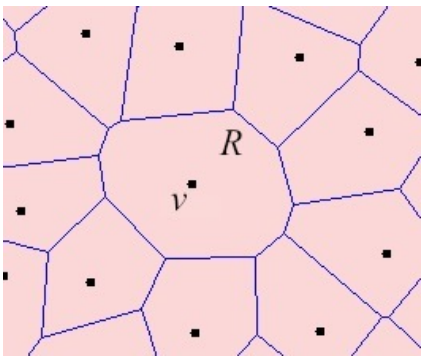


Figure 4.1: The region R bounded by the 10-sided polygon is the Voronoi cell of the vertex v. Any point inside of R is closer to v than to the other vertices represented by black dots. This diagram was created with the software available at [57].

In fact, Lloyd's algorithm is known to minimize an energy function equal to the sum of the moments of inertia of the Voronoi cells, with the vertices being the axes of rotation [47]. This energy function and its gradient can be computed. The limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS) optimization procedure can then be employed to minimize the energy function [47]. The LBFGS procedure is favored because it does not need the second derivatives. The resulting method has been shown to require less iterations than Lloyd's algorithm [47].

The standard moment of inertia involves a Euclidean distance. However, this Euclidean distance can be replaced by a higher-order L_p distance [45]. Equation 4.1 defines the L_p distance in two dimensions:

$$||\boldsymbol{y} - \boldsymbol{x}||_p = (|y_1 - x_1|^p + |y_2 - x_2|^p)^{\frac{1}{p}} \tag{4.1}$$

Various L_p distances are illustrated on Fig. 4.2. Each of these curves contains a set of points equidistant to the origin according to its particular L_p distance. The curves become more and more rectangular as p increases. For any p different from two, the L_p distance becomes anisotropic.

Lévy-Liu's algorithm seeks to align mesh vertices in precise cardinal directions by optimizing the Voronoi cells shapes. The cardinal direction fields can be generated by the method described in chapter 3. An energy function equal to the sum of the higher-order moments of inertia of the

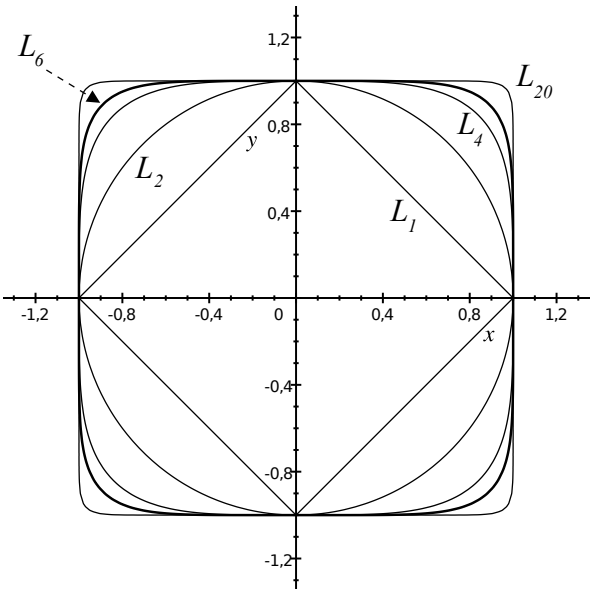


Figure 4.2: Unit circles in various L_p distances

Voronoi cells is minimized [45]. The L_p distances are only used for the energy function. The Voronoi cells are still constructed with the habitual Euclidean distance. Fig. 4.3 shows a Voronoi diagram before and after its optimization with different L_p distances. The L_2 distance leads to hexagonal Voronoi cells, while the L_6 distance leads to squarely Voronoi cells.

Fig. 4.4 presents another example of optimization with the L_6 distance. A p of 6 will be employed throughout this chapter.

As seen from Fig. 4.4, the Voronoi cells take a rectangular shape, which has the effect of aligning the mesh vertices.

Lévy-Liu's algorithm employs the LBFGS procedure to minimize the energy function [45]. As said before, LBFGS only requires the values of the energy and the gradient. However, the second derivative of the function still needs to be continuous. The L_2 energy function has already been proven to be C^2 continuous at certain conditions [47]. Nevertheless, there is currently no proofs available for the general L_p case.

The vertices of the triangular mesh shown on Fig. 4.5 (1) are not aligned. However, by applying Lévy-Liu's algorithm prior to the recombination step, good quality aligned quadrilaterals can be created, as seen on Fig. 4.5 (2).

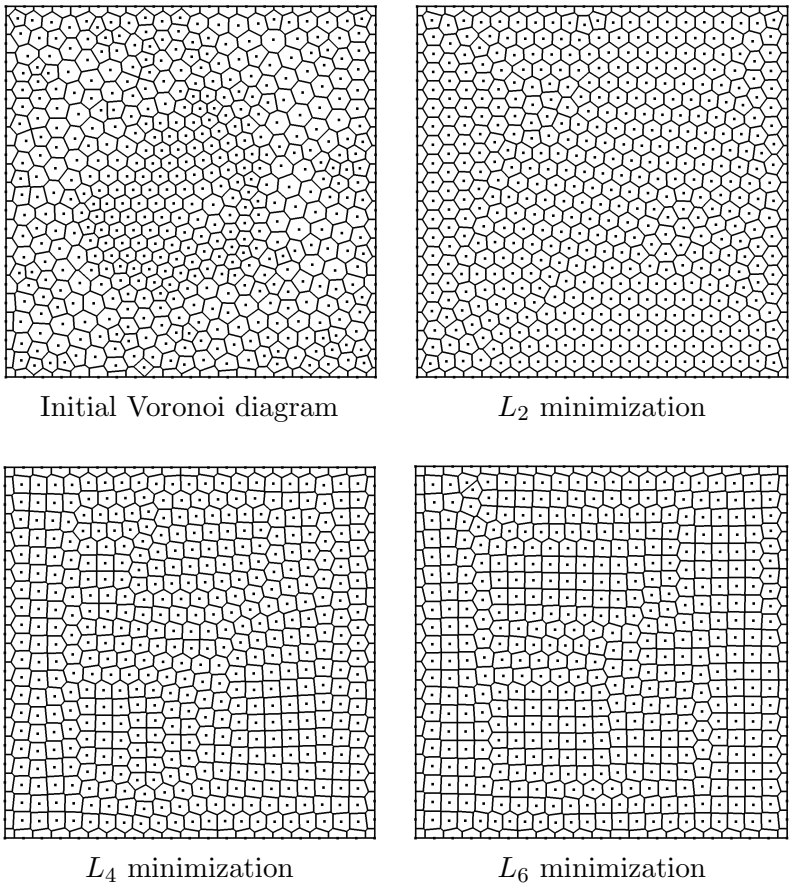
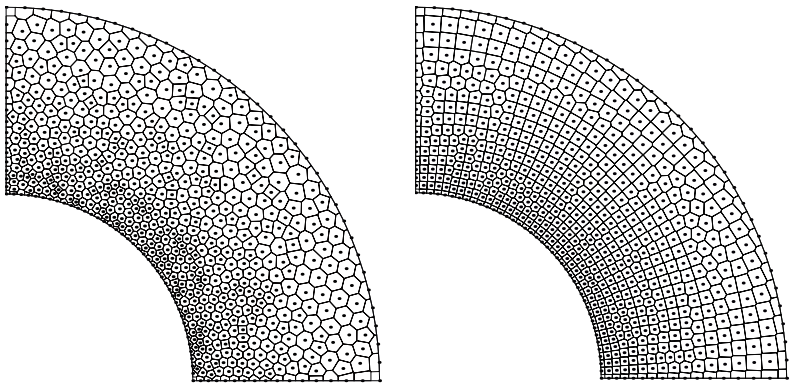


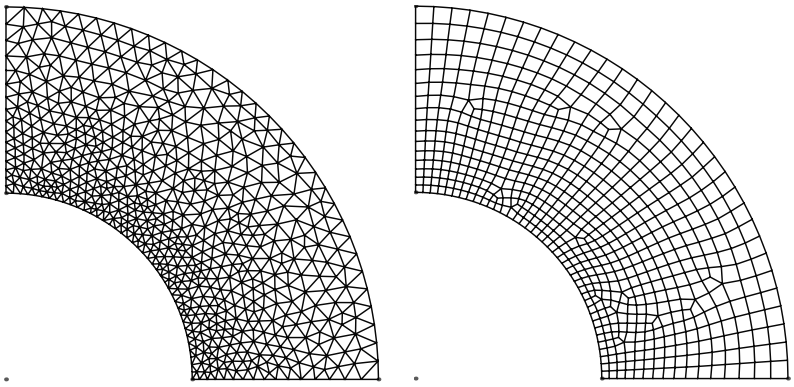
Figure 4.3: A Voronoi diagram optimized with different L_p distances

The Blossom-quad algorithm creates quadrilaterals by recombining triangles. Therefore, the mesh vertices are always re-triangulated with a Delaunay method after Lévy-Liu's optimization. A two-dimensional implementation of the Lévy-Liu's algorithm adapted to the field of finite element computations will be presented in this chapter. The implementation does not modify the one-dimensional meshes discretizing the CAD model edges. It takes into input cardinal direction fields and mesh size fields in order to define the orientation and size of the quadrilaterals. By employing conformal parametrization techniques, the algorithm can also be applied to three-dimensional surfaces [50]. The energy and



(1) (2)

Figure 4.4: The Voronoi cells before (1) and after (2) Lévy-Liu's optimization with the L_6 distance



(1) (2)

Figure 4.5: The quadrilateral mesh (2) has been created by applying Lévy-Liu's and Blossom-quad algorithms to the triangular mesh (1).

the gradient are computed with well-known Gauss integration methods. The Alglib library [14] is used for the LBFGS optimization part.

4.1 Lévy-Liu's algorithm

The clipped Voronoi diagram is the part of the Voronoi diagram that is inside the domain, as shown on Fig. 4.4 (a) and (b). It is indispensable to the computation of accurate values for the energy and the gradient. The method used to generate clipped Voronoi diagrams is inspired from [90] and is explained in subsection 4.1.1. Subsection 4.1.2 shows how to compute the energy and its gradient. Subsection 4.1.3 discusses the various difficulties related to the optimization step.

4.1.1 Clipped Voronoi diagrams

Fig. 4.6 shows an example of a clipped Voronoi diagram.

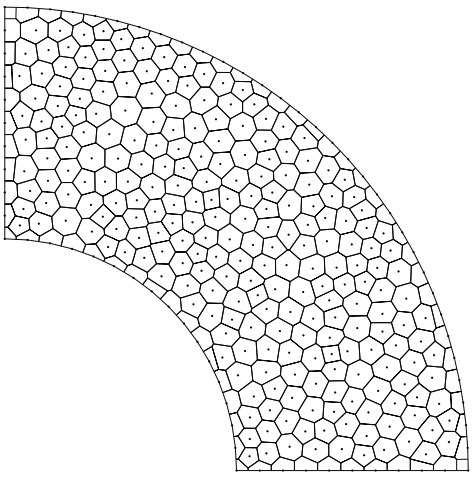


Figure 4.6: A clipped Voronoi diagram

Both boundary and interior vertices have Voronoi cells. Most of the time, it is only the Voronoi cells generated by boundary vertices that intersect the boundary. However, if an interior vertex is sufficiently close to a boundary, its Voronoi cell might intersect it, as shown on Fig. 4.7. This is unlikely, but the implementation needs to take this phenomenon into account.

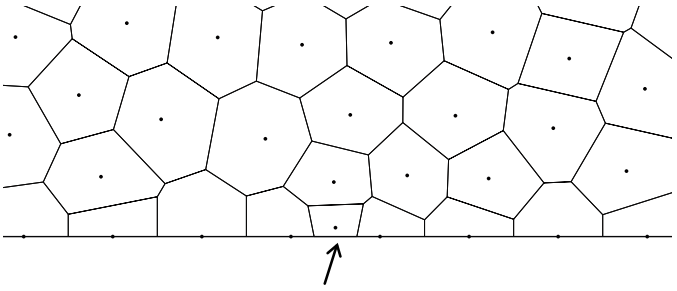


Figure 4.7: The arrow shows an interior Voronoi cell intersecting the boundary.

Subsection 4.1.1 gives some results about the convexity of the clipping polygons. Subsection 4.1.1 presents a method capable of identifying all Voronoi cells that intersect the boundary. Subsection 4.1.1 explains the global clipping algorithm.

Convexity of the clipping polygons

The Sutherland-Hodgman algorithm is a simple procedure that can *clip both concave and convex polygons against convex clipping polygons* [13]. Its steps are illustrated on Fig. 4.8. This algorithm could be useful for the computation of clipped Voronoi diagrams [90]. As required, the clipping polygon needs to be convex while the polygon being clipped can be either convex or concave. By construction, a Voronoi cell is always convex. The two-dimensional domain, however, is arbitrary and could be concave. Therefore, the logical choice would consist of clipping the domain with the Voronoi cells [90]. Nevertheless, it was the opposite that was implemented: the Voronoi cells are clipped by the boundary. In these conditions, the Sutherland-Hodgman algorithm can only be applied when the clipping polygons made of boundary line segments are locally convex. This approach is not ideal and would be too difficult to implement in three dimensions.

The Voronoi cell of an interior mesh vertex can sometimes intersect the boundary, as shown on Fig. 4.9. The Voronoi cell on the figure is clipped by a single line segment, which can be viewed as an half-plane. An half-plane is evidently a convex polygon. It is the only possible scenario. It is true that a concave clipping polygon could be built by placing a boundary mesh vertex inside the Voronoi cell. Nevertheless, it would

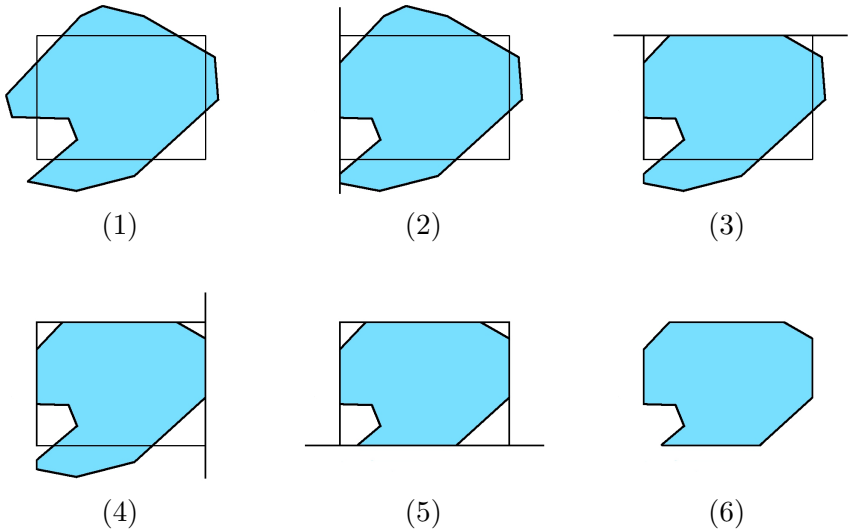


Figure 4.8: A six steps demonstration of the Sutherland-Hodgman algorithm

mean that there are more than one vertex inside a single Voronoi cell, which is impossible by definition. The Sutherland-Hodgman algorithm can therefore be applied to interior Voronoi cells.

The Voronoi cell of a boundary mesh vertex always intersects the two boundary line segments at its left and at its right, as seen on Figs. 4.6 and 4.7. These particular intersections cannot be treated by the Sutherland-Hodgman algorithm. However, they can be computed by cycling through the Delaunay elements incident to the boundary mesh vertex, as shown on Fig. 4.10.

A boundary Voronoi cell can also intersect supplementary line segments, such as segment #2 of Fig. 4.11. Line segment #2 is equivalent to a convex polygon. The Sutherland-Hodgman algorithm can be applied here as well.

Research by propagation

The Voronoi cells of interior mesh vertices can sometimes intersect the boundary. Nevertheless, checking all interior Voronoi cells for intersection with all boundary line segments would take too long. Fortunately, Yan and *al*. discovered an efficient way to perform this task [90].

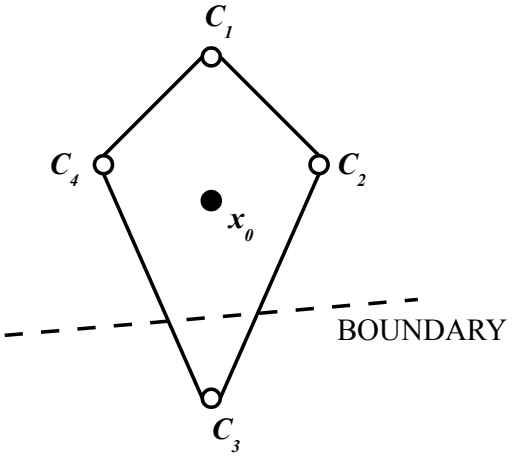


Figure 4.9: Interior Voronoi cell intersected by a line segment (throughout this chapter, the black dots are assumed to be mesh vertices and the hollow dots are assumed to be Voronoi vertices).

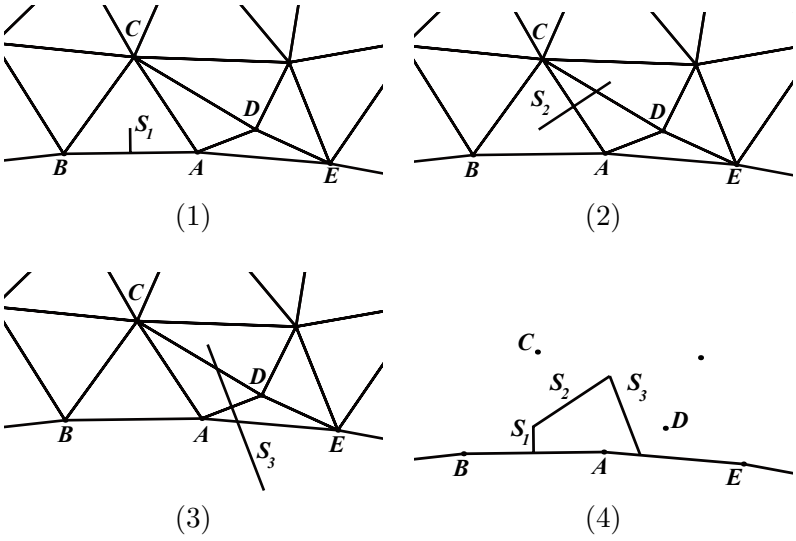


Figure 4.10: S_1, S_2 and S_3 are the bisectors of the Delaunay edges AB, AC and AD.

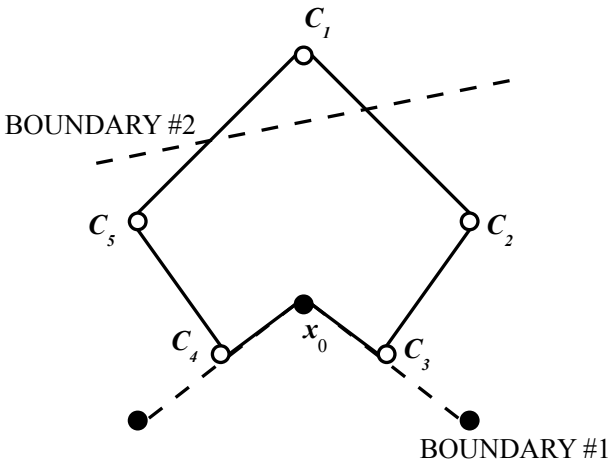


Figure 4.11: Boundary Voronoi cell intersected by a line segment

As seen on Fig. 4.12, the facet f of the Voronoi cell of mesh vertex $\boldsymbol{x}_1$ intersects the boundary line segment l. By definition, f is the orthogonal bisector of the Delaunay edge joining $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, $\boldsymbol{x}_2$ being another mesh vertex. f is also a facet of $\boldsymbol{x}_2$'s Voronoi cell. Therefore, $\boldsymbol{x}_2$'s Voronoi cell will intersect l. By starting from the Voronoi cells of the boundary mesh vertices, it is possible to identify all interior Voronoi cells that intersect the boundary while only verifying a small amount of them.

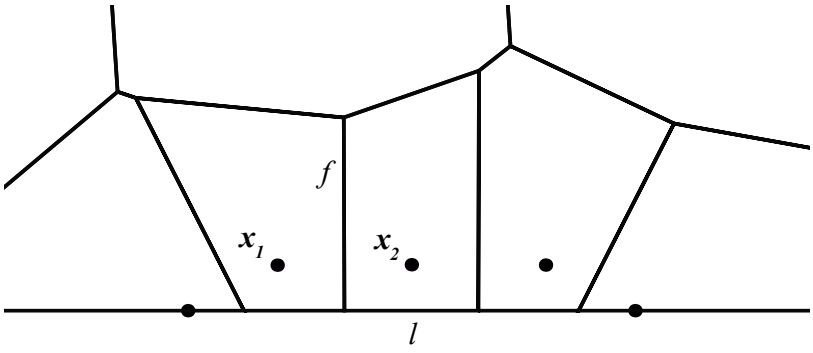


Figure 4.12: Three interior Voronoi cells intersecting the boundary

Voronoi diagram clipping algorithm

The four steps of the complete clipping algorithm can now be described. These steps are inspired from [90]. Two data structures will be used: a queue and an array of lists. The queue will contain the mesh vertices to be treated. The array will contain for each mesh vertex the list of boundary line segments that intersect its Voronoi cell.

1. The first step consists of creating the interior Voronoi cells, as shown in Fig. 4.13 (a). Each Voronoi cell is a polygon composed of Voronoi vertices. The Voronoi vertices are the centers of the circles circumscribing the Delaunay triangles. Some interior Voronoi cells might intersect the boundary, but this will be addressed later in steps 3 and 4.

2. There is a boundary line segment at the left and at the right of each boundary mesh vertex. The second step consists of clipping the boundary Voronoi cells with these line segments, as shown in Fig. 4.13 (b). Neighboring interior Voronoi cells could intersect the same boundary line segments. If it is the case, then the interior mesh vertex in question is pushed in the queue and the boundary line segment is added to its list.

3. The third step consists of identifying all the boundary line segments that intersect each Voronoi cell with the procedure described earlier. Both the queue and the array of lists are used for this step. The mesh vertices are popped one after the other until the queue is empty. Every time that a mesh vertex is popped, each facet of its Voronoi cell is tested for intersection with each boundary line segment in its list. If there is an intersection, then the other mesh vertex that shares the same facet is pushed in the queue, but only if its list did not originally contain the boundary line segment. This precaution is taken in order to ensure that the algorithm does not continue indefinitely.

4. The list of boundary line segments that intersect each Voronoi cell is now known. The last step consists of using the Sutherland-Hodgman algorithm to clip the Voronoi cells with the boundary line segments. Fig. 4.13c) shows the final clipped Voronoi diagram.

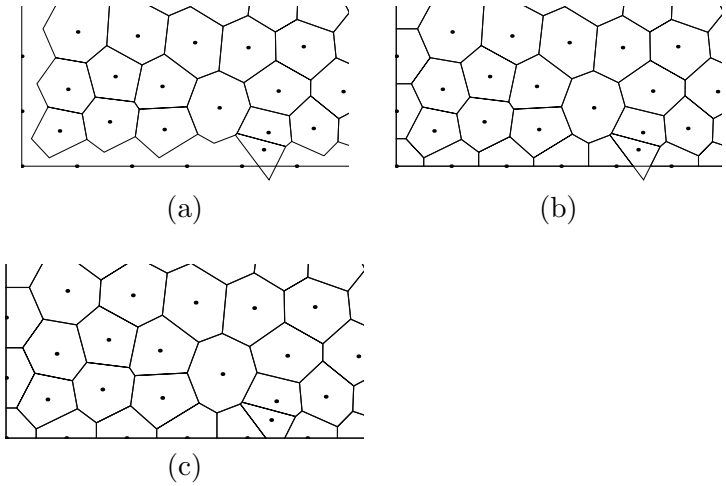


Figure 4.13: a) The unclipped Voronoi cells of the interior mesh vertices, b) The clipped Voronoi cells of the boundary mesh vertices, c) The final clipped Voronoi diagram

4.1.2 Energy and gradient evaluation

This subsection starts by introducing the energy function and the gradient. It then shows how the domain is divided into triangular elements. It finally explains how to compute the energy and the gradient with Gauss integration techniques.

The implementation described in this chapter can only find local energy minimums. In general, it cannot find global minimums even for simple geometries such as squares and rectangles. It can nevertheless align points in a very satisfactory manner.

For example, in the case of a 11 by 11 square mesh containing 81 interior vertices, the global minimum is readily identifiable. It is a perfect mesh composed of only right triangles. Fig. 4.14 illustrates four different initial triangular meshes and their corresponding optimized meshes after 100 iterations. As expected, the global minimum can be found for mesh A. Even though meshes B and C look similar, only mesh B can be optimized to the perfect solution. This simple example shows that the resulting mesh depends on the initial configuration. However, the alignment can be improved even in the case of low quality initial meshes.

The variable τ is used to measure how close to perfect right triangles the elements are:

$$\tau = \frac{1}{N}\sum_{i=1}^{N}\max_{j=1,2,3}(\text{abs}(\sin\theta_{ij})) \tag{4.2}$$

N is the total number of triangular elements in the mesh. θ_{i1} is the first interior angle of triangular element i, θ_{i2} is the second one and θ_{i3} is the third one. τ is equal to 1 for a mesh containing only perfect right triangles. It is equal to 0.866 for a mesh containing only equilateral triangles.

Chapter 9 contains the pseudo-code relevant to the energy and gradient calculations.

Energy function

Before defining the energy function, two parameters need to be introduced: M and ρ.

M is the change of basis matrix that goes from S to S'. It is a function of the angle θ.

$$M(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \tag{4.3}$$

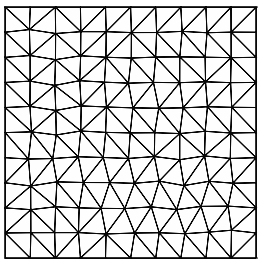
S' is the local coordinate system and S is the global one. As shown on Fig. 4.15, θ is the angle between S' and S. S' is locally aligned with the cardinal direction field. The cardinal direction field specifies the orientation of the quadrilaterals. Methods capable of generating cardinal direction fields were described in chapter 3.

The parameter ρ is the density. It specifies the size of the quads.

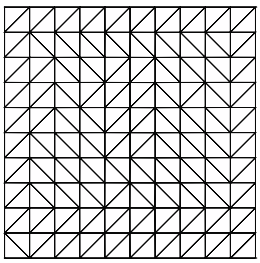
The energy function can now be defined [45] as:

$$F(\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N) = \sum_{i=1}^{N} \underbrace{\int_{R_i} \rho(\boldsymbol{y})||M(\boldsymbol{y}-\boldsymbol{x}_i)||_p^p d\boldsymbol{y}}_{I^{R_i}(\boldsymbol{x}_i)} \tag{4.4}$$

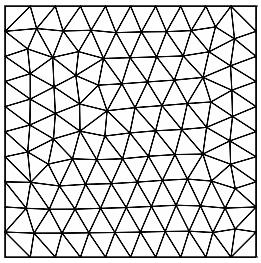
The index i runs over all N mesh vertices $\boldsymbol{x}_i$. R_i is the domain corresponding to the i-th Voronoi cell generated by mesh vertex $\boldsymbol{x}_i$. In what follows, the energy integral will be denoted by $I^{R_i}(\boldsymbol{x}_i)$. The Voronoi diagram is computed with the L_2 distance, because libraries that can create Voronoi diagrams in other L_p distances are not available. This situation may change in the future, with the publication in 2012 of an article about L_∞ Voronoi diagrams [56].



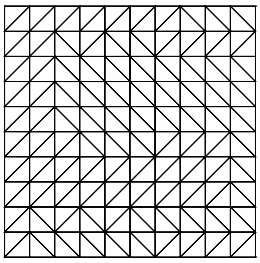
mesh A, $\tau = 0.989$



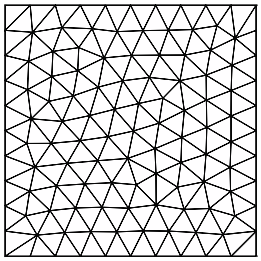
optimization of mesh A, $\tau = 0.999$



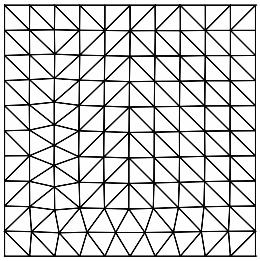
mesh B, $\tau = 0.937$



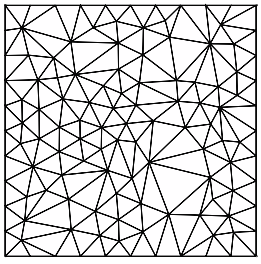
optimization of mesh B, $\tau = 0.999$



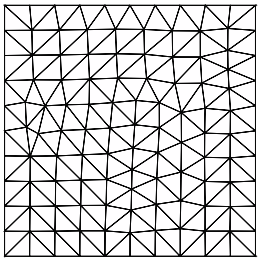
mesh C, $\tau = 0.926$



optimization of mesh C, $\tau = 0.990$



mesh D, $\tau = 0.948$



optimization of mesh D, $\tau = 0.980$

Figure 4.14: Meshes A, B, C and D all contain 81 interior vertices.

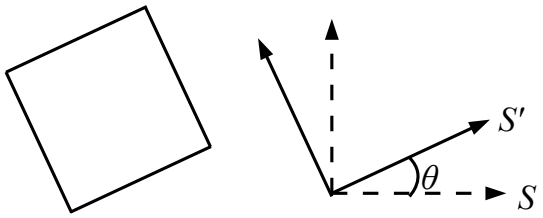


Figure 4.15: An oriented quadrilateral element

The integral $I^{R_i}(\boldsymbol{x}_i)$ is in the order of ρl^{p+2}, with l being the characteristic length of the Voronoi cell. For this derivation, the density ρ was assumed to be constant over R_i.

$$I^{R_i}(\boldsymbol{x}_i) = \int_{R_i} \rho ||M(\boldsymbol{y} - \boldsymbol{x}_i)||_p^p d\boldsymbol{y} \sim \int_{R_i} \rho l^p d\boldsymbol{y} \sim \rho l^{p+2} \tag{4.5}$$

The Gersho conjuncture states that for large distributions of vertices, minimal total energy translates into energy equidistribution between Voronoi cells [49]. Therefore, all the integrals $I^{R_i}(\boldsymbol{x}_i)$ should have approximately the same value when the characteristic length l corresponds to the local mesh size h [22]. This requirement leads to a relationship between ρ and h. More precisely, ρ needs to be proportional to $\frac{1}{h^{p+2}}$. A similar reasoning exists in the literature [22].

$$\rho \propto \frac{1}{h^{p+2}} \tag{4.6}$$

A constant of 1 was chosen, making ρ equal to $\frac{1}{h^{p+2}}$ for the remaining part of this chapter.

The user fixes the mesh size h at the geometrical vertices of the surface through an input file. A smooth and gradual one-dimensional mesh size field can then be automatically calculated along the geometrical edges of the surface. Moreover, because h is known at the boundary, Dirichlet conditions can be imposed. The mesh size field inside the domain is assumed to obey a Laplace equation. Finite elements of the first order are employed for the resolution.

Energy gradient

The energy gradient is the total derivative of F with respect to the position of each non-boundary mesh vertex. Boundary vertices are considered to be unmovable. Displacing edge vertices would modify the

initial geometry, which is undesirable in a context of finite element analysis. It is assumed that $\boldsymbol{x}_1$, $\boldsymbol{x}_2$,..., $\boldsymbol{x}_n$ are non-boundary vertices and that $\boldsymbol{x}_{n+1}$, $\boldsymbol{x}_{n+2}$,..., $\boldsymbol{x}_N$ are boundary vertices. The energy gradient is given by:

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{x}_k}F(\boldsymbol{x}_1,\boldsymbol{x}_2,...,\boldsymbol{x}_N)=\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{x}_k}\sum_{i=1}^{N}I^{R_i}(\boldsymbol{x_i})=\sum_{i=1}^{N}\frac{\mathrm{d}I^{R_i}(\boldsymbol{x_i})}{\mathrm{d}\boldsymbol{x}_k}\quad k=1,...,n \tag{4.7}$$

The total derivative on the right hand side can be rewritten in terms of partial derivatives.

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{x}_k}F(\boldsymbol{x}_1,\boldsymbol{x}_2,...,\boldsymbol{x}_N)=\sum_{i=1}^{N}\frac{\partial I^{R_i}(\boldsymbol{x_i})}{\partial\boldsymbol{x}_k}+\sum_{i=1}^{N}\frac{\partial I^{R_i}(\boldsymbol{x_i})}{\partial R_i}\frac{\mathrm{d}R_i}{\mathrm{d}\boldsymbol{x}_k} \tag{4.8}$$

The total derivative symbol takes into consideration the fact that the Voronoi cells shapes change when the mesh vertex $\boldsymbol{x_i}$ is displaced infinitesimally. On the other hand, the partial derivative symbol assumes that the Voronoi cells are undeformable, even when the mesh vertex $\boldsymbol{x_i}$ moves. The total derivative does not consider that the position of one mesh vertex could be a function of the position of another mesh vertex.

Equation (4.8) can be further simplified. First, the partial derivative of $I^{R_i}(\boldsymbol{x_i})$ with respect to $\boldsymbol{x}_k$ is non-null only when $i=k$. Secondly, R_i can be expanded in terms of the Voronoi vertices $\boldsymbol{C}_{i1}$, $\boldsymbol{C}_{i2}$,...,$\boldsymbol{C}_{iM_i}$ of the i-th Voronoi cell. All Voronoi cells do not have the same number of vertices M_i. Fig. 4.16 shows a Voronoi cell with six Voronoi vertices. Equation (4.8) can be rewritten as [45]:

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{x}_k}F(\boldsymbol{x}_1,\boldsymbol{x}_2,...,\boldsymbol{x}_N)=\frac{\partial I^{R_k}(\boldsymbol{x_k})}{\partial\boldsymbol{x}_k}+\sum_{i=1}^{N}\sum_{j=1}^{M_i}\frac{\partial I^{R_i}(\boldsymbol{x_i})}{\partial\boldsymbol{C}_{ij}}\frac{\mathrm{d}\boldsymbol{C}_{ij}}{\mathrm{d}\boldsymbol{x}_k} \tag{4.9}$$

Most of the terms $\frac{\mathrm{d}\boldsymbol{C}_{ij}}{\mathrm{d}\boldsymbol{x}_k}$ are null, as explained in the next section.

Gradient assembly

In order to evaluate the integrals with Gauss techniques, the Voronoi cells are divided into triangular elements. Each triangular element is composed of a mesh vertex and two successive Voronoi vertices, as shown in Fig. 4.16.

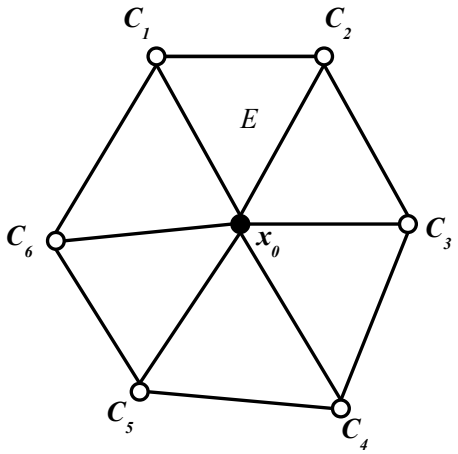


Figure 4.16: A Voronoi cell divided into six triangular elements

Most of the terms inside the double summation found in equation (4.9) vanish. By considering the relationship between the mesh vertices and the Voronoi vertices, the relevant contributions can be identified. Each Voronoi vertex falls into one of three categories.

1. A Voronoi vertex can be the center of the circle circumscribing a Delaunay element. Voronoi vertex $\boldsymbol{C}_1$ from Fig. 4.17 belongs to this category. As long as the displacements are infinitesimal, $\boldsymbol{C}_1$ depends only on $\boldsymbol{x}_0$, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. In Fig. 4.17 to 4.19, the dotted segments are Delaunay edges.

2. A Voronoi vertex can be the intersection point between a Voronoi facet and a boundary line segment. Voronoi vertex $\boldsymbol{C}_2$ from Fig. 4.18 belongs to this category. Again, as long as the displacements are infinitesimal, $\boldsymbol{C}_2$ depends only on $\boldsymbol{x}_0$, $\boldsymbol{x}_2$ and the boundary line segment.

3. A Voronoi vertex can be the median point between two boundary mesh vertices. Voronoi vertex $\boldsymbol{C}_2$ from Fig. 4.19 belongs to this category. It depends only on $\boldsymbol{x}_0$ and $\boldsymbol{x}_2$.

The following formulas were derived from equation (4.9). They state the contribution to the energy gradient of the element E found in Fig. 4.17. It is assumed that $\boldsymbol{x_0}$, $\boldsymbol{x_1}$, $\boldsymbol{x_2}$ and $\boldsymbol{x_3}$ are non-boundary mesh vertices. A += sign is used because there will be contributions from other elements as well.

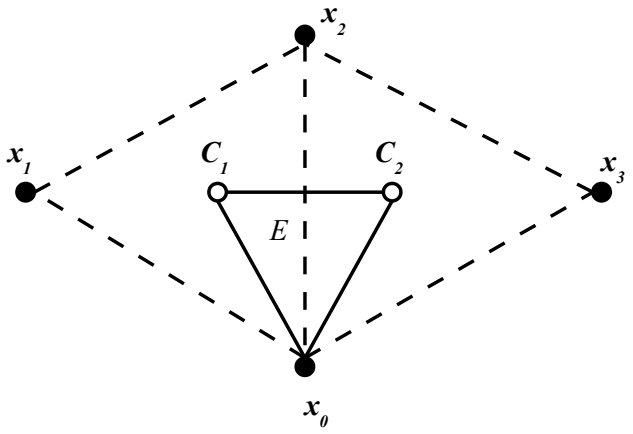


Figure 4.17: $\boldsymbol{C}_1$ is the center of the circle circumscribing the Delaunay element $\boldsymbol{x}_0$-$\boldsymbol{x}_1$-$\boldsymbol{x}_2$.

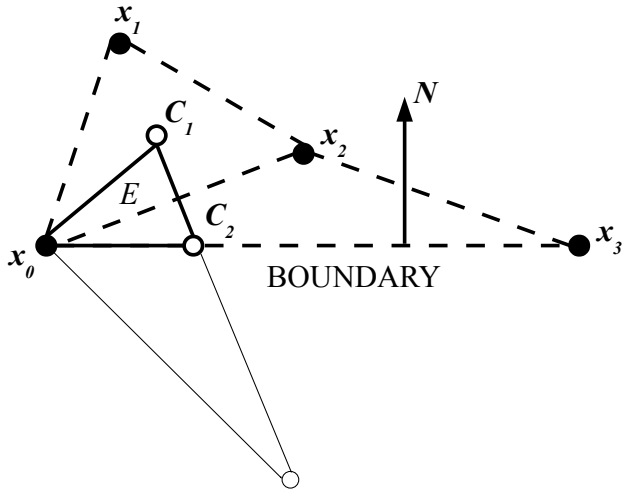


Figure 4.18: $\boldsymbol{C}_2$ is the intersection point between the bisector of the Delaunay edge $\boldsymbol{x}_0$-$\boldsymbol{x}_2$ and the boundary line segment $\boldsymbol{x}_0$-$\boldsymbol{x}_3$.

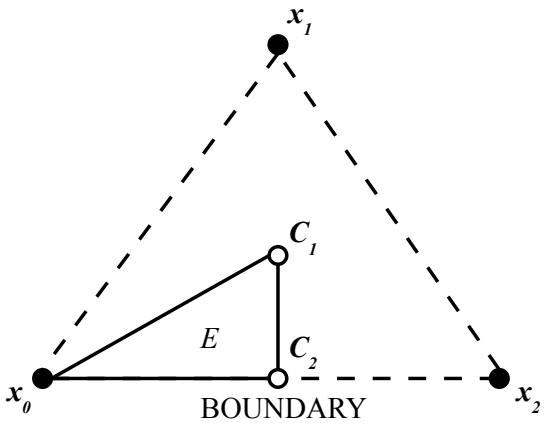


Figure 4.19: The unclipped Voronoi cell of $\boldsymbol{x}_0$ has a semi-infinite Voronoi facet that starts at $\boldsymbol{C}_1$ and bisects the Delaunay edge $\boldsymbol{x}_0$-$\boldsymbol{x}_2$. The median point between $\boldsymbol{x}_0$ and $\boldsymbol{x}_2$ is therefore a Voronoi vertex of the clipped Voronoi cell.

$$\frac{\mathrm{d}F}{\mathrm{d}\boldsymbol{x}_0} += \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{x}_0} + \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_1}\frac{\mathrm{d}\boldsymbol{C}_1}{\mathrm{d}\boldsymbol{x}_0} + \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_2}\frac{\mathrm{d}\boldsymbol{C}_2}{\mathrm{d}\boldsymbol{x}_0} \tag{4.10}$$

$$\frac{\mathrm{d}F}{\mathrm{d}\boldsymbol{x}_1} += \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_1}\frac{\mathrm{d}\boldsymbol{C}_1}{\mathrm{d}\boldsymbol{x}_1} \tag{4.11}$$

$$\frac{\mathrm{d}F}{\mathrm{d}\boldsymbol{x}_2} += \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_1}\frac{\mathrm{d}\boldsymbol{C}_1}{\mathrm{d}\boldsymbol{x}_2} + \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_2}\frac{\mathrm{d}\boldsymbol{C}_2}{\mathrm{d}\boldsymbol{x}_2} \tag{4.12}$$

$$\frac{\mathrm{d}F}{\mathrm{d}\boldsymbol{x}_3} += \frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_2}\frac{\mathrm{d}\boldsymbol{C}_2}{\mathrm{d}\boldsymbol{x}_3} \tag{4.13}$$

Gauss integration

The terms inside equations (4.10) to (4.13) are integrals on the element E. A linear transformation T will be used to go from the reference triangle E' to the triangle E, as shown on Fig. 4.20. It will then be possible to evaluate the various integrals with Gauss techniques.

$$\boldsymbol{y} = \boldsymbol{T}(u,v) = \boldsymbol{x}_0(1-u-v) + \boldsymbol{C}_1 u + \boldsymbol{C}_2 v \tag{4.14}$$

16 Gauss points are used. As a result, polynomials up to the order 8 can be integrated exactly.

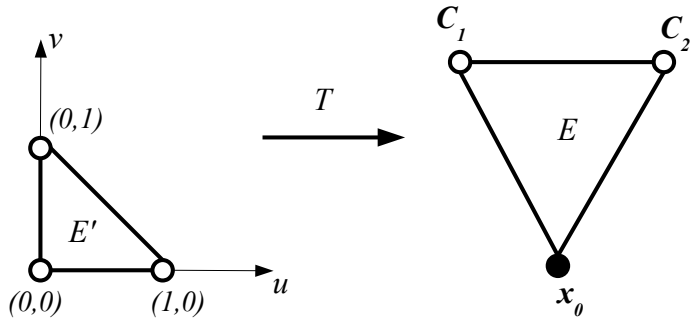


Figure 4.20: The linear transformation $\boldsymbol{T}$

Equation (4.4) defines $I^E(\boldsymbol{x_0})$ as the energy contribution of the element E. This equation can be rewritten in terms of the reference element E', where J is the Jacobian of the transformation T.

$$I^E(\boldsymbol{x_0}) = \int_{E'} \rho(\boldsymbol{T}(u,v))||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p J du dv \tag{4.15}$$

Using equation (4.15), the partial derivative of $I^E(\boldsymbol{x_0})$ with respect to the position of mesh vertex $\boldsymbol{x}_0$ can be written as:

$$\frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{x}_0} = \frac{\partial}{\partial \boldsymbol{x}_0} \int_{E'} \rho(\boldsymbol{T}(u,v))||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p J du dv \tag{4.16}$$

$$= \int_{E'} \rho(\boldsymbol{T}(u,v)) \frac{\partial ||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p}{\partial \boldsymbol{x}_0} J du dv$$

The partial derivative of $I^E(\boldsymbol{x_0})$ with respect to the position of Voronoi vertex $\boldsymbol{C}_1$ is given by:

$$\frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_1} = \frac{\partial}{\partial \boldsymbol{C}_1} \int_{E'} \rho(\boldsymbol{T}(u,v)) ||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p J du dv \tag{4.17}$$

$$= \int_{E'} \frac{\partial \rho(\boldsymbol{T}(u,v))}{\partial \boldsymbol{T}(u,v)} \frac{\mathrm{d}\boldsymbol{T}(u,v)}{\mathrm{d}\boldsymbol{C}_1} ||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p J +$$

$$\rho(\boldsymbol{T}(u,v)) \frac{\partial ||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p}{\partial \boldsymbol{T}(u,v)} \frac{\mathrm{d}\boldsymbol{T}(u,v)}{\mathrm{d}\boldsymbol{C}_1} J +$$

$$\rho(\boldsymbol{T}(u,v)) ||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p \frac{\partial J}{\partial \boldsymbol{C}_1} du dv$$

The Voronoi cells are divided into triangular elements. The mesh size is known at each element node and linear interpolation is used to determine the mesh size at a precise point. Equation 4.6 is employed to obtain the density from the mesh size. The orientation angle is considered to be constant by element.

A Voronoi vertex can depend on three mesh vertices, as shown on Fig. 4.17. The following matrix is the derivative of Voronoi vertex $\boldsymbol{C}_1$ with respect to mesh vertex $\boldsymbol{x}_0$ [45]. The derivatives of $\boldsymbol{C}_1$ with respect to mesh vertices $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ can be obtained by replacing $\boldsymbol{x}_0$ with $\boldsymbol{x}_1$ or $\boldsymbol{x}_2$.

$$\frac{\mathrm{d}\boldsymbol{C}_1}{\mathrm{d}\boldsymbol{x}_0} = \begin{bmatrix} (\boldsymbol{x}_1 - \boldsymbol{x}_0)^T \\ (\boldsymbol{x}_2 - \boldsymbol{x}_0)^T \end{bmatrix}^{-1} \begin{bmatrix} (\boldsymbol{C}_1 - \boldsymbol{x}_0)^T \\ (\boldsymbol{C}_1 - \boldsymbol{x}_0)^T \end{bmatrix} \tag{4.18}$$

A Voronoi vertex can also depend on two mesh vertices and one boundary line segment, as shown on Fig. 4.18. The following matrix needs to be used in this situation [45]. The derivative of $\boldsymbol{C}_1$ with respect to mesh vertex $\boldsymbol{x}_1$ can be obtained by replacing $\boldsymbol{x}_0$ with $\boldsymbol{x}_1$. $\boldsymbol{N}$ is the normal vector to the boundary line segment. It can be multiplied by any non-zero constant without affecting the value of the derivative.

$$\frac{\mathrm{d}\boldsymbol{C}_1}{\mathrm{d}\boldsymbol{x}_0} = \begin{bmatrix} (\boldsymbol{x}_1 - \boldsymbol{x}_0)^T \\ \boldsymbol{N}^T \end{bmatrix}^{-1} \begin{bmatrix} (\boldsymbol{C}_1 - \boldsymbol{x}_0)^T \\ \boldsymbol{0} \end{bmatrix} \tag{4.19}$$

The derivations of these matrices can be found in Lévy and Liu's article [45]. It is only the derivatives with respect to non-boundary mesh vertices that need to be calculated.

The inverse of the Jacobian is present in the derivative of the density with respect to the position. Flattened elements can therefore lead to indefinite values, which in turn can prevent the LBFGS library from functioning correctly. When the mesh vertices are very well aligned, the Voronoi cells become square-like. However, these squares can still have very small faces at the corner. These small faces will translate into slim elements. The issue can be solved by removing the elements that are too slim. Taking the ratio of the shortest edge to the longest one is a good criteria [27]. While this approach should not affect the total energy value, it may lead to incorrect gradient values.

4.1.3 LBFGS optimization

There are many potential issues with the LBFGS optimization step. In this section, the various problems that can be encountered are discussed.

The problem of points leaving domain

The LBFGS library performs line searches in order to find new lower energy solutions. There is no guarantees that all solutions are going to be restrained to the domain. A mesh vertex could leave the domain, which would cause the code to fail. To prevent this from happening, it is absolutely necessary to check that all mesh vertices are inside the domain before clipping the Voronoi diagram or computing the energy and the gradient. If there is a single vertex that falls outside the domain, the implementation should not try to perform any calculations. Instead, it should automatically set the energy to a very large value, such as 10^9. This will ensure that the LBFGS library does not select this particular solution. The library will continue the line search and will eventually find a lower energy solution with all mesh vertices inside the domain. An octree data structure can be used for efficient spatial searches.

Gradient normalization

The L_6 distance is used in practice. However, even thought a high exponent is necessary in order to align the vertices in a rectangular way, it can also cause numerical problems. If the density ρ is missing from the formulation, the gradient can reach values as low as 10^{-20}. These values are too close to zero and the optimization library will be unable to find acceptable solutions. The addition of a density to the energy function solves the problem. As seen in equation (4.6), the density is

proportional to $h^{-(p+2)}$ and will tend to normalize the energy and the gradient. Both will be closer to one and the optimization library will work properly.

Convergence of the optimization procedure

The energy will get lower and lower after each iteration. Fig. 4.21 shows the convergence curve of a car hood problem. The curve is very steep at the beginning, but nearly horizontal at the end. It is not necessary to reach complete convergence in order to obtain a good solution. Eighty iterations are usually sufficient. Sometimes, instead of converging, a solution can begin to oscillate. This usually happens only after an important decrease in energy. The result is going to be acceptable even in these conditions.

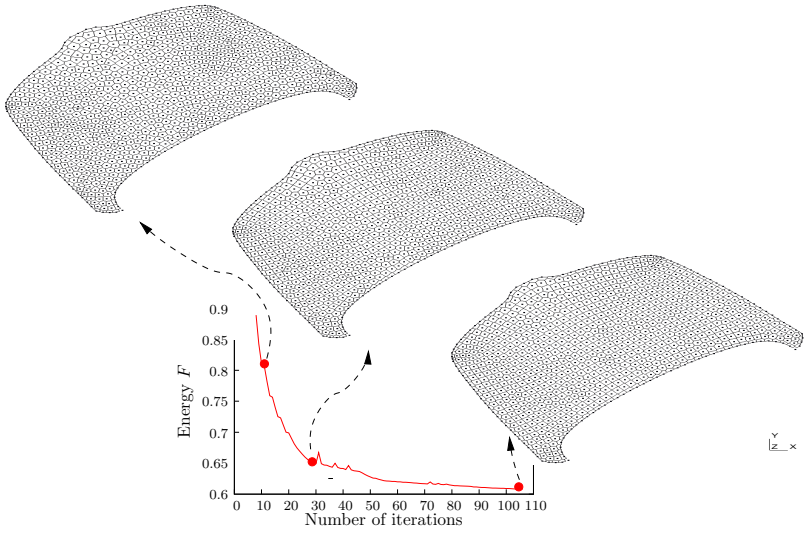


Figure 4.21: A convergence curve (credits: Emilie Marchandise). At each iteration, the clipped Voronoi diagram, the energy and the gradient need to be computed.

4.2 Results

In this section, different examples of high quality quadrilateral surface meshes with well defined orientations are presented.

Five steps are necessary to produce quadrilateral meshes:

1. Map conformally the three-dimensional surface to a two-dimensional parametric space [50].
2. Create a triangular mesh in the parametric space.
3. Use this mesh to compute a cardinal direction field with the method described in chapter 3.
4. Use Lévy-Liu's algorithm to optimize the location of the mesh vertices.
5. Apply the Blossom-quad algorithm to combine the triangles into quadrilaterals.

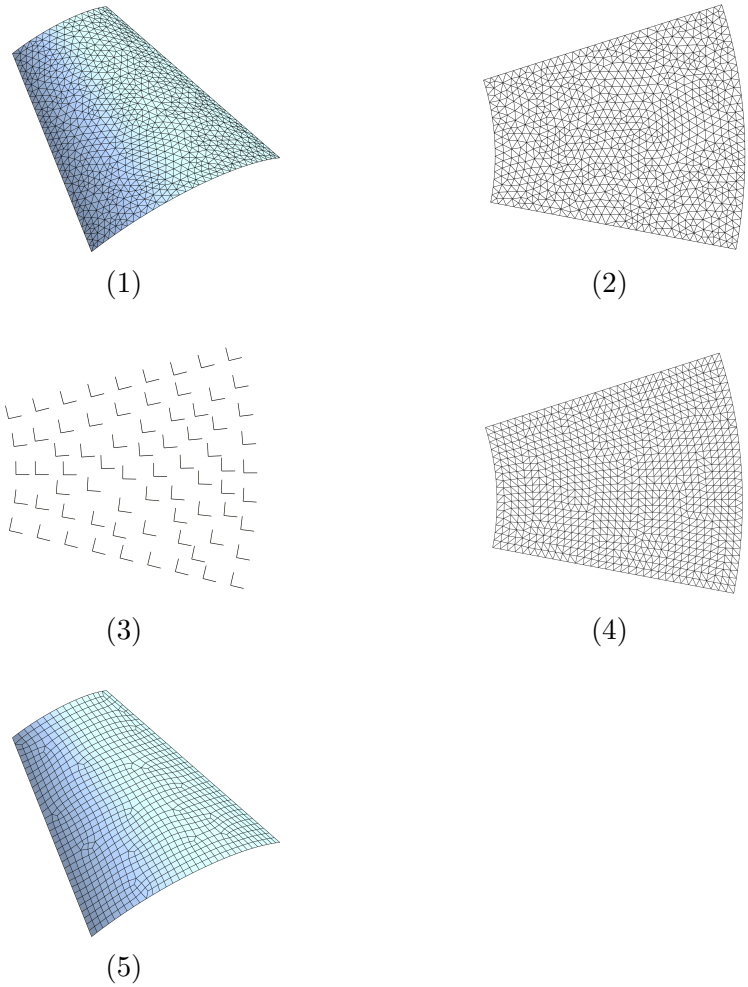
Fig. 4.22 illustrates the different steps of the global process. (1) is the triangular mesh of the geometry in the three-dimensional space. (2) is the triangular mesh of the geometry in the parametric space. (3) shows the cardinal direction field determining the orientation of the quads. (4) is the triangular mesh in the parametric space after the application of Lévy-Liu's algorithm. (5) is the final quadrilateral mesh.

Fig. 4.23 shows the energy density inside each Voronoi cell after the application of Lévy-Liu's algorithm. As seen from equation (4.4), the energy density is in fact the value of the L_p distance raised to the power of p and normalized by the density. The alignment is perfect in pale blue regions. It is more or less incorrect in yellow, orange or red regions.

The L_6 distance was chosen for all the examples of this chapter. However, it might be useful to compare the results obtained with other distances. Fig. 4.24 shows a car hood mesh optimized with three different distances: L_2, L_4 and L_6.

The variable τ is equal to 0.910 for the L_2 mesh, 0.967 for the L_4 mesh and 0.982 for the L_6 mesh. It has been calculated with Eq. 4.2. As expected, τ is higher for the L_6 mesh than for the L_4 mesh. The number of Gauss integration points necessary to evaluate the integrals exactly increases with the exponent of the L_p distance. The L_6 distance is therefore a good compromise between speed and squareness.

In Figs. 4.25 and 4.26, the meshes labelled as (b) were created with Lévy-Liu's algorithm, while the meshes labelled as (a) were created without it. The effect of Lévy-Liu's algorithm can be verified by comparing (a) and (b).



(1) (2)

(3) (4)

(5)

Figure 4.22: The various steps necessary to produce quadrilateral meshes

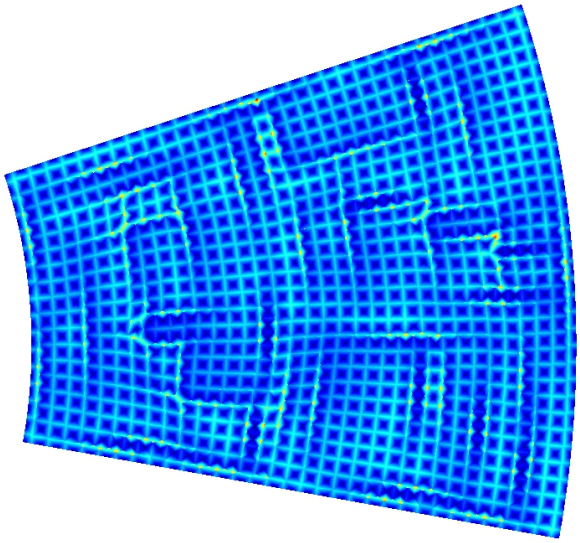


Figure 4.23: The energy density in the parametric space after the application of Lévy-Liu's algorithm

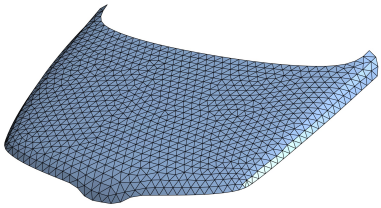
Fig. 4.25 shows a car body part. (b) contains 10051 quads. It took 5 minutes and 58 seconds to perform the 199 iterations of Lévy-Liu's algorithm on a standard 2010 laptop. A similar result could have been obtained with approximately 70 iterations in 2 minutes and 12 seconds.

(a) has an average quality of $\bar{\eta} = 0.80$ and (b) has an average quality of $\bar{\eta} = 0.90$. The quality $\eta(q)$ is defined in equation 2.1.

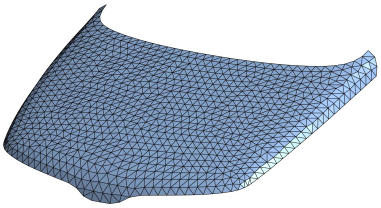
In an ideal quadrilateral mesh, each non-boundary mesh vertex is 4-valent, which means that it is connected to four neighbors. In (a), 71% of the non-boundary mesh vertices are 4-valent. In (b), this number reaches 89%.

Fig. 4.26 shows a mechanical part. (b) contains 44489 quads and 250 geometric faces. It took 17 minutes and 43 seconds to perform 199 iterations on each of the faces. (a) has an average quality of $\bar{\eta} = 0.77$, while (b) has an average quality of $\bar{\eta} = 0.84$. (b) contains 962 left-over triangles. These triangles may be eliminated by using an improved version of Blossom-quad.

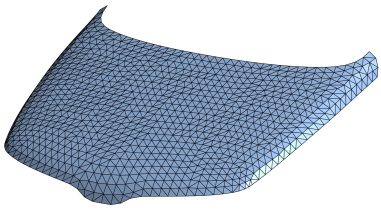
A time profiler was used to analyze the performance of the implementation. In particular, the curved sheet displayed on Fig. 4.22 was considered. The data show that calculating the energy and the gradient represents 45.44% of the execution time. 31.95% of the execution



L_2 optimization, $\tau = 0.910$



L_4 optimization, $\tau = 0.967$

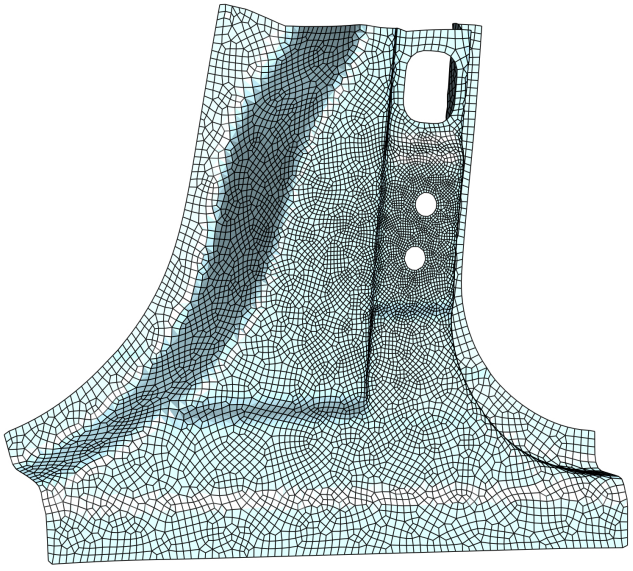


L_6 optimization, $\tau = 0.982$

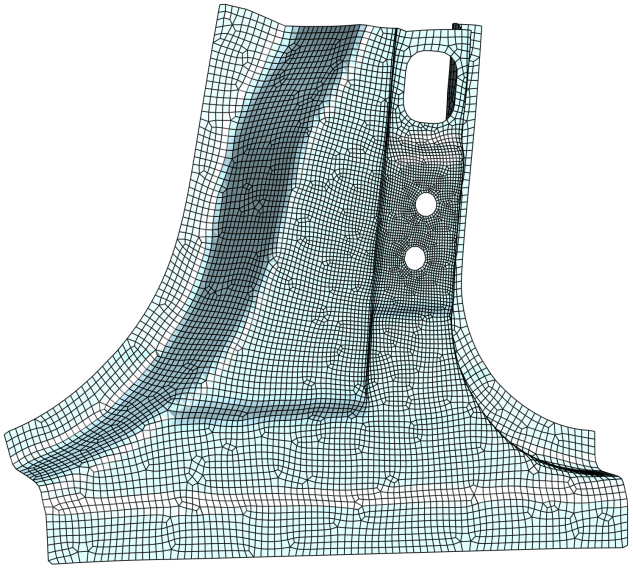
Figure 4.24: Three meshes of the same car hood optimized with different L_p distances

time is spent on cardinal directions and size field queries. These queries require the use of a quadtree data structure. The execution time could be lowered by employing less Gauss points. However, such a strategy would diminish the accuracy of the numerical integrations. For comparison, an attempt was made to use only three Gauss points, which can integrate exactly polynomials up to the order 2. Again, the geometry of Fig. 4.22 was chosen. The quality of the resulting quadrilaterals and their alignment was as good as the previous solution obtained with 16 Gauss points. The execution time fell from 29.17 to 19.45 seconds. Only 17.84% of the execution time was spent on the energy and gradient calculations.

The next chapter will present a three-dimensional implementation of Lévy-Liu's algorithm.

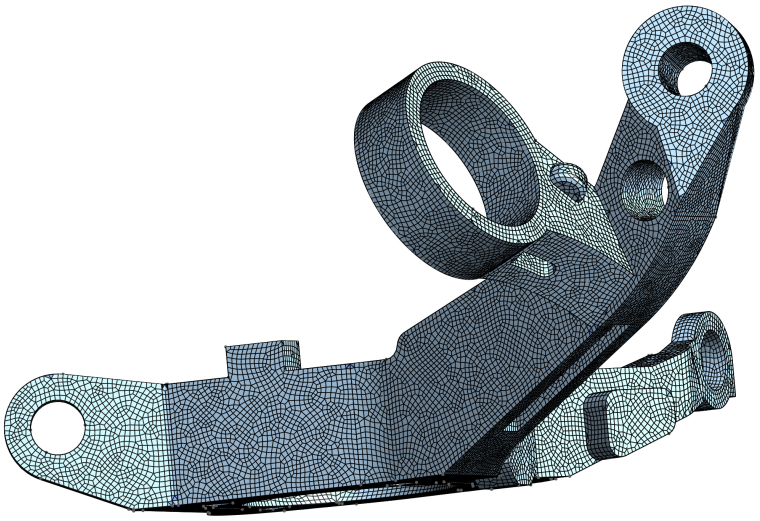


(a)

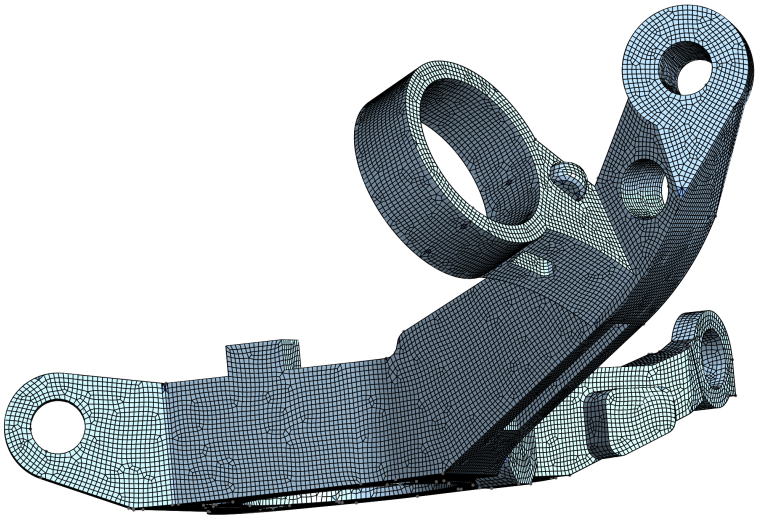


(b)

Figure 4.25: The mesh of a car body part generated with (a): Blossom-quad only (b): Blossom-quad in combination with Lévy-Liu



(a)



(b)

Figure 4.26: The mesh of a mechanical part generated with (a): Blossom-quad only (b): Blossom-quad in combination with Lévy-Liu

Chapter 5

Lloyd's energy minimization in three dimensions

A two-dimensional version of Lévy-Liu's algorithm was described in chapter 4. Lévy-Liu's algorithm can also be applied in three dimensions [45]. The energy and the gradient are not more difficult to evaluate. The clipping of three-dimensional Voronoi diagrams was part of Lévy and Liu's original implementation. However, it can be avoided by keeping the boundary mesh vertices and their neighbors unmovable. Let's consider a square mesh, as shown on Fig. 5.1. The mesh is two-dimensional for easier visualization.

The mesh edges are black and the Voronoi edges are purple. The only mesh vertices allowed to move are those in blue, because they are not boundary vertices or connected to boundary vertices.

As said in chapter 4, the Voronoi cells are divided into triangular elements in order to evaluate the energy and the gradient. Each triangular element contains a mesh vertex and two consecutive Voronoi vertices. In Fig. 5.1, the gradient is composed of six energy derivatives because there are three movable mesh vertices and two coordinates. The Voronoi vertices of the triangular elements contributing to the gradient were both marked in green on the picture. In other words, moving in-

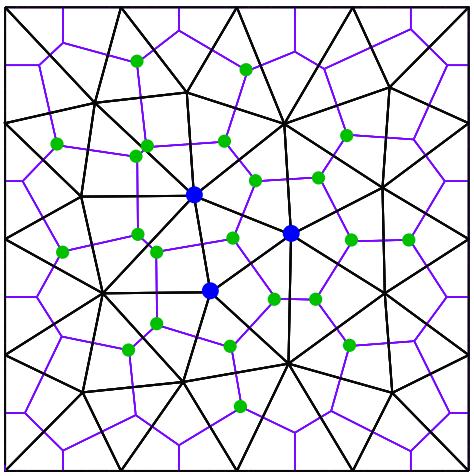


Figure 5.1: A square mesh and its Voronoi diagram. The only movable mesh vertices are those in blue.

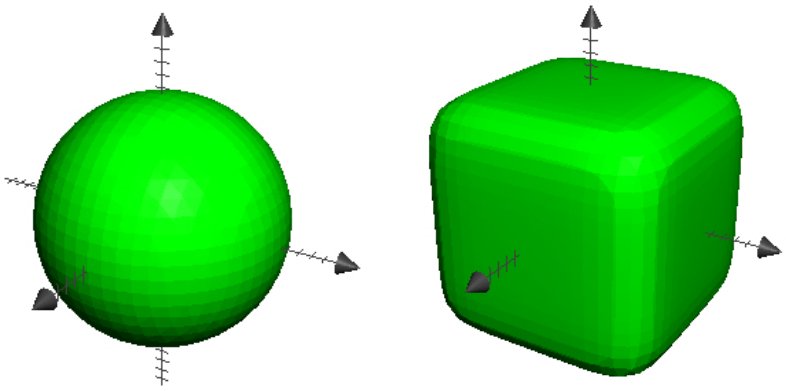
finitesimally the blue mesh vertices can only influence the Voronoi edges composed of two green Voronoi vertices. All these green Voronoi vertices are strictly inside the domain, not on the boundary. Therefore, clipping the Voronoi diagram with the domain would not be necessary in this case. Instead, the Voronoi diagram could be clipped by a bounding box englobing the domain. It is still necessary to prevent the Voronoi diagram from extending to infinity.

After a certain number of iterations, the blue mesh vertices could move closer to the boundary, which could render the gradient values incorrect. Nevertheless, this phenomenon will be assumed to be sufficiently uncommon to be ignored in this chapter. The same reasoning will be applied to the three-dimensional meshes.

Because the neighbors of the boundary mesh vertices are unmovable, they need to be well placed right from the start. For example, they could be created by projecting the boundary mesh vertices in the direction of the surface normal, at a distance h, with h being the local mesh size. Evidently, precautions would have to be taken to avoid generating vertices too close to each other. Another algorithm capable of similar results will be discussed in the later chapters.

Fig. 5.2 shows the unit spheres of the L_2 and L_6 distances in three dimensions. The three-dimensional energy function will employ the L_6

distance, just like in two dimensions. The energy and the gradient are evaluated with Gauss techniques. The same LBFGS library is used.



(1) (2)

Figure 5.2: The unit spheres of the L_2 (1) and L_6 (2) distances

This chapter is divided into four sections. The first section briefly discusses the generation of three-dimensional Voronoi diagrams. The second section explains how the energy and the gradient are calculated. The third section gives some indications about the LBFGS optimization step. The last section presents the results.

5.1 Voronoi diagrams

The three-dimensional Voronoi diagrams are generated with the Voro++ free software [74]. Voro++ has the ability to clip Voronoi diagrams with simple geometries. In this chapter, the clipping geometries are always bounding boxes, i.e. rectangular parallelepipeds.

Fig. 5.3 shows a three-dimensional mesh and Fig. 5.4 shows its corresponding Voronoi diagram. The Voronoi diagram does not extend to infinity. It is clipped by a bounding box englobing the geometry and not by the geometry itself. As explained earlier, the energy gradient is assumed to remain correct because the boundary mesh vertices and their neighbors are kept immobile.

The Voronoi cells are divided into tetrahedral elements in order to permit numerical integration. The validity of a Voronoi diagram can therefore be verified by comparing volumes. The volume of the bound-

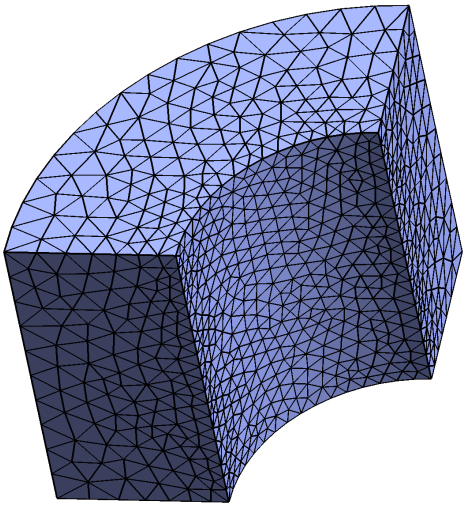


Figure 5.3: A three-dimensional mesh

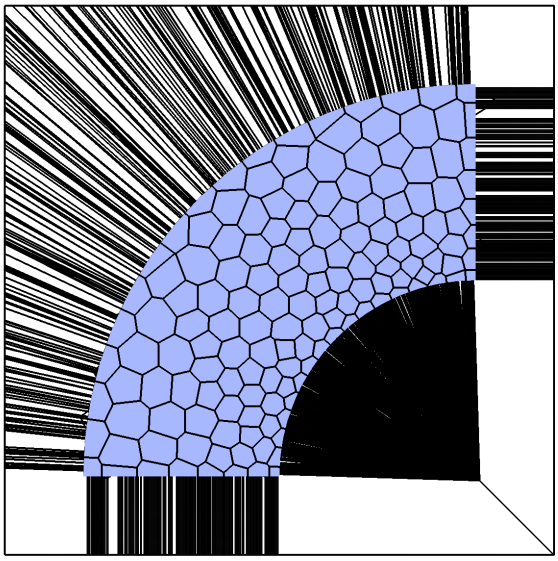


Figure 5.4: A top view of the Voronoi diagram

ing box should be equal to the sum of the volumes of the tetrahedral elements. In the case of Fig. 5.4, both of these volumes are equal to 2.744000.

The total energy is equal to the sum of the energy of all the tetrahedral elements, even if they do not contribute to the gradient. There are 108,634 tetrahedral elements in Fig. 5.4 compared to 11,881 mesh tetrahedra in Fig. 5.3. This is one of the disadvantages of clipping Voronoi diagrams with bounding boxes instead of geometries. The boundary Voronoi cells have more faces, which lead to more tetrahedral elements. As a result, the execution time is larger.

5.2 Energy and gradient evaluation

M is the change of basis matrix that goes from the global system of coordinates S to the local system of coordinates S'. The axes of S' are aligned with the local cardinal directions. Let $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ and $\boldsymbol{v}_3$ be three orthonormal vectors selected among the local cardinal directions. M is then equal to $\begin{bmatrix}\boldsymbol{v_1} & \boldsymbol{v_2} & \boldsymbol{v_3}\end{bmatrix}^{-1}$.

F is the energy function minimized by Lévy-Liu's algorithm [45]. R_i is the region bounded by the i-th Voronoi cell generated by mesh vertex $\boldsymbol{x}_i$. There are N Voronoi cells.

$$F(\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N) = \sum_{i=1}^{N} \underbrace{\int_{R_i} \rho(\boldsymbol{y}) ||M(\boldsymbol{y} - \boldsymbol{x}_i)||_p^p d\boldsymbol{y}}_{I^{R_i}(\boldsymbol{x}_i)} \tag{5.1}$$

Eq. (5.2) illustrates the relation between the density ρ and the mesh size h in three dimensions [22]. A similar procedure to the one described in chapter 4 was employed to obtain this result.

$$\rho \propto \frac{1}{h^{p+3}} \tag{5.2}$$

A constant of 1 was chosen, making ρ equal to $\frac{1}{h^{p+3}}$ for the remaining part of this chapter.

On the surface of the three-dimensional domain, the mesh size h is known. As a result, Dirichlet conditions can be imposed. The mesh size field inside the domain is assumed to obey a Laplace equation.

The energy gradient is the derivative of F with respect to each movable mesh vertex. The mesh vertices numbered from 1 to n are assumed

to be movable and the mesh vertices numbered from $n+1$ to N are assumed to be unmovable.

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{x}_k}F(\boldsymbol{x}_1,\boldsymbol{x}_2,...,\boldsymbol{x}_N)=\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{x}_k}\sum_{i=1}^{N}I^{R_i}(\boldsymbol{x_i})=\sum_{i=1}^{N}\frac{\mathrm{d}I^{R_i}(\boldsymbol{x_i})}{\mathrm{d}\boldsymbol{x}_k}\quad k=1,...,n \tag{5.3}$$

The total derivatives can be separated into partial derivatives.

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{x}_k}F(\boldsymbol{x}_1,\boldsymbol{x}_2,...,\boldsymbol{x}_N)=\sum_{i=1}^{N}\frac{\partial I^{R_i}(\boldsymbol{x_i})}{\partial\boldsymbol{x}_k}+\sum_{i=1}^{N}\frac{\partial I^{R_i}(\boldsymbol{x_i})}{\partial R_i}\frac{\mathrm{d}R_i}{\mathrm{d}\boldsymbol{x}_k} \tag{5.4}$$

Similarly to the two-dimensional case, the total derivative symbol takes into consideration the fact that moving a mesh vertex modify the Voronoi cells shapes. On the contrary, the partial derivative symbol considers the Voronoi cells to be undeformable.

The i-th Voronoi cell is a polyhedron composed of the Voronoi vertices $\boldsymbol{C}_{ij}$.

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{x}_k}F(\boldsymbol{x}_1,\boldsymbol{x}_2,...,\boldsymbol{x}_N)=\frac{\partial I^{R_k}(\boldsymbol{x_k})}{\partial\boldsymbol{x}_k}+\sum_{i=1}^{N}\sum_{j=1}^{M_i}\frac{\partial I^{R_i}(\boldsymbol{x_i})}{\partial\boldsymbol{C}_{ij}}\frac{\mathrm{d}\boldsymbol{C}_{ij}}{\mathrm{d}\boldsymbol{x}_k} \tag{5.5}$$

The first type of Voronoi vertices are the centers of the Delaunay tetrahedra circumspheres. The second type of Voronoi vertices are the intersection points between the bounding box boundaries and the Delaunay edges bisecting planes. The movable mesh vertices are presumed to have no influence on the second type of Voronoi vertices.

In order to evaluate the integrals with Gauss techniques, each Voronoi cell is divided into tetrahedral elements. The integral $I^{R_i}(\boldsymbol{x}_i)$ of Eq. (5.1) can be expressed as a sum of integrals over the tetrahedral elements E_{ij}:

$$I^{R_i}(\boldsymbol{x}_i)=I^{E_{i1}}(\boldsymbol{x}_i)+I^{E_{i2}}(\boldsymbol{x}_i)+I^{E_{i3}}(\boldsymbol{x}_i)+... \tag{5.6}$$

A tetrahedral element is composed of three Voronoi vertices and one mesh vertex, i.e. the vertex generating the Voronoi cell. Evidently, the derivative of $I^{E_{ij}}$ with respect to a Voronoi vertex is null if the Voronoi vertex is not a member of E_{ij}.

Unclipped Voronoi cells are convex polyhedra composed of convex polygonal faces [44]. It is also known that *the intersection of two or*

more convex polyhedra is a (possibly empty) convex polyhedron [23]. Because the bounding boxes are convex polyhedra, the clipped Voronoi cells remain convex. A three-dimensional Voronoi cell is illustrated on Fig. 5.5. Each face can be triangulated by connecting one of its Voronoi vertex with the other Voronoi vertices, as shown on Fig. 5.6. The whole Voronoi cell can then be tetrahedralized by connecting its generating mesh vertex with each Voronoi vertices [45], just like in Fig. 5.7.

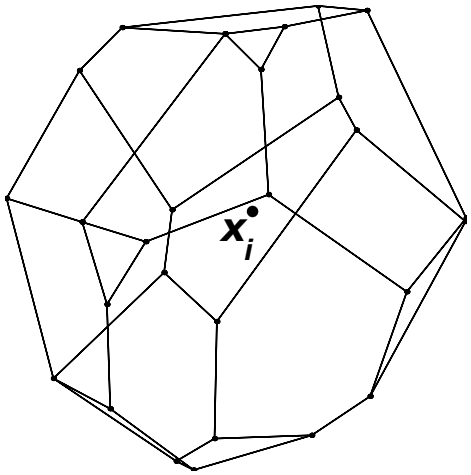


Figure 5.5: The Voronoi cell generated by the mesh vertex $\boldsymbol{x}_i$

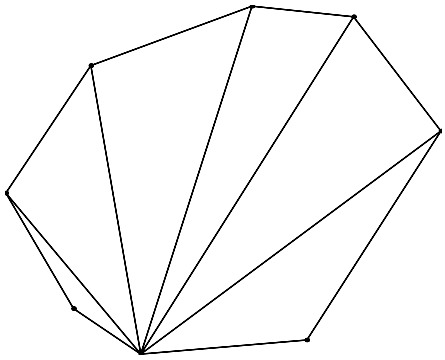


Figure 5.6: A face triangulation

Gauss integration is used to evaluate the integrals $I^{E_{ij}}$ and their derivatives. Fig. 5.8 shows the linear transformation T that goes from the reference element E' to the physical element E. 43 Gauss points are

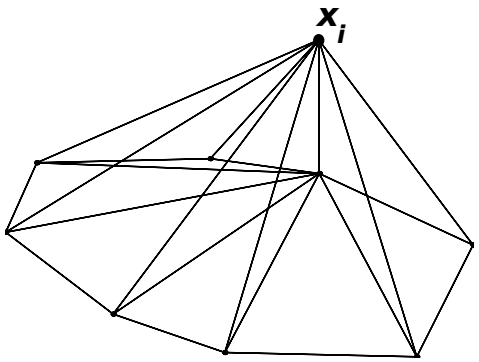


Figure 5.7: A Voronoi cell can be decomposed into tetrahedral elements by connecting the generating mesh vertex $\boldsymbol{x}_i$ with the Voronoi vertices of each triangulated face.

employed. As a result, polynomials up to the order 8 can be integrated exactly.

$$\boldsymbol{y} = \boldsymbol{T}(u, v, w) = \boldsymbol{x}_0(1 - u - v - w) + \boldsymbol{C}_1 u + \boldsymbol{C}_2 v + \boldsymbol{C}_3 w \tag{5.7}$$

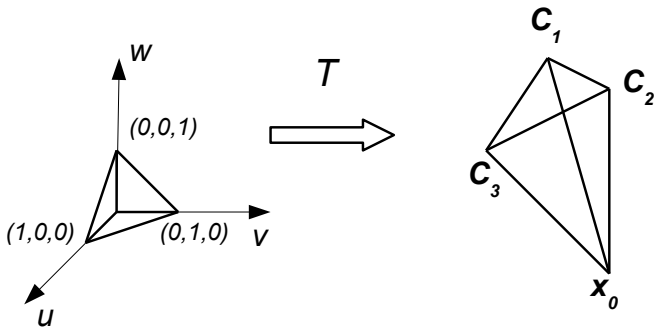


Figure 5.8: The linear transformation T

The next three equations express the integral I^E and its derivatives in terms of the reference element E'. $\boldsymbol{C}_1$ is a Voronoi vertex.

$$I^E(\boldsymbol{x_0}) = \int_{E'} \rho(\boldsymbol{T}(u, v)) ||M(\boldsymbol{T}(u, v) - \boldsymbol{x}_0)||_p^p J du dv \tag{5.8}$$

$$\frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{x}_0} = \frac{\partial}{\partial \boldsymbol{x}_0} \int_{E'} \rho(\boldsymbol{T}(u,v)) ||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p J du dv \tag{5.9}$$

$$= \int_{E'} \rho(\boldsymbol{T}(u,v)) \frac{\partial ||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p}{\partial \boldsymbol{x}_0} J du dv$$

$$\frac{\partial I^E(\boldsymbol{x_0})}{\partial \boldsymbol{C}_1} = \frac{\partial}{\partial \boldsymbol{C}_1} \int_{E'} \rho(\boldsymbol{T}(u,v)) ||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p J du dv \tag{5.10}$$

$$= \int_{E'} \frac{\partial \rho(\boldsymbol{T}(u,v))}{\partial \boldsymbol{T}(u,v)} \frac{\mathrm{d}\boldsymbol{T}(u,v)}{\mathrm{d}\boldsymbol{C}_1} ||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p J +$$

$$\rho(\boldsymbol{T}(u,v)) \frac{\partial ||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p}{\partial \boldsymbol{T}(u,v)} \frac{\mathrm{d}\boldsymbol{T}(u,v)}{\mathrm{d}\boldsymbol{C}_1} J +$$

$$\rho(\boldsymbol{T}(u,v)) ||M(\boldsymbol{T}(u,v) - \boldsymbol{x}_0)||_p^p \frac{\partial J}{\partial \boldsymbol{C}_1} du dv$$

The mesh size is interpolated linearly. It is not queried at every integration point for efficiency reasons. Eq. 5.2 is used to obtain the density and its derivatives. Interpolating directly the density should be avoided because it leads to convergence difficulties. The density is proportional to the inverse of h elevated to the power of $p+3$. Therefore, the variations of density are far more important than the variations of mesh size inside an element.

The mesh size derivatives with respect to physical coordinates are obtained from the mesh size derivatives with respect to parametric coordinates through the linear transformation T. The inverse of the Jacobian is therefore present. Very slim tetrahedral elements have almost null Jacobians, which can prevent the LBFGS library from functioning correctly. When the mesh vertices are aligned in a rectangular way, the Voronoi cells appear like cubes. Nevertheless, these cubes can have very small faces at the corners. These small faces will translate into needles. The problem can be solved by neglecting the needles. The value of the total energy should not be affected, but the gradient could become incorrect. The slimness of a tetrahedral element can be evaluated by taking the ratio of the shortest edge to the longest one [27].

The following matrix is the derivative of the Voronoi vertex $\boldsymbol{C}_1$ with respect to the mesh vertex $\boldsymbol{x}_0$ [45].

$$\frac{\mathrm{d}\boldsymbol{C}_1}{\mathrm{d}\boldsymbol{x}_0} = \begin{bmatrix} (\boldsymbol{x}_1 - \boldsymbol{x}_0)^T \\ (\boldsymbol{x}_2 - \boldsymbol{x}_0)^T \\ (\boldsymbol{x}_3 - \boldsymbol{x}_0)^T \end{bmatrix}^{-1} \begin{bmatrix} (\boldsymbol{C}_1 - \boldsymbol{x}_0)^T \\ (\boldsymbol{C}_1 - \boldsymbol{x}_0)^T \\ (\boldsymbol{C}_1 - \boldsymbol{x}_0)^T \end{bmatrix} \tag{5.11}$$

5.3 LBFGS optimization

Because the mesh size is variable, the scale of the vertices coordinates is also variable. The scale refers to the expected amplitude of the vertices displacements during the optimization. There are $3N$ variables to optimize, with N being the number of movable mesh vertices. Each variable is the coordinate of a movable mesh vertex. The LBFGS solver should be preconditioned with the scale of each variable [14]. In the implementation, the scale at a given point of the domain is equal to the mesh size multiplied by a certain arbitrary factor, such as 0.5. If the scale is not specified, the convergence could be difficult to achieve.

5.4 Results

The aim of this section is to verify the validity of the three-dimensional implementation of Lévy-Liu's algorithm. The results will be analyzed in terms of vertices alignment and hexahedra percentages. In two dimensions, the orientation of the quadrilaterals could be assessed visually, but not in three dimensions. The three-dimensional cardinal direction fields and mesh size fields are computed according to the techniques described in chapter 3.

Fig. 5.9 shows the vertices distribution inside a cube after 0, 3, 7 and 21 iterations. The initial distribution corresponds to a perfect solution that has been slightly perturbed. As the number of iterations increases, the vertices alignment clearly improves. The distribution is almost perfect after 21 iterations.

Fig. 5.10 shows the vertices distribution inside an inclined cube after 0, 3, 7 and 19 iterations. This test was performed to verify the calculations related to the change of basis matrix M. Again, the vertices alignment improves progressively.

Pairs of mixed hexahedral meshes are displayed on Figs. 5.11, 5.13, 5.15 and 5.17. Each pair was created by applying Yamakawa-Shimada's

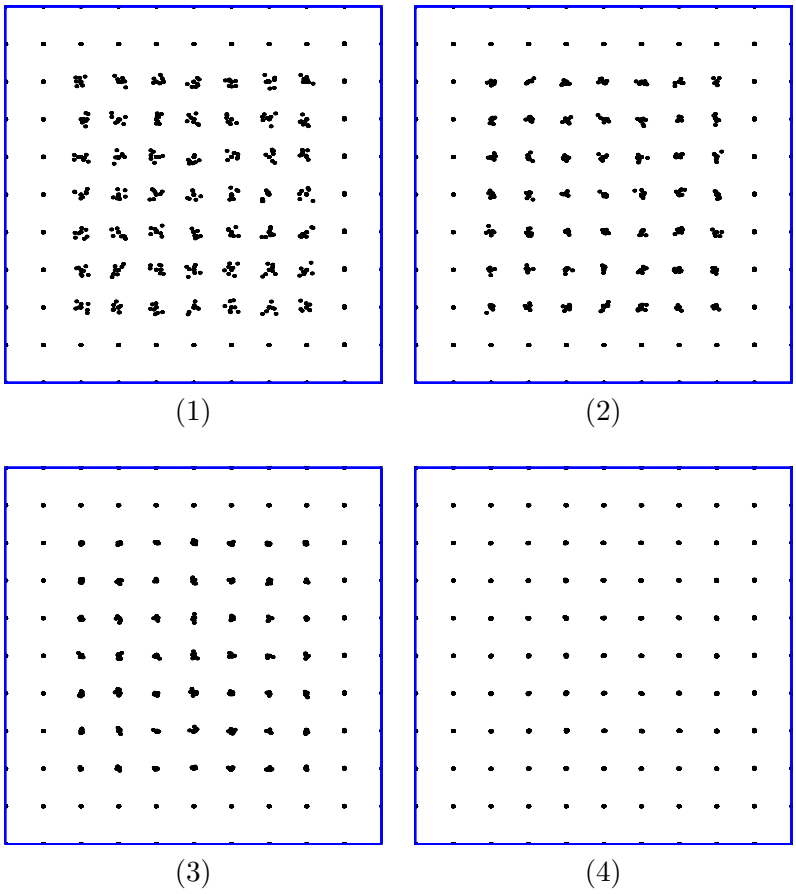
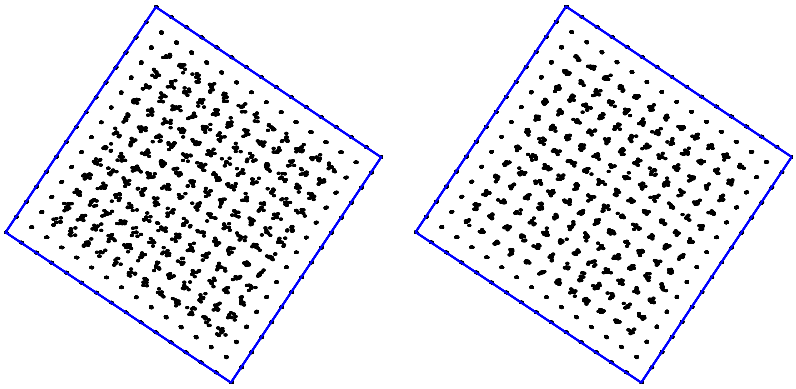
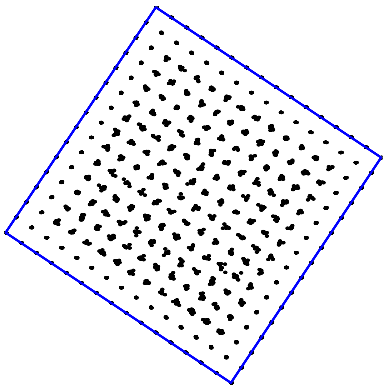


Figure 5.9: The vertices distribution after 0 iterations (1), 3 iterations (2), 7 iterations (3) and 21 iterations (4)

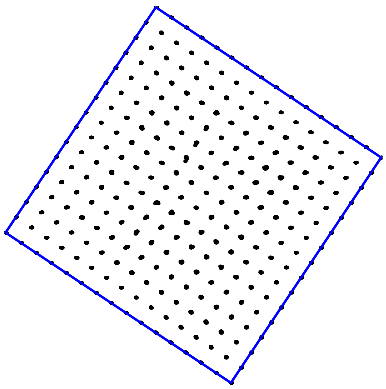


(1)

(2)



(3)



(4)

Figure 5.10: The vertices distribution after 0 iterations (1), 3 iterations (2), 7 iterations (3) and 19 iterations (4)

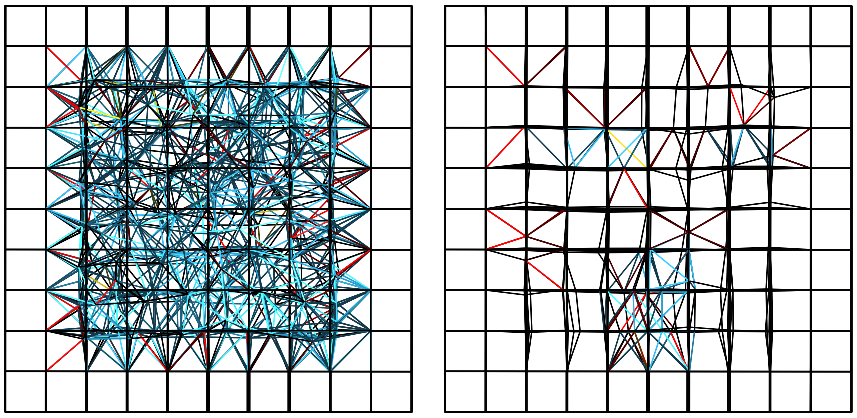
algorithm to two different vertices distributions. The second vertices distribution was generated by optimizing the first one with Lévy-Liu's algorithm. Figs. 5.12, 5.14, 5.16 and 5.18 are the corresponding convergence curves. A convergence curve is a graph of the energy in terms of the number of iterations. Each iteration is associated with a different vertices configuration. The LBFGS library performs line searches to identify lower energy configurations. Vertices configurations leading to higher energy are not accepted.

To facilitate visibility, only the mesh edges are shown. Hexahedra have black edges, tetrahedra have blue edges, prisms have yellow edges and pyramids have red edges. The quantity of black edges is an indicator of the percentage of hexahedra. Because the immediate neighbors of the boundary mesh vertices are well aligned, the first layers consist mostly of hexahedra. As said in the beginning of the chapter, the boundary mesh vertices are projected in the direction of the surface normal.

Three quantities are used to evaluate the quality of the meshes: the percentage of hexahedra by number H_{nbr}, the percentage of hexahedra by volume H_{vol} and the average hex quality Q defined in Eq. 2.2. Table 5.17 contains the relevant mesh data. In all cases, H_{nbr}, H_{vol} and Q were improved by Lévy-Liu's optimization. The convergence curves also display the expected tendency: the energy decreases as the number of iterations increases. However, there is a high peak near the end of the curve in Fig. 5.14. This peak was most likely caused by the LBFGS library moving vertices to unfavorable positions.

	Fig. 5.11	Fig. 5.13	Fig. 5.15	Fig. 5.17
H_{nbr} (1)	42.17%	37.67%	33.07%	35.23%
H_{vol} (1)	73.97%	70.41%	65.97%	66.15%
Q (1)	0.89	0.88	0.87	0.89
H_{nbr} (2)	85.95%	67.37%	63.10%	51.96%
H_{vol} (2)	95.90%	88.97%	84.48%	78.26%
Q (2)	0.98	0.94	0.93	0.94
T	$\approx$ 10 min.	$>$ 20 min.	$>$ 20 min.	$\approx$ 14 min.

Table 5.1: This table highlights the improvements brought to the hexahedra percentages. The first mesh of each pair has the label (1) and the second mesh has the label (2). The execution time T was measured on a 2010 laptop computer.



(1) (2)

Figure 5.11: The vertices distribution of mesh (2) was created by applying 200 iterations of Lévy-Liu's algorithm to the vertices distribution of mesh (1)

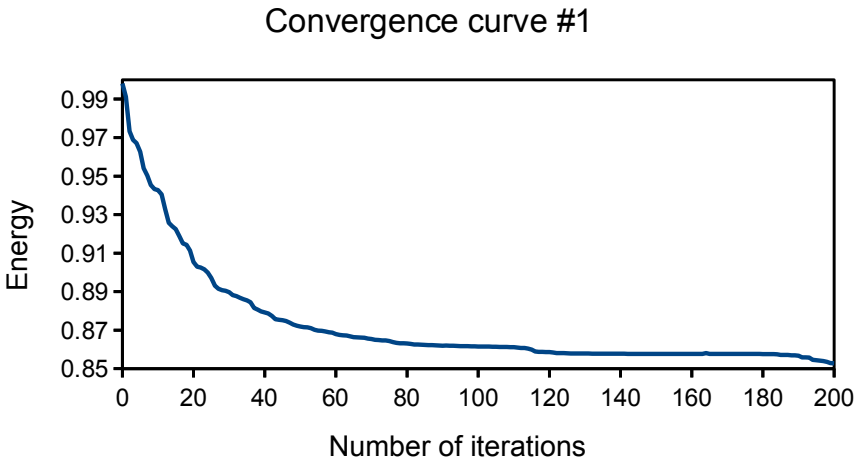
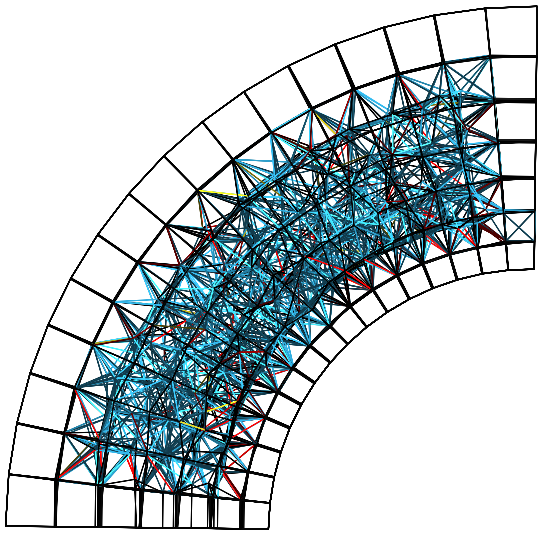
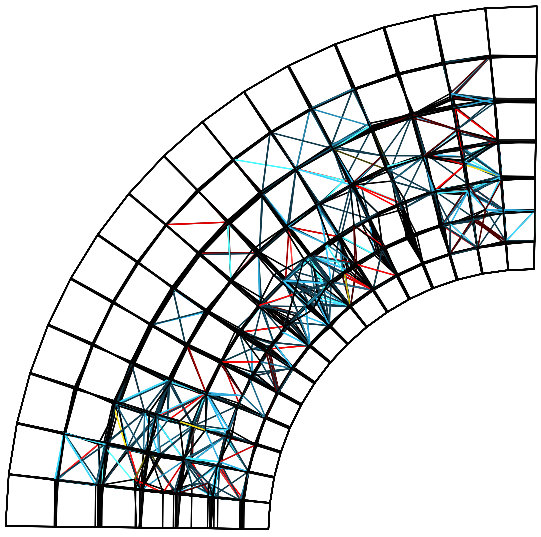


Figure 5.12: Lloyd's energy in terms of the number of iterations (Fig. 5.11 (2))



(1)



(2)

Figure 5.13: The vertices distribution of mesh (2) was created by applying 200 iterations of Lévy-Liu's algorithm to the vertices distribution of mesh (1)

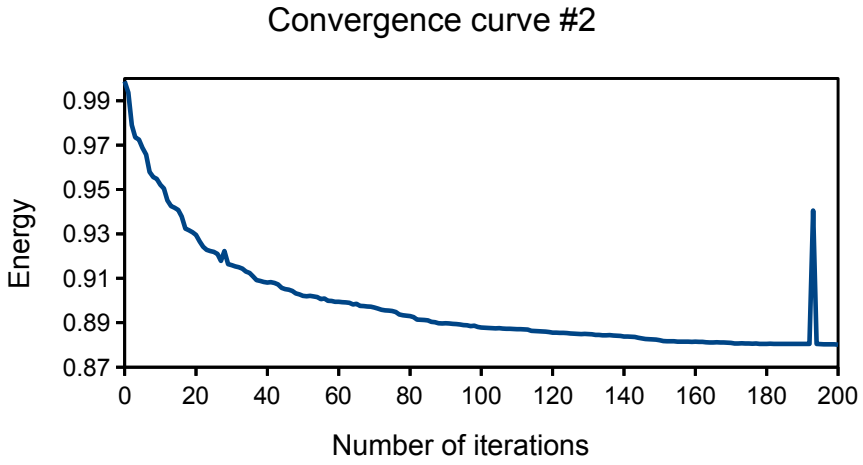
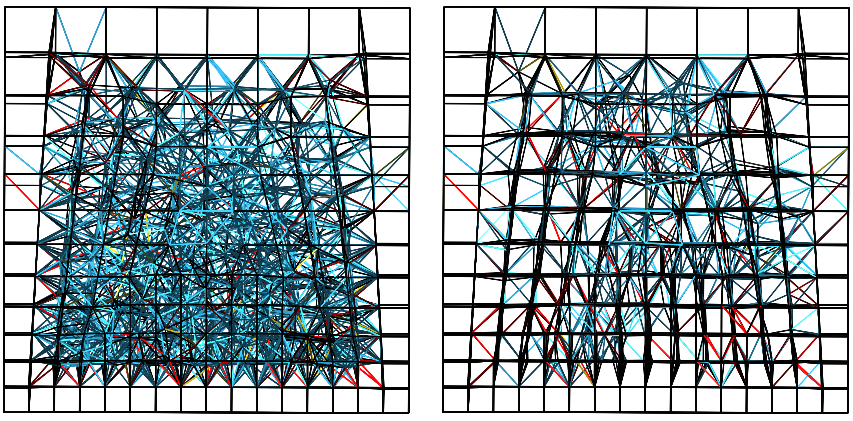


Figure 5.14: Lloyd's energy in terms of the number of iterations (Fig. 5.13 (2))



(1) (2)

Figure 5.15: The vertices distribution of mesh (2) was created by applying 161 iterations of Lévy-Liu's algorithm to the vertices distribution of mesh (1)

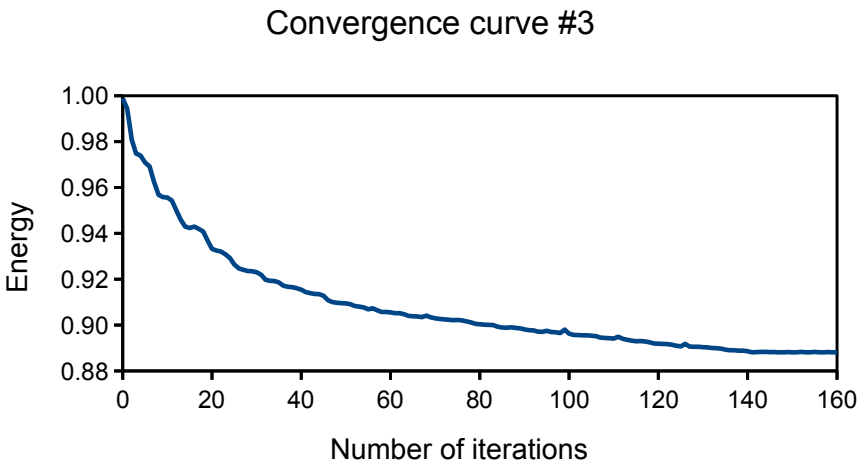
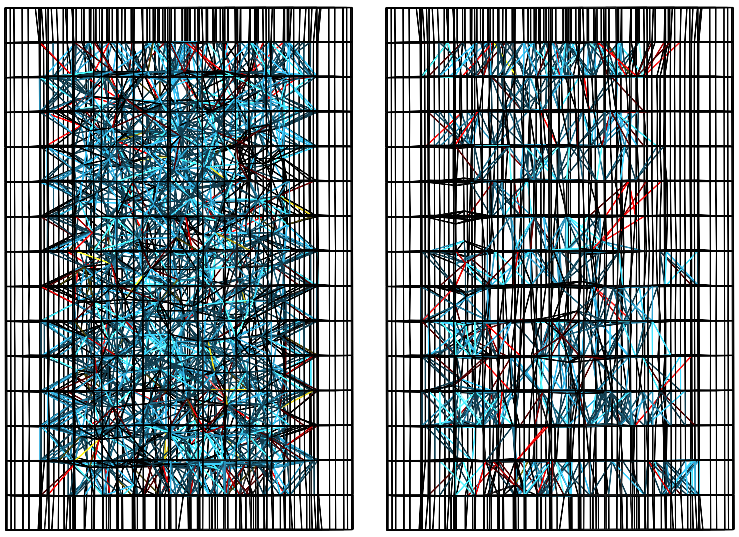


Figure 5.16: Lloyd's energy in terms of the number of iterations (Fig. 5.15 (2))



(1) (2)

Figure 5.17: The vertices distribution of mesh (2) was created by applying 200 iterations of Lévy-Liu's algorithm to the vertices distribution of mesh (1)

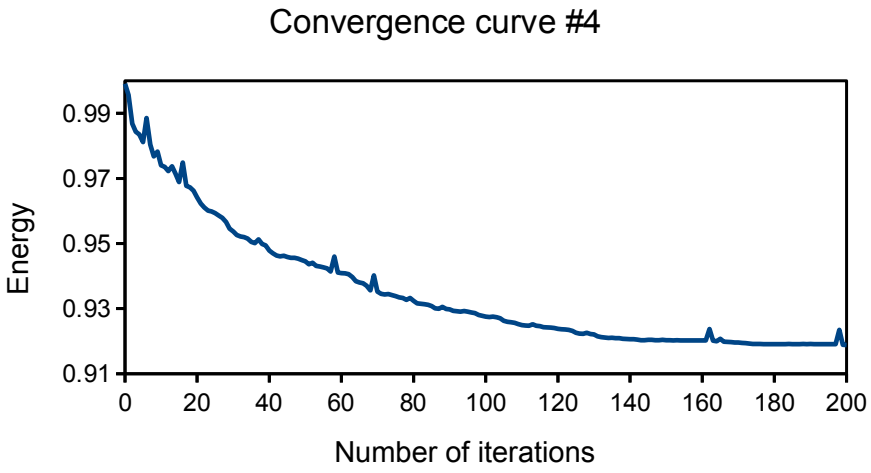


Figure 5.18: Lloyd's energy in terms of the number of iterations (Fig. 5.17 (2))

A profiler was used to identify the most time consuming methods. The problem illustrated on Fig. 5.9 was considered. The data show that 33.56% of the time is spent on numerical integrations. 23.68% of the time is spent on cardinal directions and size field queries. These queries require an octree data structure. 22.73% of the time is spent on the matrices operations related to equation 5.11. This percentage could certainly be greatly reduced by using specialized formulas for 3×3 matrices instead of general algorithms for $n\times n$ matrices. 10.83% of the time is spent on generating the Voronoi diagram. When four Gauss points are employed instead of 43, the percentage of the time spent to evaluate the integrals falls from 33.56% to 6.37%. As a reminder, four Gauss points can integrate exactly polynomials up to the order 2. The perfect configuration shown on Fig. 5.9 (4) was attained even with four Gauss points.

The execution times of table 5.1 are very high even for small meshes. On the contrary, Lévy and Liu's implementation was shown to be faster by several order of magnitudes [45]. More careful programming and lower order numerical integration could certainly accelerate the implementation presented in this chapter. However, the next two chapters will explore a completely different alternative. More precisely, the frontal approach will be examined in the hope of attaining practical execution times. Modern examples of practical execution times can be found in

the literature. For example, the automatic method developed by Vyas and Shimada was capable of meshing a control arm with 62,300 mixed elements in less than 15 minutes [85]. The mesh was later employed for an elasto-plastic analysis [85].

Chapter 6

The frontal approach in three dimensions

In a perfect hexahedral mesh, each vertex is surrounded by six other vertices: one at the left, one at the right, one above, one below, one at the front and one at the back. The position of these vertices depends on the local mesh size and local orientation. This observation can be used as a starting point for designing an algorithm capable of generating clouds of well aligned vertices. The algorithm proceeds as follow: each mesh vertex attempts to create six other vertices, one in each cardinal direction. However, a prospective vertex is added to the cloud only if it satisfies two conditions: it is inside the domain and it is not too close to an existing vertex. The cloud is later tetrahedralized.

The surfacic boundary of the geometry is assumed to be already meshed. At the beginning of the process, all boundary mesh vertices are pushed into a queue. A queue is a first-in first-out data structure. Queued vertices are then popped one after the other. Each vertex popped attempts to create six other vertices. As said earlier, if a prospective vertex is inside the domain and not too close to an existing vertex, it is added to the cloud and pushed into the queue. The process eventually stops by itself. The final step consists of tetrahedralizing the vertices using a standard Delaunay procedure [28, 79].

The algorithm shares some similarities with the advancing front method. The vertices are created layer by layer toward the center. However, contrary to the advancing front method, it does not construct a connectivity gradually. All the tetrahedra are generated only at the end. Such a frontal algorithm could be a much faster alternative than the implementation presented in chapter 5, which was too slow to be practical.

The quality of the alignment inside the geometry depends strongly on the quality of the alignment on the surface. The surfacic triangles should be as squared as possible. Such meshes can be created with Lévy-Liu's [45] or Delquad [69, 68] algorithms. However, a two-dimensional version of the frontal algorithm was employed for most of the examples presented here. This two-dimensional version will be discussed in the next chapter. The frontal algorithm is not particularly more difficult to implement in three dimensions than in two dimensions.

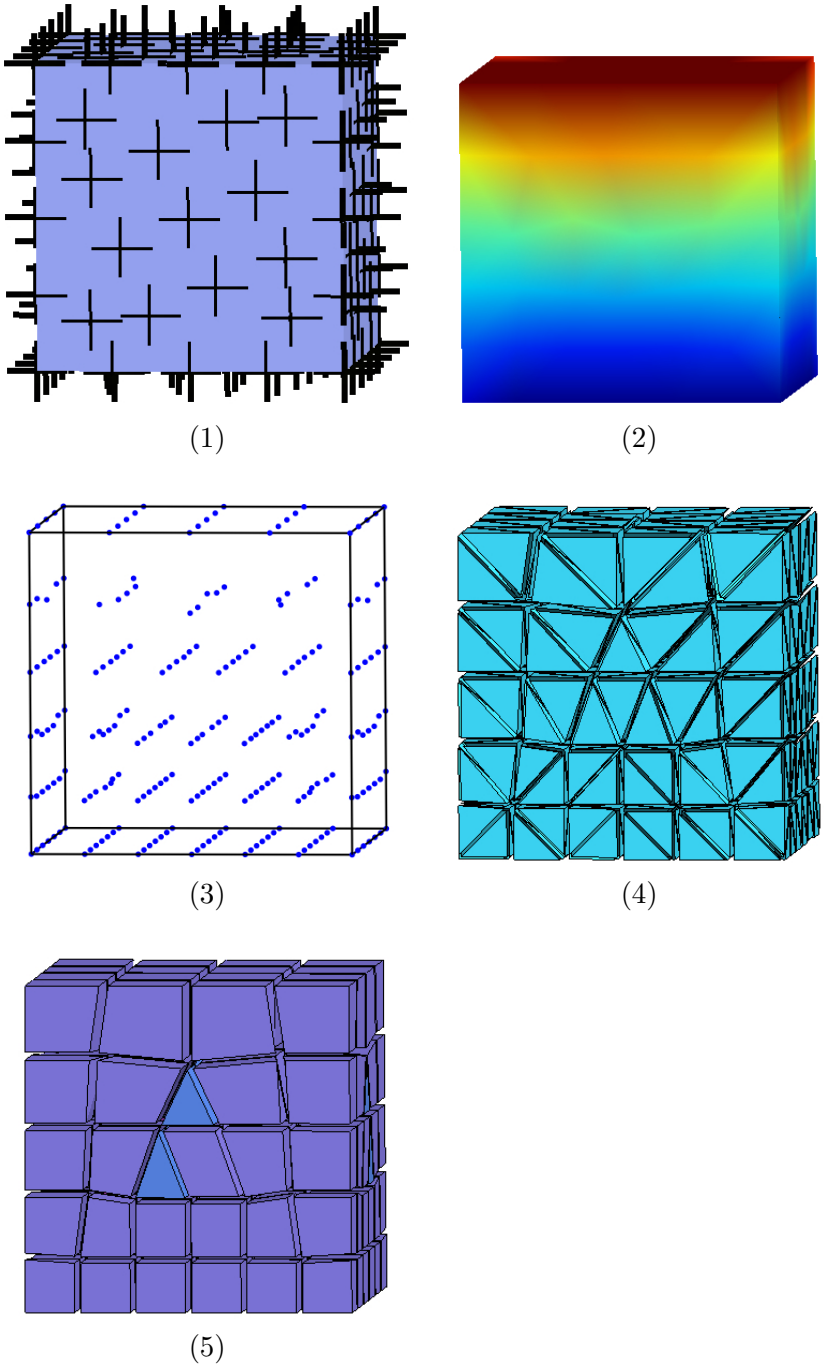
Fig. 6.1 illustrates the main steps of the mixed hexahedral mesh generation process. The frontal algorithm corresponds to step 3. Chapter 9 contains a pseudo-code version of the frontal algorithm.

A prospective vertex is added to the cloud only if there is a certain unoccupied area around it. The size of this exclusion area depends on the local mesh size. Each vertex is enclosed into a box large enough to contain its exclusion area. If there is no box intersection between two vertices, then they are not too close to each other. R-trees [31, 20] can efficiently determine which boxes intersect a given box. Therefore, it is not necessary to compute the distance between the prospective vertex and all the other ones.

This chapter is divided as follow. Section 6.1 gives more details about the frontal algorithm. Section 6.2 introduces the infinity distance. Section 6.3 explains how R-trees are used. Section 6.4 discusses the smoothing of three-dimensional cardinal direction fields.

6.1 Vertices creation

As said earlier, the boundary mesh vertices are first pushed into a queue. They are then popped in order. As shown on Fig. 6.2, each vertex popped attempts to create one vertex in each cardinal directions, at a distance h from itself. h is the local mesh size. The local cardinal directions are fetched from the cardinal direction field.



(1) (2)

(3) (4)

(5)

Figure 6.1: The different steps necessary to produce mixed hexahedral meshes. (1): cardinal direction field, (2): mesh size field, (3): vertices cloud, (4): vertices tetrahedralization, (5): mixed hexahedral mesh. 123 out of the 163 final elements are hexahedra.

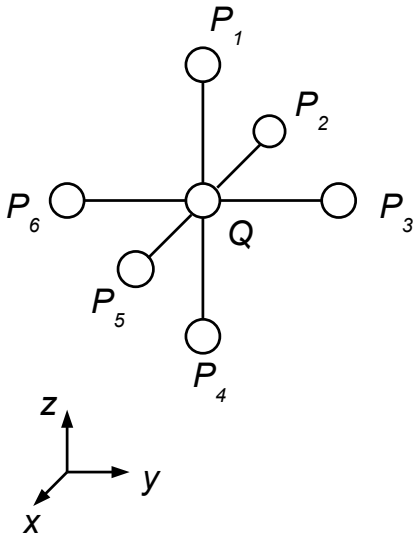


Figure 6.2: The vertex Q is already present in the cloud. The frontal algorithm verifies if the vertices P_1 to P_6 can be added, in an attempt to emulate the configuration of an ideal hexahedral mesh

A prospective vertex is added to the cloud and to the queue only if it satisfies these two conditions:

1. It is inside the domain.

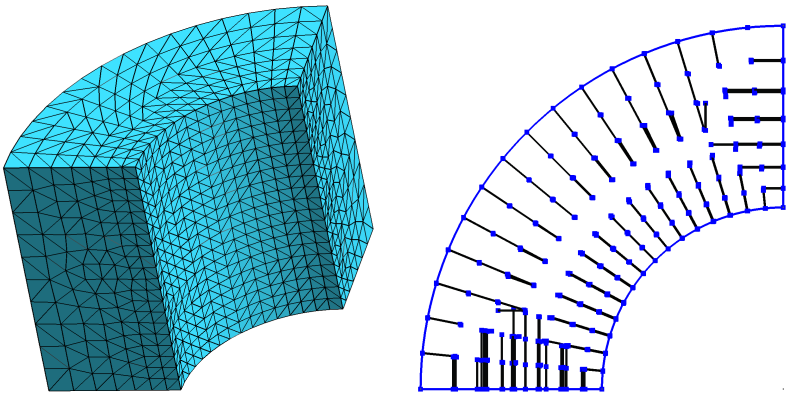
2. It is not too close to an existing mesh vertex.

An octree data structure can be employed to efficiently determine if a vertex is inside the domain [28].

The second condition is necessary in order to prevent the vertices from being too close to each other. The minimum acceptable distance between a prospective vertex and an existing one is equal to kh, with h being the local mesh size at the prospective vertex and k ranging from 0 to 1. k absolutely needs to be inferior to one. If not, too many valid vertices will be missing from the cloud. In this implementation, k is equal to 0.7.

Eventually, no more prospective vertices can be added to the cloud without being too close to existing ones. The process then stops and the cloud is tetrahedralized with a Delaunay procedure [28, 79].

The frontal algorithm was applied to the quarter cylinder shown on Fig. 6.3 (1). In Fig. 6.3 (2), black lines are traced between each vertex and its parent in order to observe the progression of the algorithm.



(1) (2)

Figure 6.3: A cloud of well aligned vertices was generated inside the quarter cylinder. Black lines connect each vertex with its parent. (2) is an interior view of (1).

As said earlier, each vertex attempts to create six other vertices at a distance $d = h$ from itself. For smoother size transitions, d can instead be an average between the local mesh size at the parent vertex and the local mesh size at the prospective vertex.

Fig. 6.4 shows a mixed hexahedral mesh created with the frontal algorithm. Proportionally speaking, it contains many more hexahedra than the meshes presented in chapter 2.

Four pairs of mixed hexahedral meshes were presented in the results section of chapter 5. Each of these pairs contained two meshes labelled (1) and (2). In all cases, the vertices distribution of mesh (2) was created by optimizing the vertices distribution of mesh (1) with Lévy-Liu's algorithm. The vertices of mesh (1) were not well aligned, except for the neighbors of the boundary vertices. This lack of alignment was intentional, because the objective was to verify the correctness of the implementation. The vertices of mesh (1) were created by successively applying the frontal algorithm and Lévy-Liu's algorithm in the L_2 norm.

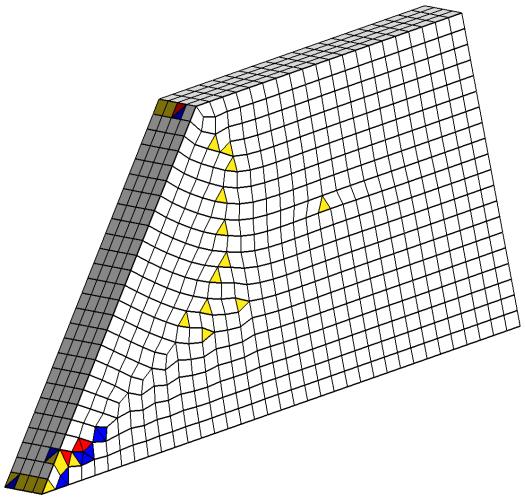


Figure 6.4: A mixed hexahedral mesh. The hexahedra are white, the prisms are yellow, the pyramids are red and the tetrahedra are blue.

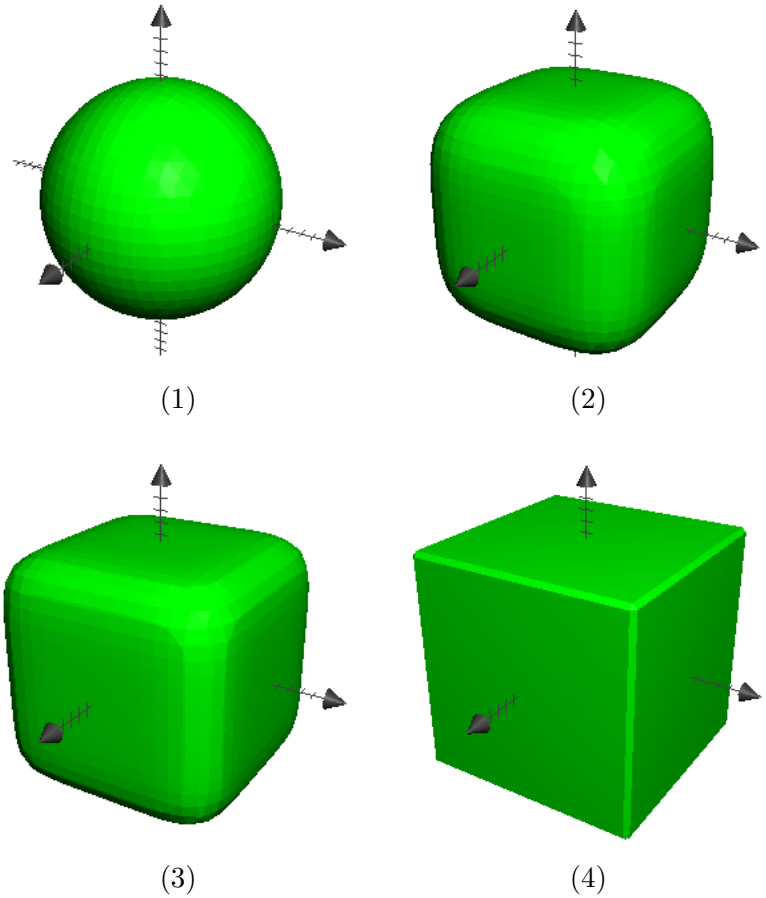
6.2 Distance measurements

The minimum acceptable distance between a prospective vertex and an existing one is equal to kh, with h being the local mesh size at the prospective vertex. In order to favor aligned vertices, the distances are computed with the infinity norm instead of the Euclidean norm. These two norms are illustrated on Fig. 6.5.

The exclusion area around each prospective vertex is therefore a cube. The infinity distance can be computed with equation 6.1.

$$||\boldsymbol{x} - \boldsymbol{y}||_\infty = \max(|x_1 - y_1|, |x_2 - y_2|, |x_3 - y_3|) \tag{6.1}$$

Contrary to the Euclidean norm, the infinity norm is anisotropic, i.e. it has an orientation. In the implementation, the orientation is fetched from the same cardinal direction field used for vertices creation. Eq. 6.2 defines the oriented infinity distance. $M_{\boldsymbol{x}}$ is the change of basis matrix going from the global system of coordinates to the local system of coordinates at $\boldsymbol{x}$. The axes of the local system of coordinates are parallel to the local cardinal directions.



(1) (2)

(3) (4)

Figure 6.5: Three-dimensional unit spheres in the L_2 (1), L_4 (2), L_6 (3) and L_∞ norms (4)

$$d_{\infty}^{orien}(\boldsymbol{x},\boldsymbol{y}) = ||M_{\boldsymbol{x}}(\boldsymbol{x}-\boldsymbol{y})||_{\infty} \tag{6.2}$$

In most cases, M is non-constant. As a consequence, the function d_{∞}^{orien} becomes non-symmetrical: $d_{\infty}^{orien}(\boldsymbol{x},\boldsymbol{y}) \neq d_{\infty}^{orien}(\boldsymbol{y},\boldsymbol{x})$. From a rigorous point of view, a function cannot be referred as a distance if it is not symmetrical.

Using the infinity distance instead of the Euclidean distance can increase the hexahedra percentage. The quarter cylinder illustrated on Fig. 6.6 provides an example of such an improvement.

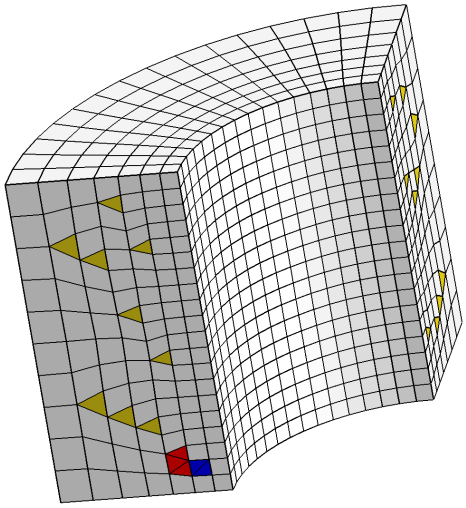


Figure 6.6: The percentage of hexahedra by number is equal to 76.03% for the L_{∞} norm (pictured) and 71.25% for the L_2 norm.

Like it was said before, kh represents the minimum distance to be tolerated between two vertices. Different values of k were tested on the geometry featured on Fig. 6.6. In each case, the percentage of hexahedra by number was computed. The resulting data is shown on table 6.1. As seen from these limited results, 0.7 appears to be an advantageous choice.

6.3 Spatial searches with R-trees

An R-tree data structure [31] is employed to verify the second condition of Section 6.1 efficiently. R-trees take into input straight side boxes

k	%
0.50	63.58
0.55	69.04
0.60	79.26
0.65	83.13
0.70	81.15
0.75	72.07
0.80	73.69
0.85	58.21
0.90	39.56

Table 6.1: The percentages of hexahedra by number obtained for different values of k

and can identify all boxes intersecting a given box. As said before, a prospective vertex is added to the cloud only if there is a certain unoccupied cubic area around it. The sides of this oriented exclusion cube have a length of $2{\cdot}0.7h$. The number of computations can be considerably diminished if each exclusion cube is enclosed into a box. When checking if a prospective vertex is too close to existing vertices, it is only necessary to compute the distance between pairs of vertices whose boxes intersect each other.

Fig. 6.7 shows two vertices: $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. For easier visualization, these vertices are on a plane. However, the principle remains exactly the same in three dimensions. $\boldsymbol{x}_1$ is the prospective vertex and $\boldsymbol{x}_2$ is a vertex already present in the cloud. The dotted lozenge around vertex $\boldsymbol{x}_1$ represents its exclusion cube. Even if the boxes intersect each other, the distance between $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ is sufficiently large. $\boldsymbol{x}_1$ can therefore be added to the cloud and to the queue.

Fig. 6.8 also shows two vertices. Again, the boxes intersect each other. This time, however, $\boldsymbol{x}_2$ is too close to $\boldsymbol{x}_1$ and $\boldsymbol{x}_1$ cannot be added to the cloud or to the queue.

On Figs. 6.7 and 6.8, $d_\infty^{orien}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ is not necessarily equal to $d_\infty^{orien}(\boldsymbol{x}_2, \boldsymbol{x}_1)$. The local mesh sizes at $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ can also be different. However, if $\boldsymbol{x}_2$ is outside the dotted lozenge of $\boldsymbol{x}_1$, it is considered sufficient.

The box surrounding a prospective vertex needs to be larger than its oriented exclusion cube of side $2kh$, with k ranging from 0 to 1. This

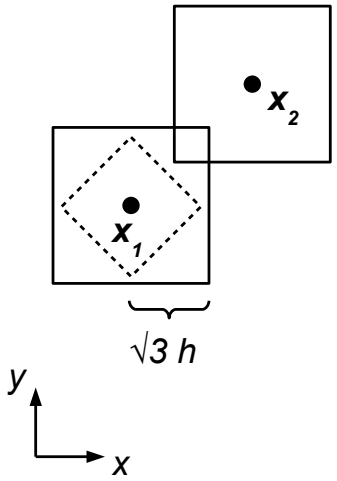


Figure 6.7: The distance between $\boldsymbol{x}_2$ and $\boldsymbol{x}_1$ is superior to $0.7h$, because $\boldsymbol{x}_2$ is outside the dotted lozenge of $\boldsymbol{x}_1$. h is the local mesh size at $\boldsymbol{x}_1$.

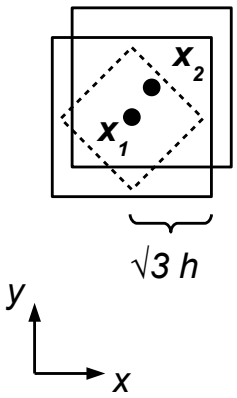


Figure 6.8: The distance between $\boldsymbol{x}_2$ and $\boldsymbol{x}_1$ is inferior to $0.7h$, because $\boldsymbol{x}_2$ is inside the dotted lozenge of $\boldsymbol{x}_1$.

condition is satisfied if the box is a cube of side $2\sqrt{3}h$. Let's assume that k is equal to 1. The distance between the farthest point on the surface of the exclusion cube and the prospective vertex is equal to $\sqrt{h^2 + h^2 + h^2}$ $= \sqrt{3}h$. If the box is a cube of side $2\sqrt{3}h$, then this point will always be inside the box.

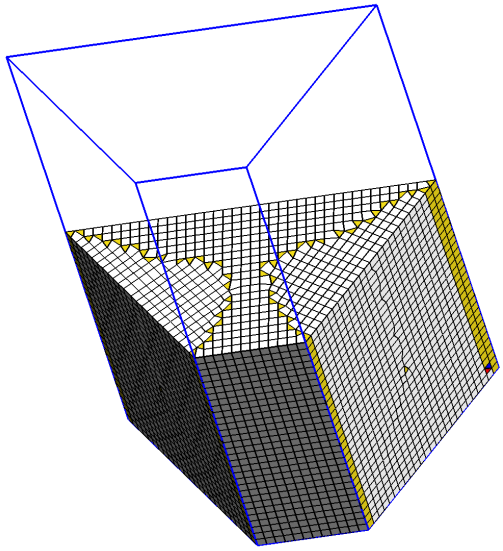
6.4 Cardinal direction field smoothing

As said in chapter 3, the cardinal directions at a precise point inside a given volume are equal to the cardinal directions of the nearest surfacic mesh vertex. Such a cardinal direction field will most likely contain discontinuities. However, a procedure invented by the Université catholique de Louvain researcher François Henrotte may prove capable of smoothing these discontinuities. According to this procedure, the cardinal directions inside a volume are represented by frames. A frame consists of six unit vectors oriented in the six cardinal directions (up, down, left, right, front and back).

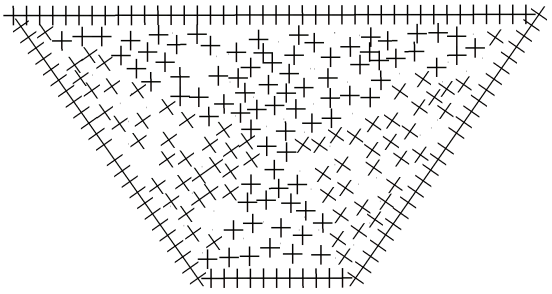
First, the volume is meshed and the closest neighbors to each vertex are identified. The frame of a given vertex can be transformed into the frame of one of its neighbors via a rotation. The smallest rotations are computed and stored as 3×3 matrices. These rotation matrices are then converted into the axis-angle representation. The axis-angle representation consists of a single vector. The vector acts as the axis of rotation and its modulus corresponds to the angle of rotation. A vector can be multiplied by a constant or added to another vector and still represents a rotation. The averages of the rotations transforming each frame into the frames of its neighbors are computed. These average rotations are subsequently applied to the frames. The procedure is repeated a certain number of times in order to smooth the cardinal direction field.

Figs. 6.9 and 6.10 both show a mixed hexahedral mesh and a slice of its three-dimensional cardinal direction field. The cardinal direction field of Fig. 6.9 is discontinuous and was generated with the method described in chapter 3. On the other hand, the cardinal direction field of Fig. 6.10 was smoothed with the relaxation procedure presented above. Ten iterations were performed.

The mixed hexahedral mesh of Fig. 6.9 contains 84.81% of hexahedra by number and 93.49% of hexahedra by volume. The mixed hexahedral mesh of Fig. 6.10 contains 80.85% of hexahedra by number and 93.01% of hexahedra by volume. Contrary to the expectations, the hexahedra

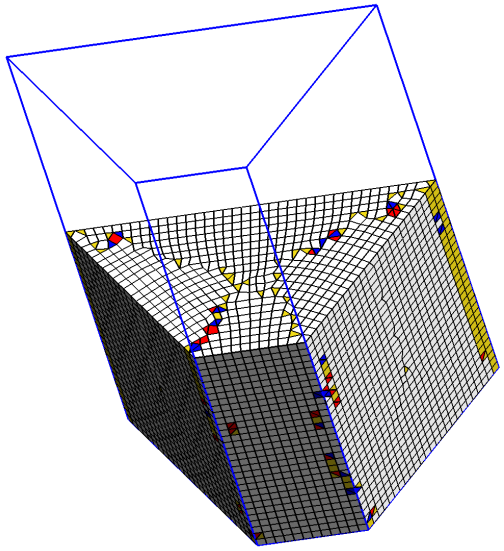


(1)

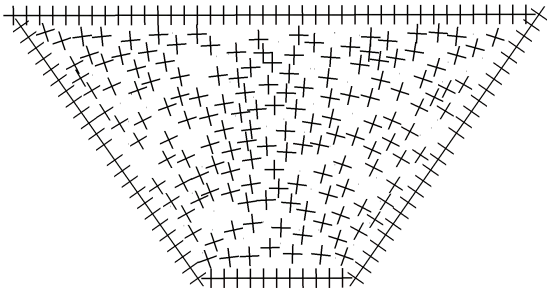


(2)

Figure 6.9: The mixed hexahedral mesh (1) is composed of 34,119 elements. It was generated using the discontinuous cardinal direction field (2).



(1)



(2)

Figure 6.10: The mixed hexahedral mesh (1) is composed of 35,915 elements. It was generated using the smoother cardinal direction field (2).

percentage did not increase as a result of the smoother cardinal direction field. The smoothing procedure was therefore not employed for the examples presented in this chapter.

The method presented in this section can also be applied to surfaces. In this case, it has an important advantage. It can generate correct cardinal direction fields for all surfaces. On the contrary, the method based on complex numbers described in chapter 3 would sometimes lead to poor results, especially for cylindrical surfaces bounded by circles. Such geometries often arise in bio-fluid simulations.

6.5 Results

This section presents several mixed hexahedral meshes created with the frontal algorithm and Yamakawa-Shimada's algorithm. Three quantities are used to evaluate the quality of the meshes: the percentage of hexahedra by number H_{nbr}, the percentage of hexahedra by volume H_{vol} and the average hex quality Q defined in Eq. 2.2. In general, H_{vol} is higher than H_{nbr}. All the data is compiled in table 6.2. The following convention is used throughout this section: the hexahedra are white, the prisms are yellow, the pyramids are red and the tetrahedra are blue.

Fig.	No. of vertices	H_{nbr}	H_{vol}	Q	T
6.11	126,922	62.91%	85.70%	0.95	488 s.
6.13	42,263	75.42%	92.85%	0.96	164 s.
6.14	11,648	58.16%	82.68%	0.94	112 s.
6.15 (1)	2,737	60.23%	83.02%	0.94	45 s.
6.15 (2)	6,006	84.54%	94.10%	0.96	80 s.
6.17	19,326	86.19%	94.93%	0.98	51 s.

Table 6.2: Mesh data. The execution time T was measured on a 2010 laptop computer. Lévy-Liu's algorithm was used for the meshes of Figs. 6.14 and 6.15.

Fig. 6.11 shows a mesh containing 142,466 elements. The mesh size is constant throughout the domain. In order to avoid altering the geometrical edges, hexahedra or prisms whose facets lie on two different geometrical faces are not created.

Each module of the mixed hexahedral mesh generation process shares a certain percentage of the total execution time. These percentages are

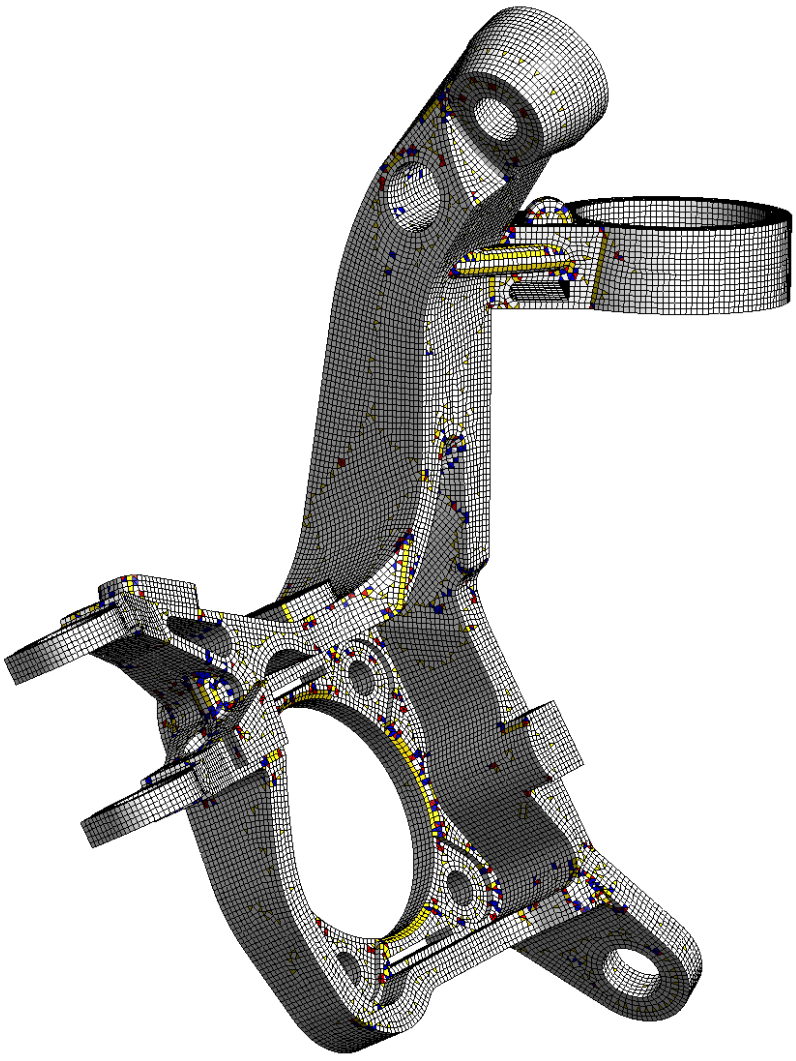


Figure 6.11: The mesh of a mechanical part composed of 250 geometrical faces

detailed on Fig. 6.12. The mesh illustrated on Fig. 6.11 was used for the analysis. For comparison, a tetrahedral mesh of equivalent density requires 55 seconds. The blue modules refer to tetrahedra recombination. Taken together, they are the most time-consuming. The green and yellow modules refer to volumic and surfacic vertices creation. The volumic vertices creation is dominated by octree queries. Eliminating unnecessary octree queries would certainly improve the performance.

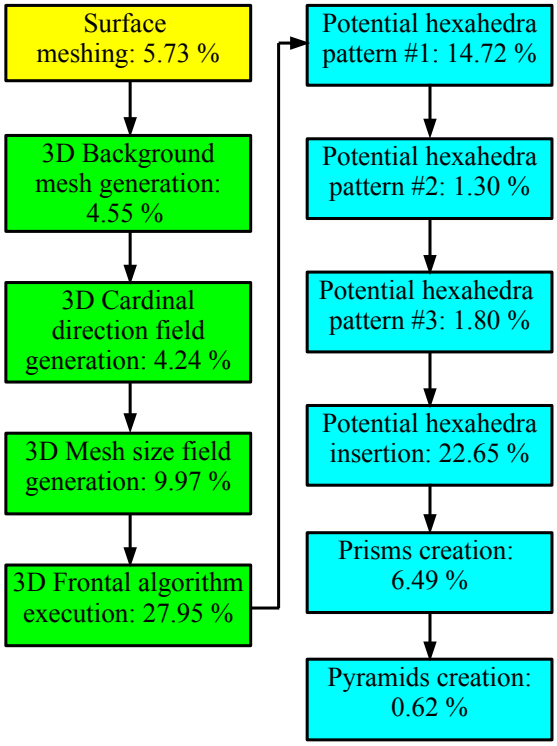
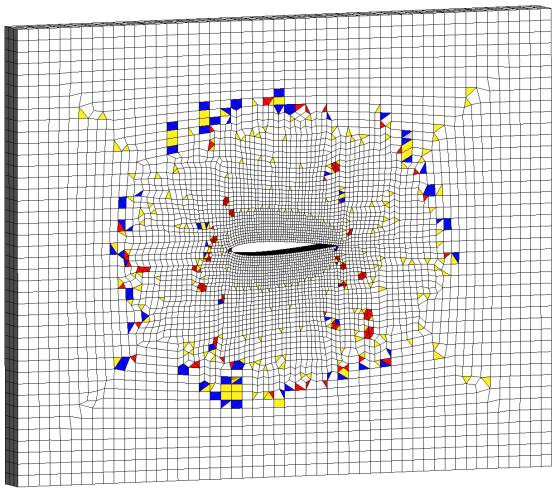


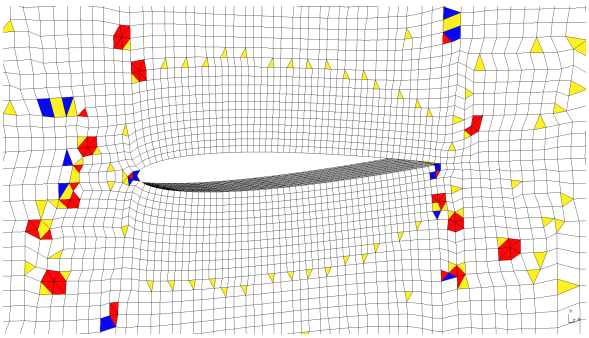
Figure 6.12: Global flowchart and time percentages

In Fig. 6.13, a rectangular parallelepiped surrounding a NACA airfoil is meshed with 43,094 elements. Subfigures (2) and (3) are zoomed images of (1). The presence of anisotropic hexahedra near the boundary would be required for resolving viscous flows.

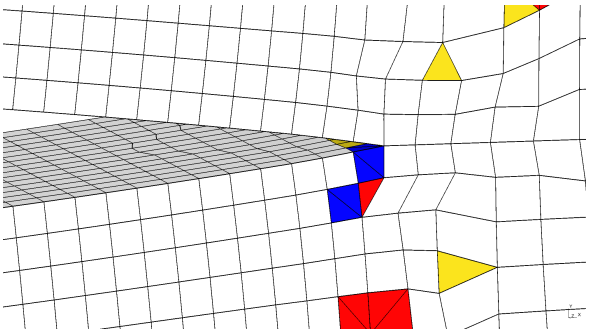
Fig. 6.14 is a cutaway view of the mixed mesh inside a cylinder. The mesh size is constant and the cardinal directions are radial.



(1)



(2)



(3)

Figure 6.13: The mesh of a NACA airfoil. (2) is a zoom of the airfoil and (3) is a zoom of the trailing edge.

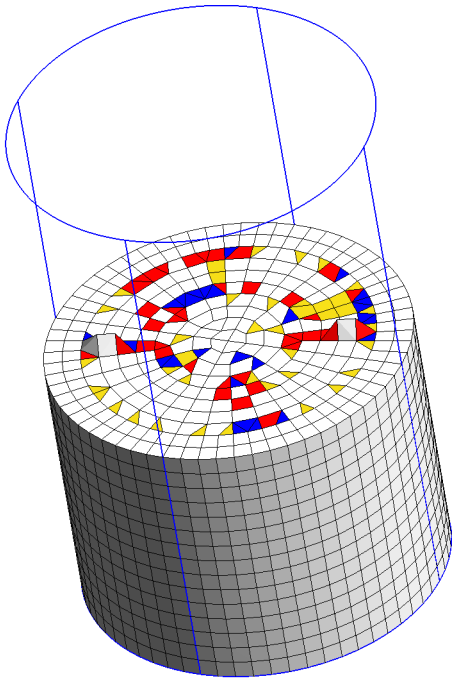
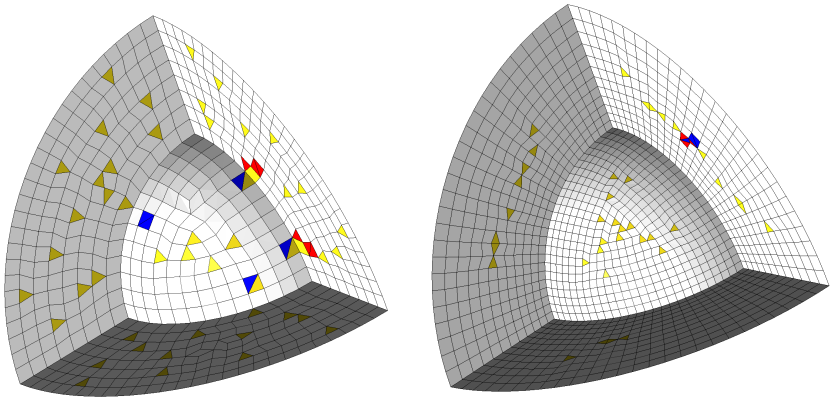


Figure 6.14: A cutaway view of a cylinder mesh. The blue lines illustrate the cylinder geometry.

Choosing a size field consistent with the geometry can improve the hexahedra percentage. For the hollow eighth ball shown on Fig. 6.15, a mesh size proportional to the radius is more suitable than a constant one.

The percentage of hexahedra can be improved by initiating the front from only a selected number of faces. In other words, the boundary mesh vertices of certain faces are not allowed to create prospective vertices. The cylinder mesh displayed on Fig. 6.16 constitutes an example of such a strategy. The front was initiated exclusively from the lower and upper disks. The mesh has the following statistics: $H_{nbr} = 81.88\%$, $H_{vol} = 91.44\%$ and $Q = 0.97$. The hexahedra percentages are higher than those of the previous cylinder.

A mesh convergence analysis was performed on the mechanical piece displayed on Fig. 6.17. The mechanical piece is made of steel and one of its extremities is fixed. The first and second frequencies were computed, as shown on Table 6.3. Mixed hexahedral meshes of various densities



(1) (2)

Figure 6.15: The meshes of two hollow eighth balls. The mesh size is constant in (1) and variable in (2).

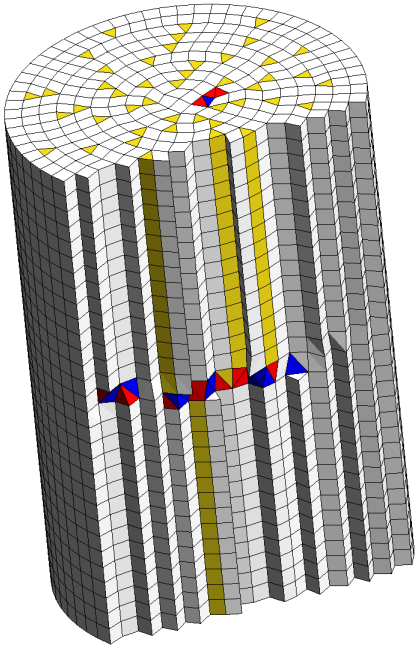


Figure 6.16: In this mesh, the front was initiated only from the lower and upper disks. As expected, there is a collision at the middle of the cylinder.

were employed. According to a similar finite element calculation performed on a structured mesh, the first frequency is equal to 757.76 Hz and the second one is equal to 1921.93 Hz. The results appear to converge. Fig. 6.18 shows the third mode of vibration of the piece.

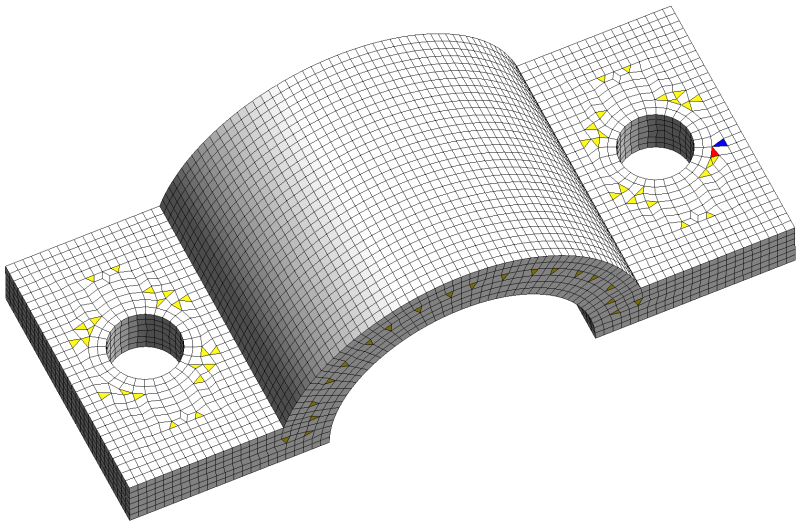


Figure 6.17: One of the mesh used for the convergence analysis. The mesh size h is equal to 1.0.

Mesh size	First frequency (Hz)	Second frequency (Hz)
2.0	770.11	1950.65
1.0	760.46	1928.52
0.5	757.55	1921.77
struct.	757.76	1921.93

Table 6.3: Mesh convergence analysis (credits: Gaëtan Compère)

Figs. 6.19, 6.20, 6.21, 6.22 and 6.23 show additional examples of mixed hexahedral meshes. Table 6.4 contains the corresponding mesh data. The CAD models of these meshes come from an online repository [72, 2, 32, 80].

Surfaces and volumes are meshed sequentially. Additionally, the two surface meshes bounding a thin region may not be identical. Many non-hexahedral elements can therefore be created in this situation. The

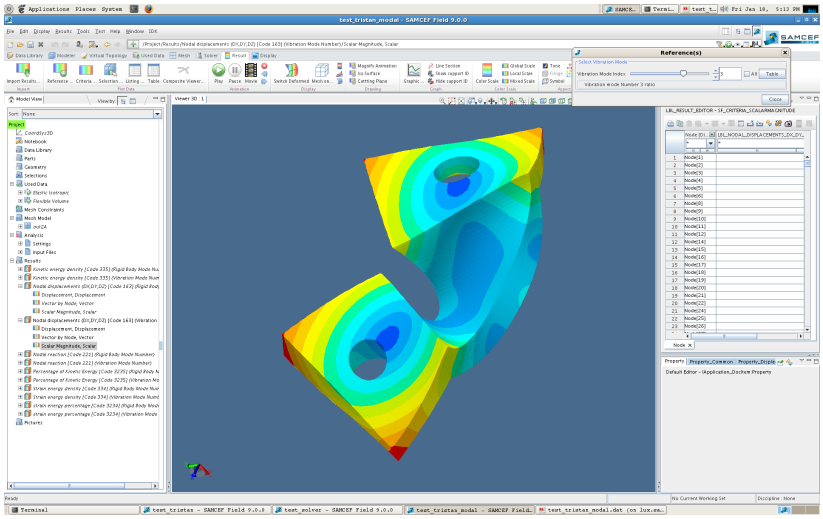
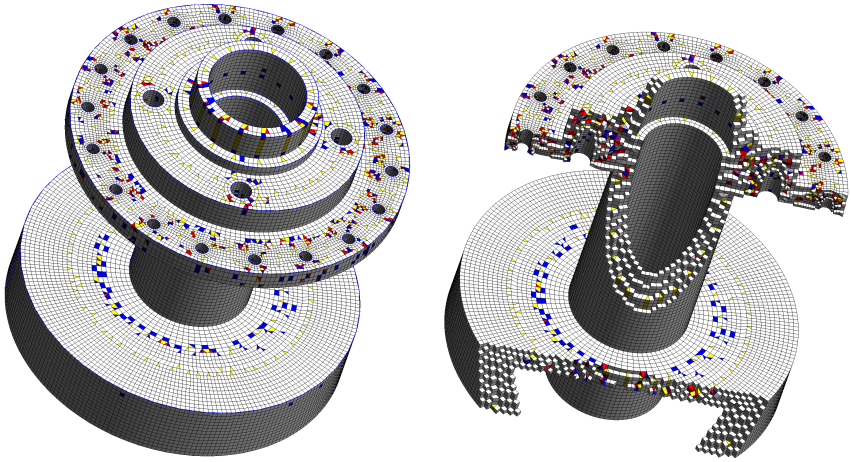
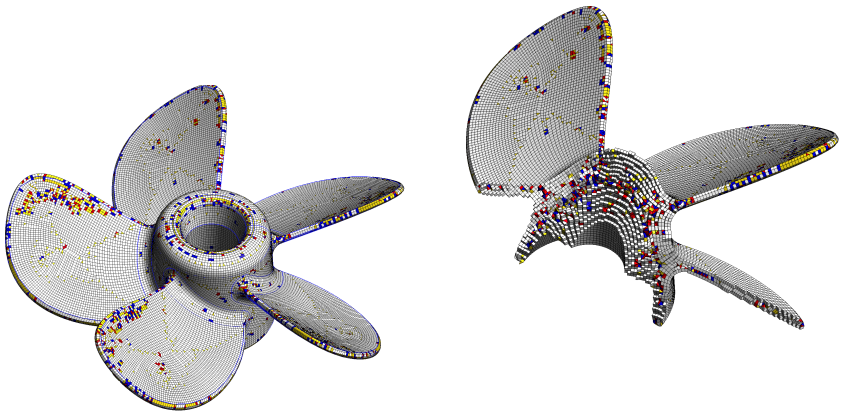


Figure 6.18: A visualization of the third mode of vibration (credits: Gaëtan Compère)



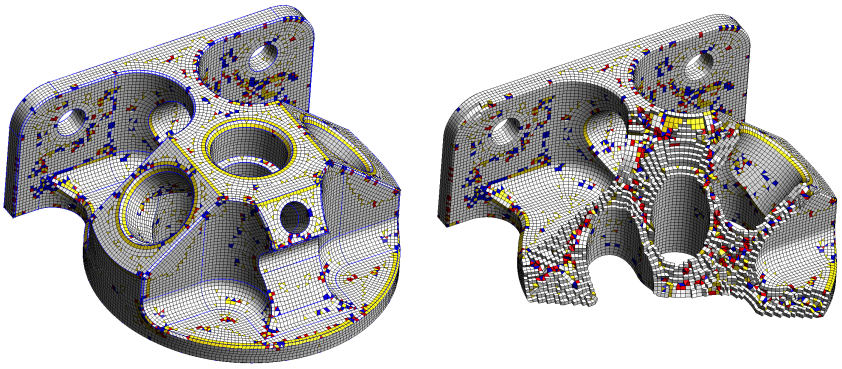
(1) (2)

Figure 6.19: A mechanical part (credits: Jean-François Remacle)



(1) (2)

Figure 6.20: A marine propeller (credits: Jean-François Remacle)

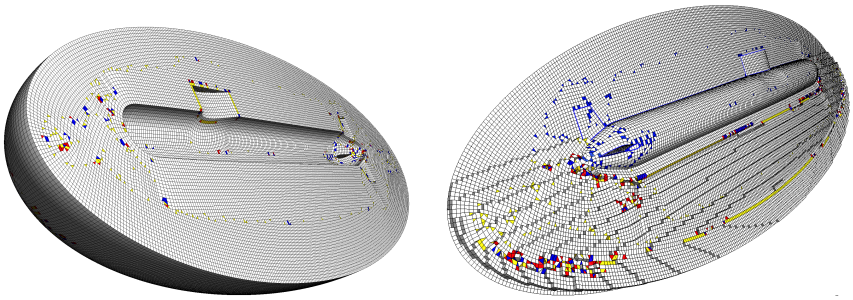


(1) (2)

Figure 6.21: A mechanical part (credits: Jean-François Remacle)

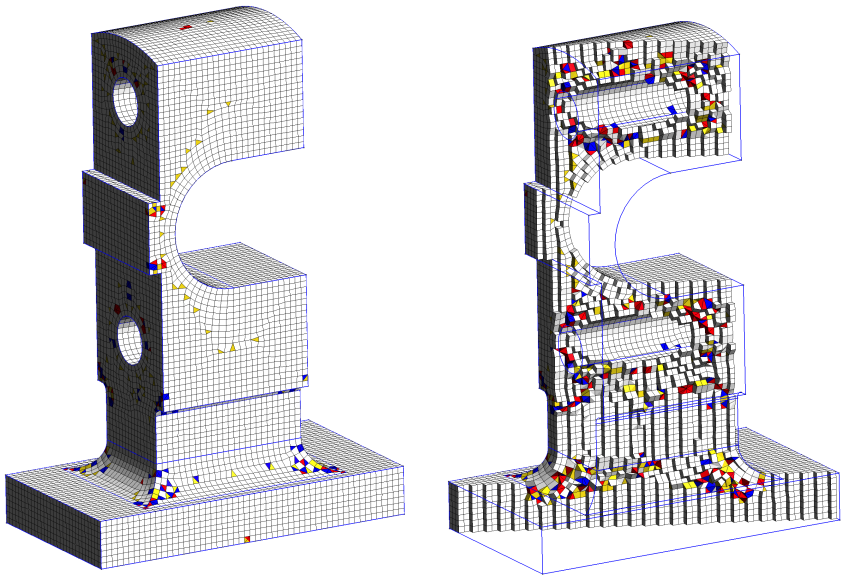
Fig.	No. of vertices	H_{vol}	T
6.19	133,436	89.74%	247 s.
6.20	133,678	83.65%	268 s.
6.21	102,946	78.55%	225 s.
6.22	598,514	90.28%	1287 s.
6.23	47,077	90.34%	74 s.

Table 6.4: Mesh data. The execution time T was measured on a 2012 laptop computer.



(1) (2)

Figure 6.22: A submarine model (credits: Jean-François Remacle)



(1) (2)

Figure 6.23: A mechanical part (credits: Jean-François Remacle)

algorithm is usually less effective for geometrical models featuring thin regions. For example, the trailing edges of the blades shown on Fig. 6.20 are very thin. As expected, they contain high proportions of non-hexahedral elements. The areas at the back of Fig. 6.21 near the two circles are also very thin.

The Lloyd's energy minimization procedure described in chapter 5 was applied to a few vertices distributions created by the frontal algorithm. In many cases, the hexahedra percentages did not increase. They could even decrease. This lack of improvement may be caused by the fact that there is too much constraints on the optimization: the vertices and their neighbors are unmovable. In the original implementation presented in Lévy and Liu's article [45], even boundary mesh vertices could move. This feature requires the clipping of three-dimensional Voronoi diagrams. However, for finite element calculations, edge vertices should remain unmovable in order to avoid altering the initial geometry.

Currently, each vertex attempts to create six other vertices in the six cardinal directions during the course of the frontal algorithm. The creation of twenty-six vertices instead of only six could also be considered. Visually, the parent vertex would be at the center of a $2\times2\times2$ hexahedral grid. The grid would be aligned with the local cardinal directions and would contain twenty-seven vertices. Twenty-six of them would be prospective vertices. In most cases, only a small fraction of these vertices would actually be added to the cloud, because existing vertices would already occupy the space. Nevertheless, it would allow the identification in advance of many potential hexahedra. This information could be used to increase the efficiency of Yamakawa-Shimada's algorithm.

The next chapter will present a surfacic version of the frontal approach, whose purpose is to create aligned vertices on surfaces. The surfaces are not required to possess a conformal parametrization, which is an advantage.

Chapter 7

The frontal approach for surfaces

The frontal approach for volumes was discussed in the previous chapter. The frontal approach can also be applied to three-dimensional surfaces. In a perfect quadrilateral mesh, each vertex is surrounded by four vertices: one at the left, one at the right, one above and one below. The surfacic frontal algorithm is based on this observation. Each vertex attempts to create four vertices in the four cardinal directions. A prospective vertex is added to the cloud only if it is inside the domain and not too close to the vertices already present in the cloud. The use of R-trees guarantees efficient spatial searches [31]. Because three-dimensional surfaces are usually non-planar, parametrization techniques are employed [67, 50, 51].

All the boundary mesh vertices on the edges of the domain are initially pushed into a queue. They are then popped in order. Each vertex popped tries to create four vertices. If a prospective vertex is inside the domain and outside the exclusion area of the existing vertices, it is added to the cloud and to the queue. An octree data structure can efficiently determine if a vertex is inside the domain or not. Once the process is over, the cloud of vertices is triangulated in the parametric plane using a Delaunay method [67, 50, 51]. Just like in three dimensions, the con-

nectivity is only constructed at the end. The Blossom-quad algorithm is employed to recombine the triangles.

This chapter is divided into four sections. The first section discusses the vertices creation. The second section describes how the distances are measured. The third section briefly explains the use of R-trees. The last section presents the results.

7.1 Vertices creation

As shown on Fig. 7.1, each vertex attempts to create four vertices in the four cardinal directions at a distance h from itself. The cardinal directions are computed with the method described in chapter 3.

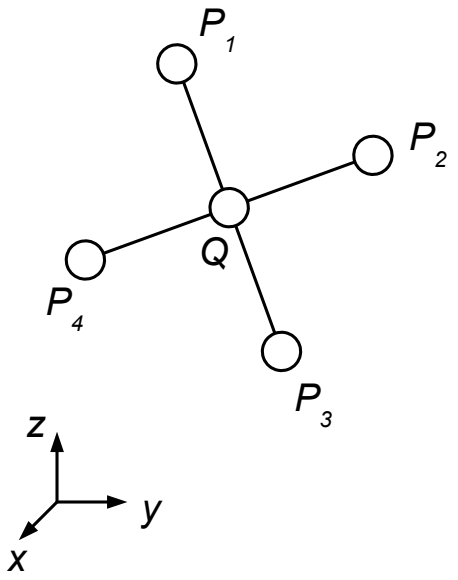


Figure 7.1: The vertex Q tries to create the vertices P_1, P_2, P_3 and P_4.

M is the parametrization of the three-dimensional surface by a plane [67, 50, 51]. M is not required to be conformal, which increases the applicability of the algorithm.

$$\mathrm{M} : (u, v) \mapsto (x, y, z) \tag{7.1}$$

The coordinates of the four vertices illustrated on Fig. 7.1 are evaluated in the parametric plane. Each vertex needs to know its parametric coordinates, because the Delaunay triangulation will be later performed in the parametric plane.

Let $\boldsymbol{t}$ be a tangent vector to the surface. It is located at the point (x, y, z) and its components are equal to $(\Delta x, \Delta y, \Delta z)$.

$$\boldsymbol{t}|_{(x,y,z)} = (\Delta x, \Delta y, \Delta z) \tag{7.2}$$

The vector $\boldsymbol{t}^*$ is the image of $\boldsymbol{t}$ in the parametric plane. It is located at the point (u, v) and its components are equal to $(\Delta u, \Delta v)$.

$$\boldsymbol{t}^*|_{(u,v)} = (\Delta u, \Delta v) \tag{7.3}$$

$\boldsymbol{t}$ can be obtain from $\boldsymbol{t}^*$ with the following equation.

$$\underbrace{(\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u})}_{\boldsymbol{t_1}} \Delta u + \underbrace{(\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v})}_{\boldsymbol{t_2}} \Delta v = (\Delta x, \Delta y, \Delta z) \tag{7.4}$$

$\boldsymbol{t_1}$ and $\boldsymbol{t_2}$ are the first derivatives of the parametrization M.

$$\boldsymbol{t_1}\Delta u + \boldsymbol{t_2}\Delta v = \boldsymbol{t} \tag{7.5}$$

By taking the dot product with $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$, two new equations can be derived from equation 7.5.

$$\boldsymbol{t_1} \cdot \boldsymbol{t_1}\Delta u + \boldsymbol{t_1} \cdot \boldsymbol{t_2}\Delta v = \boldsymbol{t_1} \cdot \boldsymbol{t} \tag{7.6}$$

$$\boldsymbol{t_2} \cdot \boldsymbol{t_1}\Delta u + \boldsymbol{t_2} \cdot \boldsymbol{t_2}\Delta v = \boldsymbol{t_2} \cdot \boldsymbol{t}$$

These two equations can be rewritten in matrix form.

$$\begin{bmatrix} \boldsymbol{t_1} \cdot \boldsymbol{t_1} & \boldsymbol{t_1} \cdot \boldsymbol{t_2} \\ \boldsymbol{t_2} \cdot \boldsymbol{t_1} & \boldsymbol{t_2} \cdot \boldsymbol{t_2} \end{bmatrix} \begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix} = \begin{bmatrix} \boldsymbol{t_1} \cdot \boldsymbol{t} \\ \boldsymbol{t_2} \cdot \boldsymbol{t} \end{bmatrix} \tag{7.7}$$

The images in the parametric plane of the cardinal directions $\boldsymbol{d_1}$, $\boldsymbol{d_2}$, $\boldsymbol{d_3}$ and $\boldsymbol{d_4}$ are found using equation 7.7. $\boldsymbol{d_1}$, $\boldsymbol{d_2}$, $\boldsymbol{d_3}$ and $\boldsymbol{d_4}$ are orthonormal in three dimensions. However, the images will not necessarily be orthonormal. The coordinates of the four vertices P_1, P_2, P_3 and P_4 are calculated with the following equations.

$$\boldsymbol{p_1^*} = \boldsymbol{q}^* + h\boldsymbol{d_1^*} \tag{7.8}$$

$$\boldsymbol{p_2^*} = \boldsymbol{q}^* + h\boldsymbol{d_2^*}$$

$$\boldsymbol{p_3^*} = \boldsymbol{q}^* + h\boldsymbol{d_3^*}$$

$$\boldsymbol{p_4^*} = \boldsymbol{q}^* + h\boldsymbol{d_4^*}$$

An alternative method to determine the coordinates of P_1, P_2, P_3 and P_4 is to use circle-surface intersections in three dimensions. For example, let's consider the vertex P_1. P_1 could be created by intersecting a circle of radius h with the surface. The circle would lie on the plane defined by the vector $\boldsymbol{d_1}$ and the normal $\boldsymbol{n}$. Fig. 7.2 illustrates this method.

Figs. 7.3, 7.4 and 7.5 show three examples of meshes generated with the frontal algorithm and Blossom-quad algorithm. The vertices seem to be well aligned. $\eta(q)$ designates the average quality and was defined in equation 2.1.

7.2 Distance measurements

Like it was said before, a prospective vertex is added to the cloud and to the queue only if it is inside the domain and not too close to an existing vertex. The distances are measured in the L_∞ norm instead of the L_2 norm in order to favor alignment.

Two points geodesically far from each other on a three-dimensional surface can be in fact very close. This phenomenon could prevent valid prospective vertices from being added to the cloud. However, parametrization is capable of solving this issue.

Let's assume that $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ are two orthonormal vectors selected among the four cardinal directions at a certain point on a three-dimensional surface. The vectors $\boldsymbol{v_1^*}$ and $\boldsymbol{v_2^*}$ are the images of $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ in the parametric plane. They are computed with equation 7.7.

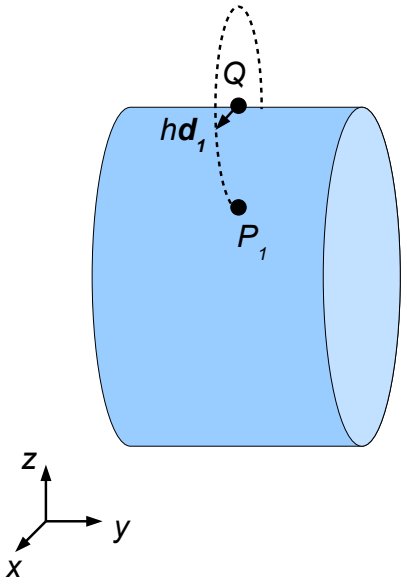


Figure 7.2: The prospective vertex P_1 is found by intersecting the surface with a circle.

$\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ are always orthonormal. However, the parametrization may not be conformal. In this case, $\boldsymbol{v_1^*}$ and $\boldsymbol{v_2^*}$ will not be orthogonal and the exclusion area will be a trapezoid.

M is the change of basis matrix going from the parametric plane to a system of coordinates whose unit vectors are $\boldsymbol{v_1^*}$ and $\boldsymbol{v_2^*}$.

$$M = \begin{bmatrix} \boldsymbol{v_1^*} & \boldsymbol{v_2^*} \end{bmatrix}^{-1} \tag{7.9}$$

A prospective vertex is added to the cloud and to the queue only if relation 7.10 is satisfied, with $\boldsymbol{x}^*$ being the parametric position of the prospective vertex and $\boldsymbol{y}^*$ being the parametric position of any other vertex already present in the cloud. k is equal to 0.7 and M is the change of basis matrix at $\boldsymbol{y}^*$.

$$||M(\boldsymbol{x}^* - \boldsymbol{y}^*)||_\infty > kh \tag{7.10}$$

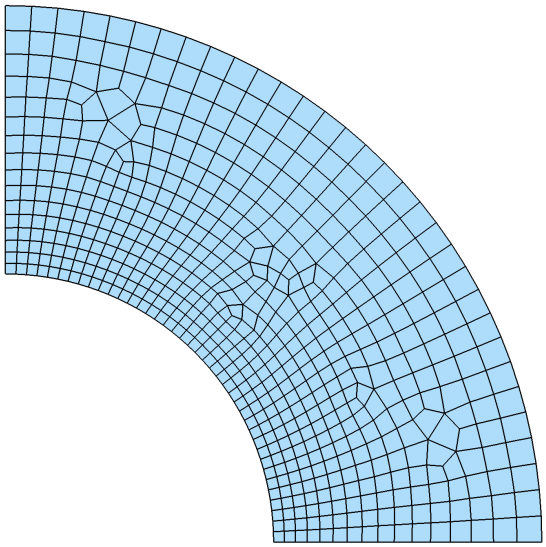


Figure 7.3: A two-dimensional mesh. $\eta(q) = 0.92$.

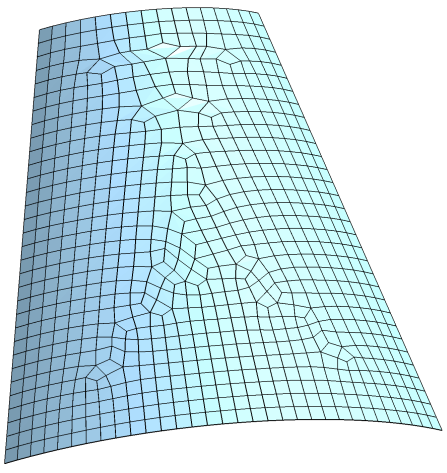


Figure 7.4: A surface mesh. $\eta(q) = 0.91$.

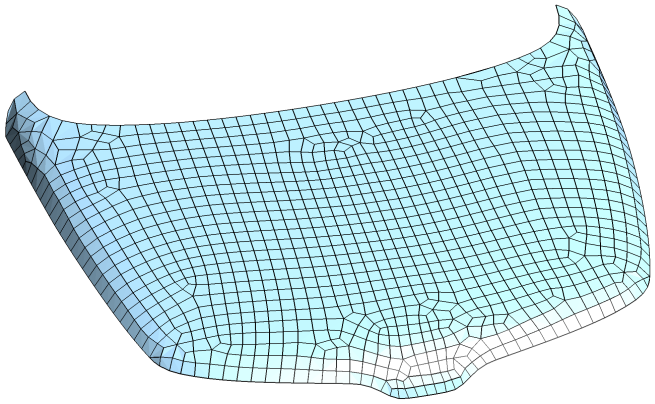


Figure 7.5: The mesh of a car hood. $\eta(q) = 0.88$.

As an example, let's assume that $\boldsymbol{v}_1^*$ is equal to (1,1) and $\boldsymbol{v}_2^*$ is equal to (-0.25,0.5). The matrix M then becomes:

$$M = \begin{bmatrix} 1 & -0.25 \\ 1 & 0.5 \end{bmatrix}^{-1} = \begin{bmatrix} 2/3 & 1/3 \\ -4/3 & 4/3 \end{bmatrix} \tag{7.11}$$

Fig. 7.6 shows the unit circle of this particular distance in the parametric plane.

The parametrization does not necessarily preserve the mesh size. However, the norms are not directly evaluated in the parametric plane. They are evaluated in a coordinate system whose unit vectors are $\boldsymbol{v}_1^*$ and $\boldsymbol{v}_2^*$. The mesh size in the relation 7.10 is therefore identical to the mesh size in the three-dimensional space.

Fig. 7.7 shows the first layers of vertices created by the frontal algorithm. The vertices are in blue. Each vertex has a trapezoidal exclusion area.

7.3 Spatial searches with R-trees

Just like in three dimensions, each exclusion area is enclosed into a box. R-trees can efficiently identify which boxes overlap a given box [31]. If there is no box intersection between two vertices, then it is not necessary to evaluate their distance.

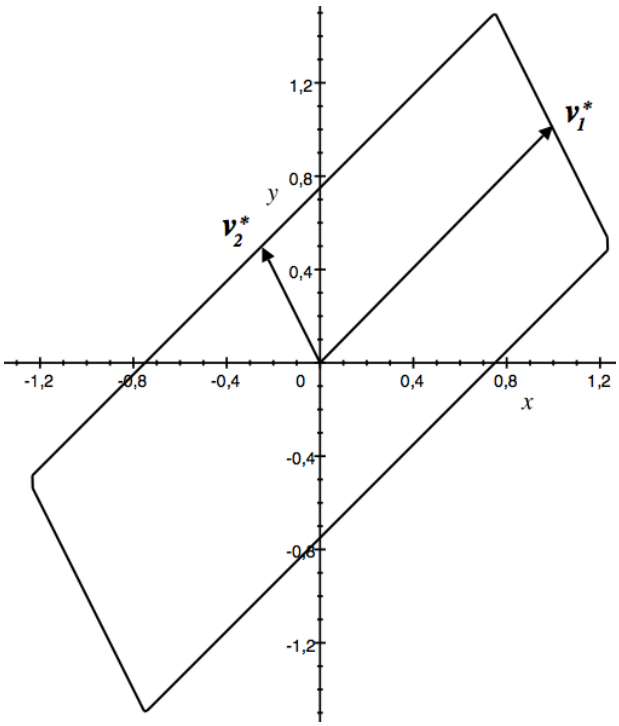


Figure 7.6: The unit circle of a trapezoidal distance in the parametric plane

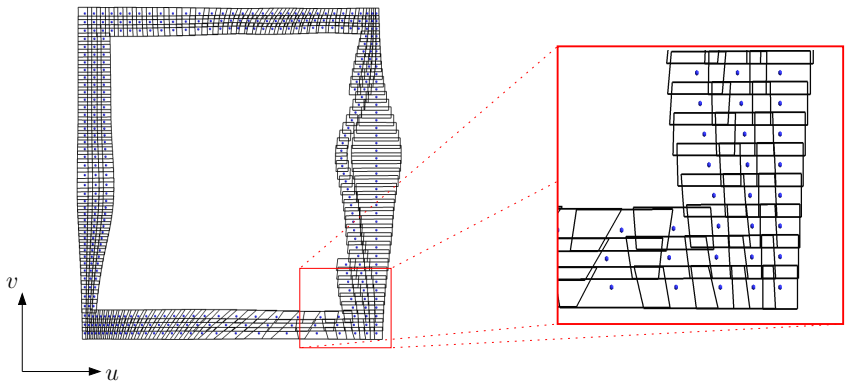


Figure 7.7: The first two or three layers of vertices created by the frontal algorithm. The trapezoids represent the minimum acceptable distance in the parametric plane (credits: Jean-François Remacle).

Trapezoidal exclusion areas are composed of four points. As shown on Fig. 7.8, the minimum u, maximum u, minimum v and maximum v of these four points determine the dimensions of the enclosing box.

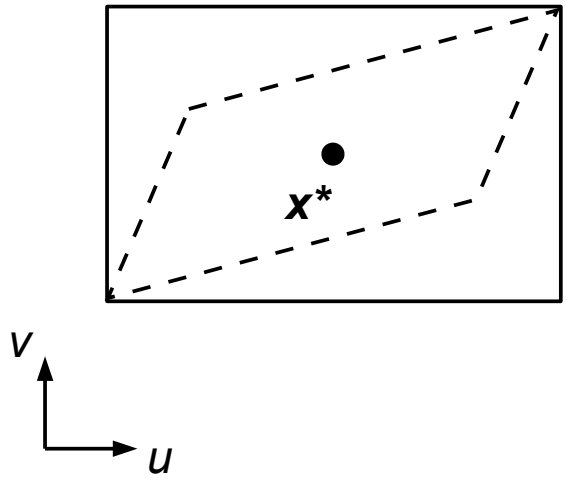


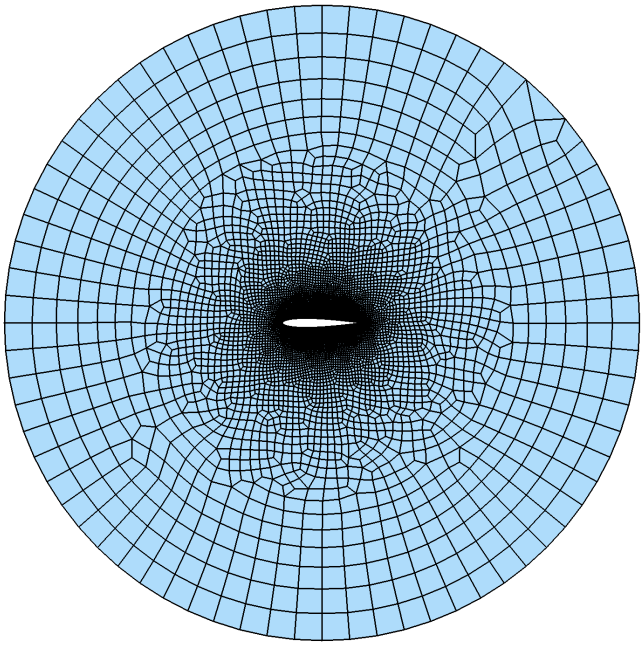
Figure 7.8: The dotted line represents the exclusion area of vertex $\boldsymbol{x}^*$ in the parametric plane.

7.4 Results

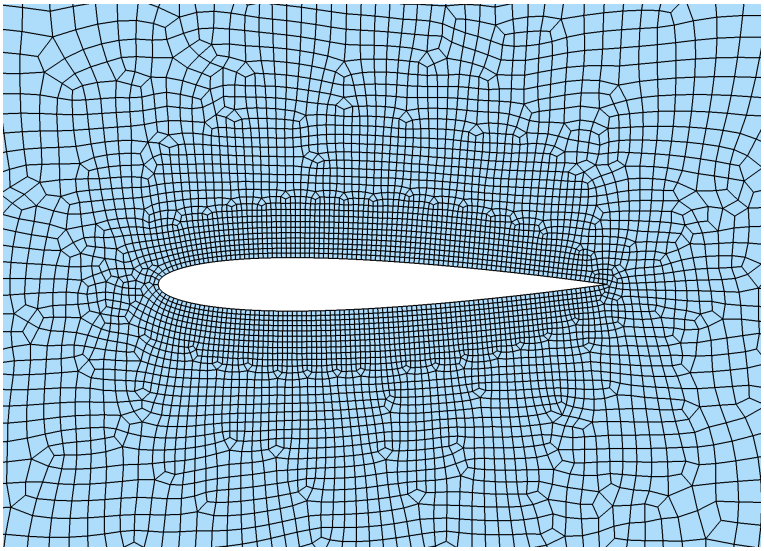
The three following meshes were generated with the frontal algorithm and the Blossom-quad algorithm. $\eta(q)$, T and N are the symbols for the average quality, the total execution time and the number of elements. $\eta(q)$ was defined in equation 2.1. The mesh of Fig. 7.9 has a variable mesh size. The presence of anisotropic quadrilaterals near the boundary would be required for resolving viscous flows.

The mesh of Fig. 7.11 contains 250 geometric faces. Out of the 46,426 elements, 160 are triangles. The total execution time was variable from one attempt to another, but not because of the frontal algorithm.

In all cases, the vertices seem to be well aligned.



(1)



(2)

Figure 7.9: The mesh of a NACA airfoil. (2) is a zoom of (1). $\eta(q) = 0.89$, $T = 2.44$ s. and $N = 7{,}119$.

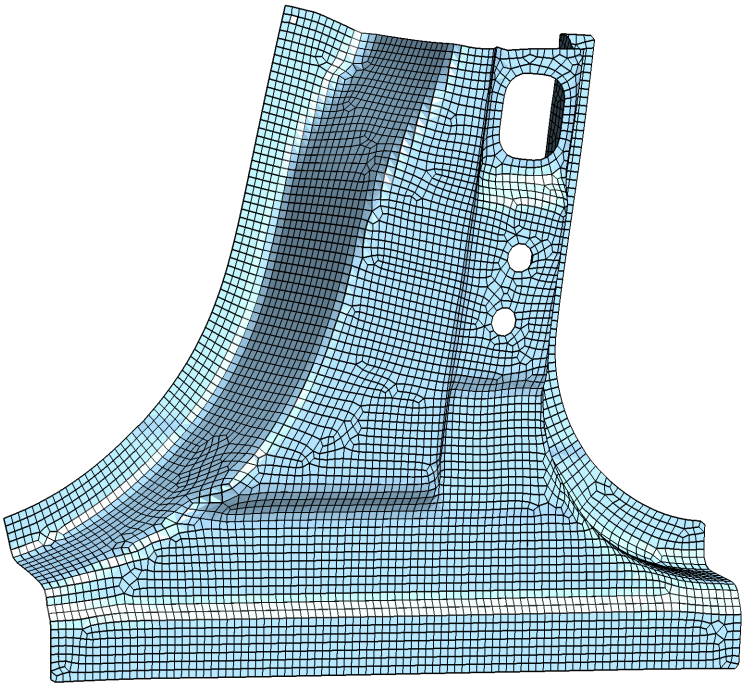


Figure 7.10: The mesh of a car body part. $\eta(q) = 0.93$, $T = 43.52$ s. and $N = 5{,}987$.

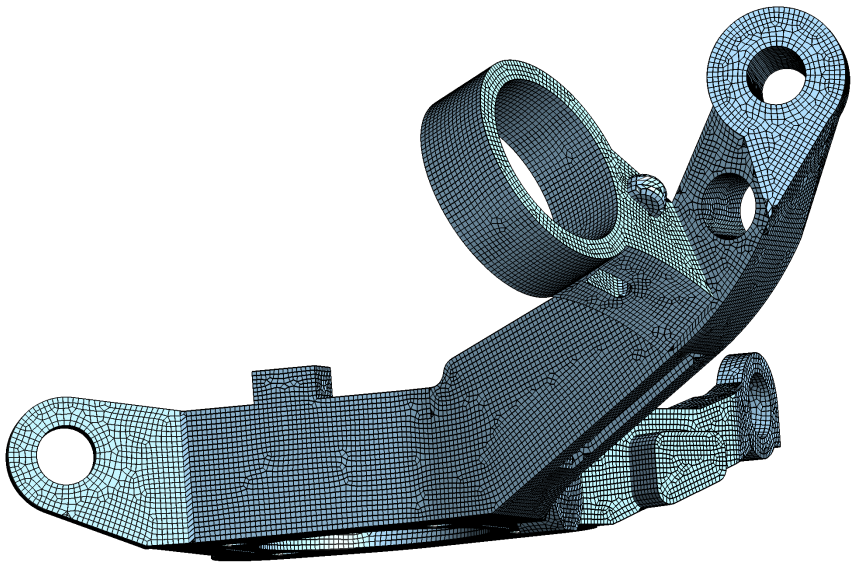


Figure 7.11: The mesh of a mechanical part. $\eta(q) = 0.86$, $T = 275.17$ s. and $N = 46{,}426$.

Chapter 8

Conclusion

The purpose of the Blossom-quad algorithm is to recombine triangles into quadrilaterals in order to create non-uniform quadrilateral meshes. In chapter 2, aligned vertices distributions were shown to lead to higher quality quadrilaterals and better orientations. As a consequence, automatic methods capable of producing aligned vertices were sought. Both the frontal and Lévy-Liu's algorithms are satisfying options. Lévy-Liu's algorithm is an iterative method based on the minimization of Lloyd's energy in the L_p norm. The two-dimensional implementation described in this text uses Gauss integration techniques and does not modify the edge meshes. Three-dimensional surfaces can also be treated, but require conformal parametrization. The implementation has two drawbacks. The first one is the execution time. Optimizing large meshes can take long. The second one is the non-globality of the minimums. The implementation can only find local energy minimums, not global ones. The frontal algorithm is very different. It does not move vertices, it creates them in a frontal manner. The front is guided by cardinal direction fields and size fields. The connectivity is only built at the end of the process. The frontal algorithm has the advantage of being very simple. Moreover, conformal parametrization is not required. R-tree data structures are employed for efficiency.

Yamakawa-Shimada's algorithm recombines tetrahedra into hexahedra in order to generate non-uniform mixed hexahedral meshes. As

explained in chapter 2, aligned vertices distributions tend to lead to higher hexahedra percentages. Evidently, a recombination rate of 100% would have been preferable, but is not attainable for most geometries. Therefore, the final meshes usually contain prisms, pyramids and tetrahedra in addition to the hexahedra. As a side effect, aligned vertices distributions also result in higher quality hexahedra and better orientations. Regarding three-dimensional domains, the volumes are always meshed after the surfaces. Consequently, the vertices alignment inside the volumes is affected by the vertices alignment on the surfaces. The hexahedra percentage is wished to be as high as possible. An implementation minimizing Lloyd's energy in the L_p norm was first tested. Again, Gauss integration was employed. However, even if the method was able to improve the alignment, the execution time was too large. As an alternative, the frontal approach was investigated. The frontal algorithm has the advantage of being simple to implement even in three dimensions. It is capable of creating aligned vertices distributions within a reasonable execution time. The hexahedra percentages are superior than those achieved by Yamakawa-Shimada's algorithm alone. One drawback is the absence of guarantees regarding the hexahedra percentage. It varies according to the geometry and the mesh size field. Another drawback is the presence of non-conformities in the final meshes. More specifically, certain quadrilateral faces can be adjacent to two triangular faces.

Lévy and Liu's original implementation can move simultaneously surfacic and volumic vertices [45]. As a result, the optimization is less constrained and may lead to lower energy minimums. Very good results were achieved [45]. For example, the complex model shown on Fig. 8.1 was partitioned into 113,000 elements, out of which 63.72% were hexahedra. Only 23 minutes were required.

Certain three-dimensional domains feature thin regions in comparison with the local mesh size. In addition, the surfacic vertices resting on the upper and lower faces of these regions may not concord. Because the volumic vertices are created perpendicularly to the surfaces, this lack of concordance can lead to low proportions of hexahedra. Fig. 8.2 shows a cylindrical bowl with a thin bottom. The edges vertices of the smaller upper disk may not concord with the surfacic vertices of the lower larger disk. If the mesh size is comparable to the thickness of the bottom, then the hexahedra percentage will be low in this area. Front collisions in thin regions is one of the drawback of the frontal algorithm.

The mixed hexahedral meshes generated by Yamakawa-Shimada's algorithm often contain non-conformities. However, these non-

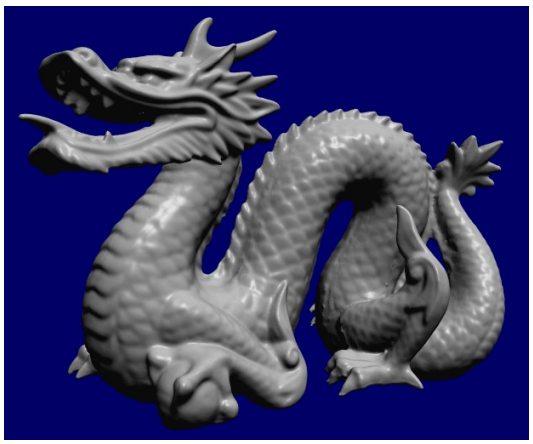
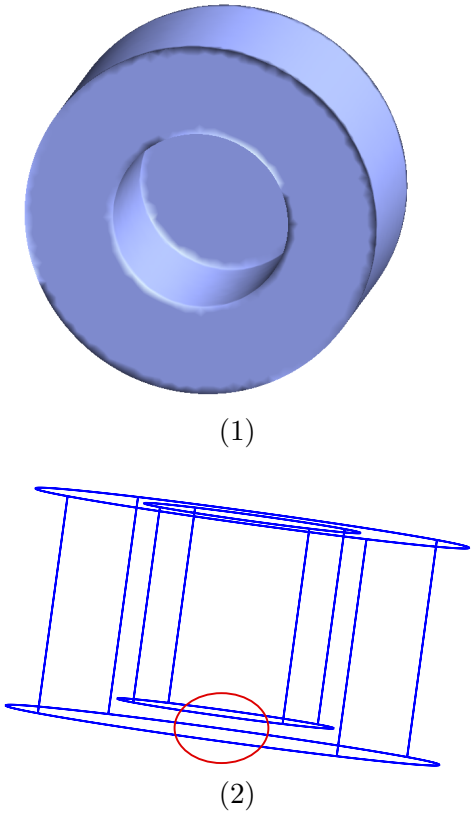


Figure 8.1: A complex model created by three-dimensional scanning (credits: [16, 1])

conformities can be repaired by Owen-Canann-Saigal's algorithm [62]. The resulting meshes should then be completely conformal, as well as free of overlaps or gaps between elements. These assumptions can be verified by comparing volumes. The volume of the domain should be equal to the sum of the elements volumes. The domain needs to have a very simple shape, such as a cubic one, in order to allow easy computation of its exact volume. This test can be employed to check the validity of the implementation.

As required by finite element solvers, mesh elements need to have positive Jacobians. The verification can be difficult because certain elements such as hexahedra have non-constant Jacobians. However, a test based on Bézier coefficients has been described in the literature [39]. If an element passes this test, then its Jacobian is guaranteed to be positive throughout its volume [39]. The validity of the mixed hexahedral meshes presented in this work is not certain. Nevertheless, the Bézier test could lift this uncertainty. For example, neighboring tetrahedra leading to potential hexahedra failing the test should not be merged.

As said in chapter 2, Yamakawa-Shimada's algorithm starts by identifying all potential hexahedra. These potential hexahedra could be viewed as the nodes of a graph [40]. Following this representation, two nodes are connected by an edge only if they are incompatible with each other. For example, two potential hexahedra that share a common tetrahedron are incompatible. Two potential hexahedra that lead to a non-



(1) (2)

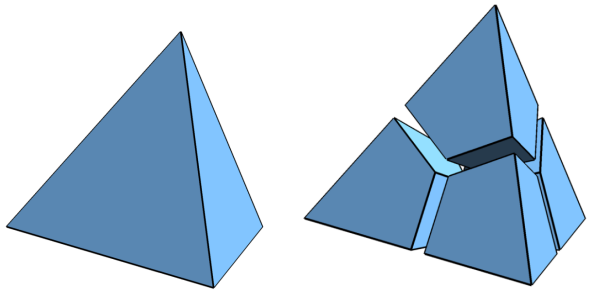
Figure 8.2: A cylindrical bowl with a thin bottom (1). The red circle indicates the location of the thin region on the CAD model (2).

conformity are also incompatible. In order to maximize the percentage of hexahedra, the largest set of nodes not connected amongst them needs to be selected. In computer science, such a set is known as a maximum independent set [5]. The maximum independent set problem is NP-hard [5]. No polynomial algorithm capable of solving a NP-hard problem has ever been discovered [30]. The use of heuristics is therefore inevitable. On another note, a weight could be assigned to each node, such as the quality of its potential hexahedron. In this case, the objective would be to find the independent set maximizing the sum of the weights [4].

The maximum independent set problem is related to the maximum clique problem [58]. In the maximum clique problem, the objective is to find the largest set of nodes connected with all the other nodes of the

set [58]. The maximum independent set of a graph is in fact the maximum clique of its complimentary graph [58, 4]. As expected, finding a maximum clique is also a NP-hard problem [58]. Polynomial constant factor approximation algorithms have two characteristics. They run in polynomial time and the quality of their solutions is guaranteed to be higher than a certain threshold [25]. However, finding a polynomial constant factor approximation algorithm for the maximum clique problem would prove the equivalence of the classes P and NP [53].

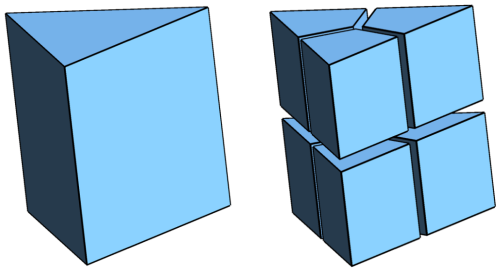
By subdividing the various elements of a mixed hexahedral mesh, it may be possible to generate a full hexahedral mesh. Figs. 8.3, 8.4 and 8.5 show respectively a tetrahedron, a prism and an hexahedron being subdivided into a certain number of hexahedra. The quadrilateral faces are assumed to be divided into four quadrilaterals and the triangular faces are assumed to be divided into three quadrilaterals. In this situation, a pyramidal element cannot be subdivided in less than 88 hexahedra [89]. Some of these hexahedra are going to be very small and the mesh size will not be respected. However, pyramids are necessary in order to maintain the conformity of the mesh. In fact, 88 hexahedra is the smallest solution ever published [89].



(1) (2)

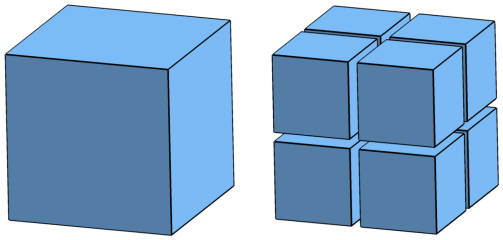
Figure 8.3: The subdivision of one tetrahedron (1) into four hexahedra (2)

Some authors have developed a subdivision technique capable of transforming a non-conformal mixed hexahedral mesh into a conformal mesh containing only hexahedra, prisms and tetrahedra [87]. As shown on Fig. 8.6, a special pattern is used when a quadrilateral face is adjacent to two triangular faces. The quadrilateral face is divided into eight triangles and the two triangular faces are divided into four triangles each [87]. The non-conformity is therefore repaired without



(1) (2)

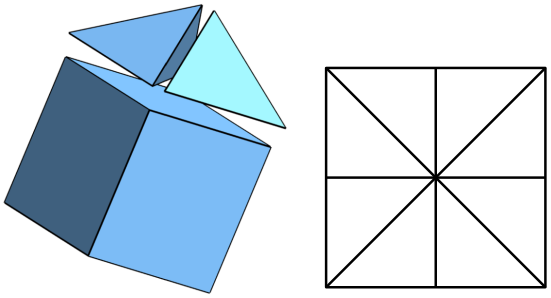
Figure 8.4: The subdivision of one prism (1) into six hexahedra (2)



(1) (2)

Figure 8.5: The subdivision of one hexahedra (1) into eight hexahedra (2)

resorting to pyramids. The elements need to be subdivided a second time in order to obtain a full hexahedral mesh. As a consequence, the mesh size is divided by four.



(1) (2)

Figure 8.6: There is a non-conformity between the hexahedron and the two tetrahedra (1). The non-conformity is resolved using the subdivision pattern (2).

Chapter 9

Appendix

The first section of this appendix contains the pseudo-code of the frontal algorithm in three dimensions. The second section contains the pseudo-code necessary to compute the L_p Lloyd's energy and its gradient in two dimensions.

9.1 The frontal algorithm in three dimensions

```
//This method computes the enclosing box of a node.
void compute_parameters(Node node){
  point = node.get_point
  x = point.x
  y = point.y
  z = point.z
  m = get_matrix(x, y, z)
  h = get_size(x, y, z) //Local mesh size

  node.set_size(h)
  node.set_metric(m)
  node.min[0] = x - sqrt3*h
  node.min[1] = y - sqrt3*h
  node.min[2] = z - sqrt3*h
  node.max[0] = x + sqrt3*h
  node.max[1] = y + sqrt3*h
```

```
  node.max[2] = z + sqrt3*h
}

//This method creates six potential nodes in the six
//cardinal directions around a given node.
void create_spawns(Node node,vector<Node> spawns){
  point = node.get_point
  x = point.x
  y = point.y
  z = point.z
  h = node.get_size
  m = node.get_matrix //Local cardinal directions

  x1 = x + h*m.get_m11
  y1 = y + h*m.get_m21
  z1 = z + h*m.get_m31

  x2 = x - h*m.get_m11
  y2 = y - h*m.get_m21
  z2 = z - h*m.get_m31

  x3 = x + h*m.get_m12
  y3 = y + h*m.get_m22
  z3 = z + h*m.get_m32

  x4 = x - h*m.get_m12
  y4 = y - h*m.get_m22
  z4 = z - h*m.get_m32

  x5 = x + h*m.get_m13
  y5 = y + h*m.get_m23
  z5 = z + h*m.get_m33

  x6 = x - h*m.get_m13
  y6 = y - h*m.get_m23
  z6 = z - h*m.get_m33

  spawns[0] = Node(SPoint3(x1, y1, z1))
  spawns[1] = Node(SPoint3(x2, y2, z2))
  spawns[2] = Node(SPoint3(x3, y3, z3))
  spawns[3] = Node(SPoint3(x4, y4, z4))
  spawns[4] = Node(SPoint3(x5, y5, z5))
  spawns[5] = Node(SPoint3(x6, y6, z6))
}
```

```
//This is the callback function passed to the R-tree library
//by the method "treat_region". The enclosing box of the
//node "neighbour" in the arguments intersects the enclosing
//box of the prospective node "individual".
bool rtree_callback(Node neighbour, Wrapper wrapper){
  individual = wrapper.get_individual
  parent = wrapper.get_parent
  h = individual.get_size
  m = individual.get_matrix

  if neighbour!=parent{
    distance = infinity_distance(individual.get_point,
      neighbour.get_point, m);
    if distance<0.7*h{
      wrapper.set_ok(0)
      return false
      //The node "neighbour" is too close to the node
      //"individual". As a result, the variable "ok" is set
      //to false. The node "individual" will not be
      //added to the cloud.
    }
  }

  return true
}

//This method creates a cloud of well aligned vertices
//inside the three-dimensional domain "gr".
void treat_region(GRegion gr){
  boundary_vertices = gr.boundary_vertices

  for i=0 to boundary_vertices.size{
    x = boundary_vertices[i].x
    y = boundary_vertices[i].y
    z = boundary_vertices[i].z

    node = new Node(SPoint3(x, y, z))
    compute_parameters(node)
    rtree.Insert(node.min, node.max, node)
    fifo.push(node)
    //"fifo" is a "first in, first out" data structure.
  }

  while(!fifo.empty()){
    parent = fifo.front
```

```
    fifo.pop

    spawns.resize(6)

    create_spawns(parent, spawns)

    for i=0 to 6{
      individual = spawns[i]
      point = individual.get_point
      x = point.x
      y = point.y
      z = point.z

      if(inside_domain(octree, x, y, z)){
        compute_parameters(individual)

        if(far_from_boundary(octree, individual)){
          wrapper.set_ok(1)
          wrapper.set_individual(individual)
          wrapper.set_parent(parent)
          rtree.Search(individual.min, individual.max,
            rtree_callback, wrapper)
          //"rtree_callback" is the callback function
          //described above.

          if(wrapper.get_ok()){
            fifo.push(individual)
            rtree.Insert(individual.min, individual.max,
              individual)
            vertex = MVertex(x, y, z)
            new_vertices.push_back(vertex)
            //The node "spawns[i]" is added to the cloud
            //because there is no other nodes
            //too close to it.
          }
        }
      }
    }
  }

  MeshDelaunayVolume(gr)
}
```

9.2 The computation of the L_p Lloyd's energy and its gradient in two dimensions

```
//Interpolation of the x coordinate
double Tx(double u, double v, SPoint2 p1, SPoint2 p2,
  SPoint2 p3){
  return (1.0 - u - v)*p1.x + u*p2.x + v*p3.x
}

//Interpolation of the y coordinate
double Ty(double u, double v, SPoint2 p1, SPoint2 p2,
  SPoint2 p3){
  return (1.0 - u - v)*p1.y + u*p2.y + v*p3.y
}

//Computation of the Jacobian of a triangular Voronoi
//element
double J(SPoint2 p1, SPoint2 p2, SPoint2 p3){
  return (p2.x - p1.x)*(p3.y - p1.y)
    - (p3.x - p1.x)*(p2.y - p1.y)
}

//Distance between p1 and p2 in the Lp norm raised to the
//power of p
double f(SPoint2 p1, SPoint2 p2, matrix m, int p){
  //"m" is the change of basis matrix going from S
  //to S', with S' being the coordinate system
  //aligned with the local cardinal directions.
  x1 = p1.x
  y1 = p1.y
  x2 = p2.x
  y2 = p2.y
  a = m.get_a
  b = m.get_b
  c = m.get_c
  d = m.get_d
  val1 = a*(x1 - x2) + b*(y1 - y2)
  val2 = c*(x1 - x2) + d*(y1 - y2)
  val = pow_int(val1, p) + pow_int(val2, p)
  return val
}

//Derivative of the distance raised to the power of p with
```

```
//respect to the x coordinate of p1
double df_dx(SPoint2 p1, SPoint2 p2, matrix m, int p){
  x1 = p1.x
  y1 = p1.y
  x2 = p2.x
  y2 = p2.y
  a = m.get_a
  b = m.get_b
  c = m.get_c
  d = m.get_d
  val1 = a*(x1 - x2) + b*(y1 - y2)
  val2 = c*(x1 - x2) + d*(y1 - y2)
  val = p*pow_int(val1, p - 1)*a + p*pow_int(val2, p - 1)*c
  return val
}

//Derivative of the distance raised to the power of p with
//respect to the y coordinate of p1
double df_dy(SPoint2 p1, SPoint2 p2, matrix m, int p){
  x1 = p1.x
  y1 = p1.y
  x2 = p2.x
  y2 = p2.y
  a = m.get_a
  b = m.get_b
  c = m.get_c
  d = m.get_d
  val1 = a*(x1 - x2) + b*(y1 - y2)
  val2 = c*(x1 - x2) + d*(y1 - y2)
  val = p*pow_int(val1, p - 1)*b + p*pow_int(val2, p - 1)*d
  return val
}

//Computation of the Lloyd's energy of a triangular Voronoi
//element
double F(voronoi_element element, int p){
  v1 = element.get_v1
  v2 = element.get_v2
  v3 = element.get_v3
  generator = v1.get_point
  C1 = v2.get_point
  C2 = v3.get_point
  energy = 0.0
  m = element.get_matrix
```

```
  //"rho" is the density and "h" is the mesh size.
  for i=0 to gauss_num{
    u = gauss_points(i, 0) //Parametric coordinate u
    v = gauss_points(i, 1) //Parametric coordinate v
    x = Tx(u, v, generator, C1, C2)
    y = Ty(u, v, generator, C1, C2)
    point = SPoint2(x, y)
    weight = gauss_weights(i)
    rho = compute_rho(element.get_h(u, v), p)
    energy = energy + weight*rho*f(generator, point, m, p)
  }

  energy = J(generator, C1, C2)*energy
  return energy
}

//This method computes the derivatives of the Lloyd's energy
//of a triangular Voronoi element with respect to the x and y
//coordinates of the generator. The triangular element is
//assumed to be undeformable. One of its points is located
//at the generator. However, it is not the generator.
SVector3 simple(voronoi_element element, int p){
  v1 = element.get_v1
  v2 = element.get_v2
  v3 = element.get_v3
  generator = v1.get_point
  //"element" is a triangular part of the Voronoi cell
  //belonging to the vertex "generator".
  C1 = v2.get_point
  C2 = v3.get_point
  comp_x = 0.0
  comp_y = 0.0
  jacobian = J(generator, C1, C2)
  m = element.get_matrix

  for i=0 to gauss_num{
    u = gauss_points(i, 0)
    v = gauss_points(i, 1)
    x = Tx(u, v, generator, C1, C2)
    y = Ty(u, v, generator, C1, C2)
    point = SPoint2(x, y)
    weight = gauss_weights(i)
    rho = compute_rho(element.get_h(u, v), p)
    comp_x = comp_x
      + weight*rho*df_dx(generator, point, m, p)
```

```
    comp_y = comp_y
      + weight*rho*df_dy(generator, point, m, p)
  }

  comp_x = jacobian*comp_x
  comp_y = jacobian*comp_y
  return SVector3(comp_x, comp_y, 0.0)
}

//This method computes the derivatives of the Lloyd's energy
//of a triangular Voronoi element with respect to the x and y
//coordinates of one of its points (C1).
SVector3 dF_dC1(voronoi_element element, int p){
  v1 = element.get_v1
  v2 = element.get_v2
  v3 = element.get_v3
  generator = v1.get_point
  C1 = v2.get_point
  C2 = v3.get_point
  comp_x = 0.0
  comp_y = 0.0
  jacobian = J(generator, C1, C2)
  m = element.get_matrix

  for i = 0 to gauss_num{
    u = gauss_points(i, 0)
    v = gauss_points(i, 1)
    x = Tx(u, v, generator, C1, C2)
    y = Ty(u, v, generator, C1, C2)
    point = SPoint2(x, y)
    weight = gauss_weights(i)
    rho = compute_rho(element.get_h(u, v), p)
    drho_dx = (-p - 2)*pow_int(1.0/element.get_h(u, v), p + 3)
      *element.get_dh_dx
    drho_dy = (-p - 2)*pow_int(1.0/element.get_h(u, v), p + 3)
      *element.get_dh_dy
    distance = f(point, generator, m, p)
    comp_x = comp_x
      + weight*rho*df_dx(point, generator, m, p)*u*jacobian
    comp_x = comp_x
      + weight*rho*distance*(C2.y - generator.y)
    comp_x = comp_x
      + weight*drho_dx*u*distance*jacobian
    comp_y = comp_y
      + weight*rho*df_dy(point, generator, m, p)*u*jacobian
```

```
    comp_y = comp_y
      + weight*rho*distance*(generator.x - C2.x)
    comp_y = comp_y
      + weight*drho_dy*u*distance*jacobian
  }

  return SVector3(comp_x, comp_y, 0.0)
}

//This method computes the derivatives of the Lloyd's energy
//of a triangular Voronoi element with respect to the x and y
//coordinates of one of its points (C2).
SVector3 dF_dC2(voronoi_element element, int p){
  v1 = element.get_v1
  v2 = element.get_v2
  v3 = element.get_v3
  generator = v1.get_point
  C1 = v2.get_point
  C2 = v3.get_point
  comp_x = 0.0
  comp_y = 0.0
  jacobian = J(generator, C1, C2)
  m = element.get_matrix

  for i = 0 to gauss_num{
    u = gauss_points(i, 0)
    v = gauss_points(i, 1)
    x = Tx(u, v, generator, C1, C2)
    y = Ty(u, v, generator, C1, C2)
    point = SPoint2(x, y)
    weight = gauss_weights(i)
    rho = compute_rho(element.get_h(u, v), p)
    drho_dx = (-p - 2)*pow_int(1.0/element.get_h(u, v), p+3)
      *element.get_dh_dx
    drho_dy = (-p - 2)*pow_int(1.0/element.get_h(u, v), p+3)
      *element.get_dh_dy
    distance = f(point, generator, m, p)
    comp_x = comp_x
      + weight*rho*df_dx(point, generator, m, p)*v*jacobian
    comp_x = comp_x
      + weight*rho*distance*(generator.y - C1.y)
    comp_x = comp_x
      + weight*drho_dx*v*distance*jacobian
    comp_y = comp_y
      + weight*rho*df_dy(point, generator, m, p)*v*jacobian
```

```
    comp_y = comp_y
      + weight*rho*distance*(C1.x - generator.x)
    comp_y = comp_y
      + weight*drho_dy*v*distance*jacobian
  }

  return SVector3(comp_x, comp_y, 0.0)
}

//This method computes the Lloyd's energy of the whole
//vertices distribution and its gradient.
void eval(DocRecord triangulator,
  vector<SVector3> gradients, double energy, int p){
  for i=0 to gradients.size{
    gradients[i] = SVector3(0.0,0.0,0.0)
  }
  energy = 0.0
  e = 0.000001

  for it=clipped.begin to it=clipped.end{
    if it.get_quality<e{
      continue
      //If a triangular Voronoi element is too slim, it is not
      //taken into consideration in order to avoid null
      //Jacobians. This strategy might lead to inaccuracies
      //in the gradient.
    }
    v1 = it.get_v1
    v2 = it.get_v2
    v3 = it.get_v3
    generator = v1.get_point
    C1 = v2.get_point
    C2 = v3.get_point
    index = v1.get_index1
    energy = energy + F(it, p)
    gradients[index] = gradients[index] + simple(it, p)
    grad1 = dF_dC1(it, p)
    grad2 = dF_dC2(it, p)
    if v2.get_index3!=-1{
      index1 = v2.get_index1
      index2 = v2.get_index2
      index3 = v2.get_index3
      //Here, C1 is the circumcenter of the Delaunay
      //triangle composed of the vertices index1,
      //index2 and index3.
```

```
  p1 = convert(triangulator, index1)
  p2 = convert(triangulator, index2)
  p3 = convert(triangulator, index3)
  gradients[index1] = gradients[index1]
    + inner_dFdx0(grad1, C1, p1, p2, p3)
  gradients[index2] = gradients[index2]
    + inner_dFdx0(grad1, C1, p2, p1, p3)
  gradients[index3] = gradients[index3]
    + inner_dFdx0(grad1, C1, p3, p1, p2)
}
else if v2.get_index2!=-1{
  index1 = v2.get_index1
  index2 = v2.get_index2
  normal = v2.get_normal
  //Here, C1 is the intersection point between the
  //bisector of the Delaunay edge index1-index2
  //and a boundary edge. "normal" is the normal
  //vector of this boundary edge.
  p1 = convert(triangulator, index1)
  p2 = convert(triangulator, index2)
  gradients[index1] = gradients[index1]
    + boundary_dFdx0(grad1, C1, p1, p2, normal)
  gradients[index2] = gradients[index2]
    + boundary_dFdx0(grad1, C1, p2, p1, normal)
}
if v3.get_index3!=-1{
  index1 = v3.get_index1
  index2 = v3.get_index2
  index3 = v3.get_index3
  //Here, C2 is the circumcenter of the Delaunay
  //triangle composed of the vertices index1,
  //index2 and index3.
  p1 = convert(triangulator, index1)
  p2 = convert(triangulator, index2)
  p3 = convert(triangulator, index3)
  gradients[index1] = gradients[index1]
    + inner_dFdx0(grad2, C2, p1, p2, p3)
  gradients[index2] = gradients[index2]
    + inner_dFdx0(grad2, C2, p2, p1, p3)
  gradients[index3] = gradients[index3]
    + inner_dFdx0(grad2, C2, p3, p1, p2)
}
else if v3.get_index2!=-1{
  index1 = v3.get_index1
  index2 = v3.get_index2
```

```
      normal = v3.get_normal
      //Here, C2 is the intersection point between the
      //bisector of the Delaunay edge index1-index2
      //and a boundary edge. "normal" is the normal
      //vector of this boundary edge.
      p1 = convert(triangulator, index1)
      p2 = convert(triangulator, index2)
      gradients[index1] = gradients[index1]
        + boundary_dFdx0(grad2, C2, p1, p2, normal)
      gradients[index2] = gradients[index2]
        + boundary_dFdx0(grad2, C2, p2, p1, normal)
    }
  }
}
```

Bibliography

[1] (1996-2011). The stanford 3d scanning repository. `http://graphics.stanford.edu/data/3Dscanrep/`.

[2] (2011). 5 bladed propeller. `http://grabcad.com/library/5-bladed-propeller`.

[3] Alves, M. L., Fernandes, J. L. M., da Conceição Rodrigues, J. M., and Martins, P. A. F. (2003). Finite element remeshing in metal forming using hexahedral elements. *J Mater Process Tech*, 141:395–403.

[4] Appa, G., Pitsoulis, L., and Williams, H. P., editors (2006). *Handbook on Modelling for Discrete Optimization*. Springer Science+Business Media, USA.

[5] Avis, D., Hertz, A., and Marcotte, O., editors (2005). *Graph Theory and Combinatorial Optimization*. Springer Science+Business Media, USA.

[6] Baudouin, T. C., Remacle, J.-F., Marchandise, E., Henrotte, F., and Geuzaine, C. (submitted). A frontal approach to hex-dominant mesh generation. *Adv Model Sim Eng Sci*.

[7] Baudouin, T. C., Remacle, J.-F., Marchandise, E., Lambrechts, J., and Henrotte, F. (2012). Lloyd's energy minimization in the l_p norm for quadrilateral surface mesh generation. *Eng Comput*, DOI 10.1007/s00366-012-0290-x.

[8] Benzley, S. E., Perry, E., Merkley, K., Clark, B., and Sjaardema, G. (1995). A comparison of all hexagonal and all tetrahedral finite element meshes for elastic and elasto-plastic analysis. In Tautges, T., editor, *Proceedings of the 4th International Meshing Roundtable*, Albuquerque.

[9] Biswas, R. and Strawn, R. C. (1998). Tetrahedral and hexahedral mesh adaptation for cfd problems. *Appl Numer Math*, 26:135–151.

[10] Blacker, T. D. (2000). Meeting the challenge for automated conformal hexahedral meshing. In Owen, S., editor, *Proceedings of the 9th International Meshing Roundtable*, New Orleans.

[11] Blacker, T. D. and Myers, R. J. (1993). Seams and wedges in plastering: A 3d hexahedral mesh generation algorithm. *Eng Comput*, 2:83–93.

[12] Blacker, T. D. and Stephenson, M. B. (1991). Paving: A new approach to automated quadrilateral mesh generation. *Int J Numer Meth Eng*, 32:811–847.

[13] Blundell, B. G. (2008). *An Introduction to Computer Graphics and Creative 3-D Environments*. Springer-Verlag, London.

[14] Bochkanov, S. and Bystritsky, V. (1999-2011). Alglib. `http://www.alglib.net`.

[15] Borouchaki, H. and Frey, P. J. (1996). Adaptive triangular-quadrilateral mesh generation. Technical report, Institut National de Recherche en Informatique et en Automatique.

[16] Curless, B. and Levoy, M. (1996). A volumetric method for building complex models from range images. In *Proceedings of ACM SIGGRAPH*, New Orleans.

[17] Dewhirst, D. L., Grinsell, P. M., Tucker, J. R., and Mahajan, A. (1993). Joining tetrahedra to hexahedra. In *Proceedings of MSC World Users' Conference*.

[18] Dheeravongkit, A. and Shimada, K. (2007). Inverse adaptation of a hex-dominant mesh for large deformation finite element analysis. *Comput Aided Design*, 39:427–438.

[19] Dhondt, G. (2004). *The Finite Element Method for Three-Dimensional Thermomechanical Applications*. John Wiley & Sons, Chippenham.

[20] Douglas, G., Green, M., Guttman, A., and Stonebraker, M. (2004). R-trees: A dynamic index structure for spatial searching. `http://www.superliminal.com/sources/RTreeTemplate.zip`.

[21] Du, Q., Faber, V., and Gunzburger, M. (1999). Centroidal voronoi tessellations: Applications and algorithms. *Siam Rev*, 41:637–676.

[22] Du, Q. and Wang, D. (2003). Tetrahedral mesh generation and optimization based on centroidal voronoi tessellations. *Int J Numer Meth Eng*, 56:1355–1373.

[23] Duneau, M. and Gratias, D. (2003). Covering clusters in icosahedral quasicrystals. In Kramer, P. and Papadopolos, Z., editors, *Coverings of Discrete Quasiperiodic Sets*. Springer-Verlag.

[24] Fernandes, J. L. M. and Martins, P. A. F. (2007). All-hexahedral remeshing for the finite element analysis of metal forming processes. *Finite Elem Anal Des*, 43:666–679.

[25] Fleischer, R., Moret, B., and Schmidt, E. M., editors (2002). *Experimental Algorithmics: from Algorithm Design to Robust and Efficient Software*. Springer-Verlag, Germany.

[26] Fortin, A. and Garon, A. (1997-2011). Les eléments finis : de la théorie à la pratique. `http://giref.ulaval.ca/files/afortin/Publications/elements_finis.pdf`.

[27] Frey, P. and George, P.-L. (1999). *Maillages. Applications aux éléments finis*. Hermes Science Publishing, London.

[28] Geuzaine, C. and Remacle, J.-F. (2009). Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *Int J Numer Meth Eng*, 79:1309–1331.

[29] Gowers, T., Barrow-Green, J., and Leader, I., editors (2008). *The Princeton Companion to Mathematics*. Princeton University Press, USA.

[30] Gupta, P., Agarwai, V., and Varshney, M. (2008). *Design and Analysis of Algorithms*. PHI Learning, New Delhi.

[31] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *ACM Special Interest Group on Management of Data*, pages 47–57, Boston.

[32] Hall, C. (2013). Cv hp1. `http://grabcad.com/library/cv-hp1`.

[33] Hausner, A. (2001). Simulating decorative mosaics. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, New York.

[34] Hosch, W. L., editor (2011). *The Britannica Guide to Geometry*. Britannica Educational Publishing and Rosen Educational Services, New York.

[35] Huang, J., Jiang, T., Wang, Y., Tong, Y., and Bao, H. (2012). Automatic frame field guided hexahedral mesh generation. Technical report, Zhejiang University.

[36] Huang, J., Tong, Y., Wei, H., and Bao, H. (2011). Boundary aligned smooth 3d cross-frame field. In Bala, K., editor, *Proceedings of ACM SIGGRAPH Asia*, Hong Kong.

[37] Ito, Y., Shih, A. M., and Soni, B. K. (2008). Octree-based reasonable-quality hexahedral mesh generation using a new set of refinement templates. *Int J Numer Meth Eng*, 77:1809–1833.

[38] Jedrzejewski, F. (2005). *Introduction aux méthodes numériques*. Springer-Verlag, France.

[39] Johnen, A., Remacle, J.-F., and Geuzaine, C. (2013). Geometrical validity of curvilinear finite elements. *J Comput Phys*, 233:359–372.

[40] Kerckhove, C. V. (2013). Génération de maillages hexa-dominants par optimisation combinatoire. Master thesis.

[41] Kikuchi, N. (1986). *Finite Element Methods in Mechanics*. Cambridge University Press, USA.

[42] Kowalski, N., Ledoux, F., and Frey, P. (2012). A pde based approach to multidomain partitioning and quadrilateral meshing. In Jiao, X. and Weill, J.-C., editors, *Proceedings of the 21th International Meshing Roundtable*, San Jose.

[43] Kraft, P. (1999). Automatic remeshing with hexahedral elements: Problems, solutions and applications. In Shimada, K., editor, *Proceedings of the 8th International Meshing Roundtable*, South Lake Tahoe.

[44] Ledoux, H. and Gold, C. M. (2007). The 3d voronoi diagram: A tool for the modelling of geoscientific datasets. In *GéoCongrès Québec*, Québec.

[45] Lévy, B. and Liu, Y. (2010). l_p centroidal voronoi tessellation and its applications. In Hoppe, H., editor, *ACM Transactions on Graphics*, Los Angeles.

[46] Li, Y., Liu, Y., Xu, W., Wang, W., and Guo, B. (2012). All-hex meshing using singularity-restricted field. In Sloan, P.-P., editor, *Proceedings of ACM SIGGRAPH Asia*, Singapore.

[47] Liu, Y., Wang, W., Lévy, B., Sun, F., Yan, D.-M., Lu, L., and Yang, C. (2009). On centroidal voronoi tessellation-energy smoothness and fast computation. *ACM T Graphic*, 28:1–17.

[48] Longest, P. W. and Vinchurkar, S. (2007). Effect of mesh style and grid convergence on particle deposition in bifurcating airway models with comparisons to experimental data. *Med Eng Phys*, 29:350–366.

[49] Lu, L., Sun, F., Pan, H., and Wang, W. (2012). Global optimization of centroidal voronoi tessellation with monte carlo approach. *IEEE T Vis Comput Gr*, 18:1880–1890.

[50] Marchandise, E., de Wiart, C. C., Vos, W. G., Geuzaine, C., and Remacle, J.-F. (2011). High-quality surface remeshing using harmonic maps-part II : Surfaces with high genus and of large aspect ratio. *Int J Numer Meth Eng*, 86:1303–1321.

[51] Marchandise, E., Remacle, J.-F., and Geuzaine, C. (2013). Optimal parametrizations for surface remeshing. *Eng Comput*, DOI 10.1007/s00366-012-0309-3.

[52] Maréchal, L. (2009). Advances in octree-based all-hexahedral mesh generation: Handling sharp features. In Clark, B. W., editor, *Proceedings of the 18th International Meshing Roundtable*, Salt Lake City.

[53] Mayr, E. W., Prömel, H. J., and Steger, A., editors (1998). *Lectures on Proof Verification and Approximation Algorithms*. Springer-Verlag, Germany.

[54] Meshkat, S. and Talmor, D. (2000). Generating a mixed mesh of hexahedra, pentahedra and tetrahedra from an underlying tetrahedral mesh. *Int J Numer Meth Eng*, 49:17–30.

[55] Mount, D. M. and Arya, S. (1997). Ann: A library for approximate nearest neighbor searching. In *CGC Workshop on Computational Geometry*, pages 33–40, Durham.

[56] Mouton, T. and Béchet, E. (2012). Lloyd relaxation using analytical voronoi diagram in the l_∞ norm and its application to quad optimization. In Jiao, X., editor, *Proceedings of the 21st International Meshing Roundtable*, San Jose.

[57] Muhammad, R. B. (2013). Voronoi / delaunay / convex hull incremental method - applet. `http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/Voronoi/Incremental2/incremental2.htm`.

[58] Nagurney, A., editor (2003). *Innovations in Financial and Economic Networks*. Edward Elgar Publishing, UK.

[59] Nieser, M., Reitebuch, U., and Polthier, K. (2011). Cubecover - parameterization of 3d volumes. In Botsch, M. and Schaefer, S., editors, *Proceedings of the Eurographics Symposium on Geometry Processing*, Lausanne.

[60] Nivoliers, V. (2012). Echantillonnage pour l'approximation de fonctions sur des maillages. `http://alice.loria.fr/~nivoliev/These/these_nivoliers_impression.pdf`.

[61] Owen, S. J. (1998). A survey of unstructured mesh generation technology. In Freitag, L., editor, *Proceedings of the 7th International Meshing Roundtable*, Dearborn.

[62] Owen, S. J., Canann, S. A., and Saigal, S. (1997). Pyramid elements for maintaining tetrahedra to hexahedra conformability. In Canann, S. A. and Saigal, S., editors, *ASME Trends in Unstructured Mesh Generation*, pages 123–129, Evanston.

[63] Owen, S. J. and Saigal, S. (2000). H-morph: an indirect approach to advancing front hex meshing. *Int J Numer Meth Eng*, 49:289–312.

[64] Owen, S. J., Staten, M. L., Canann, S. A., and Saigal, S. (1999). Q-morph: an indirect approach to advancing front quad meshing. *Int J Numer Meth Eng*, 44:1317–1340.

[65] Paluszny, A., Mätthai, S. K., and Hohmeyer, M. (2007). Hybrid finite element-finite volume discretization of complex geologic structures and a new simulation workflow demonstrated on fractured rocks. *Geofluids*, 7:186–208.

[66] Reichenberger, V., Jakobs, H., Bastian, P., and Helmig, R. (2006). A mixed-dimensional finite volume method for two-phase flow in fractured porous media. *Adv Water Resour*, 29:1020–1036.

[67] Remacle, J.-F., Geuzaine, C., Compère, G., and Marchandise, E. (2010). High quality surface meshing using harmonic maps. *Int J Numer Meth Eng*, 83:403–425.

[68] Remacle, J.-F., Henrotte, F., Baudouin, T. C., Béchet, E., Marchandise, E., Geuzaine, C., and Mouton, T. (2013). A frontal delaunay quad mesh generator using the l_∞ norm. *Int J Numer Meth Eng*, 94:494–512.

[69] Remacle, J.-F., Henrotte, F., Baudouin, T. C., Geuzaine, C., Béchet, E., Mouton, T., and Marchandise, E. (2011). A frontal delaunay quad mesh generator using the l_∞ norm. In Quadros, W. R., editor, *Proceedings of the 20th International Meshing Roundtable*, Paris.

[70] Remacle, J.-F., Lambrechts, J., Seny, B., Marchandise, E., Johnen, A., and Geuzaine, C. (2012). Blossom-quad: a non-uniform quadrilateral mesh generator using a minimum cost perfect matching algorithm. *Int J Numer Meth Eng*, 89:1102–1119.

[71] Richeson, D. S. (2008). *Euler's Gem*. Princeton University Press, USA.

[72] Rieling, R. (2013). Yamaha xtz-125. `http://grabcad.com/library/yamaha-xtz-125`.

[73] Rossier, Y. (1991). *Systèmes de gestion industrielle: une étude de cas*. Presses Polytechniques et Universitaires Romandes, Lausanne.

[74] Rycroft, C. H. (2009). Voro++: A three-dimensional voronoi cell library in c++. *Chaos*, 19:041111.

[75] Schneiders, R., Schindler, R., and Weiler, F. (1996). Octree-based generation of hexahedral element meshes. In Mitchell, S., editor, *Proceedings of the 5th International Meshing Roundtable*, Pittsburgh.

[76] Shepherd, J. F. (2006). Hexahedral mesh generation constraints. Technical report, Scientific Computing and Imaging Institute.

[77] Shewchuk, J. R. (1998). Tetrahedral mesh generation by delaunay refinement. In *Proceedings of the 14th Annual Symposium on Computational Geometry*, pages 86–95. ACM.

[78] Shimada, K., Liao, J.-H., and Itoh, T. (1998). Quadrilateral meshing with directionality control through the packing of square cells. In Freitag, L., editor, *Proceedings of the 7th International Meshing Roundtable*, Dearborn.

[79] Si, H. (2002-2006). Tetgen: A quality tetrahedral mesh generator and three-dimensional delaunay triangulator. `http://tetgen.berlios.de/files/tetgen-manual.pdf`.

[80] Singh, V. P. (2013). Machine-part body. `http://grabcad.com/library/machine-part-body`.

[81] Tanton, J. (2005). *Encyclopedia of Mathematics*. Facts on File, USA.

[82] Tautges, T. J., White, D. R., and Leland, R. W. (2004). Twelve ways to fool the masses when describing mesh generation performance. In Üngör, A., editor, *Proceedings of the 13th International Meshing Roundtable*, Williamsburg.

[83] Veyhl, C., Belova, I. V., Murch, G. E., Öchsner, A., and Fiedler, T. (2010). On the mesh dependence of non-linear mechanical finite element analysis. *Finite Elem Anal Des*, 46:371–378.

[84] Vinchurkar, S. and Longest, P. W. (2008). Evaluation of hexahedral, prismatic and hybrid mesh styles for simulating respiratory aerosol dynamics. *Comput Fluids*, 37:317–331.

[85] Vyas, V. and Shimada, K. (2009). Tensor-guided hex-dominant mesh generation with targeted all-hex regions. In Clark, B. W., editor, *Proceedings of the 18th International Meshing Roundtable*, Salt Lake City.

[86] Wang, E., Nelson, T., and Rauch, R. (2004). Back to elements - tetrahedra vs. hexahedra. In *Proceedings of the International ANSYS Conference*.

[87] Yamakawa, S., Gentilini, I., and Shimada, K. (2008). Subdivision templates for converting a non-conformal hex-dominant mesh to a conformal hex-dominant mesh without pyramid elements. In Garimella, R., editor, *Proceedings of the 17th International Meshing Roundtable*, Pittsburgh.

[88] Yamakawa, S. and Shimada, K. (2003). Fully-automated hex-dominant mesh generation with directionality control via packing rectangular solid cells. *Int J Numer Meth Eng*, 57:2099–2129.

[89] Yamakawa, S. and Shimada, K. (2010). 88-element solution to schneiders' pyramid hex-meshing problem. *Int J Numer Meth Bio Eng*, 26:1700–1712.

[90] Yan, D.-M., Wang, W., Lévy, B., and Liu, Y. (2010). Efficient computation of 3d clipped voronoi diagram. In Mourrain, B., Schaefer, S., and Xu, G., editors, *Proceedings of the GMP Conference*, Castro Urdiales.

[91] Zhang, Y., Bazilevs, Y., Goswami, S., Bajaj, C. L., and Hughes, T. J. R. (2007). Patient-specific vascular nurbs modeling for isogeometric analysis of blood flow. *Comput Method Appl M*, 196:2943–2959.