Mesh Generation

Jeanne Pellerin

UNIVERSITÉ DE LORRAINE

Ecole Doctorale IAEM

MESH GENERATION

Mémoire présenté pour obtenir l'Habilitation à Diriger des Recherches
de l'Université de Lorraine
Mention Informatique

Soutenance publique prévue le 14 juin 2021
par

Jeanne PELLERIN

Composition du jury :

Rapporteurs :	Raphaëlle CHAINE	Professeur à l'Université de Lyon
	Christophe GEUZAINE	Professeur à l'Université de Liège
	Vladimir GARANZHA	Professeur à l'Académie des Sciences de Russie
Examinateurs :	Monique TEILLAUD	Directrice de Recherche à l'Inria Nancy Grand-Est
	Franck LEDOUX	CEA, Professeur associé à l'Université d'Evry-Val d'Essonne
	Dmitry SOKOLOV	Maître de Conférences à l'Université de Lorraine
Invités :	Jean-François REMACLE	Professeur à l'Université catholique de Louvain
	Bruno LEVY	Directeur de Recherche à l'Inria Nancy Grand-Est

Résumé

Les maillages sont omniprésents en informatique car ils permettent de définir la géométrie des objets virtuels, de les visualiser, ainsi que de prédire les propriétés physiques de leurs contreparties réelles. En ingénierie, les maillages ont un rôle central dans la plus grande partie des méthodes de calcul scientifique et sont notamment un prérequis pour obtenir des simulations numériques fiables et efficaces. Les maillages non-structurés, particulièrement flexibles, permettent des variations de taille, d'orientation et de forme de leurs éléments qui sont nécessaires pour certaines simulations. L'objectif de mes travaux de recherche est de développer des algorithmes de génération de maillages non-structurés en trois dimensions, sujet qui combine des défis de géométrie algorithmique, de physique computationnelle et d'informatique. Dans une première partie, je présente mes contributions au pré-traitement des modèles surfaciques dans l'espace 3D ainsi qu'une méthode de remaillage basée sur le diagramme de Voronoï. La deuxième partie se concentre sur la génération de maillages volumiques. Des algorithmes pour générer deux types de maillages non-structurés sont décrits: les maillages tétraédriques avec une approche type Delaunay et les maillages éléments finis hybrides avec une approche dite indirecte. Ces recherches m'ont conduite à l'étude de problèmes théoriques sous-jacents à la génération de maillage, travail présenté dans la troisième partie de ce manuscrit avec mes contributions à l'énumération des subdivisions combinatoires des polyèdres en tétraèdres ou hexaèdres. Le dernier chapitre conclut ces travaux de génération de maillage et propose un ensemble de perspectives combinant une approche théorique et pratique.

Abstract

Meshes are ubiquitous in computer science, they define the geometry of the virtual representations of objects and permit to study the physical properties of their real world counterparts. In engineering, meshes have a key role in most scientific computation methods and are a prerequisite to reliable and efficient numerical simulations. Unstructured meshes allow size, orientation and shape variations, that are necessary for some numerical methods. The objective of my research work is to develop algorithms to generate unstructured meshes in three dimensions, a topic that combine computational geometry, computational physics and computer science challenges. In a first part, I describe my contribution to the pre-processing of surface meshes in the 3D space as well as a remeshing method based on the Voronoi diagram. The second part focuses on 3D mesh generation. Algorithms to generate two types of meshes are described: tetrahedral meshes using a Delaunay based approach and hybrid mesh generation using an indirect approach. This work in 3D led me to study on the underlying theoretical questions of computational geometry which are presented in the third part where my contributions to the enumeration of the combinatorial subdivisions of polyhedra into tetrahedra or hexahedra are described. The final chapter draws some conclusions and presents perspectives of this research work in combining a practical and theoretical approaches in mesh generation.

Contents

Chapter 1

Introduction

1.1 An Introduction to Mesh Generation

What is a mesh? When watching the latest 3D animation movie or playing your favorite video game, you are probably not thinking about the meshes hidden behind the beautiful landscapes or your beloved characters. But without meshes, you would not be able to see these landscapes or characters whose actual 3D shapes they define. Meshes are virtual geometrical representations of objects, and are ubiquitous in computer science. In engineering, meshes are used for the prediction of the physical behavior of manufactured, or natural objects by numerical simulations. When running physical simulations, meshes are at the forefront, and you should look at them, because they have an important impact on the results. If a mesh is not good enough, computations based on it will fail or the results will be incorrect. Meshes have indeed a central role in most scientific computation methods. The great variability of the objects studied in engineering (e.g. sand grain, mechanical piece, wind turbine, subsurface model) as well as the great variability of physical processes (e.g. flows, wave propagation, mechanics, electromagnetics, heat transfer) and numerical methods (e.g. finite element, finite volumes, spectral elements) are a major challenge for mesh generation methods which should in all these cases generate the best mesh possible, or at least, one good enough.

The mesh generation challenge Generating a mesh may be an easy task: draw a circle, place ten points on this circle, and finally connect the neighboring points by segments. The result is a mesh of the circle. More points may be added to improve the circle shape approximation by the segments. In the general case, the mesh generation problem is significantly more difficult, but the main steps remain the same: (1) process the input shape, (2) place the mesh vertices, (3) connect the vertices, and (4) optimize the mesh. Depending on the constraints on the size, shape, and number of elements, the problem can be easily solved, rather difficult, extremely difficult, or have no solution. Trivial in one dimension, mesh generation is fairly easy in two dimensions. The high difficulty of generating three dimensional meshes is however surprising and widely underestimated. Mesh generation is considered by some practitioners as the most expensive step of engineering analysis in terms of human interaction time. NASA considered in 2014 that mesh generation is one of the six technologies whose development is critical for future numerical simulations stating that "*Current meshing methods are neither reliable enough, nor sufficiently robust, and cannot automatically produce high quality meshes at the desired resolution for complex configurations*" (Slotnick et al. 2014).

Defining our terms In mesh generation, it is particularly important to define precisely the terms we use. Indeed, the difficulty, and sometimes the feasibility, of one mesh is dependent on the use, definition, and understanding of a few words. Given an object defined by its boundaries, generating a mesh of this object consists in generating a set of elements (points, segments, quadrilaterals, triangles, tetrahedra, pyramids, prisms, hexahedra, etc.) whose union is an adequate approximation of the object and whose interiors do not intersect. Depending on the application, a mesh should also enforce a set of constraints on the cell types, total number of cells, their shape, size, and orientation.

The mesh type is defined from the types of its elements and their connectivity. Structured meshes

have a regular connectivity, i.e. all their vertices have the same number of neighbors. They are generally constituted of quadrilateral or hexahedral elements. To the contrary, unstructured meshes do not have a regular connectivity. When unstructured 2D mesh are extruded along the third dimension, the mesh is semi-structured (also called 2.5D). The diversity of unstructured meshes is large: the mesh cells can be general polyhedra, or specified polyhedra (tetrahedra, hexahedra, prisms, pyramids). When all the elements have the same type, the mesh type is defined from that element: quadrangular mesh, triangular mesh, hexahedral mesh, tetrahedral mesh, prismatic mesh, etc. When the mesh is constituted of elements of different types, it is said to be hybrid or mixed. A mesh is conformal when the non-empty intersection any of two of its elements is an element common the boundary of these two elements. The mesh is constrained if some of its elements (edges and facets) are imposed to be part of the final mesh before its construction (very often elements the domain boundaries).

Objectives The objective of all mesh generation methods is to generate a valid mesh that fulfills user expectations. The challenge is that this adequate mesh, one that ensures reliable and efficient numerical simulation results for one application, will probably not be adequate for another application. Indeed, meshing is always an exercise in tradeoffs between the characteristics of the mesh that allows the numerical simulations to be faithful to the physics at the desired precision and the efficiency of the computation (in other words the mesh quality). Mesh requirements are typically incompatible and generating a mesh is always a compromise. Geometrical characteristics of the objects to mesh (low angles, thin features and thin sections, high curvature) are in direct conflict with the geometrical characteristics desired for the mesh. The increasing complexity and size of the models, as well as the increasing complexity of the physics simulated are major challenges for mesh generation. This is without even mentioning the question of the existence (and feasibility) of an optimal mesh whose characteristics cannot be known in advance. It is important to keep in mind that (1) mesh generation is intrinsically an ill-posed geometrical problem, and that (2) the difficulty of the problem is infinitely superior in three dimensions than in two dimensions. In that context, it is essential to develop tools with solid theoretical basis that are quite general, but which remain motivated by given applicative needs.

The meshing workflow Despite major differences between applications and desired properties for optimal meshes, practical mesh generation generally follows four steps:

1. Input geometry pre-processing
2. Generate the mesh
3. Optimize the mesh quality (remove a priori problematic elements, fit size map)
4. Adapt the mesh to the solution (refine or coarsen the mesh using a posteriori error evaluation)

Depending on the type of mesh and on the application, the relative importance and difficulty of each of these four steps vary a lot. For visualization purposes, e.g. in computer graphics applications, the primary interest is to automatize the procedure, fidelity to the actual physics not being the crucial point. The trend is therefore to automate the correction of the input models and to develop fast provable meshing algorithms that generate automatically meshes. Such approaches are not (and may never be) reliable options for the engineering oriented meshing community. Practical meshing methods should be robust, applicable to a large range of real models, and allow control to the user. When evaluating complex physical phenomena about complex geometries, strong approximations are not acceptable, e.g. when simulating air flows around a plane wing. In complex cases, generating the mesh is an iterative process based on its adaptation to capture the small crucial features of the numerical solution. The actual objectives of the computational engineering and computer graphics communities are different, and the developed approaches, though complementary, deeply differ in terms of methods and applicability.

1.2 Main Contributions

Mesh generation has been the focus of my research work since my PhD thesis. I am more particularly interested in the development of algorithms to generate unstructured meshes in three dimensions. During my PhD thesis I developed mesh generation methods that allowed modifications of the input model, an approach motivated by geological models with very thin layers that vanish laterally. More recently, I have worked on robust meshing volumetric methods which leave the input mesh unchanged: tetrahedral meshing, hybrid meshing and hexahedral meshing. These practical topics lead me to study on the underlying theoretical questions that may be the key to new meshing algorithms.

An abstract point of view on meshing The problem of mesh generation may be subdivided into two interdependant steps: (i) determining the coordinates of the mesh vertices (ii) connecting the vertices to build valid mesh cells. Given a domain Ω and a set of points $\mathcal{V}$, I study the problem of the construction of a set of cells C that connect all the points of $\mathcal{V}$ and enforce the two fundamental properties of a mesh:

1. The union of the cells is equal to the domain: $\bigcup C = \Omega$ (Union property)
2. The intersection of two cells is either empty or is a face/edge/vertex common to the two cells (Intersection property).

These two properties have two components: one combinatorial and one geometrical whose interdependance is one of the main challenge in 3D mesh generation. In theory, the set of meshes that can be built from $\mathcal{V}$, denoted $\mathcal{H}_{\mathcal{V}}$, is finite, but is in general impossible to evaluate in 3D. Each mesh with vertices $\mathcal{V}$ is a combination of cells in C that verify the union and intersection properties. Each mesh is then a maximal clique of the compatibility graph G of C in which there is one node for each cell of C and one edge if two cells are compatible, i.e. the intersection property is valid for this pair. This approach is however far from being practical, since even for small polyhedra the set of cells C has a prohibitive size.

This abstract view of the meshing problem may however be used to analyse and classify my contributions to unstructured mesh generation methods. Practical methods heavily rely on the Delaunay triangulation and its geometrical dual the Voronoi diagram. The Delaunay triangulation of a point-set in general position, $\mathcal{V}$, is the unique triangulation such that the interior of the circumsphere of any tetrahedra is empty. The Voronoi diagram is defined from its Voronoi cells: the subsets of the space closer to a point $p \in \mathcal{V}$ than to any other point of $\mathcal{V}$.

Mesh generation from the Voronoi diagram We propose meshing methods for geological models, Ω that are defined by their triangulated boundary surfaces $\delta\Omega$, using the intersection of the Voronoi diagram of points $\mathcal{V}$ with these surfaces. Our contribution is the definition of strategies to build a mesh whatever are the topology of the intersections between the surfaces and the Voronoi cells. Indeed, depending on the point sampling resolution, a Voronoi cell can cut several model entities and define several connected components. To control the number of cells in the output mesh, our algorithms allow to locally modify the model to reduce the difference between the model Ω level of detail and the point set $\mathcal{V}$ resolution. The generated meshes are constituted of different types of cells: quadrilaterals and triangles for surfaces, tetrahedra, prisms, and pyramids for volumes (Section 3.2 and Chapter 5).

Mesh generation from the Delaunay triangulation We propose an efficient implementation of the sequential 3D Delaunay triangulation algorithm that is about three times faster than existing open-source implementations and is able to triangulate a million points in about 2 seconds. Our main contribution is a scalable parallel version of the Delaunay triangulation algorithm devoid of heavy synchronization overheads. The domain is partitioned using the Hilbert curve and conflicts are detected with a simple coloring scheme. On a AMD® EPYC 64-core machine, we have been able to generate three billion tetrahedra in less than a minute. The Delaunay triangulation is the base of a tetrahedral mesh generation process where the input is the boundary surfaces of the domain to mesh (Chapter 4).

Indirect hybrid mesh generation We propose a method to build hexahedra, prisms and pyramids by merging subsets of cells of a tetrahedral mesh $\mathcal{T}$ of vertices $\mathcal{V}$. Our contribution is an algorithm that build the set $\mathcal{H}_\mathcal{T}$ of all possible combinations of tetrahedra of $\mathcal{T}$ in hexahedra, i.e. the subsets $t \subset \mathcal{T}$ that define a topological ball and whose boundary is a triangulation of the cube. The sets of prisms $\mathcal{P}_\mathcal{T}$ and of pyramids $\mathcal{S}_\mathcal{T}$ are determined similarly to obtain $\mathcal{H} = \mathcal{H}_\mathcal{T} \cup \mathcal{P}_\mathcal{T} \cup \mathcal{S}_\mathcal{T} \cup \mathcal{T}$. The final mesh is built by determining a clique of the compatibility graph G where there is a node for each cell of $\mathcal{H}$ and one edge if the cells are compatible. (Chapter 6).

Enumerating tetrahedral meshes Using the indirect meshing algorithm, we discovered subdivisions of the hexahedron into 8 and more tetrahedra that contradicted previous beliefs on that problem. Given $\mathcal{V} = \{1, 2, 3, 4, 5, 6, 7, 8\}$, we further investigated the enumeration of the combinatorial tetrahedral meshes of the hexahedron. The set of tetrahedral cells C has $\binom{8}{4} = 70$ elements. The set of all parts of C, denoted $P(C)$ has 2^{70} elements. A naive brute force approach testing the validity of each combination succeeds but is extremely time consuming.

To show that there are 174 combinatorial triangulations of the hexahedron up to isomorphism with 5 to 15 tetrahedra, we consider the triangulations of the 3-sphere with nine vertices determined by discrete mathematicians in the 1970's. The second contribution is to demonstrate that exactly 171 of these 174 triangulations have a geometrical realization in $\mathbb{R}^3$. Our proof consists in (1) solving the underlying combinatorial problem of the point configurations that can realize a given triangulation T, before (2) finding point coordinates in $\mathbb{R}^3$ corresponding to one configuration (Chapter 7).

Enumerating hexahedral meshes We now consider the extremely constrained hexahedral meshing problem, the generation of hexahedral meshes whose boundary facets match a input quadrilateral mesh. For a given number n of vertices $\mathcal{V}$, the set of all possible hexahedral cells C has $\binom{n}{8}$ elements. We first propose a backtracking search to enumerate the subsets of C that constitute a valid combinatorial hexahedral meshes of a given polyhedron.

We then propose to enumerate directly a subset of all possible hexahedral meshes. The key idea is to exploit the equivalence between quadrangle flips and hexahedron insertion in a mesh. The tree of all sequences of flipping operations is explored, searching for a path that transforms the input quadrangulation Q to the boundary of a cube. For larger meshes the search stops when a sequence transforming Q to a set of pre-computed hexahedral meshes is found. This is the first practical algorithm to build constrained combinatorial hexahedral meshes. We compute small hexahedral meshes of quadrangulations for which the previously known best solutions could only be built by hand or contained thousands of hexahedra.

Our major theoretical result is to prove that an arbitrary ball bounded by n quadrangles can be meshed using only 78 n hexahedra. This significantly lowers the previous upper bound of 5396 n (Chapter 8).

Part I

Surface Models: Correction, Simplification & Meshing

Chapter 2

Surface Mesh Models

Among the various possibilities to construct and represent geometrical objects in 2D and 3D, boundary representations are one of the most flexible. The idea of the boundary representation (or BRep) is (not a surprise) the representation of a solid by its boundary. In 2D these boundary are segments and curves and in 3D planes or more complex surfaces. Boundary representations are an ubiquitous representation scheme for solids in most geometric modelers, since they are able to represent in a portable light way solids exhibiting a high level of geometric complexity. They are used in many industrial applications as well as to produce computer animations in movies.

Dedicated methods and software have been developped for dedicated applications. Software for automotive, shipbuilding, oil and gas, or aeronautics industries may share algorithms but developments are mostly driven by the specific applications. My research work was first motivated by geological applications where the surfaces of the solid models are meshed with triangles (Figure 2.1b). Surface models are particularly adapted to geological models that are generally built from the surfaces delimiting rock layers (horizons, faults, unconformities, etc).

In this chapter, we define the 3D models and related notions used throughout this manuscript, their components, validity, implementation. I describe my contributions to the management of geological models outside geological modeling software.

2.1 Surface Models

In a boundary representation, 3D solid models are constituted of a set of volumes (or regions) bounded by a set of surfaces, each surface is bounded by a set of lines, and each line is bounded by two end points. The choice of a datastructure is key to manipulate efficiently these models and the simplest representation is often the best choice. A surface model is built as sets of entities. The base entities define completely the topology and geometry of the model, while additional physical entities define the features that have a meaning for the user (Figure 2.3). In geological modeling, those are rock layers. When implementing the library RingMesh we adopted a simple representation that revealed itself to be similar to the one of Gmsh (Geuzaine et al. 2009). Several alternative computer representations for these models have been proposed in the literature, see for instance (Caumon, Lepage, et al. 2004) and the references therein.

2.1.1 Entities

Base entities Four base entities constitute the surface model: Corner (dimension 0), Line (dimension 1), Surface (dimension 2), and Region (dimension 3). Each base topological entity is restricted to be a simply connected manifold of arbitrary genus. Simply connected means that the entity has a unique connected component (any two points of the entity can be connected by a path contained in the entity), manifold means that the entity does not contain non-manifold point (for example points at a T intersection). Arbitrary genus means that an entity may have holes and internal boundaries. A specific structure stores the model extension.

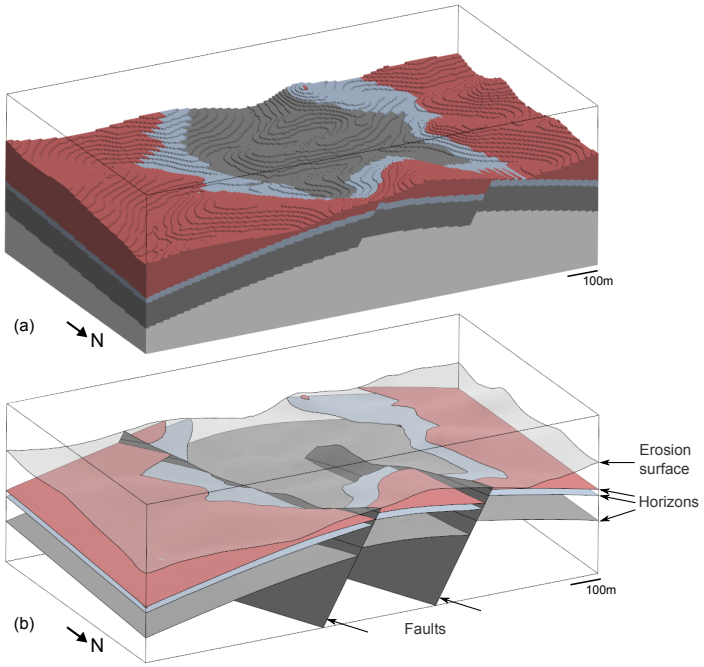


Figure 2.1: A structured mesh and a surface model representation of the same synthetic geological model.

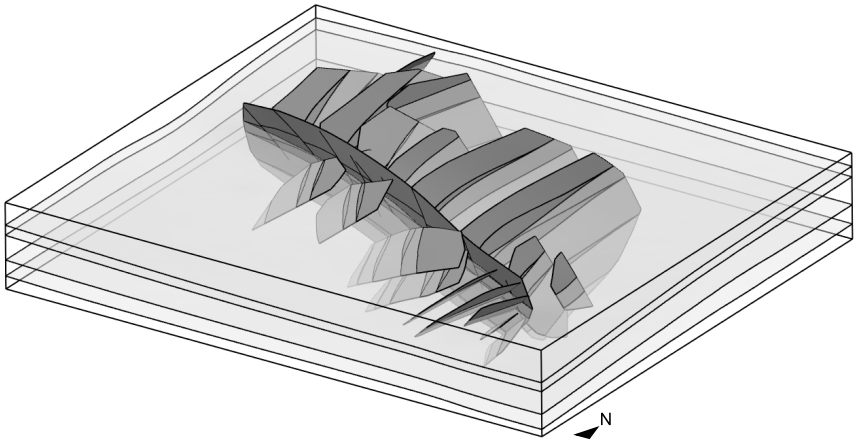


Figure 2.2: A 3D subsurface surface delimited by surfaces that represent boundaries between rock layers - horizons - and by a network of discontinuities that displace these layers

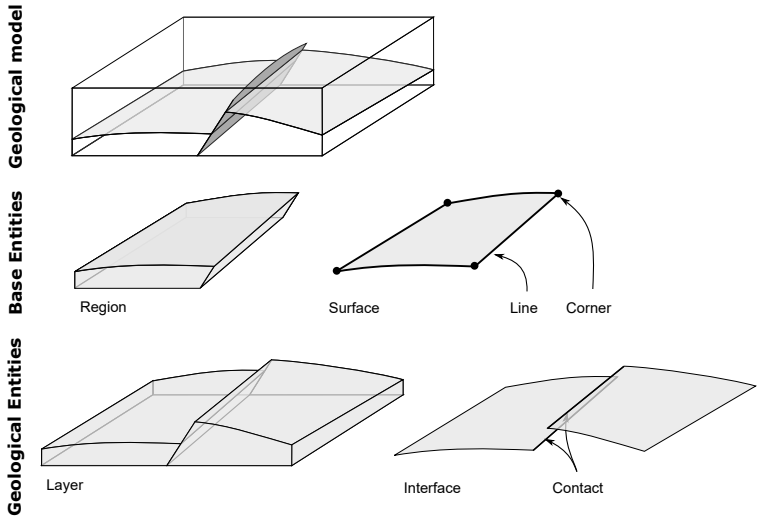


Figure 2.3: Representation of surface models in this thesis: base entities and physical (geological) entities.

Adjacencies in the model are represented with a bi-directional data structure: each entity stores a list of upward adjacencies and a list of downward adjacencies.

CORNER (1 or 2) $\rightleftharpoons$ (1..n) LINE (0..n) $\rightleftharpoons$ (1..n) SURFACE (1 or 2) $\rightleftharpoons$ (1..n) REGION

- A REGION is bounded by a set of oriented SURFACES.
- A SURFACE is either closed (no boundary) or bounded by a set of LINES and is incident to one or two REGIONS.
- A LINE is either closed or bounded by one or two CORNERS (closed LINES are cut at one of their vertices), and is incident to at least one SURFACE.
- A CORNER is incident to at least one LINE.

Physical entities Physical features, that are typically boundaries between different materials (in geology: faults, horizons, fault-horizon contact, stratigraphic layers, etc.), are represented by physical entities. Each physical entity is a group of base entities to which are associated a name and a feature recording its role in the model (in geology: normal fault, reverse fault, horizon, unconformity, boundary of the model). Each Contact corresponds to a group of LINES, each Interface to a group of SURFACES, and each Layer to a group of REGIONS. Adjacency relationships between physical entities are not stored.

2.1.2 Meshes

In the models considered in this work, the geometry of all base entities is represented by a mesh of the same dimension as the entity. Each CORNER corresponds to a 3D point. Each LINE is represented by a set of adjacent segments, and each SURFACE by a polygonal surface mesh. REGIONS can either be defined by their oriented boundary SURFACES, or by a volumetric mesh. To ensure a correct definition of the model, the geometry of the entities should enforce two conditions: (i) the boundary of each entity is a union of entities of lower dimension and (ii) the intersection of two entities is a union of entities of equal or lower dimension. These conditions are the same than those required to have a Piecewise Linear Complex (PLC), the BRep representation used in Tetgen (Si 2015).

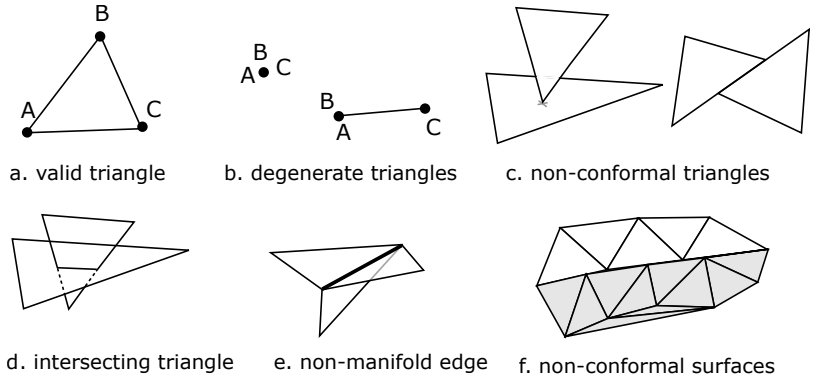


Figure 2.4: Some mesh invalidity configurations.

2.1.3 Validity

Before any manipulation or processing of a surface model, one must be sure that this model is consistent. Without this verification, problem identification and debugging of algorithms applied on the models is difficult and cumbersome if not impossible. Indeed the visual inspection of a 3D model is generally not sufficient to check its validity. The validity of a model is tightly linked to the chosen representation, to the application and to the applied algorithms. On the meshing side, the most efficient methods will stop or crash because of bad input.

Entity validity Before all, a valid model should be constituted of valid individual entities. When the input models are already meshed, these meshes must be valid. The entity meshes should be valid and conformal meshes, i.e. they are defined by a set of elements (vertices, edges, facets, and cells) such that (i) the interior of each element is not empty, and (ii) the intersection of two elements is either empty or is an element common to their boundaries (Figure 2.4c.&d.). Empty elements are typically edges, facets, or cells incident twice to the same point (Figure 2.4b.). Second, a base entity must not contain any non-manifold point (Figure 2.4e.), and must have a unique connected component. Finally, Regions defined by their boundary surfaces are to be watertight closed volumes. A model that looks good may be invalid, e.g. the model of the Annot sandstones has few local mesh issues break its validity (Figure 2.5).

Model validity The solid model must have a finite extension, i.e. the region defining the exterior of the model must be tight and closed. Second, the boundary of a model entity can only be a set of entities of the model. All the vertices, edges, or facets on the boundary of a Line, Surface, or Region have to be part of a Corner, Line, or Surface. The third validity condition ensures the consistency of the stored entity adjacencies with the geometrical representation: two distinct entities should intersect exclusively along entities of their common boundary. This implies that there is no intersection between two entities except at points that are on their (geometric) boundaries and that are part of a (topological) entity in their boundary. Some additional validity conditions may be imposed by the application. for example in geological modeling, only fault or fracture surfaces might end in a volumetric region and two distinct stratigraphic interfaces cannot cross one another.

It is important to note that if it may be relatively easy to implement some corrective functions to fix small defects of the mesh and remove duplicated vertices/edges/facets, isolated vertices, degenerate edges/facets (Levy 2016), it is extremely difficult to fix multiple defects in the general case in a robust way.

Extended geometrical validity Recently, in the PhD thesis of Pierre Anquez we extended the definition of validity to capture holes, gaps and thin features in geological models by using entity exclusion zones. We define for each model entity (Corner and Line, Surface) exclusion zones to materialize volumes that cannot overlap and cannot include any other model entity. Intersections between exclusion zones define invalid features of the input model Section 3.1.1.

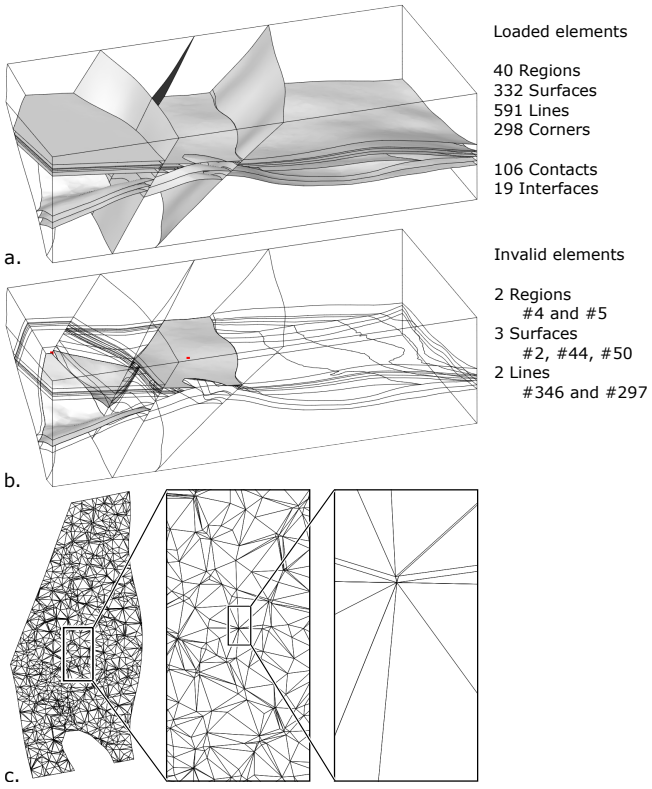


Figure 2.5: A good looking model can be invalid. This subsurface model of the Annot sandstones (Salles et al. 2011) has degenerated mesh elements 5 colocated vertices associated to 1 duplicated triangle and 8 degenerate triangles, result in the invalidity of one SURFACE and its two adjacent REGIONS. 2 degenerate LINES containing a unique degenerate edge, (2 pairs of colocated vertices) resulting in the invalidity of 2 SURFACES each containing a degenerate facet built on this edge.

2.1.4 Data Structure

A significant part of my mesh generation research was performed for geological modeling purposes. In geomodeling, the BRep geological models (also called structural models) are most of the time discrete models that are built in dedicated commercial software following specific modeling techniques (Caumon, Collon-Drouaillet, et al. 2009). These commercial software do not ease (or try to prevent) interoperability with non-geological modeling tools. That is a major hassle for the geomodeling scientific community since the use, testing and improvement of more general methods (meshing algorithms, numerical solvers, visualization software) is most active in non-commercial software. There is simply no standard file format for mesh storage. Difficulties are technical, and solutions are known, but when we started RingMesh no implementation was available for this specific type of models. In a joint effort to factorize in a common library a simple and efficient data model and to ease the tedious tasks of reading and writing files describing geological models in various formats we created RingMesh.

I developed the very first version of that code to load the surface models of Skua-Gocad (Paradigm 2016) and compute complexity measures (Pellerin, Caumon, et al. 2015). In 2014, building on that very first version, we started the project with Arnaud Botella and I was one of the main developper till september 2016, writing about half of the code. In addition to file format conversion we added functionalities to check models validity, fix mesh errors, plug in meshing software, access to information on model topology and geometry, manage attributes, and visualize the models. A convenient functionality was the reconstruction of the volumetric regions of model for which only the surfaces are available in input. The documented code is open-source and distributed under the modified BSD license.

- Pellerin, J., Arnaud Botella, A. Mazuyer, B. Chauvin, B. Levy, and G. Caumon (2015). "RINGMesh A programming library for geological model meshes". In: *Proc. 17th IAMG conference*. Freiberg, Germany
- Pellerin, J., A. Botella, F. Bonneau, A. Mazuyer, B. Chauvin, B. Lévy, and G. Caumon (2017). "RINGMesh: A programming library for developing mesh-based geomodeling applications". en. In: *Computers & Geosciences* 104, pp. 93–100
- GitHub repository: `https://github.com/ringmesh/RINGMesh`

RingMesh has been used for the development of meshing methods dedicated to geological models by the team who developed it (Pellerin, Lévy, and Caumon 2014; Pellerin, Lévy, Caumon, and Botella 2014; Botella et al. 2016), and was integrated in several workflows to: find appropriate boundary conditions in 3D geomechanical restoration (Chauvin et al. 2016), to perform homogenization for seismic wave propagation (Cupillard et al. 2015), and to determine far-field stress conditions from several borehole observations (Mazuyer et al. 2016).

The file format nightmare Meshes are compact flexible representation of potentially extremely complex object in 2D and 3D. Software choices to generate and use meshes are tightly linked to the application domain, to the available functionalities and to pragmatic reasons such as user proficiency, code availability, time, or budget. Implementing readers, writers, converters of mesh file formats conversion has been and is still a tedious part of my work. When developing RingMesh (see Section 2.1) we gave in to the temptation of writing our own file format. This was an error because there were equivalent, more general, more largely used file formats.

2.2 Synthetic 3D Geological Models

During my PhD thesis, no simple geological model was available to to develop, test, explain research methods. I built a suite of synthetic models to evaluate the impact of typical geological features on meshing algorithms (Section 2.3). These models are simple, most certainly not exactly consistent from a geologist point of view, but they have been extremely useful for testing new methods operating on geological surface models (Figure 2.7). When building them, I never suspected that they would still

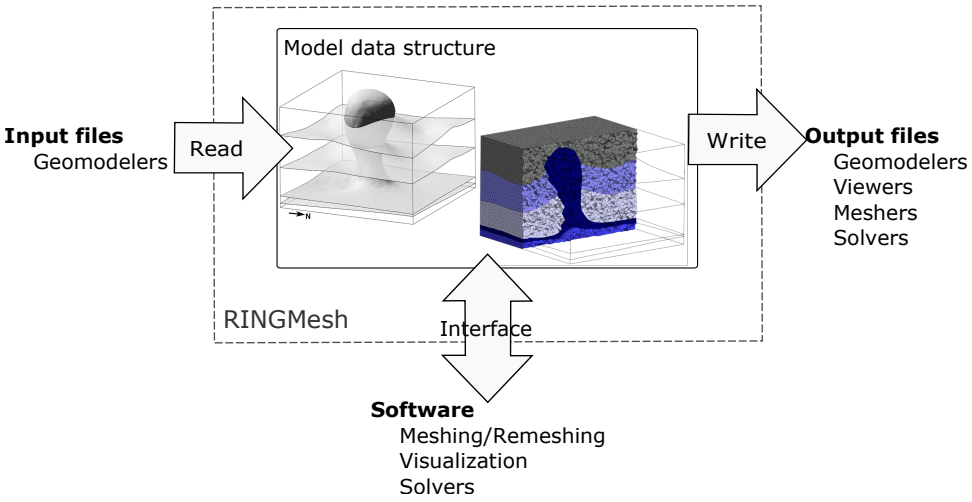


Figure 2.6: Managing meshes models is the key to interoperability between modeling software, meshing software, simulators, and visualization software. RingMesh was designed to improve the and reduce the file format.

	A1	A2	A3	A4	A5	A6	B	C	D
REGIONS	4	12	4	8	12	**14**	5	4	5
SURFACES	3	23	16	23	31	**42**	8	4	6
LINES	0	13	24	22	30	**44**	4	1	2
CORNERS	0	0	12	6	10	**15**	0	0	0

Table 2.1: Number of entities of the open models

be used today (to the contrary of some of my meshing algorithms). They are available freely on the website of the Ring consortium.

I proposed a suite of six simple models derived from a simple cylindrical anticline composed of three gently folded horizons where each layer has a constant thickness called **model A1**. In **model A2**, two regional normal faults affect the anticline. These faults are planar, parallel one to another, cut the whole volume of interest, and have dips close to 60 degrees toward the west. Moreover, they have a constant total slip, corresponding to parallel horizon cutoff lines. In **model A3**, the regional faults are restricted to an ellipsoid shape and terminate in the model. These faults do not compartmentalize the domain and fault slips vary from a maximum near fault centers to zero at fault tips. In **model A4**, one fault is regional while the eastern fault dies out to the north. Fault displacements increase to the south. As a result, horizon cutoff lines intersect with a small angle. The faults of **model A5** intersect along a branch line, resulting in a Y configuration. Fault slips are regular and the model can be restored to model A1 by rigid block motion. However, the slip on the west fault is close to the thickness of the top layer, which generates thin features in the Allan diagram. **Model A6** is obtained by cutting model A4 with a topography surface. This results in numerous isolated layer parts, some of them very small when compared to the model dimensions.

The three other models illustrate challenges arising in other contexts. **Model B** corresponds to a compressive fault-propagation fold. In the lower part, the fault has a low dip and branches onto a horizontal décollement level. The thrust dip changes to a medium angle (ramp) in the upper part and stops in the upper layer in which the shortening is accommodated by internal layer deformation. **Model C** is built from the folded basal horizon of model A1 overlaid by onlapping horizontal layers deposited at a low angle. The diapiric dome of **model D** intrudes and cuts three subhorizontal layers. Except in the intrusion influence area, the horizons are only slightly deformed. Dimensions for model B and C are similar to the ones for model A.

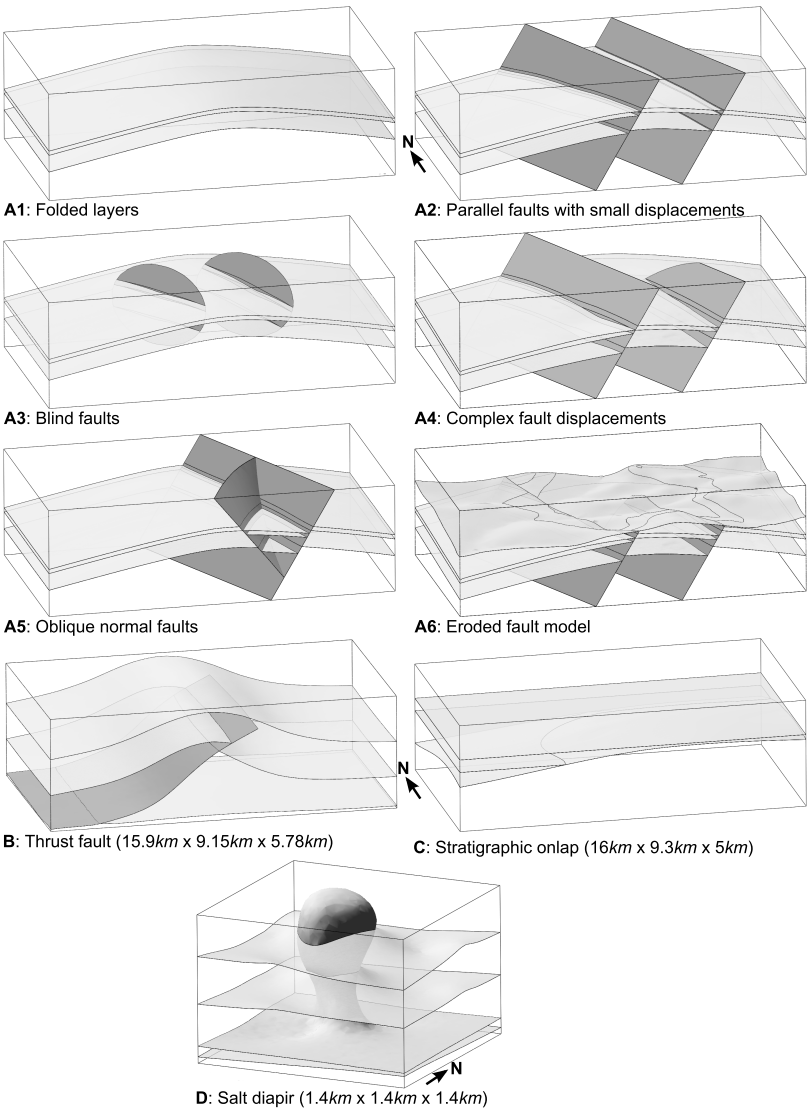


Figure 2.7: Synthetic geological models leading to typical challenging configurations for meshing algorithms. (dimensions $16km \times 9.3km \times 5km$).

2.3 Comparing Geological Models: A Meshing Standpoint

In this section, I present a contribution that is a collaboration with Guillaume Caumon, Charline Julio, Pablo Mejia-Herrera and Arnaud Botella. Our primary objective was to evaluate the relative complexity of several models to allow a fair comparison of algorithms. We analyzed the complexity of 3D geological models by describing and quantifying entities which contribute to the geometrical complexity of the geological structures.

All 3D models, in geological modeling or in other fields, are, by definition, an idealized and simplified vision of reality. How far should each model represent the complexity of the actual object? At what scale? What is exactly meant when a theory, method or algorithm is claimed to address complex domains? The term complex is indeed generally used in a qualitative and subjective way depending on the person making that statement, on his/her education and experience, on the means at his/her disposal, and on the purpose of his/her work.

The measures proposed could be useful for five purposes: (i) to help understand by quantifying our perception of structural model complexity. (ii) to compare geological models (models of the same zone at different resolutions, or models of two different zones); (iii) to improve the objective comparison of geomodeling algorithms and computer codes (an objective comparison of algorithms requires an objective comparison of the models on which they succeed or fail); (iv) as an indicator of the required effort to process these models; (v) parameterize cleaning and meshing algorithms (thin feature size, mesh size).

- Pellerin, J., G. Caumon, C. Julio, P. Mejia-Herrera, and A. Botella (2015). "Elements for measuring the complexity of 3D structural models: Connectivity and geometry". In: *Computers & Geosciences* 76. Publisher: Elsevier, pp. 130–140

Previous work In geosciences, several papers propose model complexity evaluations using a subdivision of the models into different cells. (Andrle 1996) uses radius-varying circles to compute a resolution-adapted angle measure and evaluate the complexity of geomorphic lines. (Lindsay et al. 2013) propose a local connectivity complexity measure for models represented as regular Cartesian grids. For each cell, the number of stratigraphic layers sampled by the cell and its adjacent cells is determined, and the average value of this local measure is computed over the model. This local counting principle is also used by box-counting methods to compute fractal dimension (Kruhl 2013).

In other modeling fields using a similar 3D representations, several papers propose global evaluations of model complexity (J. Rossignac 2005; Sukumar et al. 2008; White et al. 2005; Quadros et al. 2004). In geometric modeling, (J. Rossignac 2005) discusses several aspects which contribute to the complexity of 3D models and influence design, implementation, stability, and performance of 3D modeling systems. In visualization, (Sukumar et al. 2008) identify surface variations, symmetry, number of parts, details and topology as the six aspects that influence visual shape complexity. To compute a mesh sizing function, (Quadros et al. 2004) identify the small geometrical characteristics in a model from the proximity between two surfaces (distance to medial surface), the proximity between two lines (distance to medial axis), surface curvature, length and curvature of the surface boundary lines.

2.3.1 My Model Is More Complex Than Yours

2.3.1.1 Global Measures

(1) Number of features The complexity of a model depends on the number physical entities it is representing. In geological modeling they are continuous layers (or rock units), the discontinuities affecting these layers. The more entities, the more difficult and time consuming it is to check and guarantee their validity.

(2) Interactions between features Another obvious statement, is that the spatial distribution of features inside the model has a direct impact on the modeling, meshing, and simulation steps. The more interactions (closeness, intersections) there are between the model entities, the more complex that model is.

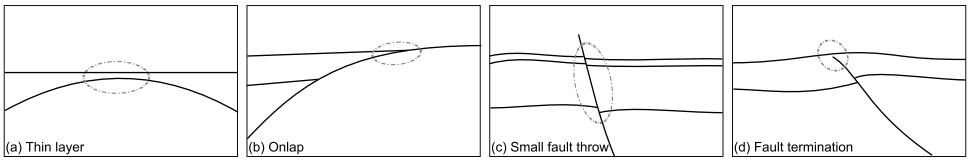


Figure 2.8: Cross-section views of some sources of complexity in geological structural models.

(3) Geological specificities Conformable layers may be challenging if one of them is locally very thin (Figure 2.8a). Abrupt thickness variations may also raise difficulties when building the model and will also be a challenge to mesh. Vertical relationships between layers controlled by unconformities are very common in stratigraphic domains which host most of the world's natural resources. They often involve thin layers and low angles between horizons (Figure 2.8c) that are extremely challenging when meshing the model, and when simulating physical processes.

Faults are discontinuities inducing a displacement of the rock units localized along a surface (Figure 2.8d). They are often difficult to characterize from subsurface data and introduce significant complexity due to their connectivity, shape, and specific properties (Jolley et al. 2007). Intersections at small angles of fault-horizons contact lines are extremely challenging for meshing software. Blind faults raise similar challenges because they stop in the model and the nil displacement at their tips is linked to very small angles between fault-horizon contacts in the Allan diagram (Figure 2.8e). When considering a fault network, the total complexity depends on the variability of the fault orientations and on the number, angle, length, and orientation of branch lines. These features have a major impact on the approximations required for meshing, e.g. Y-shaped configurations (Figure 2.8f).

2.3.1.2 Local Measures

We can simply defined a model complexity as the sum of the complexity of its components. To evaluate the complexity of the components we propose local and global measures of their connectivity (topology) and geometry.

(4) Number of components This simple measure counts the number of entities of the model (regions, surfaces, lines, and corners) excluding the ones defining the volume of interest, and giving the same weight to each entity.

(5) Variability of the component sizes This measure evaluates the complexity of the entities of a given type as a statistic on their sizes. We choose the coefficient of variation (mean over standard deviation), that characterizes the relative distribution of entity sizes and evaluates scale changes for one type of entity.

(6) Geometry We saw in the previous section that sizes, thicknesses, curvatures and angles are four important points of the geometrical complexity of structural models. We then propose to compute the geometrical complexity of each entity as the sum of four measures characterizing (1) its size C_s; (2) its shape C_f; (3) its thickness C_t; and (4) its angles C_a. Corners are not explicitly taken into account in this measure, but have an indirect impact on the values obtained for the lines and surfaces. Each of the elementary measures is chosen so that it has values between 0 and 1. The measure are made relatively to a characteristic size h given by the usage measures and a characteristic angle α (Figure 2.9) and are defined as:

- C_s compares the size of a component (line length, surface area, or region volume) to a characteristic size h given by the user.
- C_t relates to the thin features of the entity. It is defined as the percentage of the boundary length for which the thickness of the entity is smaller than h.
- C_f globally evaluates the line and surface deviation from a linear object. It is taken equal to 1 minus the size of the projection of tjhe entity on its mid-segment or mid-surface divided by the size.

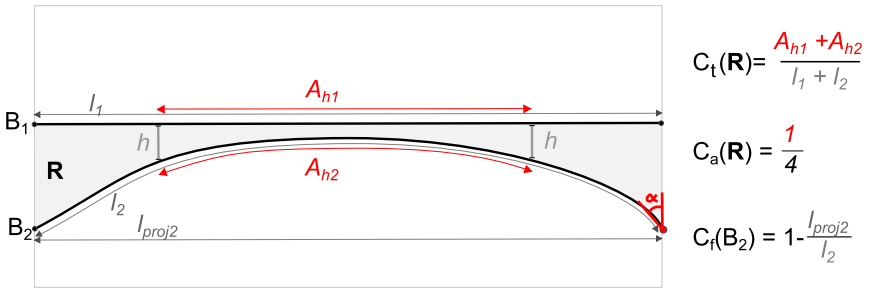


Figure 2.9: Geometrical measure computations in the plane. Thickness measure C_t and angle measure C_a for a region **R** and shape measure C_f for one of its boundaries B_2.

- C_a evaluates the percentage of the line length (respectively corners) where the angle between two surfaces (respectively two lines) is inferior to α.

(7) Volume sampling measures Instead of being computed globally, the number of model components can be computed in the neighborhoods of points sampling the model. We propose to compute the numbers of regions, surfaces, lines, and corners in the cells of a model subdivision determined by a Centroidal Voronoi diagram of points distributed inside the model (see definition in Section 3.2.2). Model complexity can then be evaluated from statistics (mean, coefficient of variation, interquartile range, percentiles, entropy) on these values.

2.3.2 Comparing the Synthetic Models

The comparison of the models constructed in Section 2.2 using the different measures gives different classifications of the models (no surprise there) by increasing complexity:

- Number of entities: A1, C, D, B, A2, A3, A4, A5, A6
- Geometry complexity with $h = 100m, \alpha = 20$ degrees: A1, C, B, A2, D, A4, A3, A5 and A6
- Size variation measure: A1, D, B, C, A2, A4, A3, A5, A6
- Volume sampling with 10,000 cells: B, D, A1, C, A3, A2, A4, A6, A5

One of the main advantages of local sampling measures is the possibility to understand the spatial organization of the complexity and estimate the extension of the zones where a given geomodeling method will fail from the cells in which the total number of entities is above a given number. These cells are the ones closest to the thin features of the models, but also to their corners and contact lines. Combining the number of cells that contains more than 6 elements and the maximum number of entities in all cells permits to establish a visual classification of the models (Figure 2.10).

These measures are far form being perfect and the question is which alternative do we have that would be able to measure volumetric features, keeping local signal like minimal distance that are crucial for meshing algorithms, and be globally consistent.

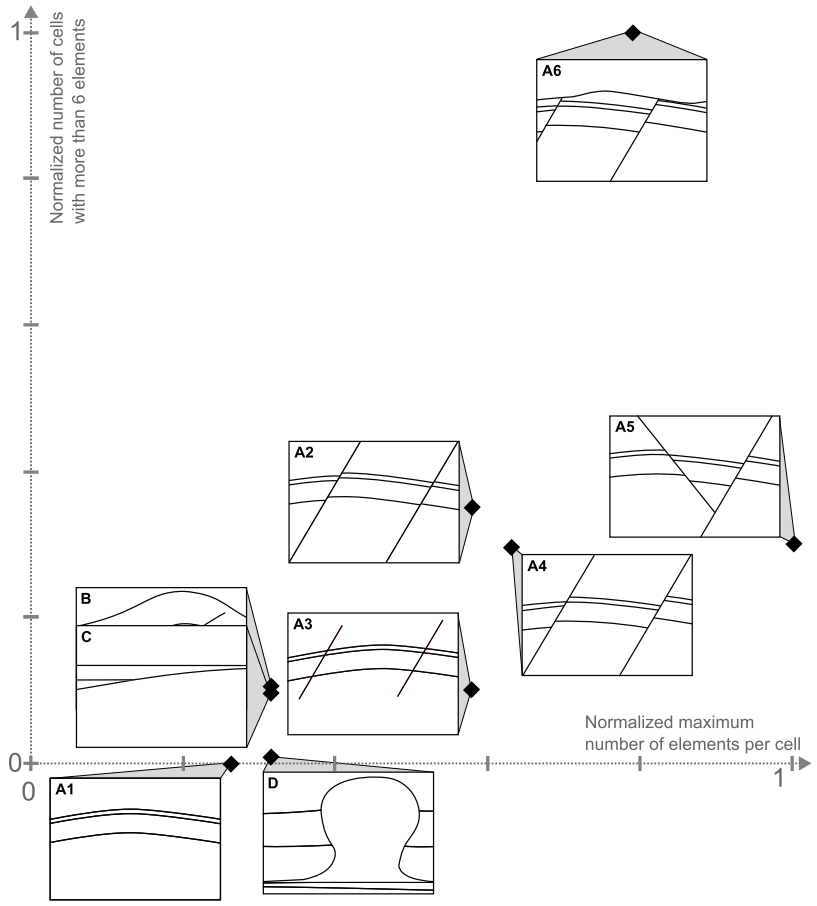


Figure 2.10: Classification of the 3D synthetic models using the number of model entities in 10 000 restricted centroidal Voronoi cells.

Chapter 3

Automatic Correction

In general, the input surface models designed during long hours with a CAD software (or another modelling software) are not valid inputs for meshing algorithms. And when they are, you can be sure that the level of detail of the surface model is not the one needed to generate the mesh. This is the eternal challenge faced by engineers. In practice, most models are manually modified by skilled practitioners who are able to ensure that the generated mesh will fulfill the project requirements. Most automatic strategies simplify the model before generating its mesh, but it is extremely challenging to ensure the validity of the output model or its consistency for the application.

3.1 Correction before Remeshing of 2D Models

This section presents the work of the PhD of Pierre Anquez that I co-advised with Guillaume Caumon and Bruno Lévy. We proposed an automatic method to repair and simplify two-dimensional (2D) geological models: geological maps and cross-sections. The output model enforces specific practical quality criteria on the model topology and geometry to allow the generation of a mesh enforcing minimum angles and minimum edge size constraints. To detect the features of the input model to correct, we define exclusion zones around input model corners and lines.

- Anquez, P., J. Pellerin, M. Irakarama, P. Cupillard, B. Lévy, and G. Caumon (2019). "Automatic correction and simplification of geological maps and cross-sections for numerical simulations". en. In: *Comptes Rendus Geoscience* 351.1, pp. 48–58
- Anquez, P. (2019). "Correction et simplification de modèles géologiques par frontières : impact sur le maillage et la simulation numérique en sismologie et hydrodynamique". PhD Thesis

3.1.1 Automatic Correction in 2D

Overview The method is based on a graph to encode the input model entities, their connectivity and the invalidities (Figure 3.1). This global topological information ensures the model consistency. The method operates on that graph to keep track of all the modifications to build a new model in which invalid features are removed. Our strategy disconnects the graph edition from the actual geometrical model editing. The advantage is that all corrections are performed once all invalid features are identified, and all decisions are taken when operating on the geometry. The input model is taken as reference when building the final geometry. Avoiding mixing topological and geometrical modifications is an advantage, because it leads to instabilities or inconsistencies (e.g. Euler et al. 1998; Pellerin, Lévy, Caumon, and Botella 2014).

The input models are edited in three steps (Figure 3.1):

1. Detection of topological and geometrical invalid features, and encode them in the graph
2. Choose model editing operations to perform, encode them on the graph.
3. Build of the output geological model from the graph and the input model.

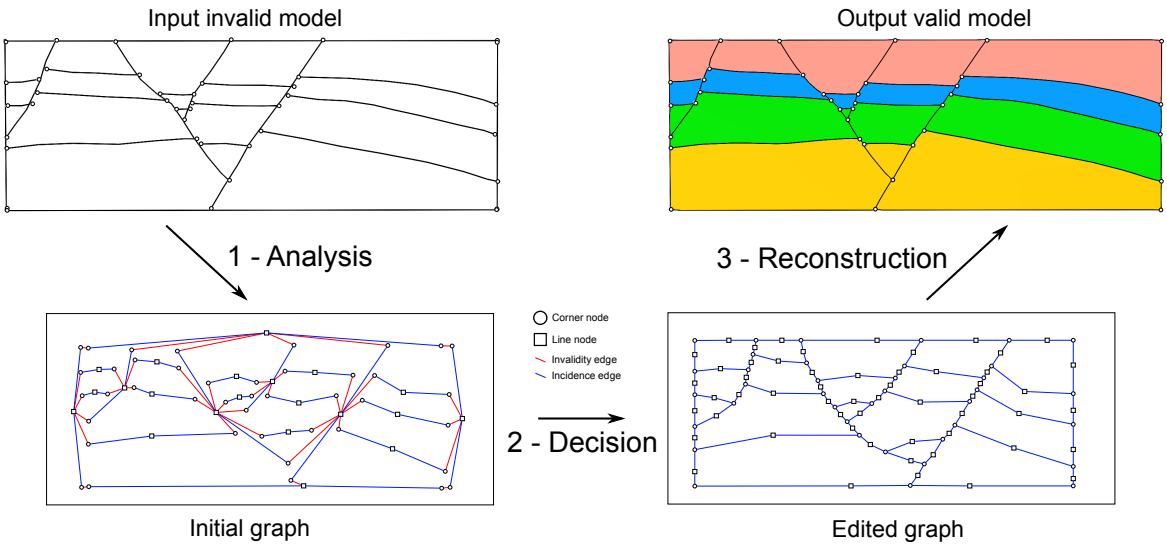


Figure 3.1: General workflow of the the method to automatically correct 2D models.

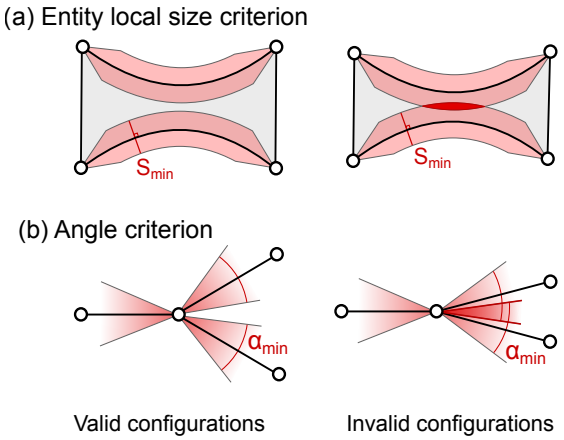


Figure 3.2: Detection of invalid model features from exclusion zone intersections.

Build the model graph The colored graph we use is an incidence graph to which we added edges representing invalid configurations. To each entity of the input model (CORNER and SURFACE) correspond a node of the model graph. There is a blue edge connecting two nodes if corresponding entities are incident to one another. There is a red edge connecting two nodes if the exclusion zones of the corresponding entities intersect (Figure 3.1). No red edges, means that the model is valid.

The construction of the graph is straightforward. To consider topological issues in the input model (cf. Section 2.1.3), such as horizon free borders, we use the graph to detect them and compute the nearest model entity so that a red edge may be built between the two. We implemented the exclusion zone of LINES as the union of the disks centered on all line points. We implemented the exclusion zones of corners as disks centered at the point. Intersections between exclusion zones can be detected by computing convex shape intersections, e.g. using the GJK algorithm (Gilbert et al. 1990). We identify all the entity points (not only its vertices) whose corresponding disks compose the intersections between exclusions zones. This geometrical information is stored alongside the graph. The detection of intersections can be implemented using exact predicates (e.g. Lévy 2015) for increased robustness.

Fix topology before geometry The next step is edit G to remove all red edges. This is the key idea of the method: operate on the connectivity of the model, encoded in the graph before modifying the defining the geometry of its entities.

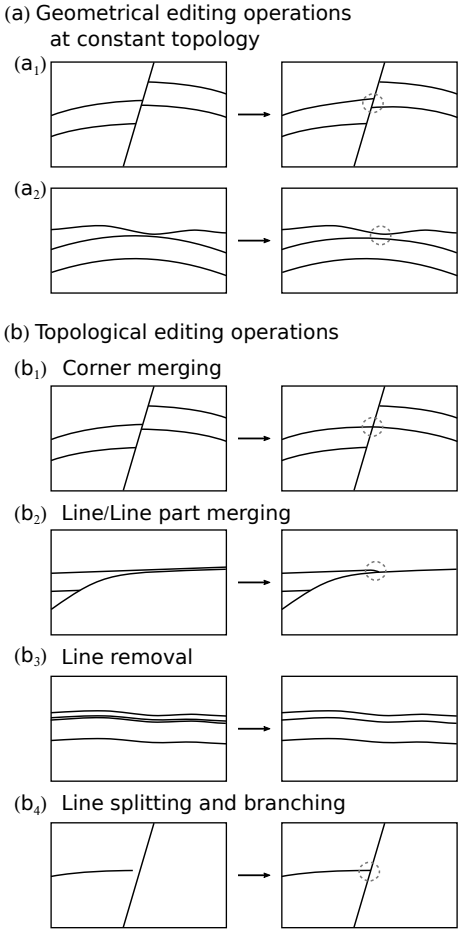


Figure 3.3: Geometrical and topological operations performed to automatically correct geological 2D models.

We perform successive graph elementary operations that correspond to operations on the geological model which will be actually performed at the next step, see Figure 3.3. The two geometrical operations that do not change the model topology aim at increasing the distance between entities. The three topological operations, merging, splitting and removal, modify the number of model entities and their connectivity; they aim at deleting thin model features by collapsing them.

(a) Edge deletion. Removing the edge connecting to entities corresponds to a modification of the model geometry that increase the distance between them and remove intersection between the excelusion zones.

(b) Edge contraction. Merging the two entities is only possible if both entities are of the same type. This corresponds to locally merge of two model entities.

(c) Node removal. Removing one entity (and subsequent red or blue edges) corresponds to the removal of a minor entity or a duplicated model feature. Prior input knowledge is required to determine which entity is removed.

(d) Node splitting in two nodes. This operation on LINES corresponds to its subdivision in two connected parts, e.g. to fix a hanging nearby corner.

The choice between the possible operation is constrained by (i) entity types (ii) a global strategy to preserve the topology or one that largley allows modifications, (iii) the physical meaning of the entities.

We first process red edges connecting a LINE and a CORNER, then the other edges, because this is the typical configuration for a horizon free-border invalid edge. Because the collision of LINE exclusion zones may not concern the whole LINES, operations on LINE- LINE invalid edges like edge contraction or node removal may need, as a pre-process, a node splitting.

Reconstruction of the geometrical model To determine the geometrical represention of the model entities encoded in the graph, we rely on the input model entity on which operations were performed. We are first set Corners, their output position is determined from the input model entities listed alongside the graph. It is set at the barycenter of input Corners, if LINES are involved that barycenter is projected on each of the LINE part, and the barycenter of these projections is chosen. To set LINES barycenter points of the LINES tomerge are computed. The final model can be meshed using various tools, e.g., Triangle (J. R. Shewchuk 1996), Gmsh (Geuzaine et al. 2009) or mmg2d[1].

3.1.2 Correction of 2D Models for Numerical Simulations

Sealing of a 2D geological model By repairing non-watertight vertical cross-sections, the method allows to mesh the model and run numerical simulations on it. A cross section in the 3D geological surface-based model built by (Collon et al. 2015) is given as an example Figure 3.4. The input model is not watertight; there are small gaps and overlaps at contacts between coal bed (pseudo-vertical lines) and fault (oblique lines) and boundary.

To seal the 2D model, minimal local entity size and minimal angle values are set to zero because the objective is to repair without simplification. We use additional geological knowledge: the coal veins are conformable one to another (they can not intersect). This constraints our method by preventing mutual branching of vein LINES. As a consequence, vein free borders can only be connected to the fault (red), to the Permo-Triassic base horizon (green) or to a boundary line. The generated output 2D model (Figure 3.4b) is watertight. The execution time for repairing this cross-section is 0.1 7 s on a laptop with a 2.50 GHz processor Intel®Core™ i7-6500U.

Accelerating numerical simulations An application presented in the manuscript of Pierre Anquez (Anquez 2019) shows how automatic correction of 2D geological cross sections can accelerated the numerical simulations for seismic risk evaluation. This work is a collaboration with Nathalie Glinsky, Diego Mercerat, and Paul Cupillard.

[1] http://www.mmgtools.org/

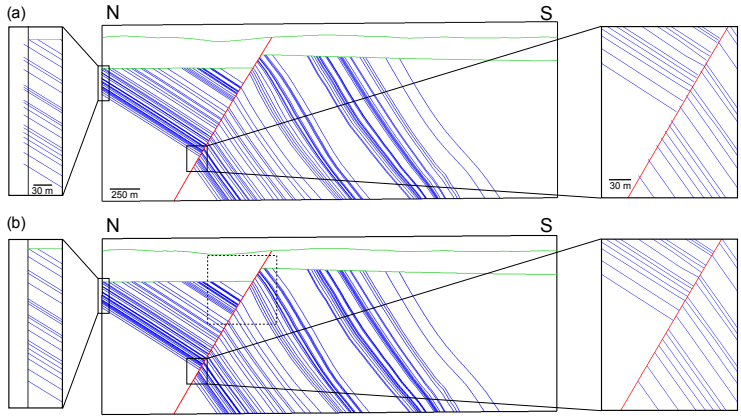


Figure 3.4: (a) Input non-watertight cross-section (note the gaps and the overlaps on detailed views). (b) Sealed output cross-section. Gaps and overlaps have been removed, resolving non-watertightness features.

Site effects are amplification phenomena of seismic waves in geological layers close to the surface in basins. They can have major consequences in terms of human lives when a seism occurs, e.g. the Michoacán seism on September 19, 1985 in Mexico. In south-east France, Nice is localized in an area where seisms are a risk and site effects can appear. The geological cross-section considered here, is a 2D profile crossing the alluvial bassin of the lower Var valley. Site effects are computed with the simulation of planar waves with a Discontinuous Galerkin solver on a triangle mesh (Peyrusse et al. 2014). This solver is time explicit, viscous-elastic, and the time step is constrained by smallest triangle height over the P-waves velocity to enforce the CFL condition. The reference model is a cross-section built from the 3D model of (Rohmer et al. 2020). A triangular mesh of this reference cross-section is generated with Mmg2d. The simulation of planar shear wave propagation takes 18 days. The details on the model construction and on the parameterization of the simulation are available in (Anquez 2019).

We compare this reference simulation result with the simulation on two models automatically simplified with our method. The first (M5) is obtained using a minimal size of de 0.18m and a minimal angle of 5 degrees. Complementary manual operations to remove small components were performed. The model was remeshed keeping the mesh size of the reference model. The second model (M6) is a further simplification of the first using the set of modifications to edit the model geometry at a fixed topology with a length of 0.18m, but changing the angle to 10 degrees. On Table 3.1, we can see that the simulation time is reduced from 18 days to less than 13 hours for (M5) and to 8 hours for M6. The simplifications targeting the small features increased the triangle height, impacting directly the time step required for the simulation. The comparisons of the results for the simplified models Figure 3.5 show that the results are slightly different but that the site effects are similar. The same frequencies are amplified at the same level leading to the same lengthening of the signals in the basin.

Models simplified automatically have strong advantages compared to the reference models or to models simplified by hand: (1) computational timings goes from 18 days to 6 - 12 hours (2) simplifications are controlled by objectives geometrical criteria, they are parsimonious and local.

	Δt(s)	Timing	Max Error Vx 12hz	Max error Vz 12Hz	Max Error Vx 25Hz	Max error Vz 25Hz
Ref	$7.29\ 10^{-7}$	18d 18h				
M5	$7.29\ 10^{-5}$	12h 37min	$3.39\ 10^{-4}$	$2.19\ 10^{-4}$	$1.34\ 10^{-3}$	$6.51\ 10^{-4}$
M6	$4.00\ 10^{-5}$	8h 10min	$3.38\ 10^{-4}$	$2.17\ 10^{-4}$	$1.34\ 10^{-3}$	$6.50\ 10^{-4}$

Table 3.1: Comparison of seismic simulation on reference model and on two simplified models. Errors on Vx and Vz are in m/s.

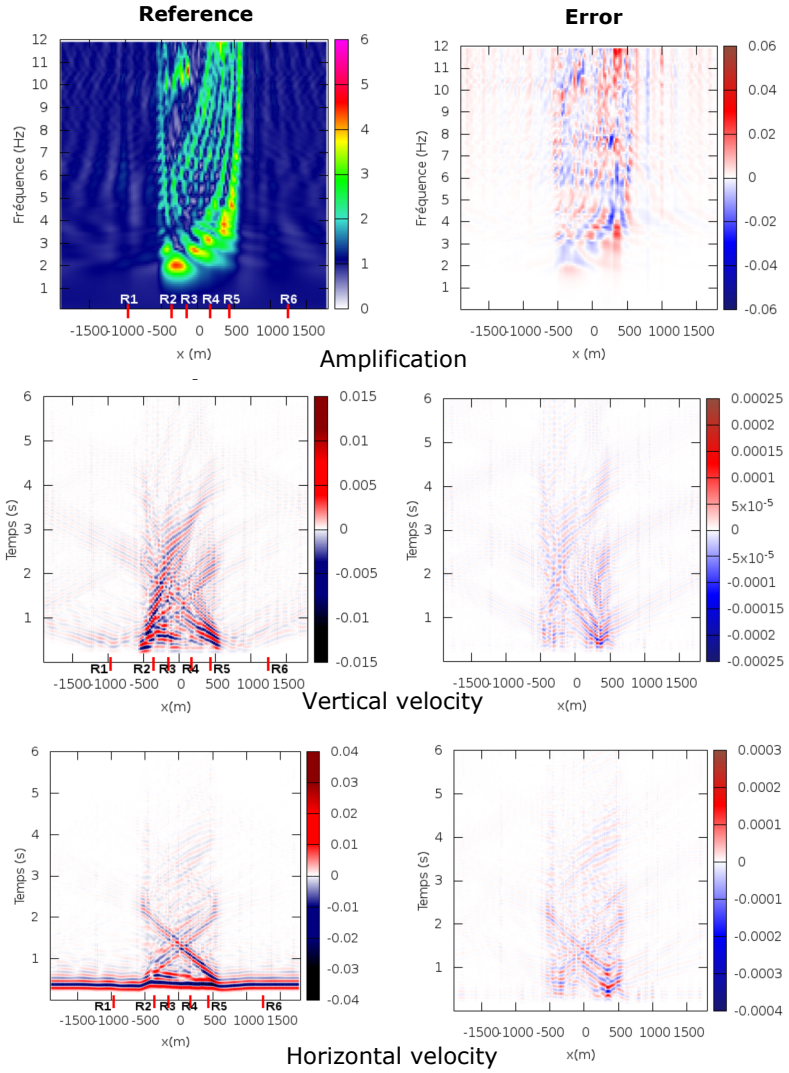


Figure 3.5: Comparison of the reference simulation with the automatically simplified model M6 in terms of amplification and velocities.

3.2 Simultaneous Surface Remeshing and 3D Model Correction

A more risky alternative to correct models is to authorize model topological modifications while remeshing. During my PhD thesis, I proposed such a method to remesh the surfaces of 3D model surfaces. From a set of conformal triangulated surfaces that are in contact along given lines and at given points, the method computes a set of surfaces meshed with triangles as equilateral as possible. The method relies on a global Centroidal Voronoi optimization to place the vertices of the final surfaces combined with combinatorial considerations to either recover or simplify the surfaces, lines and points of the input. When the final resolution is sufficient, the input contact lines, and points are also contact lines and points of the final model. However, when dealing with models with complex contacts, resolution may be insufficient and instead of a refinement strategy that may lead to too many points, we propose to locally merge some features of the input model. This ability to simplify the input model is particularly interesting when the model is to be volumetrically meshed.

- Pellerin, J., B. Lévy, and G. Caumon (2011). "Topological control for isotropic remeshing of nonmanifold surfaces with varying resolution: application to 3D structural models". In: *Proc. 13th IAMG conference*. Salzburg, Austria
- Pellerin, J., B. Lévy, G. Caumon, and A. Botella (2014). "Automatic surface remeshing of 3D structural models at specified resolution: A method based on Voronoi diagrams". In: *Computers & Geosciences* 62.0, pp. 103–116 **Best Paper Award in 2014 of the *Computers & Geosciences* journal**

Previous Work Whether it be in geological modeling or in other applications, the classical strategy to remesh a possibly high number of surfaces and their contact lines is to do so one after another. First, remesh each contact line, then parameterize each surface part to place it in a 2D space and remesh it with adequate strategy is a robust worklow implemented for example in Gmsh. The topology of the model is fixed and no modification is allowed, that is a key advantage when the input surface model is valid and at the correct level-of-detail, but may be a limitation if that is not the case. The second limitation of that strategy is that as all entities are meshed independently even when they are extremely close in space, the volumetric gridding is not the best possible.

An alternative approach was proposed in the computer graphics community to to generate high quality meshes by optimizing the coordinates of a fixed number of points to minimizing a Centroidal Voronoi Tessellation objective function (Lloyd 1982; Du, Faber, et al. 1999; Alliez, Colin de Verdiere, et al. 2005; Valette et al. 2008; Yan et al. 2009; Levy and Y. Liu 2010b; Z. Chen et al. 2012). Then, they locally recover theconnected components of the model (surface parts, contacts, and triple points) by adding one point per connected component. This simple strategy is very efficient to remesh thin layers and parallel contact lines, but fails when the resolution is not sufficient to capture other small-scale features in particular close triple points. We chose to modify these features, i.e. modify the model topology.

Remeshing methods that allow topological changes can be grouped in two main categories: those doing local modifications and those operating globally. The local modifications proposed by (Garland et al. 1997) generalize edge contractions and perform both close vertex contraction and edge contraction at the same time. Vertex clustering, introduced by (J. R. Rossignac et al. 1993), was used to remesh discrete fractures network by (Mustapha et al. 2011). A fixed size box scans the initial model in the three directions and the vertices inside it are replaced by a single vertex placed at the center of the box. Methods operating globally consider the entire model for the simplifications and subdivide into cells whose dimensions control the degree of simplification. For closed surfaces, (Andujar et al. 2002) use an octree, flag each cell as inside or outside the surface, and reconstruct a simplified closed surface from the remaining cells. Considering globally the model and working with a volumetric subdivision of the space permit to analyze locally the relationships between its different parts.

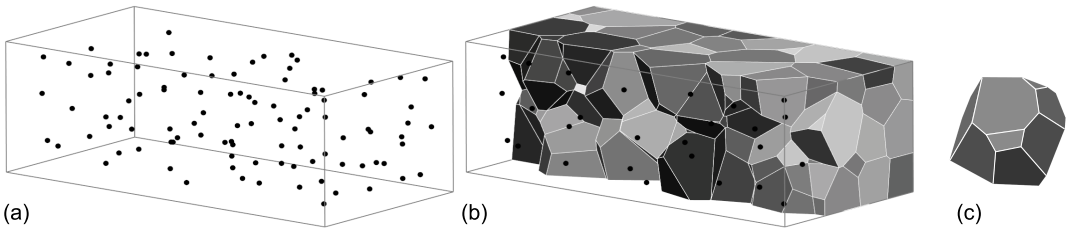


Figure 3.6: 3D Voronoi diagram (a) 200 sites are distributed in a box. (b) Solid slice in the Voronoi diagram of the sites cut by the box. (c) One Voronoi cell.

3.2.1 Overview

Input Triangulated surface models following the definition given in Section 2.1. The method is resilient to input model errors (non-water tightness, holes, non-conformal triangulation of the input surfaces along contact lines) but does not guarantee to fix them.

Output Our method computes a global remeshing of all the surfaces. The triangles of the output mesh are as equilateral as possible. Their quality does not depend on the input mesh quality and the method is resistant to degenerated triangles (skewed elements). The conformal contacts between the surfaces remain conformal, the non-conformal ones are kept non-conformal. Some modifications are done to simplify the model where contacts lines are too close.

Principle We use a Centroidal Voronoi Tessellation to place adequately a given number of points near the surfaces and contact lines of the model. A topological control is then used to determine the vertices and triangles of the final surface from the intersection of the Voronoi diagram of these points with the model while recovering the model components.

3.2.2 Fundamental Geometrical Objects

There are two fundamental geometrical objects that are ubiquitous in computational geometry and mesh generation (and in this manuscript): the Voronoi diagram and the Delaunay triangulation

3.2.2.1 Voronoi Diagram and Restricted Voronoi Diagram

A Voronoi diagram $VD(S)$ is defined relatively to set of points S To each point correspond a Voronoi cell: the subset of the space $\mathbb{R}^d$ closer to a point $p \in S$ than to any other point of S (e.g., Aurenhammer 1991) In $\mathbb{R}^d$, if we consider the Euclidean distance $\|.\|$, the Voronoi cell of p is defined as $V_p = \{x \in \mathbb{R}^n \mid \|px\| \leq \|qx\|, q \in S\}$. Voronoi cells are convex closed polyhedra that may bounded or not, and cover the space without overlapping (Figure 3.6). The Voronoi diagram was initially defined by (Voronoï 1908).

A Voronoi diagram subdivides the space and all the objects Ω included in that space. For a set of sites S and an object Ω the Restricted Voronoi diagram is defined as the intersection of the Voronoi diagram of S with Ω. The intersection of a Voronoi cell, V_p, with the object Ω is the restricted Voronoi cell of p to Ω and is defined by $V_{p\cap\Omega} = V_p \cap \Omega$. Note that to the contrary of Voronoi diagram elements, the elements of a restricted Voronoi diagram can have several connected components (Figure 3.9).

Remark Many useful Voronoi diagram generalizations have been developed by changing the distance measure between two points, changing the site natures *etc*, see the reviews of (Okabe et al. 2009) and (Aurenhammer 1991). It is also possible to work in non Euclidean spaces, for example the one defined by a surface embedded in a 3D space.

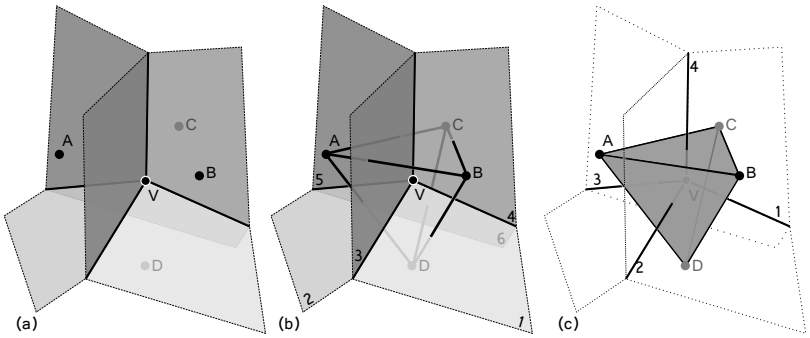


Figure 3.7: 3D Voronoi-Delaunay dual relationship (a) Voronoi vertex V is equidistant from the sites of cells A, B, C and D. (b) To each Voronoi facet containing V (numbers 1 to 6) corresponds one segment linking the sites of the 2 cells sharing the facet. (c) To each Voronoi edge containing V (numbers 1 to 4) corresponds a triangle linking the sites of the 3 cells sharing this edge.

3.2.2.2 Delaunay Triangulation and Restricted Delaunay Triangulation

A triangulation $T(S)$ of the n points $S = \{p_1, \ldots, p_n\} \in \mathbb{R}^d$ is a set of non overlapping simplices that covers exactly the convex hull $\Omega(S)$ of the point set, and leaves no point p_i isolated. The Delaunay triangulation $DT(S)$ of a point set S has the fundamental geometrical property that the interior of the circumsphere of any d-simplex is empty, i.e. it does not contain any point of S. If the point set S is in general position (in $\mathbb{R}^d$ if it contains no group of $d+1$ coplanar points and no group of $d+2$ cospherical points) then its Delaunay triangulation exists and is unique. To manage robustly degenerated cases is mandatory in mesh generation. These degeneracies disappear with an extremely small perturbations of site positions, that may only be symbolic (Edelsbrunner and Mücke 1990; Lévy 2015).

The mathematical properties of Delaunay triangulation, initially proposed by (Delaunay 1934), made it a favored object of study in computational geometry and meshing, see e.g. the excellent review in the book (Dey et al. 2009). Its well known geometric dual relationship with the Voronoi diagram is the base of many meshing algorithms Figure 3.7. In 3D, a Voronoi point correspond to a Delaunay triangulation cell, an edge to a face, a face to an edge, and a cell to a point.

The Restricted Delaunay Triangulation (Edelsbrunner and Shah 1997), is the geometric dual of the restricted Voronoi diagram (Figure 3.8). It is the subset of the elements of the Delaunay triangulation of the points S such that the dual restricted Voronoi cell/face/edge/point is not empty.

3.2.2.3 Surface Remeshing Using the Restricted Delaunay Triangulation

The restricted Delaunay triangulation of given points S to an input surface remeshes this surface. However, it does not necessarily have the same topology as the input surface (not homeomorphic), see for example Figure 3.8. To ensure that the restricted Delaunay triangulation is homeomorphic to an input manifold surface it is sufficient to verify the topological ball property (Edelsbrunner and Shah 1997): all the restricted Voronoi cells (respectively facets and edges) of S to Ω are topological disks (respectively segments and points) and all the restricted Voronoi cells (respectively facets and edges) of S to the boundary of Ω are topological segments (respectively points and the emptyset). A geometrical mean to enforce the topological ball property is to have an ε-sampling of the input surface (Amenta and Bern 1999): if for each point x of the surface Ω, there is a sampling point of S at a distance smaller than $\varepsilon \times lfs(x)$, where $\varepsilon < 0.3$ and lfs denotes the local feature size (distance to the medial axis of Ω).

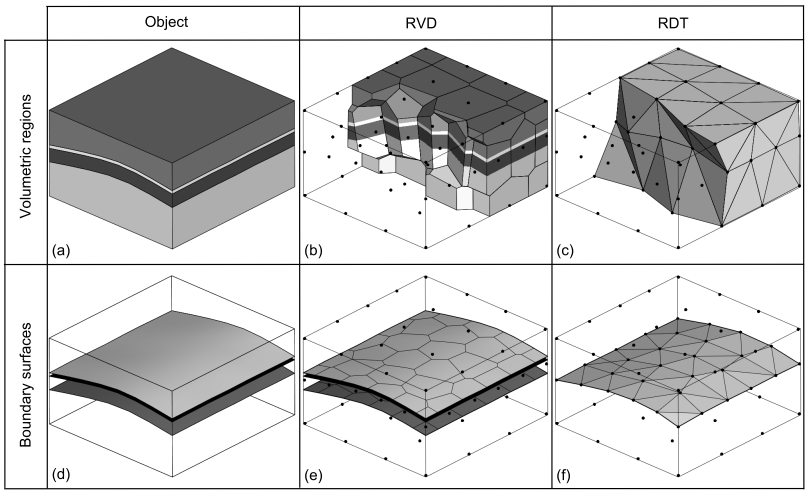


Figure 3.8: Restricted Voronoi diagrams (RVD) and their dual restricted Delaunay triangulations (RDT) to a B-rep model surfaces or regions.

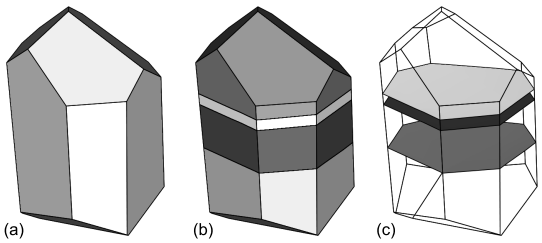


Figure 3.9: (a) A 3D Voronoi cell. (b) Its restriction to the model regions has 4 connected components. (c) Its restriction to the model surfaces has 3 connected components.

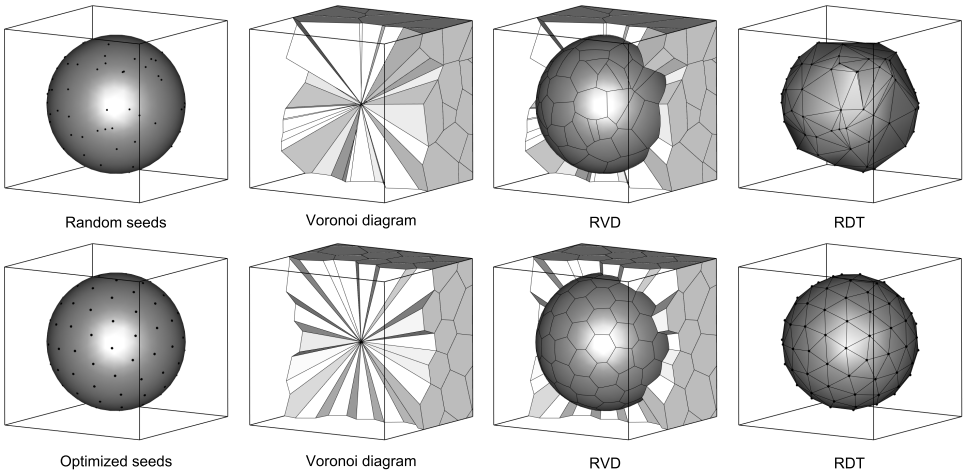


Figure 3.10: Optimization of 100 sampling points on a sphere. After optimization the restricted Voronoi diagram has compact well-shaped cells and the restricted Delaunay triangulation has almost equilateral triangles.

3.2.3 Surface Remeshing à la Voronoi

This section details the two main steps of the surface remeshing method: the optimization of a given number of sampling points on the model (Section 3.2.3.1) and the building of the final mesh from the connected components of the restricted Voronoi diagram of the sampling points (Section 3.2.3.2).

3.2.3.1 Optimization of the Mesh Vertex Coordinates

First, a fixed number of points are placed so that they are a good sampling of the surface model. Each point samples the model in the sense that it represents the part of the model closest to it than to any other point. The input number of points then determines the resolution at which the model will be remeshed, it may be computed from the square root of the model area divided by the target edge length. To have a good sampling the idea I developed was to optimize the point positions such that their restricted Voronoi diagram to the model is centroidal.

Centroidal Voronoi Diagram The space subdivision determined by a Voronoi diagram of sites randomly distributed is random, and its optimization appears in numerous statistics, image processing, or mesh generation problems. The goal is to optimize point placement to reach a specific objective, for example, minimize the distances between each one of the sites and points inside their Voronoi cell. This optimization tends toward a specific Voronoi diagram: a centroidal Voronoi diagram.

The Voronoi diagram of points S is centroidal if each site is at the centroid $p*$ of its Voronoi cell V_p, with ρ the density function we have:

$$p* = \frac{\int_{V_p} y\rho(y)dy}{\int_{V_p} \rho(y)dy} \tag{3.1}$$

Let's consider the problem of the computation of the partition of a domain Ω in k regions Ω_i and the positions of k points s_i that minimizes the function:

$$F((s_i, \Omega_i)_{i=1...k}) = \sum_{i=1}^{k} \int_{y\in\Omega_i} \rho(y)\|y - s_i\|^2 dy \tag{3.2}$$

This function evaluates the sum of the square distance between points s_i and points of the region Ω_i with the same index. (Du, Faber, et al. 1999) show that, to minimize this function, it is necessary that the regions Ω_i are the Voronoi cells of points s_i and that each point must be at the centroid of its Voronoi cell. They also show that this function has the same minimums than the function in which $\Omega_i = V_i$. Function F parameters are only point positions, the integration being done on the Voronoi cells V_i. To compute a centroidal Voronoi diagram it is then sufficient to minimize this function of the site positions. In practice, it is very difficult to obtain a global minimum, and a local minimum is often considered satisfactory. Note also that, in the same space, for a given number of sites, there are multiples centroidal Voronoi diagrams, all of them minimizing the function F.

Restricted Centroidal Voronoi Diagram The restricted voronoi diagram of a point set S to a domain Ω is centroidal if each site p is at the centroid of its restricted voronoi cell. By integrating function F only on the restricted cells we have:

$$F_\Omega = \sum_{i=1}^{k} \int_{y\in V_i\cap\Omega} \rho(y)\|y - s_i\|^2 dy \tag{3.3}$$

When points s_i belong to domain Ω, the restricted centroidal voronoi diagram is constrained (Du, Gunzburger, et al. 2003). A site optimization example on a sphere is given Figure 3.10. After the optimization restricted Delaunay triangles are almost equilateral.

The centroidal Voronoi diagram is one of the key elements of the variational meshing methods that consist in optimizing the positions of all final mesh vertices before building their (restricted) Delaunay triangulation (Figure 3.10) (Alliez, Cohen-Steiner, et al. 2005; Tournois et al. 2009; Tournois 2009; Dardenne et al. 2009; Levy and Y. Liu 2010b). The two features distinguishing this approach from more classical tetrahedron meshing methods is that (1) the number of vertices can be fixed and that (2) tetrahedra shape and quality are globally optimized by the objective function.

Implementation The three steps to perform the optimization of a given number of points over a model Ω are:

1. The initial random placement of the sampling points on the model surfaces is done using the algorithm given by (Lévy and Bonneel 2013).

2. The computation of the restricted Voronoi diagram is done using the fast parallelized method also described by (Lévy and Bonneel 2013).

3. The contributions of each cell of the restricted Voronoi diagram to the objective function and to its gradient is computed following (Yan et al. 2009) for the CVT energy and the above for the boundary term. The minimization of the objective function F is done with a L-BFGS algorithm (Nocedal 1980). The optimization can be stopped when the norm of the gradient is inferior to a given value. From our experience, convergence is very fast, and in practice we stop the optimization process after 100 iterations. Specific convergence rates are discussed by (Y. Liu et al. 2009). In all the cases we considered the input mesh resolution does not impact the convergence while increased feature density slightly decreases the convergence.

Once the points have been optimally distributed, we compute their restricted Voronoi diagram to the structural model and compute the connected components of the restricted Voronoi cells to determine the vertices and triangles of the output mesh.

3.2.3.2 Connecting the Dots

Surface remeshing Let's first consider a (non-geological) model in which the different surface connected components do not intersect and have no boundary, e.g. two nested spheres Figure 3.11. The two surfaces are sampled by 100 points whose optimized positions are between the spheres. To recover the input surface parts, we set one point for each connected component of the restricted Voronoi cell. There is then one triangle to build for each point shared by three restricted Voronoi cells (Figure 3.12). The obtained mesh is a dual of the connected components of the restricted centroidal Voronoi diagram, it is closer to the input mesh than the restricted Delaunay triangulation (one sphere in this case). The multi-nerve theorem gives that it is homotopy equivalent to the input model (Colin de Verdière et al. 2012).

Line remeshing Now, consider a surface with a boundary. Similarly to what happens for surfaces, the boundary is not correctly remeshed if the number of points is too small.. To remesh them correctly, each point is replaced by as many points as the number of connected components of the intersection between its Voronoi cell and the boundary lines (Figure 3.13).

These additional points challenge the triangle building step since one connected component of one restricted Voronoi cell may correspond to several points. As a consequence, the dual of points shared by three cells may not be a triangle and additional polygons, dual of the edges shared by two cells and intersecting twice the boundary, should be built between close boundary lines (see the gray segments on Figure 3.13c & d).

The more intersections between the restricted Voronoi cell and the model boundary, the more vertices in this cell. This may lead to configurations where the polygons to build intersect (Figure 3.14a). To avoid this, if there are more than two points for a connected component of a restricted Voronoi cell, they are merged. This makes our method more robust, but at the cost of modifications of the surface connections that are questionable from a geological point of view and depend highly on the optimized point positions.

Corner remeshing The last entities to account for are Corners (triple points), i.e. points at the intersection of at least two contact lines. To recover all the corners, one should put one point for each of them. So, to fully reconstruct the input model, we need to have for each restricted Voronoi cell one point per triple point, one point per contact, and one point per surface part.

When there are more than one Corner in a connected component of a restricted Voronoi cell to the boundary, i.e. the final resolution is not sufficient, we merge them (Figure 3.14b). When this merging operation is done, the previously described merging is also performed. This way, each restricted

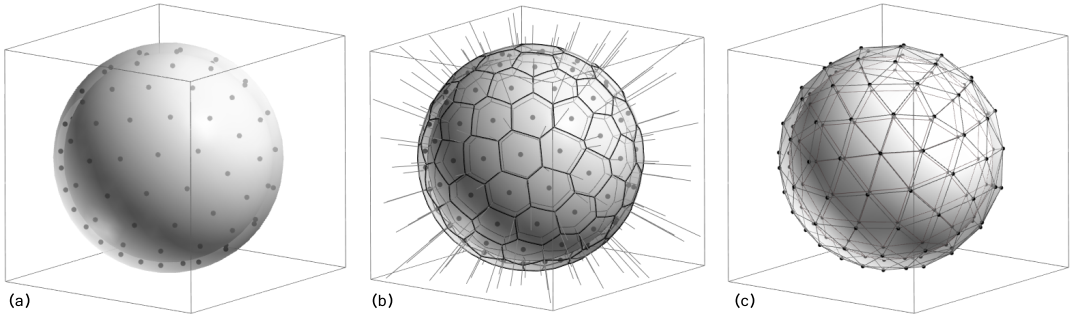


Figure 3.11: Nested spheres remeshing. (a) 100 optimized points are sandwiched between two spheres (b) Each restricted Voronoi cell has 2 connected components (c) Dual of the connected components of the restricted Voronoi diagram (see Figure 3.12).

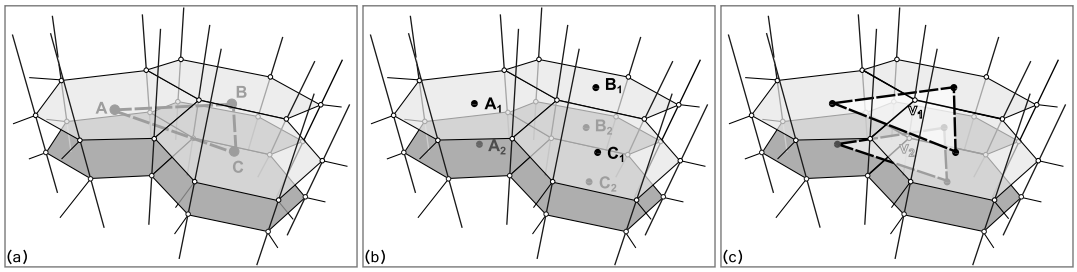


Figure 3.12: Remeshing two close surfaces. (a) 3 points (**A, B** and **C**) are sandwiched between two close surfaces, their restricted Voronoi cell s have two connected components, but in the dual restricted Delaunay triangulation there is only one triangle **ABC**. (b) Each point is replaced by two points. (c) Two triangles $\mathbf{A}_1\mathbf{B}_1\mathbf{C}_1$ and $\mathbf{A}_2\mathbf{B}_2\mathbf{C}_2$ are built corresponding to the points $\mathbf{v}_1$ and $\mathbf{v}_2$ shared by three restricted cell connected components.

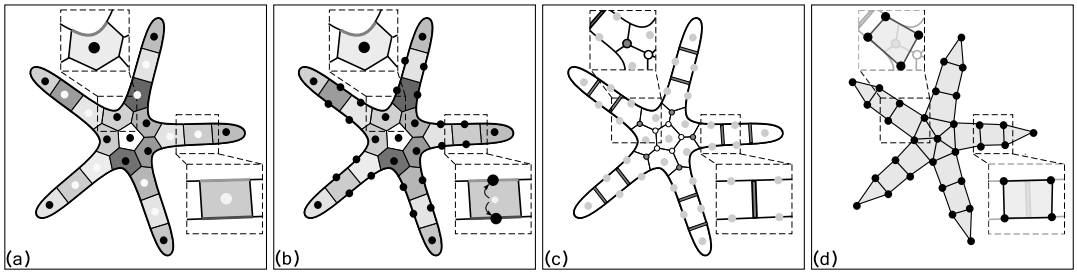


Figure 3.13: Remeshing a surface with a boundary. (a) 21 points sample the star; Voronoi cells of the white points intersect twice the boundary, those of the black points intersect it once or not at all. (b) Each white point is replaced by two points, one per intersection of the cell with the boundary (c) The elements dual of the polygons of the final mesh are: regular restricted Voronoi vertices (white), restricted Voronoi vertices neighboring at least a restricted Voronoi cell with two points (gray), and the segments on the restricted Voronoi edges intersecting twice the boundary. (d) Final mesh is made of regular Delaunay triangles and quads.

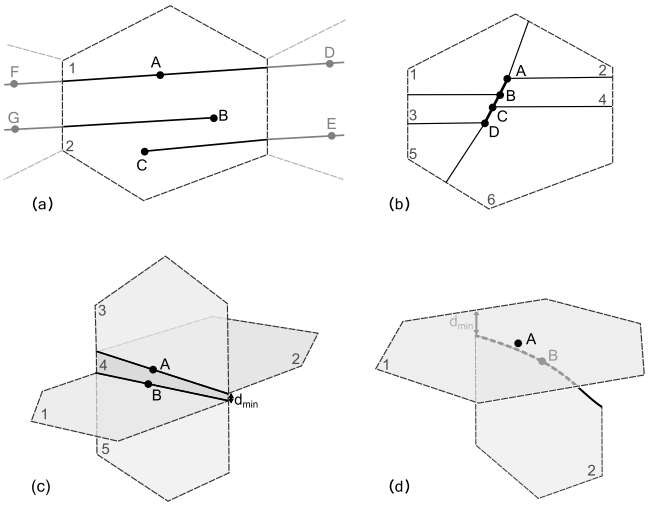


Figure 3.14: Configurations leading to modifications of the model. (a) The central restricted Voronoi cell component intersect 3 Corners (**A, B, C**), the polygons to build with these points **ABGF** and **ADEC** intersect. We merge the 3 corners. (b) Contact lines (black lines) cut the cell into 6 connected components. The 4 triple Corners (**A, B C, D**) are merged. (c & d) If $\mathbf{d}_{min}$ is inferior to the given resolution, points **A** and **B** are merged.

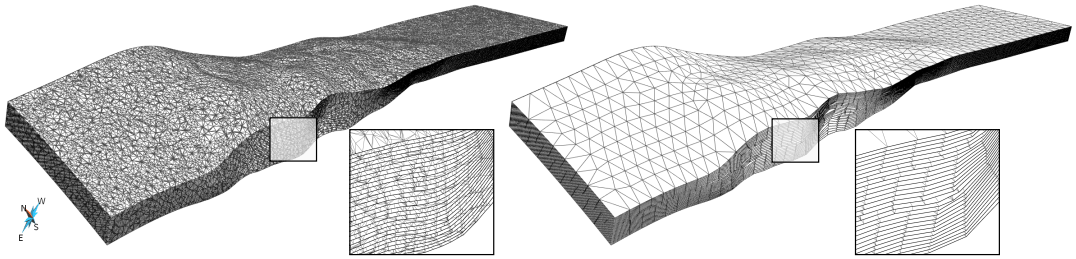


Figure 3.15: Coal veins remeshing. 29 sub-vertical surfaces delimit thin coal veins. 1000 sampling points are sufficient to remesh the model decreasing the number of triangles from nearly one million to 35 thousands.

Voronoi cell part has 1 or 2 points and the quads or triangles to build with these points do no intersect. The last modification is the merging of the vertices that correspond to close features, close meaning that the distance between them is inferior to a specified input value (Figure 3.14c & d). This is a way to make the model easier to mesh and simplify very small features by removing small fault throws and joining fault tips close to another fault.

Implementation The input of the method is a restricted Voronoi diagram, a polygonal surface obtained from the intersection of the input model with the Voronoi diagram of the optimized points. Each polygon of the restricted Voronoi diagram is the intersection of one triangle of the input model and one Voronoi cell whose ids are known.

1. Each restricted Voronoi cell and its connected components are built using that information.
2. The vertices to put for each restricted Voronoi cell are computed.
3. The last step is to build the cells linking these vertices.

Dataset	Horizons	Faults	Main challenges	Credentials	Figures
Coal veins *10.2km × 1.3km × 280m*	29	0	Thin layers	Courtesy of Gocad consortium	Figure 3.15
Forward *110m × 65m × 40m*	7	0	Thin layers, onlaps	(Laurent 2013)	Figure 3.16
Detachment *22km × 14km × 7.7km*	8	1	Thin layers	Courtesy of Chevron (Guzofski et al. 2009)	Figure 3.17
Leipzig *1.2km × 1.2km × 0.4km*	2	9	Fault network	Courtesy of Total	Figure 3.17
Lambda *6km × 4.5km × 1.9km*	2	13	Low angle faults fault throws	Courtesy of Gocad consortium	Figure 3.17
DFN *13km × 11km × 4km*	2	200	Fracture relations	Courtesy of Gocad consortium	Figure 3.18
HC *18km × 10km × 10.2km*	7	2	Thin layers, Inverse fault	Courtesy of Harvard-Chevron	Figure 3.18
Cloudspin *14.7km × 12km × 2km*	3	10	Low angle faults, fault throws	Courtesy of PDGM and Schlumberger	Figure 3.19
Clyde *12km × 10.3km × 1.7km*	4	22	Fault intersections, fault throws	Confidential	Figure 2.2 and 3.22
Nancy *11km × 3km × 1.4km*	7	26	Complex faults, fault throws	Courtesy of Total	Figure 3.20
Annot *11km × 5.5km × 2.8km*	9	3	Thin layers, onlap, fault throws	(Salles et al. 2011)	Figure 3.21
Sandbox *3.5km × 3km × 0.5km*	8	33	Fault throws	Courtesy of IFPEN (Colletta et al. 1991)	Figure 3.21

Table 3.2: Main features and challenges of the 12 remeshed models.

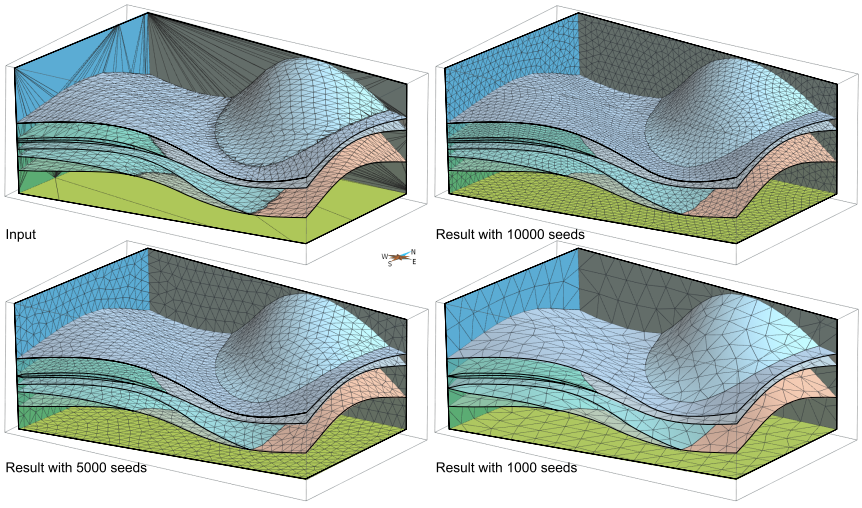


Figure 3.16: Forward model remeshing with varying resolutions. Input model shows 3 three challenges for remeshing, very thin layers, major layer thickness variations and low-angle contacts between horizons due to onlapping geometries.

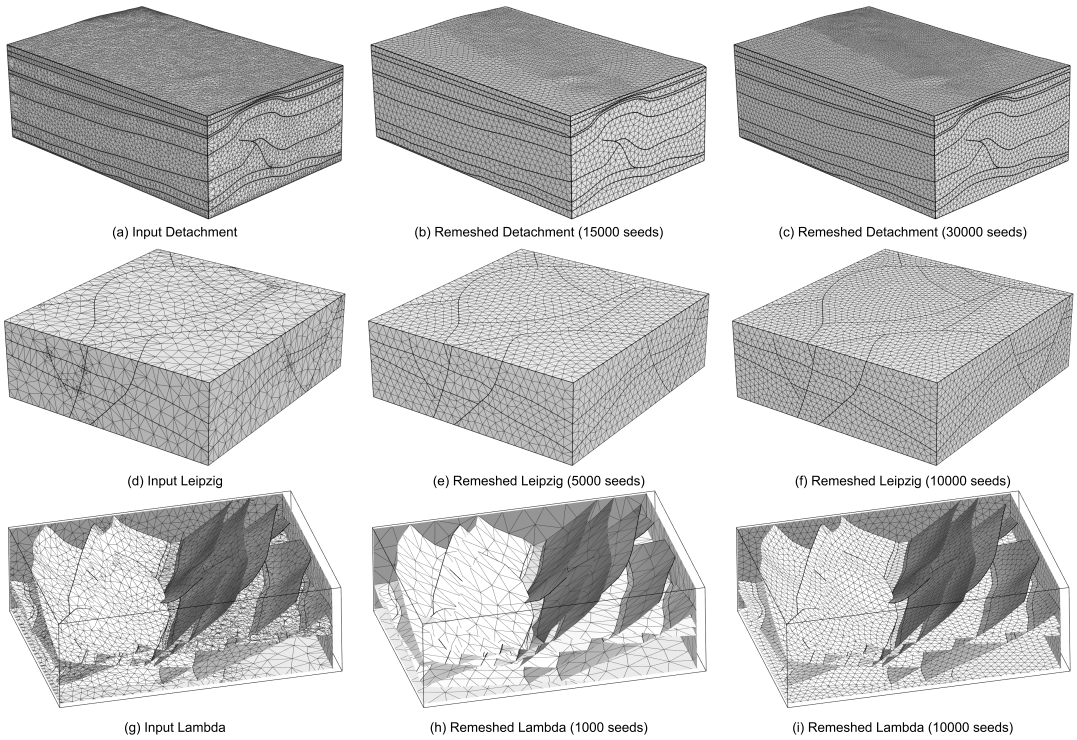


Figure 3.17: Remeshing the Detachment, Leipzig, and Lambda models

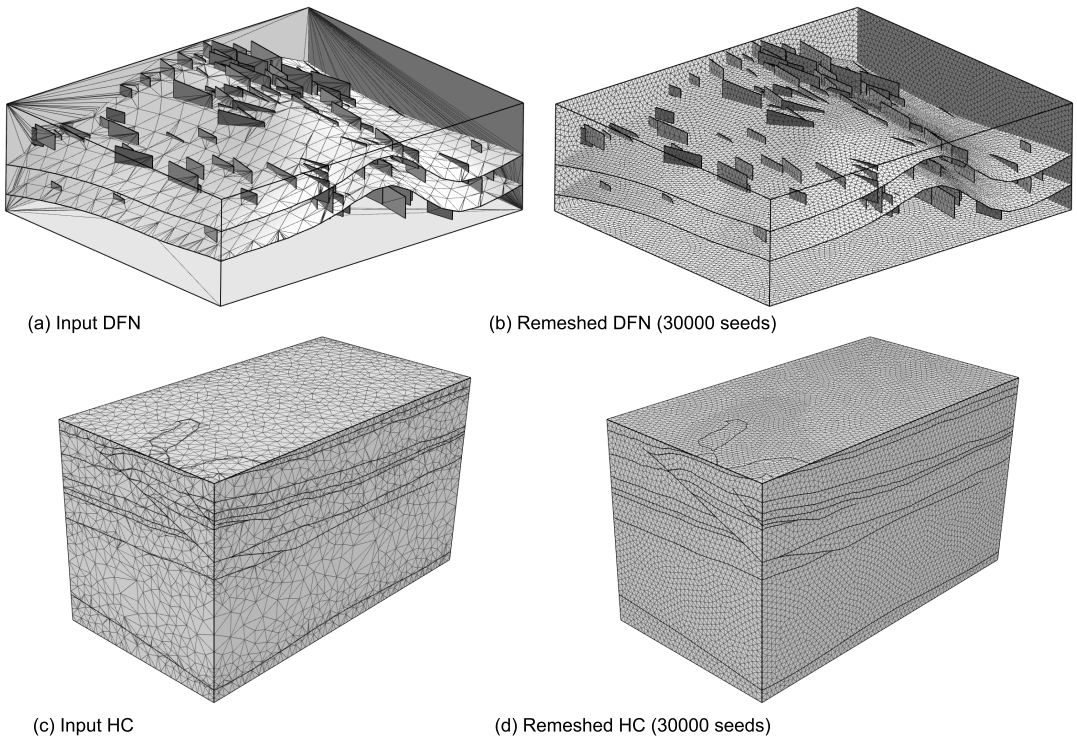


Figure 3.18: Remeshing the DFN and HC models.

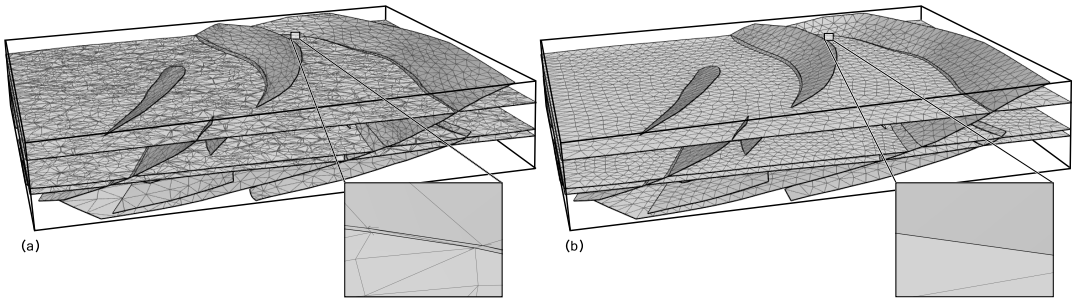


Figure 3.19: Cloudspin model remeshing. (a) Input model with very small throw near fault ends. (b) Output surfaces, remeshing was done with 5000 sampling points, contact lines are locally merged.

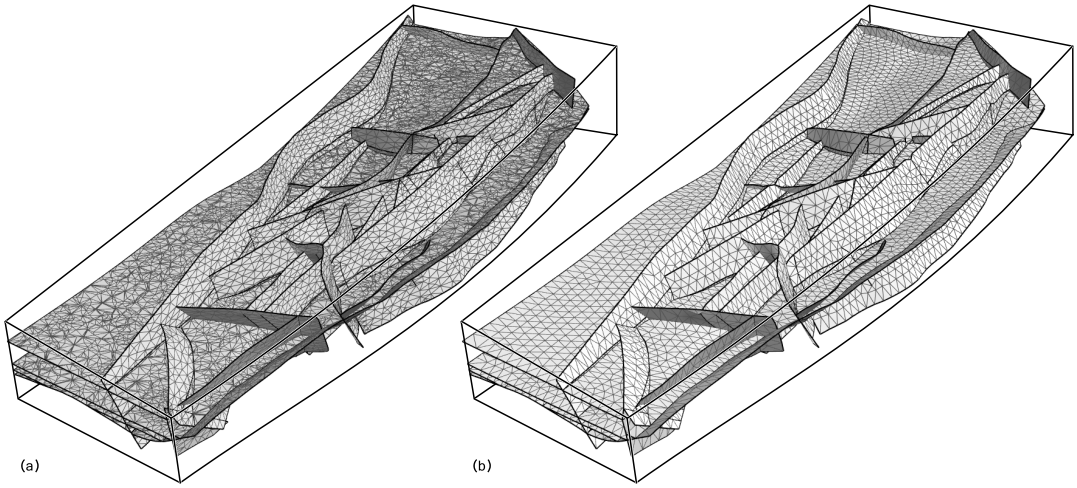


Figure 3.20: Nancy model remeshing. (a) Input model (b) Model remeshed with 10000 sampling points.

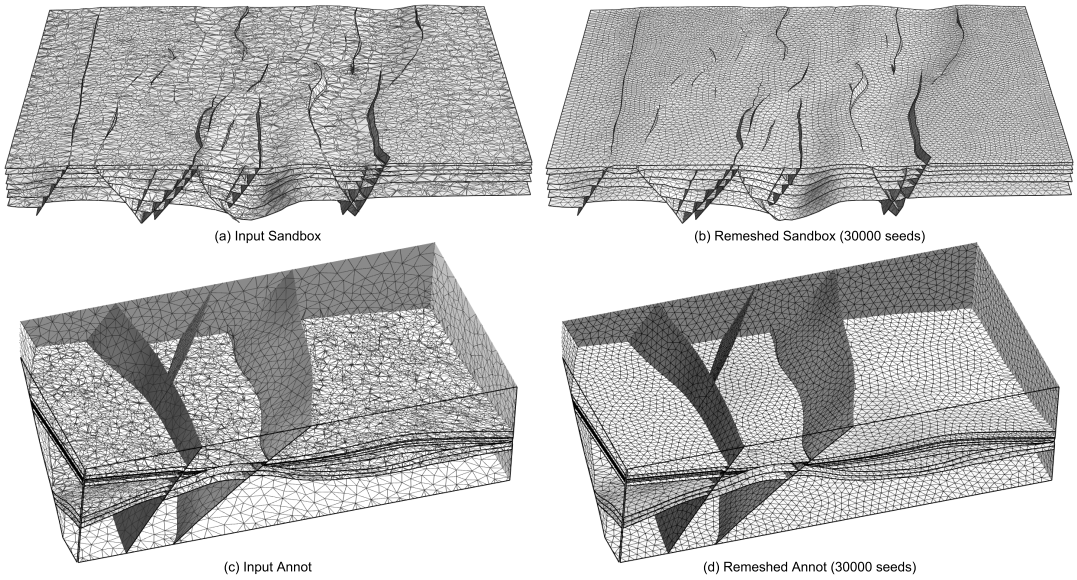


Figure 3.21: Challenging model remeshing: Sandbox and Annot.

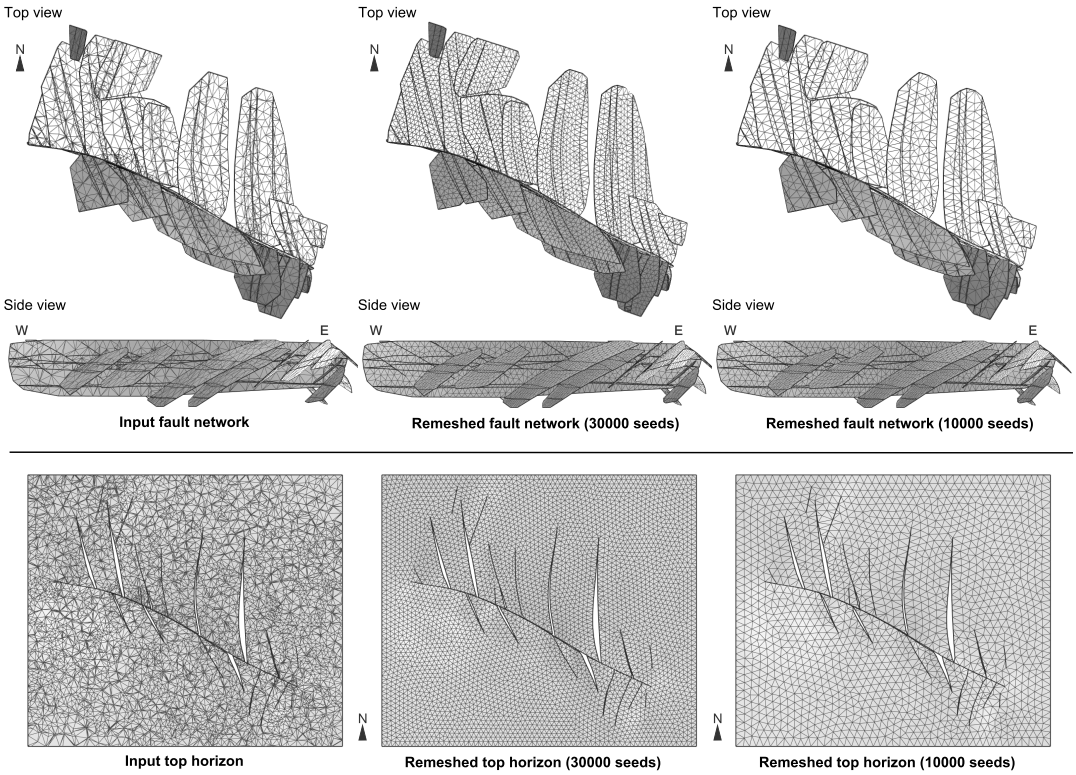


Figure 3.22: Remeshing the Clyde model with 30000 and 10000 sampling points

Dataset	#Pts	#ver	#tri	#S	#L	#C	Angles (deg)		Avg qual.	Timing (s)	
							Min	< 30 (%)		Sampling	Mesh
Veins	**Input**	**471337**	**923286**	**29**	**0**	**0**	**0.01**	**22.07**	**0.55**		
	1000	19928	35484	29	0	0	1.04	3.46	0.81	153	33
Forward	**Input**	**10151**	**13588**	**46**	**76**	**40**	**0.01**	**20.21**	**0.53**		
	1000	2457	3400	47	71	35	0.56	3.25	0.82	12	1
	5000	7961	12849	46	74	36	1.85	1.00	0.88	28	1
	10000	14096	23845	46	77	41	1.30	0.58	0.89	46	2
Detachment	**Input**	**61480**	**109098**	**50**	**84**	**46**	**7.86**	**2.19**	**0.79**		
	15000	25165	44599	50	79	41	2.60	0.50	0.90	61	3
	30000	42219	76514	50	82	44	3.85	0.24	0.91	85	3
Leipzig	**Input**	**9286**	**11344**	**188**	**320**	**166**	**0.98**	**7.54**	**0.73**		
	5000	8578	11694	186	287	135	4.05	1.61	0.84	13	1
	10000	14911	22281	186	295	143	6.08	0.96	0.86	23	2
Lambda	**Input**	**24528**	**37553**	**132**	**256**	**177**	**0.09**	**15.52**	**0.62**		
	1000	3416	3711	144	236	147	0.98	8.35	0.73	20	2
	10000	16113	24223	134	242	155	0.57	1.46	0.86	33	3
DFN	**Input**	**7876**	**7723**	**435**	**307**	**481**	**0.00**	**40.70**	**0.33**		
	30000	38081	62070	435	300	480	1.00	0.88	0.88	48	3
HC	**Input**	**39919**	**70684**	**80**	**140**	**80**	**0.12**	**19.54**	**0.60**		
	30000	37255	65198	80	141	81	0.78	0.25	0.91	72	5
Cloudspin	**Input**	**18313**	**30049**	**97**	**124**	**112**	**0.00**	**25.59**	**0.52**		
	5000	10778	16494	91	117	103	0.45	3.52	0.82	21	3
	10000	17725	28725	94	134	124	0.30	2.25	0.85	29	3
Clyde	**Input**	**41355**	**69343**	**227**	**387**	**303**	**0.01**	**20.39**	**0.56**		
	10000	15551	23367	206	318	244	0.72	2.51	0.85	38	4
	30000	38884	64850	220	354	282	0.15	1.30	0.88	66	9
Nancy	**Input**	**59115**	**85775**	**753**	**1307**	**774**	**0.00**	**25.98**	**0.50**		
	10000	24840	30445	719	1096	626	0.13	6.88	0.75	43	21
	50000	79087	119309	741	1259	774	0.05	2.53	0.84	106	49
Annot	**Input**	**76204**	**130403**	**332**	**590**	**300**	**0.00**	**20.61**	**0.56**		
	3000	12737	18253	301	455	212	0.51	5.69	0.77	40	7
	20000	41761	68240	311	522	264	0.55	2.31	0.85	80	10
Sandbox	**Input**	**72927**	**109267**	**500**	**688**	**713**	**0.00**	**21.98**	**0.53**		
	30000	52498	77688	503	890	897	0.00	1.11	0.85	93	28

Table 3.3: Remeshing result statistics. For each model, input model and produced results are compared in terms of mesh sizes, numbers of components (surfaces, lines, and corners), and quality (minimum angle, percentage of triangles with an angle inferior to 30 degrees, average quality)

3.2.4 Discussion

Some of the remeshing results of this Voronoi based method are given on (Table 3.2, Figure 3.17, Figure 3.22). For more models, see the original paper (Pellerin, Lévy, Caumon, and Botella 2014). With input mesh containing between several thousand triangles to almost one million triangles, computation times to optimize the point positions (100 iterations) and build the final mesh range between 13s and 150s on a 8-core laptop (in 2013).

Advantages Looking back, these results are quite good. The method is completely automatic and performs in several minutes or less on real data geological surface models. The method produces a result whatever the input, the potentially extremely bad quality of the input triangles and the nasty features that might have stopped another method are not an issue. These small features like small holes, gaps, thin features, whose size is smaller than the resolution of the final mesh are altered. The final global resolution in terms of number of vertices is controlled by the user. The output quality of the triangles does it depend on the quality of the input mesh. The output triangle quality $Q = 6\,S\,/(\,\sqrt{3}\,h_{max}\,p)$ where S is the area of the triangle, h_{max} the length of its longest edge, and p its half perimeter (Frey and Borouchaki 1999) is good.

Drawbacks The method has however major drawbacks that makes it as it impracticable for remeshing of industrial models. First of all, no mesh size can be respected, that is one of the key job of a meshing software in engineering and despite some attempts Voronoi based methods are bad at it. Another major problem for engineers (less so for computer graphics applications) is the fact that there is absolutely no guarantee whatsoever on which of the input entities (corners, lines, points) will be preserved, the ones that are modified are not tracked and no one knows what happened or how to measure it. No guarantee is provided for the output model validity, when using an valid input surface model the output could be invalid. there is no guarantee either on the distance between the input and output models (link with mesh size). Some incomplete post-processing was implemented to fix some defects in the final meshes (1) to improve the quality and ensure conformity along contact lines when triangles are the dual of small triangular restricted Voronoi cell (2) to fix triangle intersections that can appear when making some simplifications. The real issue with this type of method is that there is the consistency between local modifications and the global model consistency is not adequately managed (Is it possible?) Locally modifications can be made that have a impact on a global scale, or at least on the neighboring local areas. The worst is probably that the modifications performed depend on the intersection of a Voronoi cell with the surface model. Move slightly that cell and the modifications may change significantly.

One click meshing is a bad dream. In general, performing topological modifications while keeping a surface model valid is extremely difficult, and this is without mentioning the remeshing part. This is why, in practice, most of the model modifications are checked by the user that can either use semi-automatic methods or even do it by hand (component suppression, mesh repair, etc.) My current point of view (in 2020) is that mesh generation and model modification/simplification/defeaturing must be separated for robustness puposes. Clicking on a button to get a mesh may be relevant and may become true for computer graphics applications. But I do not believe in this approach for engineering applications where validity, control, and quality must be guaranteed to the user (and not in theory). This is why in the work of Pierre Anquez we focused on the simplification of the model, and left the subsequent meshing to an adequate software.

Part II

Volumetric Meshing

Chapter 4

Tetrahedral Meshing à la Delaunay

Motivations The Delaunay triangulation is a fundamental geometrical object that associates a unique triangulation $DT(S)$ to a given point set S in general position. This triangulation and its geometrical dual, the Voronoi diagram, have locality properties that makes them extremely useful to analyse points sets and of course to generate meshes. Generating robustly and efficiently the Delaunay triangulations is one key ingredient to many efficient robust and efficient tetrahedral meshing. It is important to remember that, for meshing purposes, the Delaunay triangulation is not a mesh but a mean to compute a mesh because its boundary is the convex hull of the points and not the boundary of the object to mesh.

Delaunay triangulation algorithms face a major challenge: the availability of more and more massive point sets. LiDAR or other photogrammetry technologies are used to survey the surface of entire cities and even countries, like the Netherlands (Martinez-Rubi et al. 2016). The size of finite element mesh is also growing with the increased availability of massively parallel numerical solvers. It is now common to deal with meshes of over several hundred millions of tetrahedra (Rasquin et al. 2014; Ibanez et al. 2016; Vázquez et al. 2016).

Can we parallelize tetrahedral meshing? Parallelizing 3D Delaunay triangulations and Delaunay-based meshing algorithms is however very challenging because the algorithm is essentially sequential. The most part of today's clusters have two levels of parallelism. Distributed memory systems contain thousands of nodes, each node being itself a shared memory system with multiple cores. In recent years, nodes have seen their number of cores and the size of their memory increase, with some clusters already featuring 256-core processors and up to 12TB of RAM (Nystrom et al. 2015; Fu et al. 2016). As many-core shared memory machines are becoming standard, Delaunay triangulation algorithms designed for shared memory should not only scale well up to 8 cores but to several hundred cores.

Contributions This chapter presents one of the main result of the PhD of Célestin Marot whom I co-advised during my post-doc in UCLouvain in Belgium.

Before parallelizing, we ensure that our sequential implementation of the Delaunay triangulation algorithm in 3D is efficient. It is about three times faster than existing open-source implementations and is able to triangulate a million points in about 2 seconds Section 4.1. Our implementation is similar to those, being based on the incremental Delaunay insertion algorithm, but the gain in performance is to ascribe to the details of the specific data structures we have developed, to the optimization of geometric predicates evaluation, and to specialized adjacency computations. We then propose a scalable parallel version of the Delaunay triangulation algorithm devoid of heavy synchronization overheads (Section 4.2). The domain is partitioned using the Hilbert curve and conflicts are detected with a simple coloring scheme. On a AMD® EPYC 64-core machine, we have been able to generate three billion tetrahedra in less than a minute (about 10^7 points per second). We finally demonstrate how this efficient Delaunay triangulation algorithm can be easily integrated in a tetrahedral mesh generation process where the input is the boundary surfaces of the domain to mesh (Section 4.3).

A first version of this work was presented in a conference research note in 2017, the final results were published in 2018:

- Marot, C., J. Pellerin, J. Lambrechts, and J.-F. Remacle (2017). "Toward one billion tetrahedra per minute". In: *26th International Meshing Roundtable, Research Notes*. Barcelona, Spain

- Marot, C., J. Pellerin, and J. Remacle (2018). "One machine, one minute, three billion tetrahedra". en. In: *International Journal for Numerical Methods in Engineering* 117.9, pp. 967–990

The corresponding reference implementation is open-source and available in Gmsh 4 at `http://gmsh.info`. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement ERC-2015-AdG-694020).

4.1 Sequential Delaunay

The fastest serial algorithm to build the 3D Delaunay triangulation $DT(S)$ is the Bowyer-Watson algorithm, which works by incremental insertion of points in the triangulation. The Bowyer-Watson algorithm was devised independently by Bowyer and Watson in 1981 (Bowyer 1981; Watson 1981). Very fast open-source implementations who benefited from various optimizations through the decades are available: Tetgen (Si 2015), CGAL (Boissonnat, Devillers, Teillaud, et al. 2000) and Geogram (Levy 2016). They are designed similarly and offer therefore similar performances, they triangulate a million points in about 6 seconds on one core of a high-end laptop.

The work of Célestin was to first accelerate, if possible, the sequential implementation of the incremental Delaunay insertion algorithm in 3D. He succeeded, to be 3 times faster, this gain in performance is to ascribe to the details of the specific data structures we have developed, to the optimization of geometric predicates evaluation, and to specialized adjacency computations.

4.1.1 Bowyer-Watson Algorithm

Given a point set S, let DT_k be the Delaunay triangulation of the subset $S_k = \{p_1, \dots, p_k\} \subset S$. The *Delaunay kernel* is the procedure to insert a new point $p_{k+1} \in \Omega(S_k)$ into DT_k, and construct a new valid Delaunay triangulation DT_{k+1} of $S_{k+1} = \{p_1, \dots, p_k, p_{k+1}\}$. The *Delaunay kernel* can be written in the following abstract manner:

$$DT_{k+1} \leftarrow DT_k - C(DT_k, p_{k+1}) + \mathcal{B}(DT_k, p_{k+1}), \tag{4.1}$$

where the Delaunay cavity $C(DT_k, p_{k+1})$ is the set of all tetrahedra whose circumsphere contains p_{k+1} (Figure 4.1b), whereas the Delaunay ball $\mathcal{B}(DT_k, p_{k+1})$ is a set of tetrahedra filling up the polyhedral hole obtained by removing the Delaunay cavity $C(DT_k, p_{k+1})$ from DT_k (Figure 4.1c).

The state-of-the-art incremental insertion algorithm (Algorithm 1) that we implemented has five main steps that are described in the following.

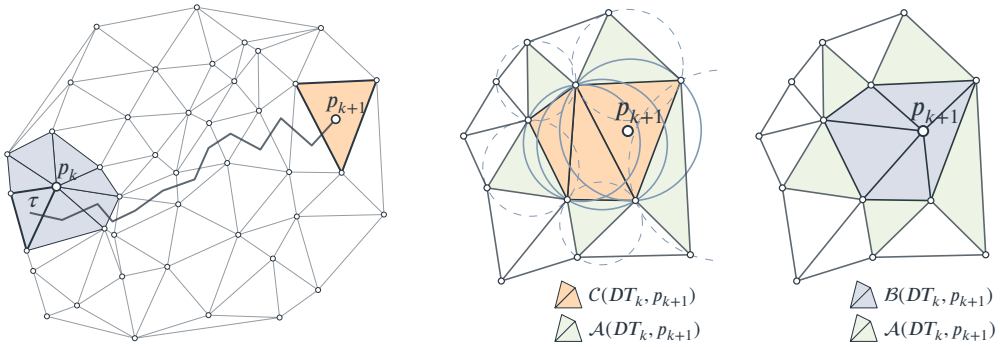
Init The triangulation is initialized with the tetrahedron formed by the first four non-coplanar vertices of the point set S. These vertices define a tetrahedron τ with a positive volume.

Sort The complexity of the Delaunay triangulation algorithm depends on the number of tetrahedra to traverse during the Walk and on the number of tetrahedra in the cavities. To minimize the former, the points are sorted before insertion so that two points that have close indices are close in space. For very large point sets, the Sort function becomes the bottleneck, we implemented a very fast sorting procedure (Section 4.1.3) that eliminates this limitation.

Algorithm 1 Sequential computation of the Delaunay triangulation DT of a set of vertices S

Input: S
Output: $DT(S)$

```
1:  function SEQUENTIAL_DELAUNAY(S)
2:      τ ← INIT(S)                                  ▷ τ is the current tetrahedron
3:      DT ← τ
4:      S' ← SORT(S \ τ)                             ▷ Section 4.1.3
5:      for all p ∈ S' do
6:          τ ← WALK(DT, τ, p)
7:          C ← CAVITY(DT, τ, p)                     ▷ Section 4.1.4
8:          DT ← DT \ C
9:          B ← DELAUNAYBALL(C, p)                   ▷ Section 4.1.5
10:         DT ← DT ∪ B
11:         τ ← t ∈ B
12:     return DT
```



(a) WALK **(b)** CAVITY **(c)** DELAUNAYBALL

Figure 4.1: Insertion of a vertex p_{k+1} in the Delaunay triangulation DT_k. **(a)** The triangle containing p_{k+1} is obtained by walking toward p_{k+1}. The WALK starts from $\tau \in \mathcal{B}_{p_k}$. **(b)**The CAVITY function finds all cavity triangles (orange) whose circumcircle contains the vertex p_{k+1}. They are deleted, while cavity adjacent triangles (green) are kept. **(c)** The DELAUNAYBALL function creates new triangles (blue) by connecting p_{k+1} to the edges of the cavity boundary.

	HXT	Geogram	TetGen	CGAL
SEQUENTIAL_DELAUNAY	**12.7**	**34.6**	**32.9**	**33.8**
INIT + SORT	0.5	4.2	2.1	1.3
INCREMENTAL INSERTION	12.2	30.4	30.8	32.5
WALK	1.0	2.1	1.6	1.4
`orient3d`	0.7	1.4	1.1	≈ 0.5
CAVITY	6.2	11.4	≈ 10	14.9
`inSphere`	3.2	6.2	5.6	10.5
DELAUNAYBALL	4.5	12.4	≈ 15	15.3
Computing sub-determinants	1.3	/	/	/
Other operations	0.5	4.5	≈ 4	≈ 1

Table 4.1: Timings for the different steps of the Delaunay incremental insertion (Algorithm 1) for four implementations: HXT (Marot, Pellerin, and J. Remacle 2018), Geogram (Levy 2016), Tetgen (Si 2015) and CGAL (Boissonnat, Devillers, Teillaud, et al. 2000). Timings in seconds are given for 5 million points (random uniform distribution). The ≈ prefix indicates that no accurate timing is available.

WALK The goal of this step is to identify the tetrahedron τ_{k+1} enclosing the next point to insert p_{k+1}. The search starts from a tetrahedron τ_k in the last Delaunay ball $\mathcal{B}(DT_k, p_k)$, and walks through the current triangulation DT_k in direction of p_{k+1} (Figure 4.1a). We say that a point is visible from a facet when the tetrahedron defined by this facet and this point has negative volume. The WALK function thus iterates on the four facets of τ, selects one from which the point p_{k+1} is visible, and then walks across this facet to the adjacent tetrahedron. This new tetrahedron is called τ and the WALK process is repeated until none of the facets of τ sees p_{k+1}, which is equivalent to say that p_{k+1} is inside τ (see Figure 4.1a).

The visibility walk algorithm is guaranteed to terminate for Delaunay triangulations (De Floriani et al. 1991). If the points have been sorted, the number of walking steps is essentially constant (J.-F. Remacle 2017). Our implementation of this robust and efficient walking algorithm is similar to the existing ones.

CAVITY Once the tetrahedron $\tau \leftarrow \tau_{k+1}$ has been identified, the function CAVITY finds all tetrahedra whose circumsphere contain p_{k+1} and deletes them. The Delaunay cavity $\mathcal{C}(DT_k, p_{k+1})$ is simply connected and contains τ (J. Shewchuk 1997), it is then built using a breadth-first search algorithm. The core and most expensive operation of the CAVITY function is the `inSphere` predicate, which evaluates whether a point e is inside/on or outside the circumsphere of given tetrahedron. This CAVITY function is thus an expensive function of the incremental insertion, which accounts for about 33% of the total computation time (Table 4.1). To accelerate this, sub-components of the `inSphere` predicate can be precomputed Section 4.1.4.

DELAUNAYBALL Once the cavity has been carved, the DELAUNAYBALL function first generates a set of new tetrahedra adjacent to the newly inserted point p_{k+1} and filling up the cavity, and then updates the mesh structure. In particular, the mesh update consists in the computation of adjacencies between the newly created tetrahedra. This is the most expensive step of the algorithm, with about 40% of the total computation time (Table 4.1). To accelerate this step, the general purpose elementary operations like tetrahedron creation/deletion or adjacency computation can be replaced by batches of optimized operations making benefit from a cavity-specific data structure (Section 4.1.5).

4.1.2 Data Structure Dedicated to Triangulation

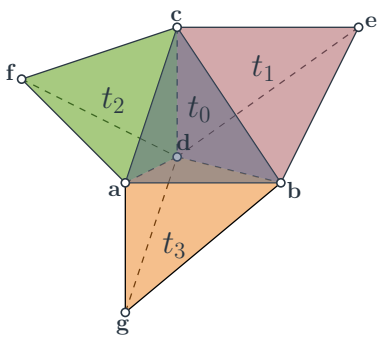
One key for enhancing performances of a Delaunay triangulation algorithm (and maybe any algorithm) is the optimization of the data structure used to store the object of interest. Various data structure designs have been proposed that are very flexible and allow representing hybrid meshes, high order meshes, add mesh elements of any type, or manage adjacencies around vertices, edges, etc. The versatility of such general purpose data structures has a huge cost, both in terms of storage and efficiency. Here, our aim is to have a structure as lightweight and fast as possible, dealing exclusively with 3D triangulations. The implementation proposed in HXT is coded in plain `C` language, with arrays of doubles, floats, and integers to store both the mesh topology and geometry. This seemingly old-style coding has important advantages in terms of optimization and parallelization because it compels us to use simple and straightforward algorithms. The mesh data only contains vertices and tetrahedra explicitly, and all topological and geometrical information can be deduced from it. One very useful and relatively expensive information is the adjacency between tetrahedra.

Vertices Vertices are stored in a single array of structures `point3d_t` (see Listing 4.1). For each vertex, in addition to the vertex coordinates, a `padding` variable is used to align the structure to 32 bytes (3 doubles of 8 bytes each, and an additional padding variable of 8 bytes sum up to a structure of 32 bytes) and conveniently store vertex related information. Memory alignment ensures that a vertex does not overlap two cache lines during memory transfer. Modern computers usually work with cache lines of 64 bytes. The padding variable in the vertex structure ensures that a vertex is always loaded in one single memory fetch. Moreover, aligned memory allows to take advantage of the vectorization capabilities of modern microprocessors.

Tetrahedra Each tetrahedron knows its 4 vertices and its 4 neighboring tetrahedra. These two types of adjacencies are stored in separate arrays. The main motivation for this storage is flexibility and,

```
typedef struct {
    double coordinates[3];
    uint64_t padding;
} point3d_t;

typedef struct {
    struct {
        uint32_t* vertex_ID;
        uint64_t* neighbor_ID;
        double* sub_determinant;
        uint64_t num;             // number of tetrahedra
        uint64_t allocated_num;   // capacity [in tetrahedra]
    } tetrahedra;

    struct {
        point3d_t* vertex;
        uint32_t num;            // number of vertices
        uint32_t allocated_num; // capacity [in vertices]
    } vertices;
} mesh_t;
```

Listing 4.1: The aligned mesh data structure `mesh_t` we use to store the vertices and tetrahedra of a Delaunay triangulation in 3D.



memory index	`vertex_ID`	`neighbor_ID`
$4t_0$	a	$4t_1+3$
$4t_0+1$	b	$4t_2+3$
$4t_0+2$	c	$4t_3+3$
$4t_0+3$	d	–
:		
$4t_1$	b	–
$4t_1+1$	c	–
$4t_1+2$	d	–
$4t_1+3$	e	$4t_0+0$
:		
$4t_2$	a	–
$4t_2+1$	d	–
$4t_2+2$	c	–
$4t_2+3$	f	$4t_0+1$
:		
$4t_3$	a	–
$4t_3+1$	b	–
$4t_3+2$	d	–
$4t_3+3$	g	$4t_0+2$

Figure 4.2: Four adjacent tetrahedra : t_0, t_1, t_2, t_3 and one of their possible memory representations in the `tetrahedra` data structure given in Listing 4.1. `tetrahedra.neighbor_ID`$[4t_i + j]/4$ gives the index of the adjacent tetrahedron opposite to `tetrahedra.vertex_ID`$[4t_i + j]$ in the tetrahedron t_i and `tetrahedra.neighbor_ID`$[4t_i + j]$ gives the index where the inverse adjacency is stored.

once more, memory alignment. Keeping good memory alignment properties on a tetrahedron structure evolving with the implementation is cumbersome. In addition, it provides little to no performance gain in this case. On the other hand with parallel arrays, additional information per tetrahedron (e.g. a color for each tetrahedron, sub-determinants etc.) can be added easily without disrupting memory layout. Each tetrahedron is identified by the indices of its four vertices in the `vertices` structure. Vertex indices of tetrahedron `t` are read between positions `4*t` and `4*t+3` in the global array `tetrahedra.vertex_ID` storing all tetrahedron vertices. Vertices are ordered so that the volume of the tetrahedron is positive. An array of double, `sub_determinant`, is used to store 4 values per tetrahedron. This space is used to speed up geometric predicate evaluation (see Section 4.1.4).

Adjacencies By convention, the i-th facet of a tetrahedron is the facet opposite the i-th vertex, and the i-th neighbor is the tetrahedron adjacent to that facet. In order to travel efficiently through the triangulation, facet indices are stored together with the indices of the corresponding adjacent tetrahedron, thanks to an integer division and its modulo. The scheme is simple. Each adjacency is represented by the integer obtained by multiplying by four the index of the tetrahedron and adding the internal index of the facet in the tetrahedron. Take, for instance, two tetrahedra t_1 and t_2 sharing facet f, whose index is respectively i_1 in t_1 and i_2 in t_2. The adjacency is then represented as `tetrahedra.neighbor_ID[4t`$_1$`+i`$_1$`]=4t`$_2$`+i`$_2$ and `tetrahedra.neighbor_ID[4t`$_2$`+i`$_2$`]=4t`$_1$`+i`$_1$. This multiplexing avoids a costly looping over the facets of a tetrahedron. The fact that it reduces by a factor 4 the maximum number of elements in a mesh is not a real concern since, element indices and adjacencies being stored as 64 bytes unsigned integers, the maximal number of element in a mesh is $2^{62} \simeq 4.6\ 10^{18}$, which is huge. Note also that division and modulo by 4 are very cheap bitwise operations for unsigned integers: $i/4 = i \gg 2$ and $i\%4 = i\&3$.

Memory footprint The key to an efficient data structure is the balance between its memory footprint and the computational cost of its modification. We do not expect to generate meshes of more than `UINT32_MAX` vertices, i.e. about 4 billion vertices on one single machine. Each vertex therefore occupies 32 bytes, 24 bytes for its coordinates and 8 bytes for the padding variable. On the other hand, the number of tetrahedra could itself be larger than 4 billion, so that a 64 bits integer is needed for element indexing. Each tetrahedron occupies 80 bytes, $4 \times 4 = 16$ bytes for the vertices, $4 \times 8 = 32$ bytes for the neighbors, 32 bytes again for the sub-determinants. In average a tetrahedral mesh of n vertices has a little more than $6n$ tetrahedra. Thus, our mesh data structure requires $\approx 6 \times 80 + 32 = 512$ bytes per vertex.

4.1.3 Fast Point Spatial Sorting

An efficient spatial sorting scheme has been proposed by Boissonnat et al. (Boissonnat, Devillers, and Hornus 2009) and is used in state-of-the-art Delaunay triangulation implementations. The main idea is to first shuffle the vertices, and to group them in rounds of increasing size (a first round with, e.g., the first 1000 vertices, a second with the next 7000 vertices, etc.) as described by the Biased Randomized Insertion Order (BRIO) (Amenta, Choi, et al. 2003). Then, each group is ordered along a space-filling curve. The space-filling curve should have the property that successive points on the curve are geometrically close to each other. With this spatial sorting, the number of walking steps between two successive cavities remains small and essentially constant (J.-F. Remacle 2017). Additionally, proceeding by successive rounds according to the BRIO algorithm tends to reduce the average size of cavities.

Space-filling curve coordinates The Hilbert curve is a continuous self-similar (fractal) curve. It has the interesting property to be space-filling, i.e., to visit exactly once all the cells of a regular grid with $2^m \times 2^m \times 2^m$ cells, $m \in \mathbb{N}$. A Moore curve is a closed Hilbert curve. in the sense that points close to each other on the curve are also close to each other in $\mathbb{R}^3$ (Haverkort et al. 2010; Abel et al. 1990). Any 3D point set can be ordered by a Hilbert/Moore curve according to the order in which the curve visits the grid cells that contains the points. The Hilbert/Moore index is thus an integer value $d \in \{0, 1, 2, \ldots, 2^{3m} - 1\}$, and several points might have the same index. The bigger m is for a given point set, the smaller the grid cells are, and hence the lower the probability of having two points

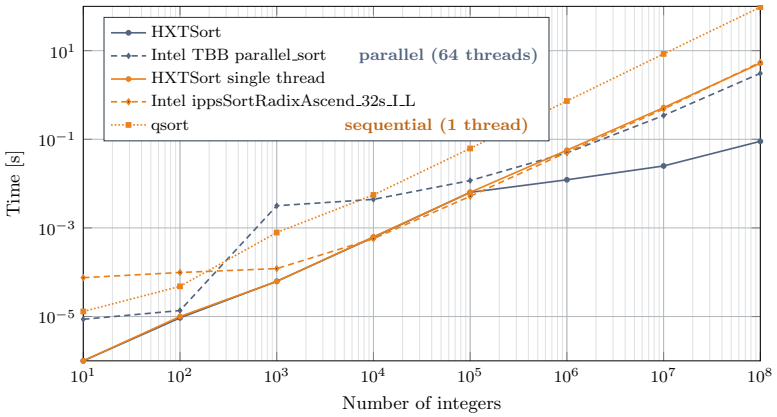


Figure 4.3: Performances of `HXTSort` for sorting $\{key, value\}$ pairs on an Intel® Xeon Phi™ 7210 CPU and comparison with widely used implementations.

with the same Hilbert/Moore index. Given a 3D point set with n points, there are in average $n/2^{3m}$ points per grid cell. Therefore, choosing $m = k\ log_2(n)$ with k constant ensures the average number of points in grid cells to be independent of n. If known in advance, the minimum mesh size can also be taken into account: m can be chosen such that a grid cell never contain more than a certain number of vertices, thus capturing non-uniform meshes more precisely. For Delaunay triangulation purposes, the objective is both to limit the number of points with the same indices (in the same grid cell) and to have indices within the smallest range as possible to accelerate subsequent sorting operations. There is thus a balance to find, so that Hilbert indices are neither too dense nor too sparse.

Computing the Hilbert/Moore index of one cell is a somewhat technical point. The Hilbert/Moore curve is indeed fractal, which means recursively composed of replicas of itself that have been scaled down by a factor two, rotated and optionally reflected. Those reflections and rotations can be efficiently computed with bitwise operations. Various transformations can be applied to point coordinates to emulate non-regular grids, a useful functionality we will resort to in Section 4.2.2. (Hamilton et al. 2008) give extensive details on how to compute Hilbert indices.

Radix sort Once the Hilbert/Moore indices have been computed, points can be sorted accordingly in a data structure where the Hilbert/Moore index is the key, and the point the associated value. An extremely well suited strategy to sort bounded integer keys is the radix sort(Blelloch et al. 1999; Zagha et al. 1991; Sohn et al. 1998; Satish, Harris, et al. 2009), a non-comparative sorting algorithm working digit by digit. The base of the digits, the radix, can be freely chosen. Radix sort has a $O(wn)$ computational complexity, where n is the number of $\{key, value\}$ pairs and w the number of digits (or bits) of the keys. In our case, the radix sort has a complexity in $O(mn) = O(n\ log(n))$, where n is the number of points and $m = k\ log_2(n)$ the number of levels of the Hilbert/Moore grid. In general, m is small because a good resolution of the space-filling curve is not needed. Typically, the maximum value among keys is lower than the number of values to be sorted. We say that keys are *short*. Radix-sort is able to sort such keys extremely quickly.

Literature is abundant on parallel radix sorting and impressing performances are obtained on many-core CPUs and GPus (Wassenberg et al. 2011; Polychroniou et al. 2014; Satish, Kim, et al. 2010; Bell et al. 2011; Sengupta et al. 2007). However, implementations are seldom available and we were not able to find a parallel radix sort implementation properly designed for many-core CPUs. We implemented `HXTSort`, that is available independently as open source at `https://www.hextreme.eu/hxtsort`. Figure 4.3 compares the performances of `HXTSort` with `qsort`, `std::sort` and the most efficient implementations that we are aware of for sorting 64-bit key and value pairs. Our implementation is fully multi-threaded and takes advantage of the vectorization possibilities offered by modern computers such as the AVX512 extensions on the Xeon PHI. It has been developed primarily for sorting Hilbert indices, which are typically short. We use different strategies depending on the key bit size. These are the reason why `HXTSort` outperforms the Boost Sort Library, GNU's and Intel's parallel implementation of the standard library and Intel TBB when sorting Hilbert indices.

4.1.4 Improving Cavity: Spending Less Time in Geometric Predicates

The central operation of the Delaunay kernel is the construction of the cavity $C(DT_k, p_{k+1})$ formed by all the tetrahedra $\{a, b, c, d\}$ whose circumscribed sphere encloses p_{k+1} (Figure 4.1). This step of the incremental insertion represents about one third of the total execution time in available implementations (Table 4.1). The cavity is initiated with a first tetrahedron τ containing p_{k+1} determined with the Walk function (Algorithm 1 and Figure 4.1a), and then completed by visiting the neighboring tetrahedra with a breadth-first search algorithm.

Optimization of the `inSphere` predicate The most expensive operation of the Cavity function is the fundamental geometrical evaluation of whether a given point e is inside, exactly on, or outside the circumsphere of a given tetrahedra $\{a, b, c, d\}$ (Table 4.1). This is evaluated using the `inSphere` predicate that computes the sign of the following determinant:

$$\texttt{inSphere}(a,b,c,d,e) = \begin{vmatrix} a_x & a_y & a_z & \|a\|^2 & 1 \\ b_x & b_y & b_z & \|b\|^2 & 1 \\ c_x & c_y & c_z & \|c\|^2 & 1 \\ d_x & d_y & d_z & \|d\|^2 & 1 \\ e_x & e_y & e_z & \|e\|^2 & 1 \end{vmatrix} = \begin{vmatrix} b_x - a_x & b_y - a_y & b_z - a_z & \|b-a\|^2 \\ c_x - a_x & c_y - a_y & c_z - a_z & \|c-a\|^2 \\ d_x - a_x & d_y - a_y & d_z - a_z & \|d-a\|^2 \\ e_x - a_x & e_y - a_y & e_z - a_z & \|e-a\|^2 \end{vmatrix}$$

This is a very time consuming computation, and to make it more efficient, we propose to expand the 4×4 determinant into a linear combination of four 3×3 determinants independent of point e.

$$\begin{aligned} \texttt{inSphere}(a,b,c,d,e) = &-(e_x - a_x) \begin{vmatrix} b_y - a_y & b_z - a_z & \|b-a\|^2 \\ c_y - a_y & c_z - a_z & \|c-a\|^2 \\ d_y - a_y & d_z - a_z & \|d-a\|^2 \end{vmatrix} + (e_y - a_y) \begin{vmatrix} b_x - a_x & b_z - a_z & \|b-a\|^2 \\ c_x - a_x & c_z - a_z & \|c-a\|^2 \\ d_x - a_x & d_z - a_z & \|d-a\|^2 \end{vmatrix} \\ &-(e_z - a_z) \begin{vmatrix} b_x - a_x & b_y - a_y & \|b-a\|^2 \\ c_x - a_x & c_y - a_y & \|c-a\|^2 \\ d_x - a_x & d_y - a_y & \|d-a\|^2 \end{vmatrix} + \|e-a\|^2 \begin{vmatrix} b_x - a_x & b_y - a_y & b_z - a_z \\ c_x - a_x & c_y - a_y & c_z - a_z \\ d_x - a_x & d_y - a_y & d_z - a_z \end{vmatrix} \end{aligned}$$

Being completely determined by the tetrahedron vertex coordinates, the four 3×3 determinants can be pre-computed and stored in the tetrahedron data structure when it is created. The cost of the `inSphere` predicate becomes then negligible. Notice also that the fourth sub-determinant is minus the tetrahedron volume. We can set it to a positive value to flag deleted tetrahedra during the breadth-first search, thereby saving memory space.

Maximal improvement The maximal improvement of the Cavity function obtained with this optimization depends on the number of times the `inSphere` predicate is invoked per tetrahedron. First, in order to simplify further discussion, we will assume that the number of tetrahedra is about 6 times the number of vertices in the final triangulation. This means that each point insertion results in average in the creation of 6 new tetrahedra. On the other hand, we have seen in Section 4.1.3 that an appropriate point ordering ensures an approximately constant number of tetrahedra in the cavities. This number is close to 20 in a usual mesh generation context (J.-F. Remacle 2017). One point insertion thus results in the deletion of 20 tetrahedra, and the creation of 26 tetrahedra (all figures are approximations). This number is also the number of triangular faces of the cavity, since all tetrahedra created by the DelaunayBall function associate a cavity facet to the inserted vertex p_{k+1}. The `inSphere` predicate is therefore evaluated positively for the 20 tetrahedra forming the cavity and negatively for the 26 tetrahedra adjacent to the faces of the cavity, a total of 46 calls for each vertex insertion.

When n points have been inserted in the mesh, a total of $26n$ tetrahedra were created and the predicate has been evaluated $46n$ times. Thus, we may conclude from this analysis that the `inSphere` predicate is called approximately $46n/26n = 1.77$ times per tetrahedron. In consequence, the maximal improvement that can be obtained from our optimization of the `inSphere` predicate is of $1 - 1/1.77 = 43\%$. Storing the sub-determinants has a memory cost (4 `double` values per tetrahedron) and a

bandwidth cost (loading and storing of sub-determinants). For instance, for $n = 4.10^6$, we observe a speedup of 32% in the `inSphere` predicate evaluations, taking into account the time spent to compute the stored sub-determinants.

Note that a second geometric predicate is extensively used in Delaunay triangulation. The `orient3d` predicate evaluates whether a point d is above/on/under the plane defined by three points $\{a, b, c\} \in \mathbb{R}^3$ (J. Shewchuk 1997). It computes the triple product $(b - a) \cdot ((c - a) \times (d - a))$, i.e. the signed volume of tetrahedron $\{a, b, c, d\}$. This predicate is mostly used in the Walk function.

Implementation details Geometrical predicates evaluated with standard floating point arithmetics may lead to inaccurate or inconsistent results. To have a robust and efficient implementations of the `inSphere` and `orient3d` predicates, we have applied the strategy implemented in Tetgen (Si 2015).

1. Evaluate first the predicate with floating point precision. This gives a correct value in the vast majority of the cases, and represents only 1% of the code.

2. Use a filter to check whether the obtained result is certain. A static filter is first used, then, if more precision is needed, a dynamic filter is evaluated. If the result obtained by standard arithmetics is not trustworthy, the predicate is computed with exact arithmetics (J. Shewchuk 1997).

3. To be uniquely defined, Delaunay triangulations requires each point to be inside/outside a sphere, under/above a plane. When a point is exactly on a sphere or a plane, the point is in a non-general position that is slightly perturbed to "escape" the singularity. We implemented the symbolic perturbations proposed by (Edelsbrunner and Mücke 1990).

4.1.5 Improving DelaunayBall: Spending Less Time Computing Adjacencies

The DelaunayBall function creates the new tetrahedra filling the cavity (Figure 4.1c), and updates the tetrahedron structures. In particular, tetrahedron adjacencies are recomputed. This is the most expensive step of the Delaunay triangulation algorithm as it typically takes about 40% of the total time (Table 4.1). In contrast to existing implementations, we strongly interconnect the cavity building and cavity retriangulation steps. Instead of relying on a set of elegant and independent elementary operations like tetrahedron creation/deletion or adjacency computation, a specific cavity data structure has been developed, which optimizes batches of operations for the specific requirements of the Delaunay triangulation (Listing 4.2).

Use cavity building information The tetrahedra reached by the breadth-first search during the Cavity step (Section 4.1.4), are either inside or adjacent to the cavity. Each triangular facet of the cavity boundary is thus shared by a tetrahedron $t_1 \in \mathcal{A}(DT_k, p_{k+1})$ outside the cavity, by a tetrahedron $t_2 \in \mathcal{C}(DT_k, p_{k+1})$ inside the cavity, and by a newly created tetrahedron inside the cavity $t_3 \in \mathcal{B}(DT_k, p_{k+1})$ (Figure 4.1). The facet of t_1 adjacent to the surface of the cavity defines, along with the point p_{k+1}, the tetrahedron t_3. We thus know a priori that t_3 is adjacent to t_1, and store this information in the `cavityBoundaryFacet_t` structure (Listing 4.2).

Fast adjacencies between new tetrahedra During the cavity construction procedure, the list of vertices on the cavity as well as adjacencies with the tetrahedra outside the cavity are computed. The last step is to compute the adjacencies between the new tetrahedra built inside the cavity, i.e. the tetrahedra of the Delaunay ball.

The first vertex of all created tetrahedra is set to be p_{k+1}, whereas the other three vertices $\{p_1, p_2, p_3\}$ are on the cavity boundary, and are ordered so that the volume of the tetrahedral element is positive, i.e., `orient3d`$(p_{k+1}, p_1, p_2, p_3) > 0$. As explained in the previous section, the adjacency stored at index $4t_i + 0$, which corresponds to the facet of tetrahedron t_i opposite to the vertex p_{k+1}, is already known for every tetrahedron t_i in $\mathcal{B}(DT_k, p_{k+1})$. Three neighbors are thus still to be determined for each tetrahedron t_i, which means practically that an adjacency index has to be attributed to $4t_i + 1$, $4t_i + 2$ and $4t_i + 3$. The internal facets across which these adjacencies have to be identified are made of the common vertex p_{k+1} and one oriented edge of the cavity boundary.

```
typedef struct {
    uint32_t new_tetrahedron_vertices[4];  //  facet vertices + vertex to
        insert
    uint64_t adjacent_tetrahedron_ID;
} cavityBoundaryFacet_t

typedef struct{
    uint64_t adjacency_map[1024];  // optimization purposes, see Section
        4.1.5

    struct {
        cavityBoundaryFacet_t* boundary_facets;
        uint64_t num;           // number of boundary facets
        uint64_t allocated_num; // capacity [in cavityBoundaryFacet_t]
    } to_create;

    struct {
        uint64_t* tetrahedra_ID;
        uint64_t num;           // number of deleted tetrahedra
        uint64_t allocated_num; // capacity
    } deleted;
} cavity_t;
```

Listing 4.2: Cavity specific data structure.

Using an elaborated hash table with complex collision handling would be overkill. We prefer to use a double entry lookup table of dimension $n \times n$, whose rows and columns are associated with the n vertices $\{p_j\}$ of the cavity boundary,to which auxiliary indices $0 \leq i_j < n$ are affected for convenience and stored in the padding variable of the vertex, With this, the unique index of an oriented edge p_1p_2 is set to be $n \times i_1 + i_2$, which corresponds to one position in the $n \times n$ lookup table. So, for each tetrahedron t_i in the Delaunay ball with vertices $\{p_{k+1}, p_1, p_2, p_3\}$, the adjacency index $4t_i + 1$ is stored at position $n \times i_2 + i_3$ in the lookup table, and similarly $4t_i + 2$ at position $n \times i_3 + i_1$ and $4t_i + 3$ at position $n \times i_1 + i_2$. A square array with a zero diagonal is built proceeding this way, in which the sought adjacencies are the pairs of symmetric components.

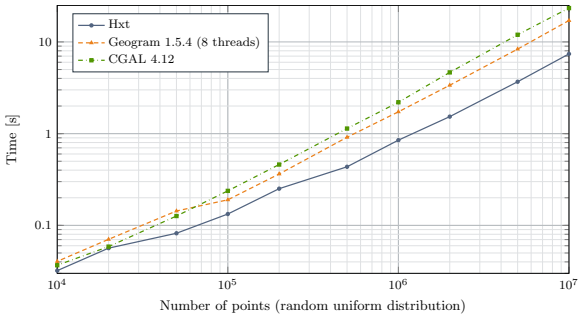
Theoretically, there is no maximal value for the number n of vertices in the cavity but, in practice, we can take advantage of the fact that it remains relatively small. Indeed, the cavity boundary is the triangulation of a topological sphere, i.e. a planar graph in which the number of edges is $e = 3f/2$, where f is the number of faces, and whose Euler characteristic is $n - e + f = 2$. Hence, the number of vertices is linked to the number of triangular faces by $n = f/2 + 2$. As we have shown earlier that f is about 26, we deduce that there are about $n = 15$ vertices on a cavity boundary. Hence, a $n_{max} \times n_{max}$ lookup table, with maximum size $n_{max} = 32$ is sufficient provided there are at most $f_{max} = 2(n_{max} - 2) = 60$ tetrahedra created in the cavity. If the cavity exceptionally contains more elements, the algorithm smoothly switches to a simple linear search.

Once all adjacencies of the new tetrahedra have been properly identified, the DelaunayBall function is then in charge of updating the mesh data structure. This ends the insertion of point p_{k+1}. The space freed by deleted tetrahedra is reused, if needed additional tetrahedra are added. Note that almost all steps of adjacencies recovery are vectorizable.

4.1.6 About a Ghost

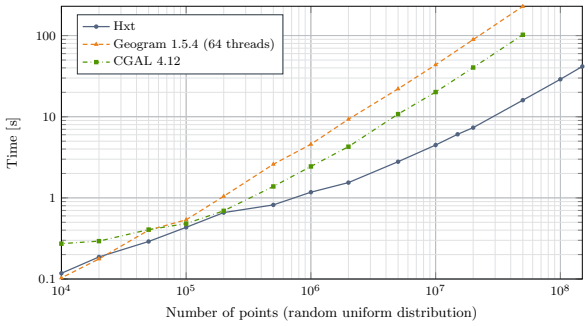
The Bowyer–Watson algorithm for Delaunay triangulation assumes that all newly inserted vertices are inside an element of the triangulation at the previous step (Algorithm 1). To insert a point p_{k+1} outside the current support of the triangulation, one possible strategy is to enclose the input vertices in a sufficiently large bounding box, and to remove the tetrahedra lying outside the convex hull of the point set at the end of the algorithm. A more efficient strategy, adopted in TetGen (Si 2015), CGAL (Boissonnat, Devillers, Teillaud, et al. 2000), Geogram (Levy 2016), is to work with the so-called ghost tetrahedra connecting the exterior faces of the triangulation with a virtual ghost vertex G. Using this elegant concept, vertices can be inserted outside the current triangulation support with almost no algorithmic change.

The ghost vertex G is the vertex "at infinity" shared by all ghost tetrahedra. The ghost tetrahedra cover the whole space outside the regular triangulation. Like regular tetrahedra, ghost tetrahedra



# vertices	10^4	10^5	10^6	10^7
Ours	0.027	0.21	2.03	21.66
Geogram	0.060	0.51	5.53	56.02
CGAL	0.062	0.64	6.65	66.24
TetGen	0.054	0.56	5.89	63.99

(a) Intel® Core™ i7-6700HQ CPU, maximum core frequency of 3.5Ghz.



# vertices	10^4	10^5	10^6	10^7
Ours	0.134	0.98	9.36	97.97
Geogram	0.240	2.47	25.34	259.74
CGAL	0.265	2.81	28.36	286.54
TetGen	0.283	2.97	31.13	336.21

(b) Intel® Xeon Phi™ 7210 CPU, maximum core frequency of 1.5Ghz.

Figure 4.4: Performances of our sequential Delaunay triangulation implementation (Algorithm 1) on a laptop **(a)** and on a slow CPU having AVX-512 vectorized instructions **(b)**. Timings are in seconds and exclude the initial spatial sort.

are stored in the mesh data structure, and are deleted whenever a vertex is inserted inside their circumsphere. The accurate definition of the circumsphere of a ghost tetrahedron, in particular with respect to the requirements of the Delaunay condition, is however a more delicate question.

From a geometrical point of view, as the ghost vertex G moves away towards infinity from an exterior facet abc of the triangulation, the circumscribed sphere of tetrahedron $abcG$ tends to the half space on the positive side of the plane determined by the points abc, i.e., the set of points x such that `orient3d`(a,b,c,d) is positive. The question is whether this inequality should be strict or not, and the robust answer is neither one nor the other. The circumcircle of a ghost triangle abG actually contains not only the open half-plane strictly above the line $\overleftrightarrow{ab}$, but also the line segment $[ab]$ itself; the other parts of the line $\overleftrightarrow{ab}$ being excluded. In 3D, the circumsphere of a ghost tetrahedron $abcG$ contains the half-space L such that for any point $d \in L$, `orient3d`(a,b,c,d) is strictly positive, plus the disk defined by the circumcircle of the triangle abc. If the point d is in the plane abc, i.e. `orient3d`(a,b,c,d)=`0`, we thus additionally test if d is in the circumsphere of the adjacent regular tetrahedron sharing the triangle abc. This composite definition of the circumscribed circle makes this approach robust with minimal changes: only the `inSphere` predicate is modified in the implementation.

Because the 3×3 determinant of `orient3d` is used instead of the 4×4 determinant of `inSphere` for ghost tetrahedra, the approach with a ghost vertex is faster than other strategies (J. R. Shewchuk 1997). Note that the Walk cannot evaluate `orient3d` on the faces of ghost tetrahedra that connect edges of the convex hull to the ghost vertex. If the starting tetrahedron is a ghost tetrahedron, the Walk directly steps into the non-ghost adjacent tetrahedron. Then, if it steps in an other ghost tetrahedron $abcG$ while walking toward p_{k+1}, we have `orient3d`$(a,b,c,p_{k+1}) > 0$ and the ghost tetrahedron is inside of the cavity. The walk hence stops there, and the algorithm proceeds as usual.

4.1.7 Serial Implementation Performances

HXT serial Delaunay triangulation algorithm has about one thousand lines (without Shewchuk's geometric predicates (J. Shewchuk 1997)). It is open-source and available in Gmsh (`www.gmsh.info`). Overall, it is almost three time faster than other sequential implementations. Figure 4.4a and 4.4b show the performance gap between our implementation and concurrent software on a laptop with a maximum core frequency of 3.5Ghz, and on a many-core computer with a maximum core frequency of 1.5Ghz and wider SIMD instructions. Table 4.1 indicates that the main difference in speed comes from the more efficient adjacencies computation in the `DelaunayBall` function. Other software use a more general function that can also handle cavities with multiple interior vertices. But this situation does not happen with Delaunay insertion, where there is always one unique point in the cavity, p_{k+1}. Since our `DelaunayBall` function is optimized for Delaunay cavities, it is approximately three time faster, despite the additional computation of sub-determinants. The remaining performance gain is explained by our choice of simple but efficient memory aligned data structure.

4.2 Parallel Delaunay Triangulation

The availability of more and more massive point sets as well as the size of finite element meshes is a major challenge for Delaunay triangulation algorithms. It is now common to deal with meshes of over several hundred millions of tetrahedra (Rasquin et al. 2014; Ibanez et al. 2016; Vázquez et al. 2016). workflow where they are adapted to the solution.

Related work As we have seen in the previous section the algorithm to build the Delaunay triangulation is essentially sequentially and parallelizing 3D Delaunay triangulations and Delaunay-based meshing algorithms is a very challenging task.

The most part of today's clusters have two levels of parallelism. Distributed memory systems contain thousands of nodes, each node being itself a shared memory system with multiple cores. In recent years, nodes have seen their number of cores and the size of their memory increase, with some clusters already featuring 256-core processors and up to 12TB of RAM (Nystrom et al. 2015; Fu et al. 2016). As many-core shared memory machines are becoming standard, Delaunay triangulation algorithms designed for shared memory should not only scale well up to 8 cores but to several hundred cores. To overcome memory and time limitations, Delaunay triangulations should be constructed in parallel making the most of both distributed and shared memory architectures. A triangulation can be subdivided into multiple parts, each constructed and stored on a node of a distributed memory cluster. Multiple methods have been proposed to merge independently generated Delaunay triangulations (Cignoni et al. 1993; M. Chen 2010; Funke et al. 2017; Blelloch et al. 1999). However, those methods feature complicated merge steps, which are often difficult to parallelize. To avoid merge operations, other distributed implementations maintain a single valid Delaunay triangulation and use synchronization between processors whenever a conflict may occur at inter-processor boundaries (Okusanya et al. 1996; Chrisochoides et al. 2003). In finite-element mesh generation, merging two triangulations can be simpler because triangulations are not required to be fully Delaunay, allowing algorithms to focus primarily on load balancing (Lachat et al. 2014).

On shared memory machines, divide-and-conquer approaches remain efficient, but other approaches have been proposed since communication costs between different threads are not prohibitive to the contrary of distributed memory machines. To insert a point in a Delaunay triangulation, the kernel procedure operates on a cavity that is modified to accommodate the inserted point (Figure 4.1). Two points can therefore be inserted concurrently in a Delaunay triangulation if their respective cavities do not intersect, $C(DT_k, p_{k1}) \cap C(DT_k, p_{k2}) = \emptyset$, otherwise there is a conflict. In practice, other types of conflicts and data-races should possibly be taken into account depending on the chosen data structures and the insertion point strategy. Conflict management strategies relying heavily on locks [1] lead to relatively good speedups on small numbers of cores (Kohout et al. 2005; Blandford et al. 2006; Batista et al. 2010; Foteinos et al. 2012). Remacle et al. presented an interesting strategy that checks if insertions can be done in parallel by synchronizing threads with barriers (J.-F. Remacle 2017).

[1] A lock is a synchronization mechanism enforcing that multiple threads do not access a resource at the same time. When a thread cannot acquire a lock, it usually waits.

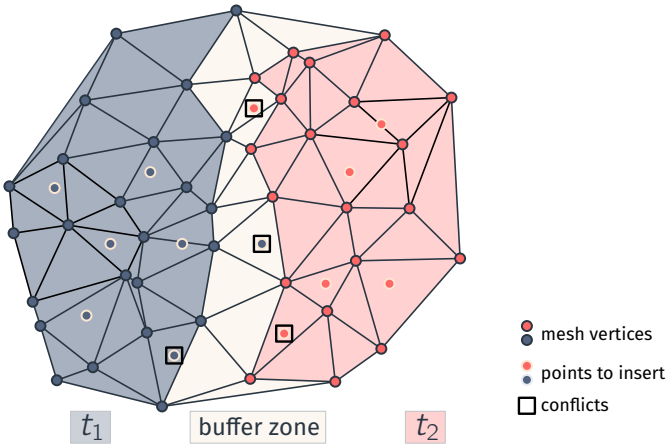


Figure 4.5: Vertices are partitioned such that each vertex belongs to a single thread. A triangle can only be modified by a thread that owns all of its three vertices. Triangles that cannot be modified by any thread form a buffer zone.

However, synchronization overheads prevent those strategies from scaling to an high number of cores. More promising approaches rely on a partitioning strategy (Lo 2012; Loseille et al. 2017). Contrarily to pure divide-and-conquer strategies for distributed memory machines, partitions can change and move regularly, and they do not need to cover the whole mesh at once.

In HXT, we propose to parallelize the Delaunay kernel using partitions based on a space-filling curve, similarly to (Loseille et al. 2017). The main difference is that we significantly modify the partitions at each iteration level. Our code is designed for shared memory architecture only, we leave its integration into a distributed implementation for future work.

4.2.1 A Parallel Strategy Based on Partitions

Our second contribution is a multi-threaded version of the Delaunay kernel that is able to concurrently insert vertices. Moore curve coordinates are used to partition the point set, avoiding heavy synchronization overheads. Conflicts are managed by modifying the partitions with a simple rescaling of the space-filling curve. The performances of our implementation have been measured on three different processors, an Intel core-i7, an Intel Xeon Phi and an AMD EPYC, on which we have been able to compute 3 billion tetrahedra in 53 seconds. This corresponds to a generation rate of over 55 million tetrahedra per second.

There are multiple conditions a program should ensure to avoid data-races and conflicts between threads when concurrently inserting points in a DT. Consider thread t_1 is inserting point p_{k1} and thread t_2 is simultaneously inserting point p_{k2}.

1. Thread t_1 cannot access information about any tetrahedron in $C(DT, p_{k2})$ and inversely. Hence:
   (a) $C(DT, p_{k1}) \cap C(DT, p_{k2}) = \emptyset$
   (b) $\mathcal{A}(DT, p_{k1}) \cap C(DT, p_{k2}) = \emptyset$ and $\mathcal{A}(DT, p_{k2}) \cap C(DT, p_{k1}) = \emptyset$
   (c) Thread t_1 cannot walk into $C(DT, p_{k2})$ and reciprocally, t_2 cannot walk into $C(DT, p_{k1})$
2. A tetrahedron in $\mathcal{B}(DT, p_{k1})$ and a tetrahedron in $\mathcal{B}(DT, p_{k2})$ cannot be created at the same memory index.

To ensure rule (1) it is sufficient to restrain each thread to work on an independent partition of the mesh (Figure 4.5). This lock-free strategy minimizes synchronization between threads and is very efficient. Each point of the Delaunay triangulation is assigned a partition corresponding to a unique thread. A tetrahedron belongs to a thread if at least three of its vertices are in that thread's partition.

To ensure (1a) and (1b), the insertion of a point belonging to thread is aborted if the thread accessed a tetrahedron that belongs to another thread or that is in the buffer zone. To ensure (1c), we forbid

threads to walk in tetrahedra belonging to another thread. Consequently, a thread aborts the insertion of a vertex when (i) the Walk reaches another partition, or when (ii) the Cavity function reaches a tetrahedron in the buffer zone. To insert these points, the vertices are re-partitioned differently (Section 4.2.2), a procedure repeated until there is no more vertices to insert or until the rate of successful insertions has become too low. In that case, the number of threads is decreased (Section 4.2.3). When the number of points to insert has become small enough, the insertion runs in sequential mode to insert all remaining vertices. The first BRIO round is also inserted sequentially to generate a first base mesh. Nevertheless, the vast majority of points are inserted in parallel.

Rule (2) is obvious from a parallel memory management point of view. It is however difficult to ensure it without requiring an unreasonable amount of memory. As explained in Section 4.2.5, synchronization between threads is required punctually.

4.2.2 Partitioning and Re-partitioning with Moore Curve

We subdivide the n_v points to insert such that each thread inserts the same number of points. Our partitioning method is based on the Moore curve, i.e. on the point insertion order implemented for the sequential Delaunay triangulation (Section 4.1.3). Space-filling curves are relatively fast to construct and have already been used successfully for partitioning meshes (Aluru et al. 1997; Lieber et al. 2010; Devine et al. 2005; Loseille et al. 2017). Each partition of the n_{thread} partitions is a set of grid cells that are consecutive along the Moore curve. To compute the partitions, i.e. its starting and ending Moore indices, we sort the points to insert according to their Moore indices (see Section 4.1.3). Then, we assign the first $n_v/n_{threads}$ points to the first partition, the next $n_v/n_{threads}$ to the second, etc (Figure 4.6c). The second step is to partition the current Delaunay triangulation in which the points will be inserted. We use once more the Moore indices to assign the mesh vertices to the different partitions. The ghost vertex is assigned a random index. The partition owning a tetrahedron is determined from the partitions of its vertices, if a tetrahedron has at least three vertices in a partition it belong to this partition, otherwise the tetrahedron is in the buffer zone. Each thread then owns a subset of the points to insert and a subset of the current triangulation.

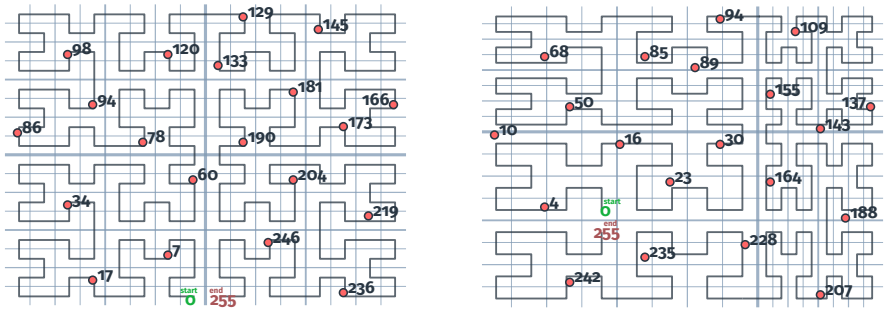
Once all threads attempted to insert all their points, a large majority of vertices is generally inserted. To insert the vertices for which insertion failed because the point cavity spans multiple partitions (Figure 4.5), we modify significantly the partitions by modifying the Moore indices computation. We apply a coordinate transformation and a circular shift to move the zero index around the looping curve (Figure 4.6). Coordinates below a random threshold are linearly compressed, while coordinates above the threshold are linearly expanded.

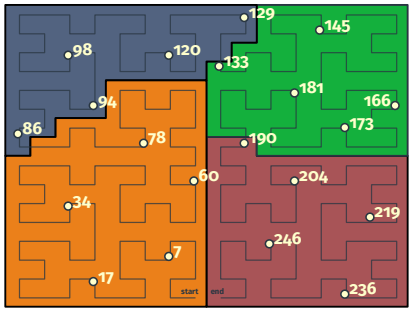
4.2.3 Ensuring Termination

When the number of vertices of the current Delaunay triangulation is small, typically at the first steps of the algorithm, the probability that the mesh vertices belong to different partitions is very high and none of the point insertions may succeed. To avoid wasting precious milliseconds, the first BRIO round is always inserted sequentially.

Moreover, there is no guarantee that re-partitioning will be sufficient to insert all the points. And even when a large part of the triangulation has already been constructed, parallel insertion may enter an infinite failure loop. After a few rounds, the remaining points to insert are typically located in restricted volumes (intersection of previous buffer zones). Because re-partitioning is done on the not-yet-inserted points, the resulting partitions are also small and thin. This leads to inevitable conflicts as partitions get smaller than cavities. In practice, we observed that the ratio ρ of successful insertions decreases for a constant number of threads. If 80 out of 100 vertices are successfully inserted in a mesh ($\rho_k = 0.8$), less that 16 of the 20 remaining vertices will be inserted at the following attempt ($\rho_{k+1} < 0.8$). Note that this effect is more important for small meshes, because the bigger the mesh, the relatively smaller the buffer zone and the higher the insertion success rate. This difference explains the growth of the number of tetrahedra created per second with the number of points in the triangulation (Figure 4.8).

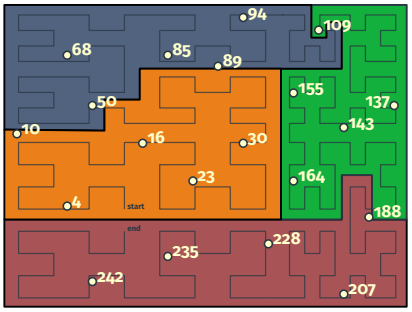
To avoid losing a significant amount of time in re-partitioning and trying to insert the same vertex multiple times we gradually decrease the number of threads. Choosing the adequate number of threads is a question of balance between the potential parallelization gain and the partitioning cost that comes with each insertion attempt. When decreasing the number of threads, we decrease the number of





	partition start	partition end	#points
1st thread	0	94	5
2nd thread	94	133	5
3rd thread	133	190	5
4th thread	190	∞	5

(c)



	partition start	partition end	#points
1st thread	0	50	5
2nd thread	50	109	5
3rd thread	109	188	5
4th thread	188	∞	5

(d)

Figure 4.6: Partitioning of 20 points in 2D using the Moore indices, on the right the supporting grid of the Moore curve is transformed and the curve is shifted. In both cases, each partition contains 5 points. Indeed, the starting and ending Moore index of each partition are defined in a way that balances the point insertions between threads.

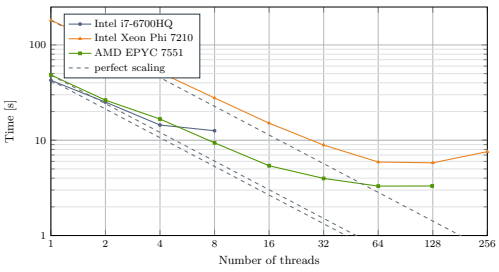


Figure 4.7: Strong scaling of our parallel Delaunay for a random uniform distribution of 15 million points, resulting in over 100 million tetrahedra on 3 machines: a quad-core laptop, an Intel Xeon Phi with 64 cores and a dual-socket AMD EPYC 2 × 32 cores.

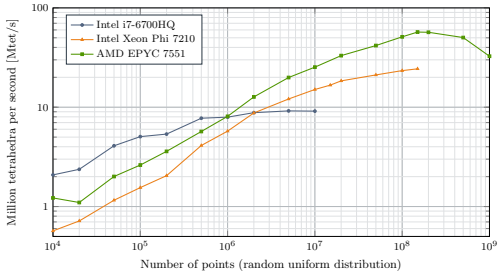


Figure 4.8: Number of tetrahedra created per second by our parallel implementation for different number of points. Tetrahedra are created more quickly when there is a lot of points because the proportion of conflicts is lower. An average rate of 65 million tetrahedra created per second is obtained on the EPYC.

	#points			#threads	#mesh
	to insert	inserted	ρ		vertices
Initial mesh					4
BRIO Round 1	2044	2044	100%	1	2048
BRIO Round 2	12 288	6988	57%	4	9036
	5300	3544	67%	2	12 580
	1756	1756	100%	1	14 336
BRIO Round 3	86 016	59 907	70%	8	74 243
	26 109	11 738	45%	8	85 981
	14 371	7092	49%	4	93 073
	7279	5332	73%	2	98 405
	1947	1947	100%	1	100 352
BRIO Round 4	602 112	503 730	84%	8	604 082
	98 382	44 959	46%	8	649 041
	53 423	31 702	59%	8	680 743
	21 721	7903	36%	8	688 646
	13 818	9400	68%	4	698 046
	4418	3641	82%	2	701 687
	777	777	100%	1	702 464
BRIO Round 5	297 536	271 511	91%	8	973 975
	26 025	16 426	63%	8	990 401
	9599	8092	84%	4	998 493
	1507	1507	100%	1	1 000 000

Table 4.2: Numbers of threads used to insert points in our parallel Delaunay triangulation implementation according to the number of points to insert, the mesh size and the insertion success at the previous step. 94.5% of points are inserted using 8 threads and 5% using 4 threads

attempts needed. When the ratio of successful insertions is too low, $\rho < 1/5$, or when the number of points to insert per thread is under 3000, we divide the number of threads by two. Furthermore, if $\rho_k \leq 1/n_{threads}$, the next insertion attempt will not benefit from multi-threading and we insert the points sequentially.

In Table 4.2 are given the number of threads used at each step of the Delaunay triangulation of a million vertices depending on the number of points to insert, the size of the current mesh, and the success insertion ratio ρ. Note that the computation of Moore indices and the sorting of vertices to insert are always computed on the maximal number of threads available (8 threads in this case) even when the insertion runs sequentially.

4.2.4 Data Structures

The data structure for our parallel Delaunay triangulation algorithm is similar to the one used by our sequential implementation (Section 4.1.2). There are two small differences. First, the parallel implementation does not compute sub-determinants for each tetrahedron. Actually, the bandwidth usage with the parallel insertions is already near its maximum. Loading and storing sub-determinant becomes detrimental to the overall performance. Instead we store a 16-bit[2] color flag to mark deleted tetrahedra. Second, each thread has its own `Cavity_t` structure (Listing 4.2) to which are added two integers identifying the starting and ending Moore indices of the thread's partition.

Memory footprint Because we do not store four sub-determinants per tetrahedra anymore but only a 2-bytes color, our mesh data structure is lighter. Still assuming that there is approximately $6n$ tetrahedra for n vertices, it requires a little more than $6 \times 50 + 32 = 332$ bytes per vertex. Thanks to this

[2] An 8-bit char would also work (not less or it would create data races) but the color flag is also used to distinguish volumes in our mesh generator described in Section 4.3

# vertices	10^4	10^5	10^6	10^7
Ours	6.9 MB	43.8 MB	404.8 MB	3.8 GB
Geogram	6.7 MB	30.5 MB	268.6 MB	2.7 GB
CGAL	14.1 MB	66.7 MB	578.8 MB	5.7 GB

Table 4.3: Comparison of the maximum memory usage of our parallel implementation, CGAL (The CGAL Project 2016) and Geogram (Levy 2016) when using 8 threads.

low memory footprint, we were able to compute the tetrahedralization of $N = 10^9$ vertices (about 6 billion of tetrahedra) on an AMD EPYC machine that has 512 GB of RAM. The experimental memory usage shown in Table 4.3 differ slightly from the theoretical formula because (i) more memory is allocated than what is used (ii) measurements represent the maximum memory used by the whole program, including the stack, the text and data segment etc.

4.2.5 Critical Operations

When creating new tetrahedra in the Delaunay ball of a point, a thread first recycles unused memory space by replacing deleted tetrahedra. The indices of deleted tetrahedra are stored in the `cavity->deleted.tetrahedra_ID`4.1 array of each thread. When the `cavity->deleted.tetrahedra_ID` array of a thread is empty, additional memory should be reserved by this thread to create new tetrahedra.

This operation is a critical part of the program requiring synchronization between threads to respect the rule (2). We need to capture the current number of tetrahedra and increment it by the requested number of new tetrahedra in one single atomic operation. *OpenMP* provides the adequate mechanism, see Listing 4.3.

To reduce the number of time this operation is done, the number of tetrahedra is incremented atomically by at least 8192, and the `cavity->deleted.tetrahedra_ID` is filled with the new indices of tetrahedra. Those tetrahedra are conceptually deleted although they have never been in the mesh. The number 8192 was found experimentally among multiples of 512[3]. Increasing it would reduce the number of time reservation of new tetrahedra is performed but it is not necessary. Indeed, since the critical operation occurs then in average every 1000^+ insertions, the time wasted is very low. Therefore, increasing the default number of deleted tetrahedra would only wastes memory space for little to no gain.

When the space used by tetrahedra exceeds the initially allocated capacity, reallocation is implemented that doubles the capacity of the arrays of mesh tetrahedra. During that operation, memory is potentially moved at another location. No other operation can be performed at that time on the mesh. Therefore, the reallocation code is placed in between two OpenMP barriers (Listing 4.4). This synchronization event is very rare and does not impact performances. In general, a good estimation of the needed memory needed to reserve is possible and that critical section is never reached.

4.2.6 Parallel Implementation Performances

We are able to compute the Delaunay triangulation of over one billion tetrahedra in record-breaking time: 41.6 seconds on the Intel Xeon Phi and 17.8 seconds on the AMD EPYC. These timings do not include I/Os. We are able to generate three billion tetrahedra in 53 seconds on the EPYC. The scaling of our implementation regarding the number of threads is detailed in Figure 4.7. We obtain a good scaling until the number of threads reach the number of cores, i.e. 4 cores for the Intel i7-6700HQ, 64 cores for the Intel® Xeon Phi 7210 and the AMD® EPYC 7551. Comparisons of implementation are given on on a laptop (Figure 4.9a) and on the Intel Xeon Phi (Figure 4.9b).

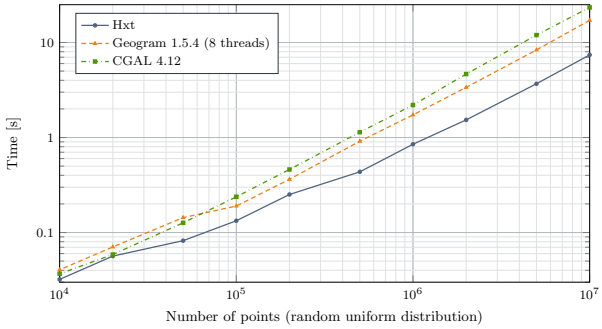
[3]It is common to choose a multiple of the page size (usually 4096 bytes) to minimize TLB misses.

```
if(cavity->to_create.num > cavity->deleted.num)
{
    uint64_t nTetNeeded = MAX(8192, cavity->to_create.num) - cavity->
        deleted.num;

    uint64_t nTet;
    #pragma omp atomic capture
    {
        nTet = mesh->tetrahedra.num;
        mesh->tetrahedra.num+=nTetNeeded;
    }

    reallocTetrahedraIfNeeded(mesh);
    reallocDeletedIfNeeded(state, cavity->deleted.num + nTetNeeded);

    for (uint64_t i=0; i<nTetNeeded; i++){
        cavity->deleted.tetrahedra_ID[cavity->deleted.num+i] = 4*(nTet+i);
        mesh->tetrahedra.color[nTet+i] = DELETED_COLOR;
    }

    cavity->deleted.num += nTetNeeded;
}
```

Listing 4.3: When there are less deleted tetrahedra than there are tetrahedra in the Delaunay ball, 8192 new "deleted tetrahedra" indices are reserved by the thread. As the `mesh` data structure is shared by all threads, `mesh->tetrahedra.num` must be increased in one single atomic operation.

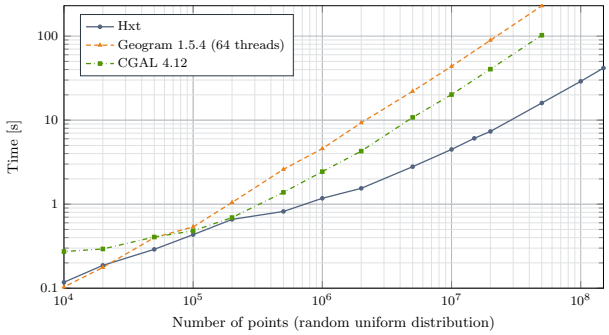
```
void reallocTetrahedraIfNeeded(mesh_t* mesh)
{
    if(mesh->tetrahedra.num > mesh->tetrahedra.allocated_num)
    {
        #pragma omp barrier

        // all threads are blocked except the one doing the reallocation
        #pragma omp single
        {
            uint64_t nTet = mesh->tetrahedra.num;
            alignedRealloc(&mesh->tetrahedra.neighbor_ID, nTet*8*sizeof(
                uint64_t));
            alignedRealloc(&mesh->tetrahedra.vertex_ID, nTet*8*sizeof(
                uint32_t));
            alignedRealloc(&mesh->tetrahedra.color, nTet*2*sizeof(uint16_t
                ));
            mesh->tetrahedra.allocated_num = 2*nTet;

        } // implicit OpenMP barrier here
    }
}
```

Listing 4.4: Memory allocation for new tetrahedra is synchronized with OpenMP barriers.



# vertices	10^4	10^5	10^6	10^7
Ours	0.032	0.13	0.85	7.40
Geogram	0.041	0.19	1.73	17.11
CGAL	0.037	0.24	2.20	23.37

(a) 4-core Intel® Core™ i7-6700HQ CPU.



# vertices	10^4	10^5	10^6	10^7	10^8
Ours	0.11	0.43	1.17	4.48	28.95
Geogram	0.10	0.54	4.58	43.70	/
CGAL	0.27	0.48	2.44	20.15	/

(b) 64-core Intel® Xeon Phi™ 7210 CPU.

Figure 4.9: Comparison of our parallel implementation with the parallel implementation in CGAL (The CGAL Project 2016) and Geogram (Levy 2016) with on a high-end laptop **(a)** and a many-core computer **(b)**. All timings are in seconds.

4.3 Tetrahedral Mesh Generation

We finally show how this very efficient parallel Delaunay triangulation can be integrated in a Delaunay refinement mesh generator which takes as input the triangulated surface boundary of the volume to mesh.

Taking into account the boundary The parallel Delaunay triangulation algorithm that we presented in the previous section is integrated in a Delaunay refinement mesh generator. A tetrahedral mesh generator is a procedure that takes as input the boundary of a domain to mesh, defined by set of triangles t that defines the boundary of a closed volume, and that returns a finite element mesh, i.e. a set of tetrahedra $\mathcal{T}$ of controlled sizes and shapes which boundary is equal to the input triangulation: $\partial\mathcal{T} = t$. From a coarse mesh that is conformal to t, a Delaunay-refinement based mesher inserts progressively vertices until element sizes and shapes follow the prescribed ranges. Generating a mesh is an intrinsically more difficult problem than constructing the Delaunay triangulation of given points because (1) the points are not given, (2) the boundary triangles must be facets of the generated tetrahedra, and (3) the tetrahedra shape and size should be optimized to maximize the mesh quality. Our objective in this section is to demonstrate that our parallel Delaunay point insertion may be integrated in a mesh generator. The interested reader is referred to the book by Frey and George (Frey and George 2007) for a complete review of finite-element mesh generation.

The mesh generation algorithm 2 proposed in this section follows the approach implemented for example in Tetgen (Si 2015). First, all the vertices of the boundary triangulation t are tetrahedralized to form the initial "empty mesh" $\mathcal{T}_0$. Then, $\mathcal{T}_0$ is transformed into a conformal mesh $\mathcal{T}$ which boundary $\partial\mathcal{T}$ is equal to $t : \partial\mathcal{T} = t$. The triangulation $\mathcal{T}$ is then progressively refined by (i) creating vertices S at the circumcenters of tetrahedra for which the circumsphere radius, r_τ, is significantly larger than the desired mesh size, h, i.e. $r_\tau/h > 1.4$ (J. R. Shewchuk 1997). (ii) inserting the vertices in the mesh using our parallel Delaunay algorithm. The sets of points S to insert are filtered *a priori* in order not to generate short edges, i.e. edges of size smaller than $0.7h$. We first use Hilbert coordinates to discard very close points on the curve, and implemented a slightly modified cavity algorithm that aborts if a point of the cavity is too close from the one to insert (Section 4.1.4). Points are thus discarded both in the filtering process and in the insertion process. Note that contrary to existing implementations, we insert as large as possible point sets S during the refinement step to take advantage of the efficiency of our parallel point insertion algorithm (Section 4.2).

We parallelized all steps of the mesh generator (Algorithm 2) except the boundary recovery procedure that is the one of Gmsh (Geuzaine et al. 2009) which is essentially based on Tetgen (Si 2015). The cavity algorithm has also been modified to accommodate possible constraints on the faces and edges of tetrahedra. With this modification, mesh refinement never breaks the surface mesh and constrained edges and faces can be included in the mesh interior. As a result, the mesh may not satisfy the Delaunay property anymore. Therefore, we must always ensure that cavities are star-shaped. This is achieved by a simple procedure that checks if boundary facets of the cavity are oriented such that they form a positive volume with the new point. If a boundary facet is not oriented correctly, indicating that the cavity is not star-shaped, the tetrahedron containing it is removed from the cavity and the procedure is repeated. In practice, these modifications do not affect the speed of point insertions significantly.

To generate high-quality meshes, an optimization step should be added to the algorithm to improve the general quality of the mesh elements and remove sliver tetrahedra. Mesh improvements are out of the scope of this part. However, first experiments have shown that optimization step should slow down the mesh generation by a factor two. For example, mesh refinement and improvement take approximately the same time in Gmsh (Geuzaine et al. 2009).

4.3.1 Small and Medium Size Test Cases on Standard Laptop

In order to verify the scalability of the whole meshing process, meshes of up to one hundred million tetrahedra were computed on a 4 core 3.5 GHz Intel Core i7-6700HQ with 1, 2, 4 and 8 threads. Those meshes easily fit within the 8Gb of RAM of this laptop.

Three benchmarks are considered in this section: (i) a cube filled with cylindrical fibers of random radii and lengths that are randomly oriented, (ii) a mechanical part and (iii) a truck tire. Surface meshes

Algorithm 2 Mesh generation algorithm

Input: A set of triangles t
Output: A tetrahedral mesh $\mathcal{T}$

```
1: function ParallelMesher(t)
2:     T_0 ← EmptyMesh(t)                          ▷ Delaunay kernel
3:     T ← RecoverBoundary(T_0)                    ▷ (Si 2015)
4:     while T contains large tetrahedra do
5:         S ← SamplePoints(T)
6:         S ← FilterPoints(S)
7:         T ← InsertPoints(T, S)                  ▷ Modified Delaunay kernel
8:     return T
```

are computed with Gmsh (Geuzaine et al. 2009). Mesh size on the surfaces is controlled by surface curvatures and mesh size inside the domain is simply interpolated from the surface mesh.

Illustrations of the meshes, as well as timings statistics are presented in Figure 4.10. Our mesher is able to generate between 40 and 100 million tetrahedra per minute. Using multiple threads allows some speedup, the mesh refinement process is accelerated by a factor ranging between 2 and 3 on this 4 core machine.

The last test case (truck tire) is defined by more than 7000 CAD surfaces. Recovering the 27 892 triangular facets missing from $\mathcal{T}_0$ takes more than a third of the total meshing time with the maximal number of threads. Parallelizing the boundary recovery process is clearly a priority of our future developments. On this same example, the surface mesh was done with Gmsh using four threads. The surface mesher of Gmsh is not very fast and it took about the same time to generate the surface mesh of 6 881 921 triangles as to generate the volume mesh that contains over one hundred million tetrahedra using the same number of threads. The overall meshing time for the truck tire test case is thus about 6 minutes.

4.3.2 Large Mesh Generation on Many Core Machine

We further generated meshes containing over 300,000,000 elements on a AMD® EPYC 64 core machine. Three benchmarks are considered: (i) two cubes filled with many randomly oriented cylindrical fibers of random radii and lengths, and (ii) the exterior of an aircraft. Surface meshes were also generated by Gmsh.

Our strategy reaches its maximum efficiency for large meshes. In the 500 thin fibers test case, over 700,000,000 tetrahedra were generated in 135 seconds. This represents a rate of 5.2 million tetrahedra per second. In the 500 thin fibers test case, boundary recovery cost was lower and a rate of 6.2 million tetrahedra per second was reached.

Remaining Challenges We optimized both the sequential and the parallel algorithm by specifying them exclusively for Delaunay triangulation purposes. Our parallel implementation of 3D Delaunay triangulation construction has one important drawback: partitioning is more costly for non-uniform point sets because Moore indices are not adaptive. Adaptive Hilbert or Moore curve computation is relatively simple (Hamilton et al. 2008; Su et al. 2016) but may not be faster than our refinement and sorting in one batch approach. Another drawback of our implementation is that it might not be well suited for distributed memory architectures. However, nothing prevents our strategy from being included in any larger divide-and-conquer approach.

We additionally presented a lightweight mesh generator based on Delaunay refinement. This mesh generator is almost entirely parallel and is able to produce computational meshes with a rate above 5 million tetrahedra per second. One issue when parallelizing mesh generation is to ensure reproducibility. For mesh generation, it is usually admitted that mesh generation should be reproducible for a given number of threads. Our mesh generator has been made reproducible in that sense, with a loss of 20% in overall performance. This feature has been implemented as an option. On the other hand, Figures 4.10 and 4.11 show that, starting from the same input, the number of elements varies.

Making the code reproducible independently of the number of threads would dramatically harm its scalability.

Note that our mesh generator does not include the final optimization step of the mesh, *mesh improvement*, to obtain a mesh adequate for finite-element simulations. This crucial step of any industrial-grade meshing procedure is the major limitation of our mesh generator. The basic operations to perform: edge swap, edge collapse, edge split, vertex relocation and face swap operations will be done using the same parallel framework than the one we described to build the Delaunay triangulation. The final challenge is the parallelization of the boundary recovery procedure. A third limitation is known in the placement of new vertices. Our current strategy, placing new vertices at the circumcenter of existing tetrahedra, does not always offer a sufficient quality when compared with frontal approaches. The frontal point insertion strategy that has been proposed in (Baudouin et al. 2014) should also be parallelizable using our partitioning strategy.

100 fibers

# threads	# tetrahedra	Timings (s) BR	Refine	Total
1	12 608 242	0.74	19.6	20.8
2	12 600 859	0.72	13.6	14.6
4	12 567 576	0.72	8.7	9.8
8	12 586 972	0.71	7.6	8.7

300 fibers

# threads	# tetrahedra	Timings (s) BR	Refine	Total
1	52 796 891	6.03	92.4	101.3
2	52 635 891	5.76	61.2	69.0
4	52 768 565	5.71	39.4	46.8
8	52 672 898	5.67	32.5	39.8

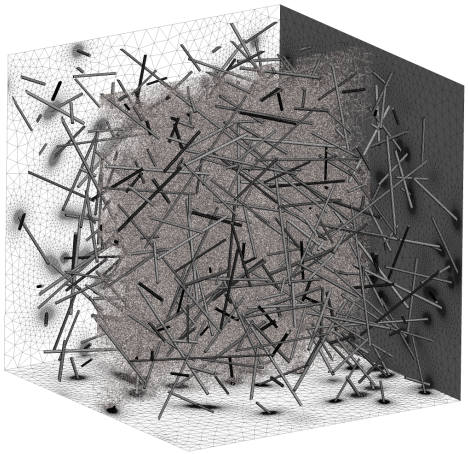
Mechanical part

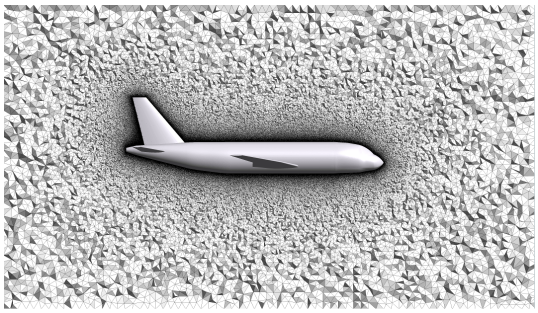
# threads	# tetrahedra	Timings (s) BR	Refine	Total
1	24 275 207	8.6	43.6	56.3
2	24 290 299	8.4	30.4	41.8
4	24 236 112	8.1	24.6	35.3
8	24 230 468	8.1	21.8	32.6

Truck tire

# threads	# tetrahedra	Timings (s) BR	Refine	Total
1	123 640 429	75.9	259.7	364.7
2	123 593 913	74.5	166.8	267.1
4	123 625 696	74.2	107.4	203.6
8	123 452 318	74.2	95.5	190.0

Figure 4.10: Performances of our parallel mesh generator on a Intel® Core™ i7-6700HQ 4-core CPU. Wall clock times are given for the whole meshing process for 1 to 8 threads. They include IOs (sequential), initial mesh generation (parallel), as well as sequential boundary recovery (BR), and parallel Delaunay refinement for which detailed timings are given.





100 thin fibers

# threads	# tetrahedra	Timings (s) BR	Refine	Total
1	325 611 841	3.1	492.1	497.2
2	325 786 170	2.9	329.7	334.3
4	325 691 796	2.8	229.5	233.9
8	325 211 989	2.7	154.6	158.7
16	324 897 471	2.8	96.8	100.9
32	325 221 244	2.7	71.7	75.8
64	324 701 883	2.8	55.8	60.1
127	324 190 447	2.9	47.6	52.0

500 thin fibers

# threads	# tetrahedra	Timings (s) BR	Refine	Total
1	723 208 595	18.9	1205.8	1234.4
2	723 098 577	16.0	780.3	804.8
4	722 664 991	86.6	567.1	659.8
8	722 329 174	15.8	349.1	370.1
16	723 093 143	15.6	216.2	236.5
32	722 013 476	15.6	149.7	169.8
64	721 572 235	15.9	119.7	140.4
127	721 591 846	15.9	114.2	135.2

Aircraft

# threads	# tetrahedra	Timings (s) BR	Refine	Total
1	672 209 630	45.2	1348.5	1418.3
2	671 432 038	42.1	1148.9	1211.5
8	665 826 109	39.6	714.8	774.8
64	664 587 093	38.7	322.3	380.9
127	663 921 974	38.1	255.0	313.3

Figure 4.11: Performances of our parallel mesh generator on a AMD® EPYC 64-core machine. Wall clock times are given for the whole meshing process for 1 to 127 threads. They include IOs (sequential), initial mesh generation (parallel), as well as sequential boundary recovery (BR), and parallel Delaunay refinement for which detailed timings are given.

Chapter 5

Hybrid Meshing à la Voronoi

When generating 3D meshes constrained by input surfaces, the shapes and sizes of the generated tetrahedra depend on the model geometry and on the input surface mesh quality. The temptation is great to remesh the surfaces and volumes simultaneously. The temptation to do this while simplifying the small features of the surface model is even greater. During my PhD thesis, unaware of the inherent extreme difficulties of mesh generation in 3D, we experimented the construction of a hybrid mesh by extending the work of surface remeshing presented in Section 3.2.3. The method we propose generates a hybrid mesh (tetrahedra, prisms, pyramids) from the intersection of the Voronoi diagram of given points and a surface model. The Voronoi diagram determines a global subdivision of the model while the mesh building strategy is local: the type of the generated elements depends on the local thickness of the region considered and on the positions of the points sampling the model. The vertices, edges, facets, and cells of the final volumetric mesh are determined from the combinatorial analysis of the intersections between the model elements and a Voronoi diagram of sampling points. Where the intersections of the Voronoi cells with the model surfaces have a unique connected component, tetrahedra are modified to fit the input triangulated surfaces. Where these intersections are more complicated, a correspondence between the elements of the Voronoi diagram and the elements of the mixed-element mesh is used to build the final volumetric mesh.

Preliminary results were presented at the International Meshing Roundtable in 2012 and a more complete version in 2014.

- Pellerin, J., B. Lévy, and G. Caumon (2012). "A Voronoi-Based Hybrid Meshing Method". In: *21st International Meshing Roundtable, Research Notes*. San Jose, CA, USA
- Pellerin, J., B. Lévy, and G. Caumon (2014). "Toward Mixed-element Meshing based on Restricted Voronoi Diagrams". en. In: *Procedia Engineering* 82. Proceedings of the 23rd International Meshing Roundtable, London, UK, pp. 279–290

5.1 Twisting the Restricted Delaunay Triangulation

5.1.1 Main Idea

The idea we developed is modifying the restricted version of the Voronoi-Delaunay duality (Section 3.2.2) and propose a correspondence that considers the connected components of both the intersection of the Voronoi diagram with the regions and the intersection of the Voronoi diagram with the surfaces.

From a centroidal Voronoi diagram Section 3.2.3.1, we propose to take into account the connected components of the restricted Voronoi diagram to the regions, surfaces to build the final mesh. In the regular restricted Delaunay triangulation the different connected components are ignored Figure 3.8 When restricted Voronoi cells to the regions may intersect one surface (Figure 5.1a). In the classical Voronoi-Delaunay relationship, the dual of a point shared by such cells is a tetrahedron linking the four sites. We propose to modify the tetrahedron vertices and instead of the sampling points, take the centroids of the adjacent restricted Voronoi diagram to the surfaces (Figure 5.1b). In thin sections of the

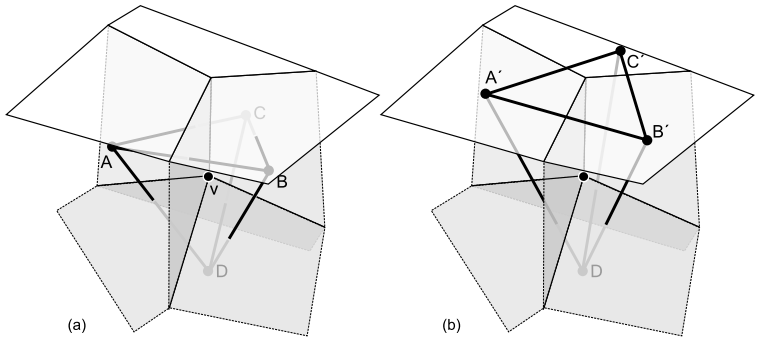


Figure 5.1: Delaunay tetrahedron modification near one surface.

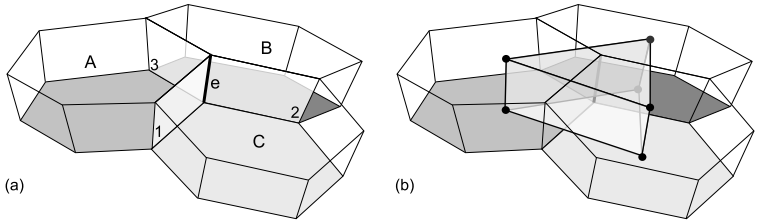


Figure 5.2: Prism built to mesh a thin section, it corresponds to the restricted Voronoi edge to this thin section e.

model, restricted Voronoi cells to regions will intersect several times the model surfaces (Figure 3.9). The connected components sandwiched between two surfaces may then not contain any Voronoi vertex (for example cells A, B, and C on Figure 5.2 share only a restricted Voronoi edge e). By putting one vertex at the centroid of each restricted Voronoi cell to the surfaces, a prism corresponding to e can be built.

Putting in correspondance the cells, facets, edges, and vertices of the Voronoi diagram cut into several components by the model surfaces with the cells, facets, edges, and vertices of a hybrid mesh is a hazardous task. We proposed however to do so. One or two vertices of the final mesh are associated to each volumetric restricted Voronoi cell depending on the number of adjacent surface connected components.

5.1.2 Building the Mesh Cells

To build all the cells automatically whatever the configuration of their intersections with the model regions and surfaces, we propose an algorithm that consists of the following steps.

1. Identify the cell from the corresponding restricted Voronoi edges and vertices.
2. Build the cells one after another by adding successively their vertices, edges, and facets

The procedure is illustrated for a restricted Voronoi vertex to a thin section sandwiched between two boundary surfaces (Figure 5.3a).

Vertices For each of the four volumetric restricted Voronoi cell containing the restricted Voronoi vertex, there is one or two points to add to the cell. If the restricted cell does not intersect model surfaces, that point is the site of the cell. If it does, the centroids of these intersections (adjacent connected component of the restricted cell to the surfaces) are taken for the points. When there are more than two intersections, some points are merged at their centroid so that there is maximum two points per restricted cell. For example, on Figure 5.3a, each restricted cell that contains vertex v is adjacent to two surface connected components and corresponds to two points (Figure 5.3b).

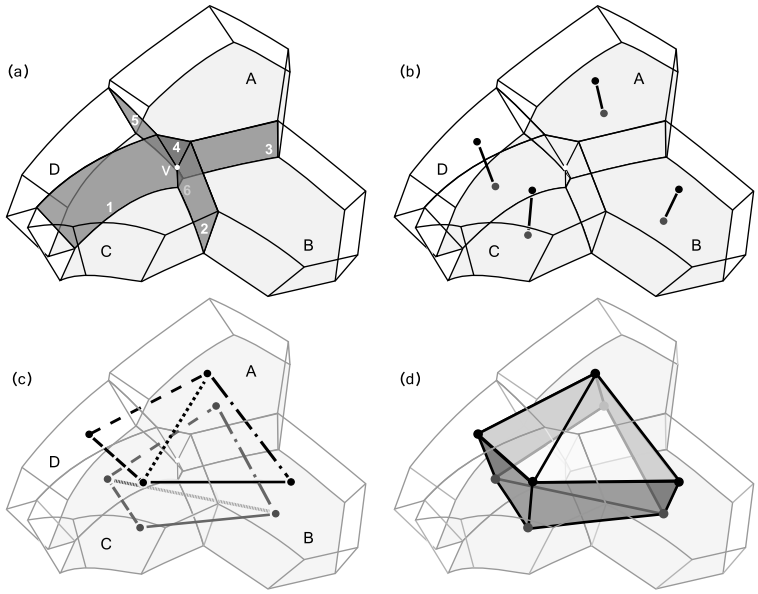


Figure 5.3: Cell building. (a) Restricted Voronoi vertex v to a thin section is shared by four restricted Voronoi cells (A, B, C, D), 6 facets, and 4 edges. (b) To each restricted cell correspond two vertices and one edge. (c) To each facet correspond one (facets 4 and 6) or two edges (facets 1, 2, 3 and 5). (d) Final cell has eight vertices, 4 triangle facets, and 4 quad facets.

Edges There are two types of edges to add: those linking the points corresponding to the same Voronoi cell (Figure 5.3b), and those corresponding to Voronoi facets containing v (Figure 5.3c). Each of the facets 1, 2, 3, 5 corresponds to two edges; the small triangular facets 4 and 6 intersect once the model surfaces and correspond to a unique edge in the cell to build.

Facets The first facets to add are those corresponding to the four restricted edges containing v. They are built from the edges corresponding to the restricted facets containing the restricted edge. In our example, they are all triangles (Figure 5.3d), but in the general case they could be polygons with up to six vertices. The other facets to add correspond to the restricted Voronoi facets containing v that are adjacent to at least two surface connected components. The four gray quad facets on Figure 5.3d correspond to restricted facets 1, 2, 3 and 5 on Figure 5.3a.

Finalizing the current cell Once all vertices, edges, and facets have been added to the cell, the cell type is determined from the number of these elements. Theoretically, a cell created with the above algorithm may have four to eight vertices, six to sixteen edges, four to ten facets, each facet having up to six vertices. In practice, most of the cells are tetrahedra, prisms, or pyramids. The other cells are processed as follows. They are first sorted into three categories: (1) the cells to subdivide (cells with 7 or 8 vertices defining a valid volume, Figure 5.3d), (2) invalid cells (in which at least one edge is a diagonal of a facet, or in which two quad facets share more than two vertices, like 5-point cells which are relatively abundant, see Figure 5.4a) and (3) the cells that belong to none of the above categories. Cells to subdivide are subdivided by adding a vertex at their centroid and building pyramids and tetrahedra with their facets. To compute tetrahedra, pyramids, and prisms from invalid cells, a two step procedure is implemented. First, all facets that are inconsistent with some edges are split along these edges. Then the validity of cell facets must be ensured. Facets with more than four vertices are triangulated and quad facets that share three vertices with another facet are split (Figure 5.4b). This processing permits to reduce the number of invalid cells, but is not sufficient to obtain a final valid mesh in all cases.

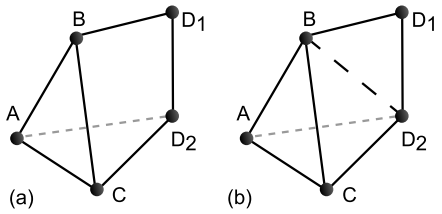


Figure 5.4: Invalid cells with 5 vertices. (a) Vertices D_1 and D_2 correspond to the same Voronoi cell. Facets ABD_1D_2 and BCD_1D_2 share three vertices. (b) Cutting these facets with BD_2 results in a tetrahedron plus a triangular facet.

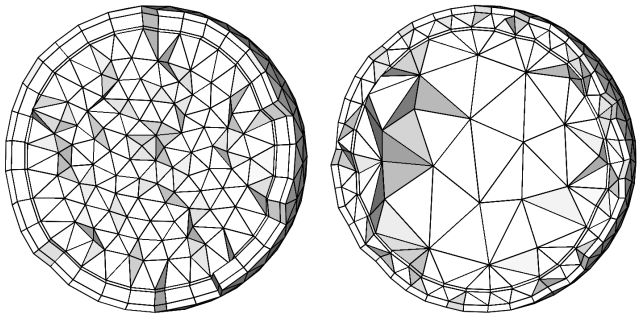


Figure 5.5: Two meshes of 4 nested spheres computed with two different samplings

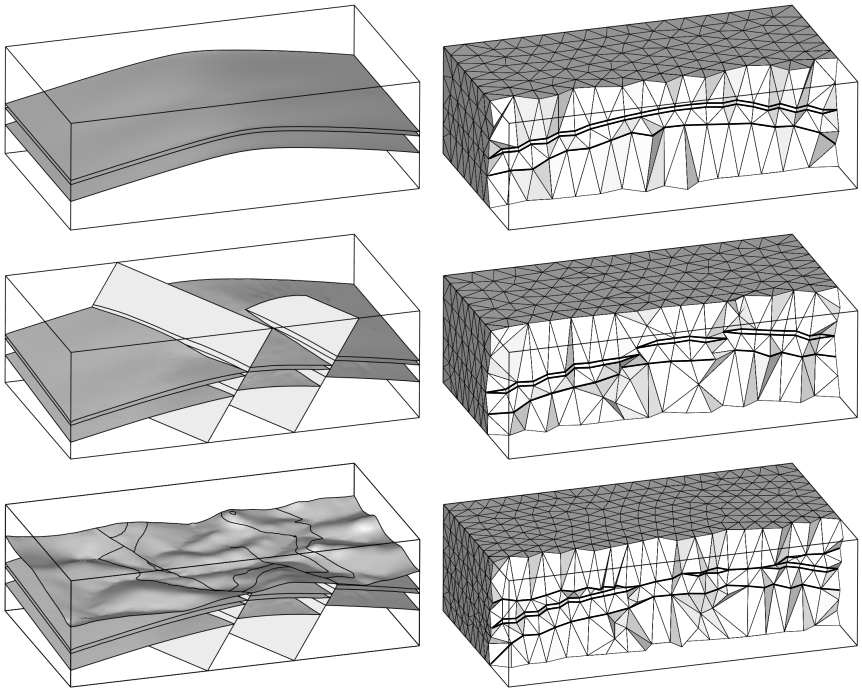


Figure 5.6: Hybrid meshes for 3 synthetic geological models. Region boundaries are highlighted.

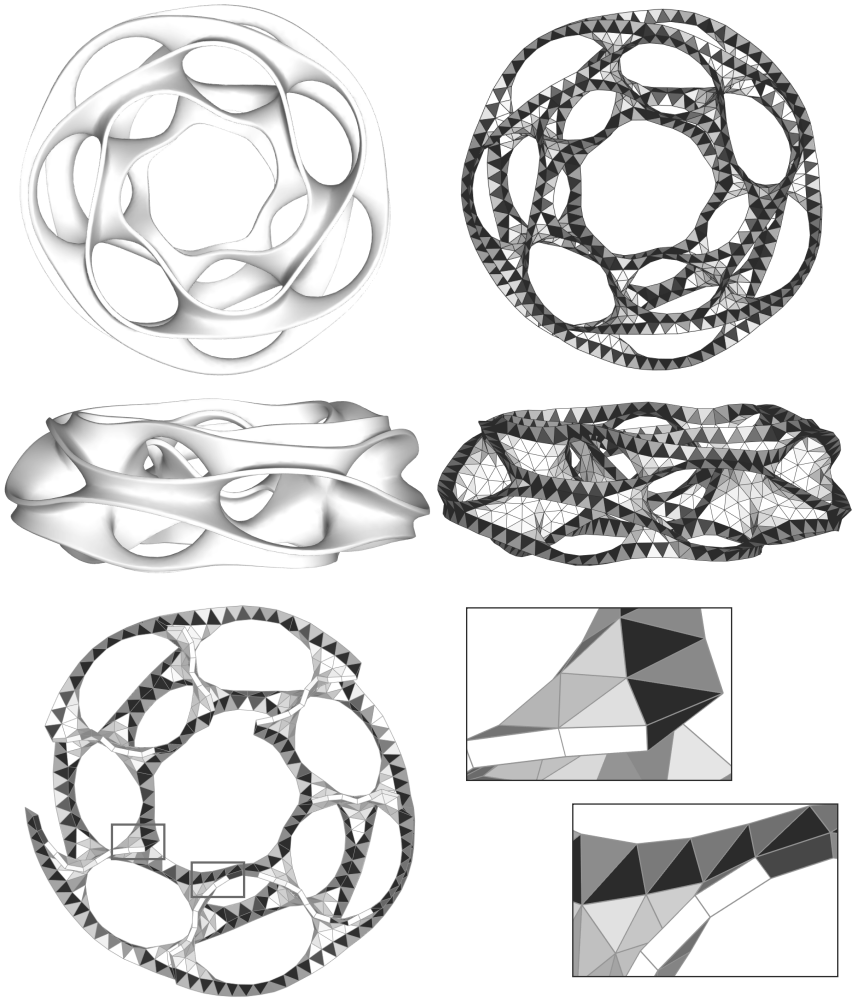


Figure 5.7: Mixed element mesh inside a smooth heptoroid surface, 516 tetrahedra (gray), 665 prisms (white), and 507 pyramids (black).

5.2 Could it Work?

The results of the algorithm are quite impressive for simple models in which layer thicknesses do not vary significantly and that have smooth boundaries, such as three nested spheres. It is not only the number of points sampling the spheres that determines which layers are thin and meshed with prisms, but also the positions of these points and the way they were optimized (Figure 5.5). The heptoroid model is meshed from only 1010 sampling points in 11 seconds (Figure 5.7). However the model is pinched near its extremities because only one site samples both creases and each restricted Voronoi cell to the surface has one connected component. The surface and volume are remeshed simultaneously. The final mesh does not depend on the input triangulation of the model surfaces (the only requirement is that their meshes are conformal) and no preliminary identification of the thin sections, that are meshed with prisms, is required.

Results for synthetic geological models are less convincing (Figure 5.6). The thinnest layer of the simple fold model on the top of Figure 5.6 is completely meshed with prisms. In the second model, built from the first by adding two faults that cut and displace the layers (one of them not cutting through the model), the same thin layer is also meshed with prisms. The discontinuities induced by the faults are respected in the mesh. However the layer is pinched near the faults and near the model box, an approximation that may be undesirable. The third model is built from the second by adding an erosion surface, below which remaining layers can be very thin and which can intersect other surfaces tangentially. Computational time (between several seconds and several minutes on a laptop with a 8 cores 1.73GHz Inter Core i7 processor and 8 GB of RAM) are quite good, it depends on the number of sampling points, on the input model number of triangles, and on the number of invalid cells to process.

The method has the same drawbacks than the surface remeshing method on which it is based Section 3.2.3. The outpush mesh is not valid because some output cell do not define a valid volume, and some other cells may intersect neighboring cells. Like for octree meshing methods, the difficulty is that all possible configurations for the intersections between subdivision cells and the model can and will occur. Having a control on these intersections is one of the reasons why the points are optimized so that their (restricted) Voronoi diagram is centroidal, and why many (re)meshing methods based on such an optimization refine the model sampling so that cell intersections with the model have a unique connected component (and verify the topological ball property (Edelsbrunner and Shah 1997)).

Missing pieces to make it work However, we did not think (then) that the fact that the algorithm fails for some cells is a show stopper, because, before a fully satisfactory solution is found, whenever the algorithm does not successfully generate a polyhedral cell, it is still possible to generate tetrahedra in cavities corresponding to the cells where the algorithm failed. The second possibility is to completely change the cell building strategy, and instead of building a mesh from scratch, progressively modify the Delaunay triangulation of the sampling points by local valid operations (vertex displacement, vertex/edge/facet duplication) that would permit to obtain the desired conformal mixed element mesh. For instance, as shown on Figure 5.4, whenever a restricted Voronoi cell intersects the set of surfaces twice, it may generate a cell with 5 vertices, due to the considered vertex that will be duplicated. In addition, note that when duplicating a vertex and moving the two instances at different locations (i.e., at the centroid of the restricted Voronoi cell connected components), special care is needed to avoid generating inverted elements (i.e. with negative orientation/Jacobian).

Food for thought Mesh generation in 3D is hard, because geometry in 3D is hard. Hoping to easily extend a meshing method from 2D to 3D is a dream that never comes true. The third dimension means trouble, and we often cannot imagine how bad it can get. So, if 2D is not a piece of cake, do not even try to think about 3D.

Chapter 6

Hybrid Meshing: Indirect Method

Motivations Hexahedral meshes are considered by many of the finite element practitioners to be superior to tetrahedral meshes. Compared to tetrahedral meshes, hexahedral meshes have important numerical properties: faster assembly (J.-F. Remacle, Gandham, et al. 2016), high accuracy in solid mechanics (E. Wang et al. 2004), or for quasi-incompressible materials (Benzley et al. 1995). Yet, no robust meshing technique is able to process general 3D domains. And generating hexahedral meshes in an automatic manner is still considered as the biggest challenge in mesh generation (Shepherd et al. 2008). The generation of ustructured hexahedral meshing is an active topic, e.g. (Gao et al. 2017; Lyon et al. 2016; Solomon et al. 2017; Fang et al. 2016) but some of the underlying theoretical challenges remain to explore as we will see in Chapter 7.

An alternative to full-hexahedral meshes are hybrid meshes that contains hexahedra (ideally a majority), prisms, pyramids and tetrahedra. To build those meshes promising, several author proposed to start from a tetrahedral mesh and merge its tetrahedra to build the output mesh (Meshkat et al. 2000; Yamakawa et al. 2003; Levy and Y. Liu 2010b; Baudouin et al. 2014; J. Huang et al. 2011; Botella et al. 2016; Sokolov, Ray, Untereiner, and Levy 2016). These methods take advantage of the existence of robust algorithms to generate tetrahedral meshes (see Chapter 4). Combining tetrahedra permit to produce meshes composed of hexahedra associated to prisms, pyramids and tetrahedra. The four steps of these indirect hex-dominant meshing methods can be summarized as follows (see also Figure 6.1):

1. A set of mesh vertices V is initially sampled in the domain.

2. A tetrahedral mesh T is built by connecting V, e.g. using a Delaunay kernel like (Si 2015).

3. The set of potential cells H (hexahedra, prisms, pyramids) that can be defined by combining tetrahedra of T is built.

4. A maximal subset $H_c \subset H$ constituted of cells that can be part of the same final mesh is determined. The final *hex-dominant mesh* is obtained adding the remaining not selected tetrahedra T'.

Contribution The ultimate goal is to combine all tetrahedra into hexahedra, i.e. obtain a final full-hexahedral mesh. All steps of indirect method are then crucial. Previous works primarily focus on the first step of placing the final mesh vertices. In this chapter, we focus on the third step. Our input is a tetrahedral mesh T of a given point set V Our contribution is an efficient algorithm that performs the identification of combination of tetrahedra into hexahedra, prisms and pyramids, and builds the set H of all possible combinations of tetrahedral elements of an input mesh T. All identified cells are valid for engineering analysis. Each potential hexahedron/prism/pyramid is computed only once. Around 3 millions potential hexahedra are computed in 10 seconds on a laptop. The method is about 10 times faster than previous methods while identifying all possible combinations to the contrary of method relying on a predefined set of patterns of the decomposition of a hexahedron into tetrahedra (Meshkat et al. 2000; Botella et al. 2016; Sokolov, Ray, Untereiner, and Levy 2016), or those relying on patterns of edge connections in a hexahedron (Yamakawa et al. 2003; Baudouin et al. 2014).

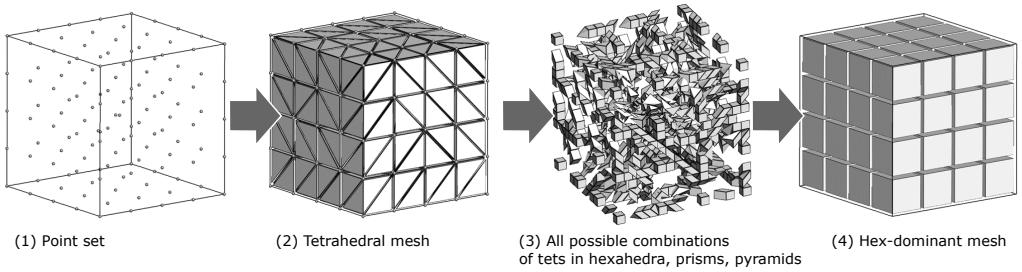


Figure 6.1: Indirect hexahedral dominant meshing principle.

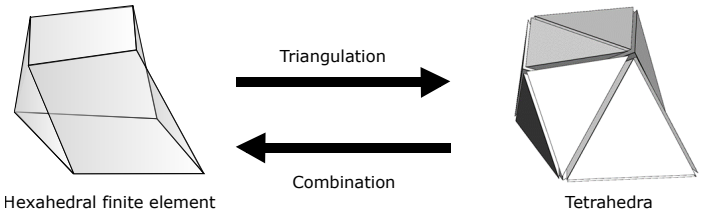


Figure 6.2: Combination of tetrahedra is the inverse of triangulating of a hexahedron. Triangular facets of tetrahedra approximate the bilinear facets of the hexahedron.

- Pellerin, J., A. Johnen, and J.-F. Remacle (2017). "Identifying combinations of tetrahedra into hexahedra: a vertex based strategy". en. In: *Procedia Engineering* 203. Proceedings of the 26th International Meshing Roundtable, Barcelona, Spain, pp. 2–13 **Selected among the 10 best papers for journal publication**

- Verhetsel, K., J. Pellerin, A. Johnen, and J.-F. Remacle (2017). "Solving the Maximum Weight Independent Set Problem: Application to Indirect Hexahedral Mesh Generation". In: *26th International Meshing Roundtable, Research Notes*. Barcelona, Spain

- Pellerin, J., A. Johnen, K. Verhetsel, and J.-F. Remacle (2018). "Identifying combinations of tetrahedra into hexahedra: A vertex based strategy". en. In: *Computer-Aided Design* 105, pp. 1–10

The C++ code implementing the methods is open-source and available at `https://www.hextreme.eu/download/`.

6.1 State of the Art

Before giving details on subdivisions of hexahedra , prisms or pyramids into tetrahedra (Section 6.1.1) and on methods used to identify these in an existing mesh (Section 6.1.2), we define the terms and notations used throughout this chapter.

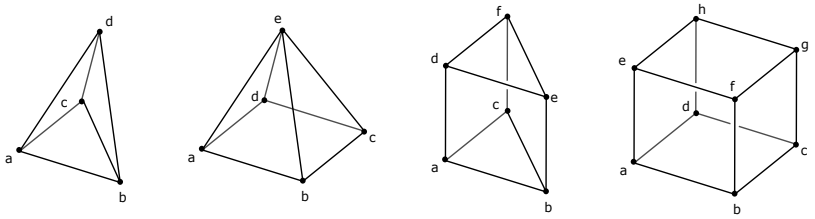


Figure 6.3: 3D cell templates: tetrahedron, pyramid, prism, and hexahedron. These linear finite element cells have straight edges, planar triangulat facets, and bilinear quadrangular facets.

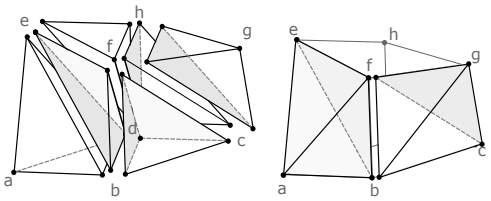


Figure 6.4: Interior tetrahedra and boundary tetrahedra of a hexahedron triangulation. Boundary tetrahedra connect the four vertices of the same quadrilateral facet.

Definitions We have to be very clear that all the cells we are considering are finite element cells hexahedra /prisms/pyramids valid for finite element simulations. A very important point is that their quadrilateral facets are not planar, but are bilinear surfaces. We require the Jacobian determinant to be strictly positive at any point inside the cell. The following conventions will be used throughout this chapter (see Figure 6.3).

A *triangulation* of a hexahedron/prism/pyramid is a triangulation of the vertices of the cell that respect the cell boundary, in other words it is a subdivision of the interior of the hexahedron/prism/pyramid into a set of conformal tetrahedra without any additional vertex (Figure 6.2). The tetrahedra induce a subdivision of each quadrilateral facet into two triangles by a diagonal boundary edge. We further define the *boundary tetrahedra* as the tetrahedra connecting the four vertices of a cell quadrilateral facet (Figure 6.4.2). In previous works (Meshkat et al. 2000; Botella et al. 2016; Baudouin et al. 2014; Sokolov, Ray, Untereiner, and Levy 2016), boundary tetrahedra are called slivers. We do not use that term which refers to a geometrical property (degeneracy) of tetrahedra. The triangulation is determined by *interior tetrahedra* (Figure 6.4.1). Indeed, the addition or removal of one or several boundary tetrahedra does not modify the cell.

6.1.1 Decomposing Hexahedra into Tetrahedra

There are as many subdivisions of the general pyramid as there are subdivisions of its planar base (De Loera et al. 2010). We are considering square pyramids, there are then exactly two triangulations of the pyramid. The ordinary triangular prism is the result of the product of a triangle with an edge: $\text{prism}(D_3) = D_3 \times D_2$. A $\text{prism}(D_n)$ has exactly $n!$ triangulations that are all equivalent to one another by affine symmetries (De Loera et al. 2010). The prism has then 6 triangulations that are all equivalent.

74 triangulations of the 3-cube $\mathbf{I}^3 = [0,1]^3$ (De Loera et al. 2010) :

1. Every triangulation of the 3-cube contains either a regular tetrahedron (i.e. a tetrahedron whose 6 edges are of equal lengths) or a diameter, i.e. an interior edge joining two opposite vertices (red edges on Figure 6.5).
2. There are 2 triangulations with a regular tetrahedron, symmetric to one another. The triangulations containing an interior edge are completely classified modulo symmetries by their dual complex which can be one of the last five shown on Figure 6.5. There are respectively 8, 24, 12, 24, 4 triangulations in each class.

A dual complex (Figure 6.5) is a practical way to visualize the 6 different possible decompositions (tetrahedrizations) of the 3-cube. In the dual complex, one vertex corresponds to one tetrahedron and two vertices are connected by an edge if the corresponding tetrahedra are adjacent through a triangular facet. A 2-cell of the dual complex (cycle in the dual graph) corresponds to an interior edge of the tetrahedrization (red on Figure 6.5). In a meshing context, these different possible decompositions of the 3-cube were identified by (Meshkat et al. 2000) who enumerate the feasible dual complex graphs, called RF-graph in their paper. In the RF-graph, additional dashed edges connect tetrahedra that are adjacent to the same quadrilateral facet (Figure 6.5).

Triangulations of the real cube Recently, the work of (Meshkat et al. 2000) was extended by (Botella et al. 2016) and (Sokolov, Ray, Untereiner, and Levy 2016) who proposed four additional

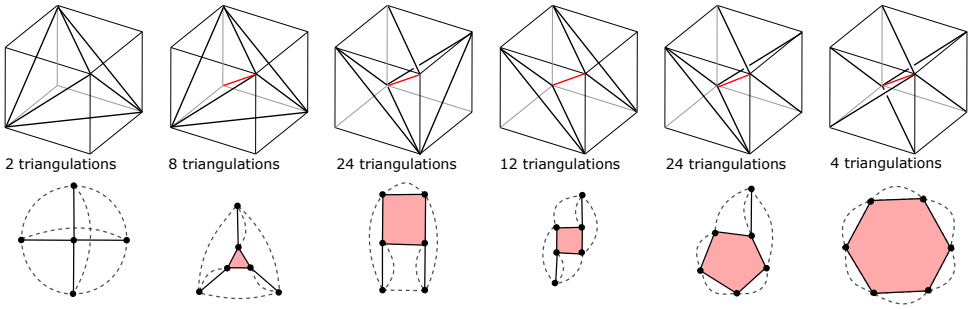


Figure 6.5: The six types of triangulations of the 3-cube and their dual complex representations.

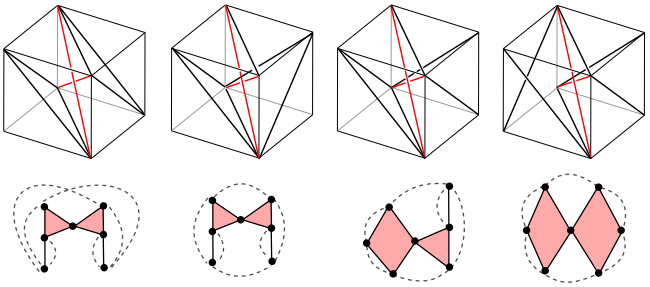


Figure 6.6: The four types of triangulations of an almost perfect cube into 7 tetrahedra proposed by (Botella et al. 2016) and their dual complex representation.

decomposition patterns into seven tetrahedra (Figure 6.6). The hexahedron is split into two prisms by a tetrahedron without any facet on the hexahedron boundary and containing two interior edges. For this tetrahedron to have a strictly positive volume, it is sufficient to work in finite precision, i.e. move slightly one of its vertices.

6.1.2 Combining Tetrahedra into Hexahedra

Two main approaches have been proposed. The first relies on a predefined set of patterns of the decomposition of a hexahedron into tetrahedra (Meshkat et al. 2000; Botella et al. 2016; Sokolov, Ray, Untereiner, and Levy 2016), the second on patterns of edge connections in a hexahedron (Yamakawa et al. 2003; Baudouin et al. 2014). Both strategies do not build the largest set of potential hexahedra H, and would miss the hexahedra on Figure 6.7.

To compute the set H of potential hexahedra and other cells that may be built by combining the elements of a tetrahedral mesh T without modifying its connectivity there are two known approaches. (Meshkat et al. 2000) propose to find combinations of tetrahedra into hexahedra by searching the adjacency graph of T for all occurrences of the cube decomposition dual complexes (Figure 6.5). The problem of matching subgraphs in large sparse graphs is solved using standard data mining algorithms that operate on graphs. The same technique is used by (Levy and Y. Liu 2010a; J. Huang et al. 2011; Botella et al. 2016) and (Sokolov, Ray, Untereiner, and Levy 2016) who consider four decompositions into seven tetrahedra (Figure 6.6).

The second approach proposed by (Yamakawa et al. 2003) relies on the vertices and edges of the tetrahedral mesh T. Local searches are performed into the vertex-edge graph of T using two patterns. These vertex connectivity patterns generalize those proposed by (Meshkat et al. 2000) and relax partially the dependency on the tetrahedral mesh. This method has been implemented by (Baudouin et al. 2014) where a third pattern taking into account configurations with an interior flat tetrahedron was added. Some other approaches like H-Morph (Owen and Saigal 2000) combine tetrahedra into hexahedra, while allowing for modifications of the connectivity and geometry of the input tetrahedral mesh (tetrahedron flips, node insertions, and node displacement). This great flexibility can make the algorithm intractable, but one advantage is that the mesh is valid throughout the procedure.

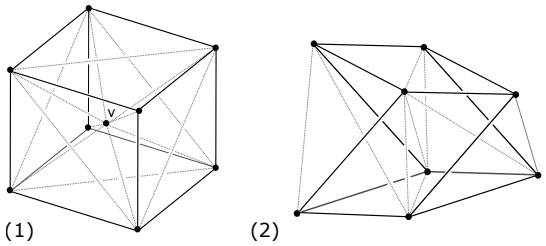


Figure 6.7: Two hexahedra not identified by existing combination methods. (1) A decomposition with an interior vertex v. (2) A decomposition into eight tetrahedra. This is a counter example to (Sokolov, Ray, Untereiner, and Levy 2016)'s claim that there is no decomposition of the hexahedron into more than seven interior tetrahedra.

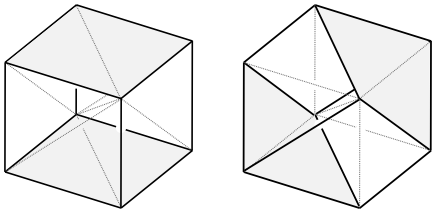


Figure 6.8: Two hexahedra with different edges and faces may be defined from the same set of tetrahedra.

6.1.3 Motivations for a New Approach

Important observations led us to work on improving these existing techniques. First, they do not identify the largest set H of potential hexahedra. On Figure 6.7 we gave two valid hexahedra that would neither be found by (Meshkat et al. 2000)'s method nor by (Yamakawa et al. 2003)'s method. The first is a decomposition that encompasses one internal vertex, a configuration that may occur when a Steiner point is added when generating the tetrahedra. The second is a decomposition that has 8 interior tetrahedra. It is a counter example to (Sokolov, Ray, Untereiner, and Levy 2016)'s claim that there is no hexahedron decomposition with more than 7 interior tetrahedra. Both decompositions are not identified when searching for hexahedra made of 5, 6, or 7 interior tetrahedra. Neither are they by (Yamakawa et al. 2003)'s algorithm since none of their constitutive tetrahedra has three facets on the hexahedron boundary.

Second, as mentioned by (Yamakawa et al. 2003), several hexahedra may be defined using the same decomposition pattern by modifying the ordering of the vertices (Figure 6.8). The hexahedra have different edges and different faces while having the same tetrahedral decomposition. The hexahedron on the left being a perfect cube, the one on the right is undoubtedly invalid (zero Jacobian determinant), but were the vertices in a more general position, both could be valid. Third, the existing methods identify the same hexahedron several times. That number is as high as the number of corner tetrahedra in the decomposition in (Yamakawa et al. 2003)'s approach and depends on dual complex symmetries in (Meshkat et al. 2000)'s approach.

6.2 Algorithm to Combine Tetrahedra into Hexahedra

6.2.1 2D Algorithm

Combinatorics Given a set V of n vertices labeled from 1 to n, let us first compute all possible quadrilaterals that can be defined from these vertices. We define a numbered quadrilateral $abcd$ by the order of its 4 vertices and we define a non-oriented quadrilateral $abcd$ by its edges ab, bc, cd and da ignoring orientation. Generating all numbered quadrilaterals that can be built from V is a combinatorial problem solved by a permutation generation algorithm that oup of 4 vertices of the n labels. When $V = \{1, 2, 3, 4\}$, the output is the set of the 24 permutations of 4 values. These 24 permutations define 3 different non-oriented quadrilaterals, each of them corresponding to 8 equivalent permutations. Adding constraints on the relative order of the vertices of one quadrilateral $a < b$, $a < c$, $b < d$, we obtain Algorithm 3 which fulfill our first objective and compute all possible non-oriented quadrilaterals that

Algorithm 3 GENERATE UNIQUE QUADRILATERALS

Input: V vertex set
Output: Q set of potential quads

```
for all a in V do
    for all b in V, b > a do
        for all c in V, c > a, c ≠ b do
            for all d in V, d > b, d ≠ c do
                Q ← Q ∪ {a, b, c, d}
```

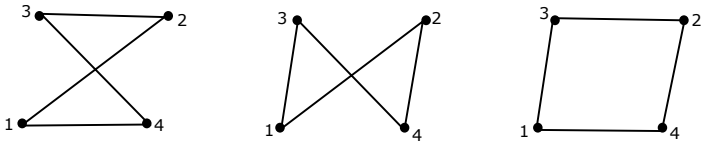


Figure 6.9: For 4 points in $\mathbb{R}^2$, only 1 over 3 possible combinatorial quadrilateral is valid. Note that in $\mathbb{R}^3$, all 3 quadrilaterals define valid bilinear quadrilateral facets.

can be built from V. The output for $V = \{1, 2, 3, 4\}$ is now the 3 non-oriented quadrilaterals that may be generated from 4 vertices: 1234, 1243, 1324.

Geometry Now, associate to each labeled vertex a point of $\mathbb{R}^2$. Among the 3 possible quadrilaterals that can be defined from 4 points of $\mathbb{R}^2$, only one is valid, i.e. has non-intersecting edges (Figure 6.9). We further associate to the set of vertices V a triangulation T and modify Algorithm 3 such that it generates all quadrilaterals whose edges are edges of the triangulation. The search for b and d is then restricted to the set of vertices connected to a. Similarly c should be connected through an edge to both b and d. The last step of the procedure is to identify the triangles subdividing each quadrilateral. Vertex selection order is now a, b, d, c instead of a, b, c, d since the choice of c depends on both b and d. All steps of the identification of quadrilaterals in a simple mesh are detailed on Figure 6.10b.

The advantage of this approach over the classical algorithms pairing adjacent triangles, e.g. (J.-F. Remacle, Lambrechts, et al. 2012) is that it identifies quadrilaterals which encompass one (or more) vertex. An example is the quadrilateral {1245} on Figure 6.10a that encompasses vertex {3}. The other advantages of the algorithm are that it is easy to add geometrical quality tests (edge lengths, angles of the quadrilateral under construction) and that its parallelization is trivial. Its complexity may seem prohibitive but a vertex of a 2D triangulation is connected to an average of 6 other vertices.

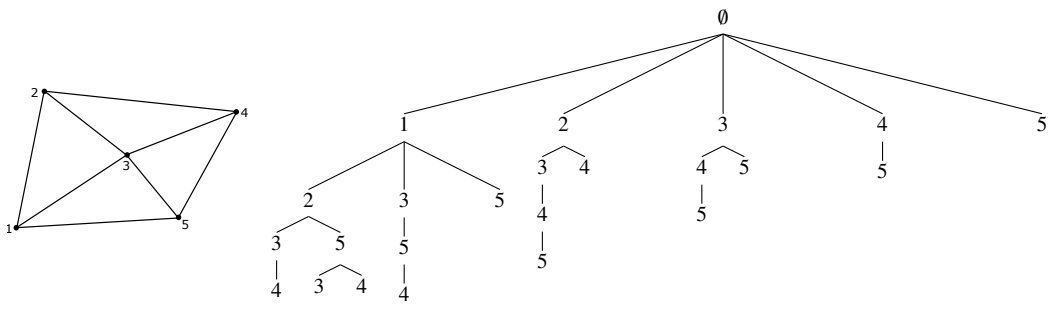


Figure 6.10: Search tree for quadraliterals in a triangulation. Left: Input 2D mesh. Right: Each branch reaching a depth of 4 defines a quadrilateral. 5 quadrilaterals are identified: {1243}, {1235}, {1345}, {1245}, {2354}.

6.2.2 3D Algorithm

Combinatorics The 3D algorithm is direct extension of the 2D algorithm to identify combinations of tetrahedra into hexahedra. We modify an algorithm generating all possible 8-subset of the set of labeled vertices V to generate exactly once all oriented hexahedra (Algorithm 4). The first corner of the hexahedron is built by choosing vertices {a, b, d, e} such that $b > a$, $d > b$ and $e > b$. This corner sets the orientation of the hexahedron (Figure 6.3). Orientation can be ignored by setting $e > d$. The four other vertices are chosen to be greater than a. The output of Algorithm 4 for $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ is a set of 1,680 oriented hexahedra. This is consistent with the well-known fact that there are 24 permutations of the labeled vertices that do not modify the orientation, the edges, or the faces of a hexahedron since $8! = 1,680 \times 24$. When orientation is ignored, there are 48 such permutations and then 840 different hexahedra. The complexity of Algorithm 4 is of course catastrophic and cannot be used as is for large point sets.

Algorithm 4 Generates exactly once each potential oriented hexahedron in a vertex set V.

Input: V vertex set
Output: H set of potential quads

for all a in V **do**
  for all b in $V, b > a$ **do**
    for all d in $V, d > b$ **do**
      for all e in $V, e > b$ **do**
        for all c in $V, c > a, e \notin \{b, d, e\}$ **do**
          for all f in $V, f > a, f \notin \{b, d, e, c\}$ **do**
            for all h in $V, h > a, h \notin \{b, d, e, c, f\}$ **do**
              for all g in $V, g > a, g \notin \{b, d, e, c, f, h\}$ **do**
                $H \leftarrow H \cup \{abcdefgh\}$

Geometry Let us now associate to each labeled vertex of V a point of $\mathbb{R}^3$. Among the $\binom{n}{8} \times 1680$ possible hexahedra, only a small subset will be geometrically valid. To restrict the search space for hexahedra, we build a triangulation (tetrahedrization) of the points. The key idea is to consider only hexahedra whose edges are triangulation edges and whose quadrilateral facets are a combination of two triangle facets. Algorithm 5 outputs all the hexahedra that may be built using the edges and facets of the triangulation. Note that the existence of the triangle facets must be checked explicitly, since, in a 3D triangulation, the existence of edges {ab}, {bc}, {bd}, {da} does not guarantee that any of the triangles {abc}, {acd}, {abd}, or {adc} exist in the triangulation. These tests are performed when computing the possible cells with Algorithm 5 in order to skip invalid configurations and accelerate the procedure. We also ensure that two merged triangles belong to the same parts of the input model, or model faces.

To compute all the hexahedra that can be generated by combining tetrahedra of an input mesh T the last step is to compute the tetrahedra of T that are inside the hexahedron. A tetrahedron is inside a cell if it either (i) have its four vertices be vertices of the cell (and the cell is convex) or (ii) have one facet on the boundary and a volume of the same sign than the cell, or (iii) be adjacent, through a facet that is not on the cell boundary, to a tetrahedron that respect (i) or (ii). At that at this step the real cell boundary is not yet determined, since there are two choices to triangulate each quadrilateral facet and 4 triangle facets are considered. We further ensure that all tetrahedra should belong to the same part of the model, otherwise the cell is discarded. Note that the boundary tetrahedra, as defined in Section 6.1, are not considered to be inside the cell and are ignored.

The algorithm to identify prisms is similar. To build pyramids it is however much faster to iterate on all triangular facets of T and to create three pyramids for each of them by changing the apex to be one of the three vertices of the facet.

Bounding the quality of generated cells To have an efficient detetection of adequate cells by Algorithm 5, it is crucial to discard invalid or bad quality cells as soon as possible. The quality and validity of a cell depend only on the coordinates of its vertices. We recall that we consider that a cell is

Algorithm 5 Vertex based search algorithm of the set of all potential hexahedra H in a tetrahedral mesh T.

Input: V vertex set, T tetrahedrization of V
Output: H set of potential hexahedra

```
for all a in V do
    for all b in neighbors {a}, b > a do
        for all d in neighbors {a}, d > b do
            for all e in neighbors {a} , e > b, e ≠ d do
                for all c in neighbors {b, d}, c > a, c ≠ e do
                    if !isQuadFace a, b, c, d then
                        continue
                    for all f in neighbors {b, e}, f > a, f ∉ {b, d, e, c} do
                        if !isQuadFace a, b, f, e then
                            continue
                        for all h in neighbors {d, e}, h > a, h ∉ {b, c, f} do
                            if !isQuadFace a, d, h, e then
                                continue
                            for all g ∈ neighbors {c, f, h}, g > a, g ∉ {b, d, e} do
                                if !isQuadFace d, c, g, h then
                                    continue
                                if !isQuadFace e, f, g, h then
                                    continue
                                if !isQuadFace b, c, g, f then
                                    continue
                                tetsInHex = computeTetrahedra{abcdefgh}
                                H ← H ∪ {abcdefg,tetsInHex}
```

valid if the Jacobian determinant is strictly positive at any point of the element. All cells that have a negative Jacobian determinant are discarded.

The quality of a finite element cell is defined as the minimal value taken by the scaled Jacobian determinant over the element. If this value is inferior or equal to zero, the cell is invalid. In the first-order finite element cells we are considering (hexahedra, prisms, and pyramids) the maximum quality of the element is bounded by the quality at the corners, itself bounded by the quality of the facets sharing this vertex $Q_{cell} < min(Q_{corners}) < min(Q_{facets})$.

- The quality of a quadrilateral face corner *abd* is evaluated as the sinus of the angle made by the incident edges:

$$sin(\vec{ab}, \vec{ad})$$

- The quality of a triangle facet corner *abc* (Johnen, Weill, et al. 2017) is evaluated as:

$$\frac{2\,||\vec{ab} \times \vec{ac}||}{3\sqrt{3}} \frac{||\vec{ab}|| + ||\vec{ac}|| + ||\vec{bc}||}{||\vec{ab}||\,||\vec{ac}||\,||\vec{bc}||}$$

- The quality of a hexahedron corner *abde* is evaluated as the scaled Jacobian:

$$\frac{|(\vec{ab} \times \vec{ad}) \cdot \vec{ae}|}{||\vec{ab}||\,||\vec{ad}||\,||\vec{ae}||}$$

- The quality of a prism corner *abcd* is evaluated as:

$$\frac{2\,(\vec{ab} \times \vec{ac}) \cdot \vec{ad}}{3\sqrt{3}} \frac{||\vec{ab}|| + ||\vec{ac}|| + ||\vec{bc}||}{||\vec{ab}||\,||\vec{ac}||\,||\vec{bc}||\,||\vec{ad}||}$$

- The quality of a pyramid base corner *abde* is evaluated as:

$$\frac{3\ det(J)^{\frac{2}{3}}}{\|J\|_F^2} \quad where \quad J = J_p J_I^{-1} \quad with \quad J_I = \begin{pmatrix} 1 & 0 & \frac{1}{2} \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & \frac{\sqrt{2}}{2} \end{pmatrix} \quad J_p = \begin{pmatrix} \vec{ab} & \vec{ad} & \vec{ae} \end{pmatrix}$$

  where $\|.\|_F$ denotes the Frobenius norm.

These quality values give an upper bound of the overall cell quality that we use to accelerate dramatically the cell identification of Algorithm 5. Indeed a new upper quality bound for the cell under construction can be computed when a vertex completing a face or a corner is added. When this bound becomes smaller than the required minimum quality, the cell construction is terminated. Additional quality tests on the planarity of quadrilateral facets, or on edge lengths could be added.

Guaranteeing that the Jacobian determinant of the final cells is strictly positive, is more challenging and time consuming and is the last test performed using the method of (Johnen, J.-F. Remacle, et al. 2013; Johnen, Weill, et al. 2017). We modified the implementation so that it is exact and robust to floating point errors.

6.3 Hex-Dominant Meshing

To demonstrate that hex-dominant meshes may be generated from the set of potential cells identified by the previously described algorithm, we choose a subset of all compatible hexahedra, prisms, and pyramids.

Cell compatibility To have a valid final mesh, chosen cells should be mutually compatible. Two cells (hexahedron, prism, pyramid) are compatible if all following conditions are satisfied:

1. They share no interior tetrahedron.
2. If they share 4 vertices, they share a quadrilateral face connecting these 4 vertices;
3. If they share 3 vertices, they share a triangle face connecting these 3 vertices;
4. If they share 2 vertices, they share an edge connecting these 2 vertices;

The incompatibilities between the remaining tetrahedra (not selected to build any cell) and the cells should also be accounted for. There are at least two possible strategies to manage them: (Owen, Canann, et al. 1997) propose to raise pyramids on each non-conformal quadrilateral face and (Botella et al. 2016) propose to subdivide the pyramid or hexahedron incident to a non conformal contact into pyramids and tetrahedra. Both methods insert a new vertex, the apex of the pyramid or a point inside the neighboring cell algorithm. Their major drawback is that they increase the proportion of tetrahedra and pyramids compared to the proportion of hexahedra. In the meshes produced for this chapter, the compatibility condition between tetrahedra and cells is relaxed. Some quadrilateral faces will be adjacent to one or two triangles. This mesh should then be used with finite element solvers capable of handling these type of non-conformities.

Graph formalization The selection of the cells of the final mesh can be reformulated as a Maximum Weight Independent Set (MWIS) problem (Botella et al. 2016). Let us consider the graph G that has one vertex for each of the cell that may be built by combining tetrahedra of the input mesh T, and one edge linking each pair of compatible cells. The objective is then to find the largest subgraph in which all vertices are linked to one another. This is the Maximum Clique Problem (MCP), which is in general NP-hard. We may further associate a weight to each vertex depending on the cell quality. Since the compatibility graph G is usually very dense, it is advantageous to replace it with its complement, the incompatibility graph G^*. The goal is then to find the solution to the Maximum Weighted Independent Set problem (MWIS).

When the graph G^* contains up to a few hundreds of vertices, the optimal solution may be found by enumerating all independent sets and comparing their total weights (Wu et al. 2015). However

such an algorithm cannot be expected to terminate in a reasonable amount of time for graphs with a few thousands of vertices, let alone graphs with a few millions of vertices like the one we obtain. Reviewing the methods to solve the relevant MWIS problem is out of the scope. The interested reader is referred to (Verhetsel, Pellerin, Johnen, et al. 2017) and references therein for applicable methods in the specific case of indirect hex-dominant meshing.

Greedy solution The strategy we develop to obtain a hex-dominant mesh is the one used by previous works: greedily compute an approximate solution to the MWIS problem (Baudouin et al. 2014; Botella et al. 2016; Sokolov, Ray, Untereiner, and Levy 2016). The vertices (potential cells) are sorted by decreasing weight (quality), and independent vertices (compatible cells) are iteratively added to the solution in decreasing order of weight. This solution could be improved by taking advantage of the locality of the problem and optimizing small disjoint subgraphs (Verhetsel, Pellerin, Johnen, et al. 2017). Our non-optimized sequential implementation runs typically in a few seconds. The resulting hex-dominant meshes for three of the input tetrahedral meshes are shown on Figure 6.11.

6.4 Results and Discussion

We have applied our algorithm to 12 different tetrahedral meshes. Those were generated using the point placement strategy described in (Baudouin et al. 2014) and implemented in Gmsh (www.gmsh.info). They have between 127 and more than 3 million vertices (Table 6.1). The results of the tetrahedra combination algorithm in terms of numbers of detected potential hexahedra, prisms, pyramids and computational times are given in Table 6.2. The number of potential hexahedra, prisms, pyramids mostly depends on the number of vertices of the input tetrahedral mesh and on the minimal required quality.

Performances As expected, for a given input mesh, the higher the minimal quality, the faster the algorithm. For example, the running time on dataset Knuckle decrease from 214s to about 9s when quality increases (Table 6.2). Discarding as soon as possible cells that have a low quality is key to the efficiency of the algorithm. The multi-threaded version of our algorithm is very fast with about 300,000 potential hexahedra built per second. The algorithm for prisms generates about 900,000 prisms per second and the one for pyramids about more than 1.5 millions per second. All timings and performances are given for a laptop with 16Go RAM and an Intel®Core™i7-6700HQ CPU @2.60 GHz processor. For all cells, the running time clearly depends almost only on the number of potential cells detected. Note that for models Knuckle and Los1 the computed potential cells are only counted since they do not fit in the 16 Go RAM. We estimate that this method is more than ten times faster than the state of the art pattern matching method to identify combinations of tetrahedra into cells (Sokolov, Ray, Untereiner, and Levy 2016). However, providing a valuable comparison is delicate because quality criteria have a strong impact on performances and are not explicitly stated in previous works, preventing comparison. Moreover no timings are provided for this one step of the hex-dominant meshing workflow.

Comparison of identified hexahedra with pattern based methods We compare the number of potential hexahedra identified by our algorithm with the number of potential hexahedra that would be identified by pattern-matching methods (Botella et al. 2016; Sokolov, Ray, Untereiner, and Levy 2016) or vertex combination (Yamakawa et al. 2003) in Table 6.3. To count the potential hexahedra matching one of the six cube decompositions or one of the four decompositions into seven tetrahedra we compare the dual complex graphs. To count the potential hexahedra that would be detected by (Yamakawa et al. 2003), we count those containing a tetrahedron that has three facets on the hexahedron facets. On small models, our algorithm detects 4 to 5% more potential hexahedra than the existing methods. That number depends on the input tetrahedral mesh. This demonstrates that using a predefined set of templates for the hexahedron triangulation does not permit to detect all the potential hexahedra in a given tetrahedral mesh.

On larger meshes, the small difference between methods can be explained by the point placement strategy of the input tetrahedral mesh. The points are generated by propagation from the boundary of

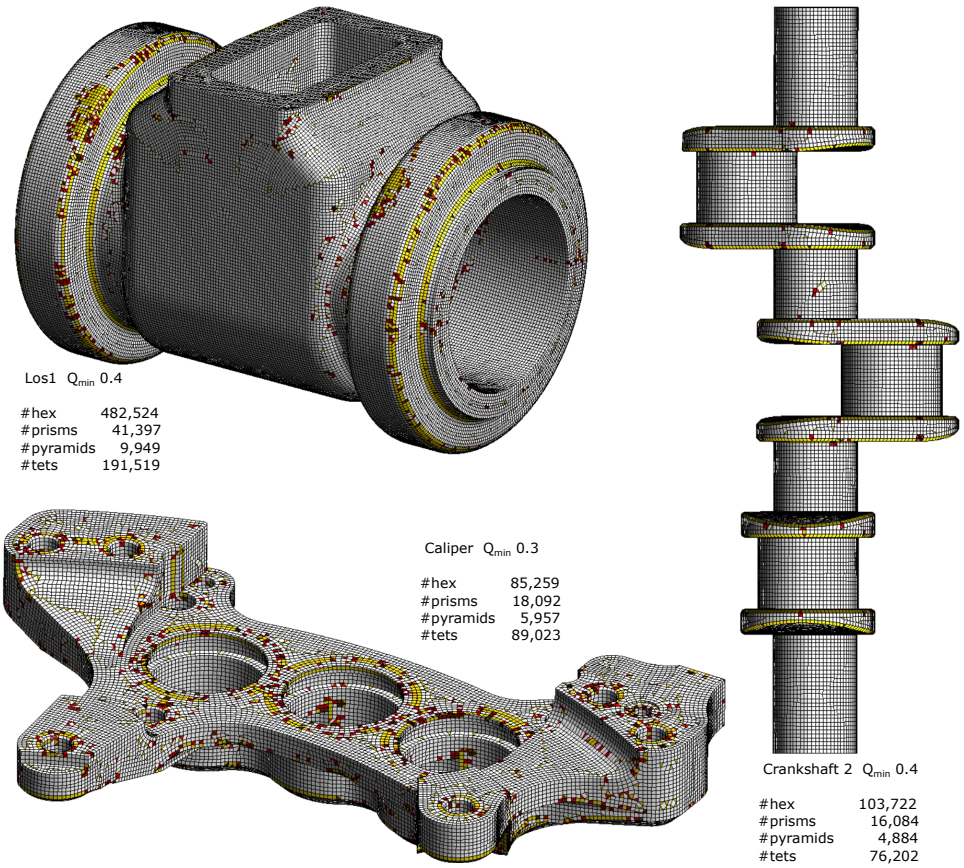


Figure 6.11: Examples of three hexahedral dominant meshes generated by a greedy selection (white: hexahedra, red: tetrahedra, yellow: prisms, black: pyramids). The complete workflow, from the loading of the tetrahedral mesh till the writing of the mixed-cell mesh, typically runs in less than a minute.

Model	#vertices	#tets	#bd tris	Load (s)	Struct. (s)
Cube	127	396	192	<0.01	<0.01
Fusee	11,975	50,750	15,128	0.07	0.04
CrankShaft	23,245	104,302	27,342	0.13	0.09
Fusee_1	71,947	349,893	55,954	0.40	0.26
Caliper	130,572	675,289	79,446	0.77	0.49
CrankShaft_2	140,985	763,870	59,656	0.87	0.52
Fusee_2	161,888	828,723	98,952	0.95	0.58
FT47_b	221,780	1,260,255	55,178	1.42	0.83
FT47	370,401	2,085,394	102,434	2.36	1.40
Fusee_3	501,021	2,694,950	217,722	3.07	1.89
Los1	583,561	3,250,705	182,814	3.76	2.28
Knuckle	3,058,481	17,466,833	640,081	17.49	16.25

Table 6.1: Characteristics of the input tetrahedral meshes on which tests are performed. For each model are given the number of vertices, the number of tetrahedra, the number of triangles defining the model boundary, the sequential timings to load the mesh and build the data structures used by our algorithm in seconds. The input meshes are available at: `www.hextreme.eu`.

		Cells		Timings (s)					Cells		Timings (s)		
Q_{min}	#Hexes	#Prisms	#Pyr.	Hex	Pri.	Pyr.	Q_{min}	#Hexes	#Prisms	#Pyr.	Hex	Pri.	Pyr.
Cube							Fusee 2						
0	710	1,596	1,172	0.03	0.01	0.01	0	4,586,779	5,866,663	3,122,057	9.46	6.88	1.21
0.2	349	1,052	661	<0.01	<0.01	<0.01	0.2	3,147,148	4,614,298	2,013,799	6.17	5.37	0.92
0.4	308	1,010	639	<0.01	<0.01	<0.01	0.4	1,940,213	3,978,611	1,746,731	4.13	4.79	0.85
0.6	129	661	615	<0.01	<0.01	<0.01	0.6	455,155	1,849,333	1,489,604	1.27	2.44	0.78
0.8	64	218	138	<0.01	<0.01	<0.01	0.8	114,224	520,350	333,250	0.40	0.86	0.46
Fusee							FT47_b						
0	149,731	251,131	158,671	0.50	0.37	0.07	0	8,374,128	9,845,963	4,954,902	16.34	11.28	2.06
0.2	95,259	193,440	120,496	0.30	0.28	0.06	0.2	6,111,946	8,046,676	3,177,657	11.11	9.11	1.45
0.4	50,931	148,390	99,949	0.19	0.23	0.05	0.4	3,413,741	6,963,922	2,798,967	6.88	8.09	1.34
0.6	14,978	73,397	74,831	0.07	0.13	0.04	0.6	869,890	3,257,991	2,349,396	2.33	4.13	1.22
0.8	4,187	20,323	18,312	0.02	0.05	0.03	0.8	184,921	808,736	745,059	0.65	1.40	0.78
CrankShaft							FT47						
0	309,938	516,217	327,071	1.06	0.78	0.14	0	13,842,934	15,994,194	8,098,013	26.79	19.32	3.42
0.2	196,441	400,698	255,842	0.65	0.60	0.12	0.2	10,225,710	13,177,806	4,968,253	18.28	14.90	2.33
0.4	97,194	291,381	205,388	0.38	0.47	0.11	0.4	5,978,752	11,638,277	4,441,345	11.65	13.54	2.18
0.6	27,549	138,308	143,964	0.15	0.26	0.09	0.6	1,481,030	5,392,542	3,886,408	3.76	6.76	1.99
0.8	6,381	35,157	33,804	0.05	0.10	0.06	0.8	325,062	1,428,695	1,080,358	1.05	2.32	1.24
Fusee 1							Fusee 3						
0	1,692,873	2,323,521	1,283,513	3.90	2.85	0.52	0	17,052,534	20,251,770	10,377,769	33.06	24.08	4.43
0.2	1,098,993	1,792,440	866,043	2.40	2.17	0.40	0.2	12,581,618	16,270,311	6,218,061	22.75	18.75	2.94
0.4	655,375	1,496,380	739,789	1.58	1.88	0.37	0.4	8,214,177	14,639,094	5,562,577	15.55	16.63	2.75
0.6	167,752	714,225	610,352	0.53	0.99	0.33	0.6	1,770,306	6,701,699	4,969,021	4.41	8.39	2.59
0.8	46,215	207,852	139,521	0.17	0.36	0.21	0.8	414,528	1,859,542	1,047,155	1.34	2.92	1.54
Caliper							Los1						
0	3,536,954	4,675,695	2,508,509	7.96	5.73	1.03	0	21,909,206	25,424,213	12,802,385	42.21	30.00	6.30
0.2	2,341,875	3,706,438	1,745,660	5.07	4.48	0.81	0.2	16,152,752	20,505,357	7,698,238	28.56	23.60	3.61
0.4	1,262,784	3,032,209	1,490,139	3.15	3.82	0.75	0.4	10,300,168	18,629,804	6,821,354	19.39	21.42	3.37
0.6	314,028	1,372,489	1,205,077	1.08	1.97	0.67	0.6	2,330,993	8,867,751	6,106,437	5.81	10.84	3.13
0.8	78,639	376,243	301,792	0.34	0.73	0.43	0.8	502,613	2,262,204	1,798,365	1.61	3.54	1.94
Crankshaft 2							Knuckle						
0	4,406,357	5,565,490	2,898,887	9.24	6.45	1.24	0	86,688,872	122,469,704	64,495,784	220.75	161.08	28.29
0.2	2,983,490	4,376,413	1,874,339	5.84	5.11	0.86	0.2	51,459,040	93,888,327	36,416,856	132.79	122.33	19.81
0.4	1,683,753	3,777,653	1,598,601	3.63	4.41	0.78	0.4	34,537,750	87,271,830	34,099,717	96.46	115.21	19.37
0.6	417,004	1,677,332	1,385,062	1.20	2.21	0.72	0.6	9,499,091	42,600,754	32,019,383	31.59	62.31	18.77
0.8	104,728	468,154	301,721	0.36	0.79	0.43	0.8	2,837,726	10,931,917	6,650,518	9.44	20.43	11.32

Table 6.2: Number of valid cells (hexahedra, prisms and pyramids) identified in 12 tetrahedral meshes (Table 6.1) with our algorithm for different minimal quality values. Running times are given for a laptop with 16Go RAM and an Intel®Core™i7-6700HQ CPU @2.60 GHz processor. Input tetrahedral meshes are available at: www.hextreme.eu

Model	Ours	[1]	[2]	[3]
Cube	710	672	696	706
Fusee	149,627	123,696	142,783	146,022
CrankShaft	310,181	251,806	294,182	302,131
Fusee_1	1,692,188	1,532,747	1,683,543	1,686,709
Caliper	3,536,997	3,175,201	3,513,863	3,522,483
CrankShaft_2	4,405,892	3,919,768	4,383,763	4,392,081
Fusee_2	4,585,236	4,110,253	4,568,332	4,574,594
FT47_b	8,374,930	7,384,695	8,343,306	8,355,231
FT47	13,846,837	12,133,218	13,806,781	13,821,507
Fusee_3	17,048,021	14,983,008	17,010,173	17,023,768
Los1	21,908,307	19,308,740	21,865,212	21,880,467
Knuckle	86,553,836	79,544,742	86,346,178	86,433,942

Table 6.3: Number of valid potential hexahedra detected by our algorithm and comparison with the number of hexahedra that correspond to patterns used by previous methods;: [1] (Meshkat et al. 2000), [2] (Botella et al. 2016; Sokolov, Ray, Untereiner, and Lévy 2017), [3] (Yamakawa et al. 2003)

the model. Where the fronts collide, a roughly 2-dimensional surface, point placement is not optimal. It is in this area that out method makes a difference, and the bigger the mesh is, the relatively smaller this area. Note that the higher the required minimum scaled Jacobian, the smaller the difference between the number of potential hexahedra detected by our method and the number of hexahedra detected by the existing methods. This is no surprise since the best quality is obtained for hexahedra that are close to the perfect cube which has a limited number of decompositions.

Challenges The algorithm presented in this chapter solves one step of the indirect hex-dominant meshing workflow that is the identification in a given tetrahedral mesh, of all the possible combinations of tetrahedral elements into hexahedra, prisms, pyramids. The algorithm identifies all the valid cells elements since no assumption is made on subdivisions into tetrahedra, each cell detected only once, bad quality cells are discarded early, and the algorithm is easy to implement. The C++ code is open-source and available at `https://www.hextreme.eu/download/` and in www.gmsh.info.

This is however only one step of the workflow to build hex-dominant meshes and all steps have a crucial impact on the final output. Indeed, the quality and number of potential hexahedra we can generate highly depend on the input tetrahedral mesh and then on the placement of its vertices. For example is the model Caliper shown on Figure 6.11 has complex geometrical features and points are non optimally placed around these preventing good quality hex-meshing. One key question is the optimization of such a mesh if that is possible, how can we increase the quality of such hybrid meshes (Chapter 9). The placement of the vertices is also one very important point for the generation of good hex-dominant mesh and is a very active research subject. The choice of the cells of the final mesh is the second key. We have shown that there may be up 30 times more cells than they are vertices in the input mesh, making the construction of the incompatibility graph very costly and the exact resolution of the MWIS problem intractable.

Part III

Theoretical Mesh Generation

Chapter 7

Theoretical Tetrahedral Meshing

Motivations The unexpected observation that there actually are much more patterns of subdivisions of the hexahedron into tetrahedra than previously thought when I implemented the algorithm that combines tetrahedra into hexahedra presented in Chapter 6 raised the (simple) question: How many more exactly? It is well known that the cube has five subdivisions into 6 tetrahedra and one subdivision into 5 tetrahedra. However, all hexahedra are not cubes and moving the vertex positions increases the number of subdivisions. Recent hexahedral dominant meshing methods try to take these configurations into account for combining tetrahedra into hexahedra, but fail to enumerate them all. The objective of this chapter is to answer this important theoretical question: How many different triangulations of the hexahedron into tetrahedra are there? And, more generally, is it possible to generate all the triangulations of a given polyhedron? and if it is, how do we do it?

How many triangulations of the hexahedron are there? A finite element hexahedron is combinatorially a 3-dimensional cube whose square facets have been triangulated and whose geometry is defined as the image of a reference cube by a trilinear mapping denoted φ on Figure 7.1. Our hexahedra, in the physical space, are not strictly cubes, moreover they are only valid as long as the mapping from the reference cube is injective. Validity is difficult to evaluate because the Jacobian determinant is not constant over the hexahedron e.g. (Chandrupatla et al. 2011).

A set of tetrahedra is a valid triangulation of the hexahedron {12345678} (Figure 7.1) if it is a valid combinatorial 3-manifold, has 8 vertices and if its boundary contains the 8 vertices, 12 edges {12}, {14}, {15}, {23}, {26}, {34}, {36}, {48}, {56}, {58}, {67}, {78} and 12 triangular facets that can be paired into 6 quadrilaterals {1234}, {1265}, {1485}, {2376}, {3487}, {5678}.

Bounds on the number of tetrahedra that can be build with 8 vertices to mesh a 3-ball may determined from the Euler characteristic. Each triangulation is a 3-ball with Euler characteristic $\chi = 1$, where $\chi = v - e + f - t$. Then $v - e_{int} - e_{bd} + f_{int} + f_{bd} - t = 1$. Since there are 4 triangular faces per tetrahedron and 2 tetrahedra per interior triangular face, we have $4t = 2f_{int} + f_{bd}$. Since the number of boundary edges e_{bd}, and boundary triangular facets f_{bd} are fixed the number of tetrahedra t in a hexahedron (without internal vertices) depends only on the number of interior edges e_{int}. Moreover, there are at most $\binom{n}{2} - e_{bd}$ edges in a cell with n vertices we then have trivial bounds on the number of tetrahedra in the triangulation of the cells. $t = 5 + e_{int}$ and $5 \leq t_{hex} \leq 15$. See also (Edelsbrunner, Preparata, et al. 1990) for additional combinatorial results.

Nb tetrahedra	5	6	7	8	9	10	11	12	13	14	15	Total
Combinatorial	1	5	5	7	13	20	35	30	28	19	11	174
Geometrical	1	5	5	7	13	20	35	30	28	19	8	171

Table 7.1: Numbers of combinatorial triangulations of the hexahedron with 5 to 15 tetrahedra up to isomorphism.

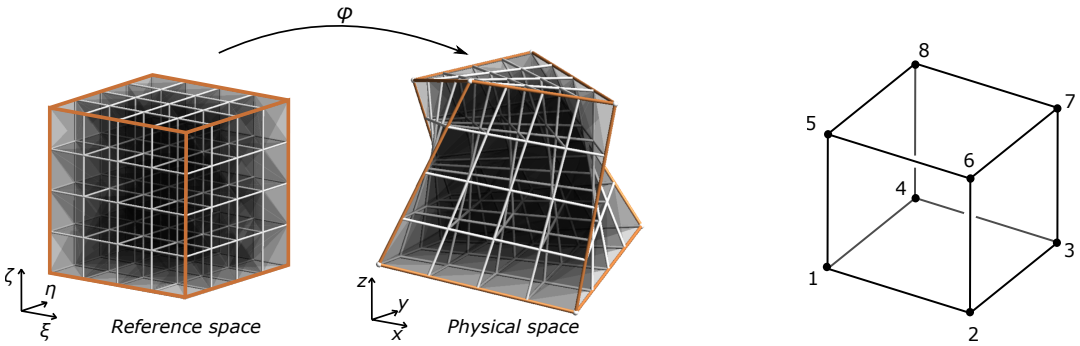


Figure 7.1: The geometry and connectivity of the hexahedra we study. A Finite Element hexahedron is the image of a reference cube by a trilinear mapping, its connectivity is the one of the cube.

Contributions In this chapter, I present research work conducted at the Université Catholique de Louvain in collaboration with Kilian Verhetsel (PhD student I co-advised) and Jean-François Remacle who was not a bit afraid of the above theoretical question because: Why don't you try brute-force and see what happens?

To enumerate the triangulations of the hexahedron we need discrete mathematics where enumeration is part of the job and where powerful concepts unheard of in the engineering world: matroids, chirotopes (no mythology, real math) permit to actually solve the problem. We solve the purely combinatorial problem of the hexahedron triangulations and show that there are 174 triangulations of the hexahedron up to isomorphism that have between 5 and 15 tetrahedra (Table 7.1). We prove this result by reviewing all the combinatorial triangulations of the 3-ball with eight vertices. We then solve the geometrical problem. The result that exactly 171 hexahedron triangulations have a geometrical realization in $\mathbb{R}^3$. To solve the realization problem we implemented the method proposed in (Firsching 2017). We exhibit triangulations with 14 and 15 tetrahedra of the hexahedron, contradicting the belief that there might only be subdivisions with up to 13 tetrahedra (Meshkat et al. 2000; Sokolov, Ray, Untereiner, and Levy 2016). This is an important theoretical result that links hexahedral meshes to tetrahedral meshes.

- Pellerin, J., K. Verhetsel, and J.-F. Remacle (2018). "There are 174 subdivisions of the hexahedron into tetrahedra". en. In: *ACM Transactions on Graphics* 37.6, pp. 1–9

The code and all result triangulations are available both in the supplementary materials of the paper and on the site of the project at `https://www.hextreme.eu`. This research was supported by the European Research Council (project HEXTREME, ERC-2015-AdG-694020).

7.1 Triangulations of the Cube

The triangulations of the hexahedron, limited to the specific geometrical configuration of the cube, were studied in discrete geometry (De Loera et al. 2010) and in mesh generation methods that combine tetrahedra into hexahedra (Meshkat et al. 2000; Botella et al. 2016; Sokolov, Ray, Untereiner, and Levy 2016).

The cube Following (De Loera et al. 2010), let us first look at the different types of tetrahedra that can be built from the eight vertices of a cube $I^3 = [0, 1]^3$. Suppose that one facet of the tetrahedron is contained in one of the six cube facets, there are four possible vertices to built a non-degenerate tetrahedra that represent three cases due to symmetry (Figure 7.2):

- *Corner* tetrahedra have three facets on the cube faces.
- *Staircase* tetrahedra have two facets on the cube faces, they have two facets on the hexahedron boundary.
- *Slanted* tetrahedra have a unique facet on the cube faces.

The last type of tetrahedra are *Core* tetrahedra that have no facet on the cube faces. For the 3-cube, this list is complete and irredundant (De Loera et al. 2010).

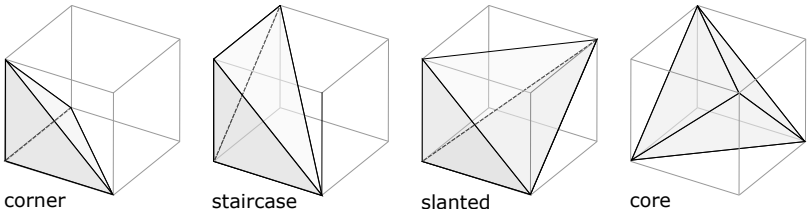


Figure 7.2: The four types of tetrahedra in a cube $I^3 = [0, 1]^3$.

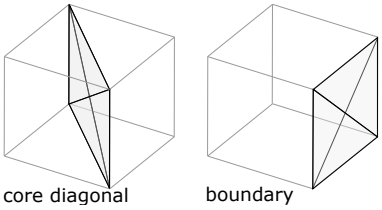


Figure 7.3: Two types of tetrahedra in a finite precision cube.

From this classification of the tetrahedra in a cube triangulation follows the classification of triangulations of the 3-cube. The 3-cube has exactly 74 triangulations that are classified in 6 classes (Figure 7.4) (see (De Loera et al. 2010) and references therein).

- Every triangulation of the 3-cube contains either a regular tetrahedron (i.e. a tetrahedron whose 6 edges are of equal lengths) or a diameter, i.e. an interior edge joining two opposite vertices.
- There are two triangulations of the first type, symmetric to one another. The triangulations of the second type are completely classified, modulo symmetries by their dual complexes (Figure 7.4).

The dual complex of a triangulation is a graph in which there is a vertex for each tetrahedron and one edge if corresponding tetrahedra share a triangle (Figure 7.4). By construction, a 2-cell of the dual complex corresponds to an interior edge of the triangulation.

There are at least two ways to prove that these six triangulations are indeed the six possible types of triangulations of the cube. Discrete geometers rely on the determination of the cell containing the barycenter of the cube and on symmetry considerations about the vertex arrangement around that cell (De Loera et al. 2010). (Meshkat et al. 2000) enumerate the possible dual complexes.

The 3-cube in finite precision Floating point numbers have a finite precision and represent the cube $I^3 = [0, 1]^3$ up to some tolerance. Moreover, sets of points that are exactly cospherical and coplanar are often specifically processed because these non-general positions are problematic for many geometrical algorithms (Edelsbrunner and Mücke 1990). Therefore, to combine elements of tetrahedral meshes, (Botella et al. 2016) and (Sokolov, Ray, Untereiner, and Levy 2016) consider subdivisions of the cube into two prisms linked by a *sliver*, i.e. a very flat tetrahedra connecting four points that are coplanar up to a given tolerance. Note that (Sokolov, Ray, Untereiner, and Levy 2016) aim at enumerating the triangulations of more general configurations, but the geometrical argument used in the proof limits its validity to the cube represented in finite precision.

In a finite precision 3-cube, two new types of non-degenerate tetrahedra appear (Figure 7.3):

- *Core diagonal* tetrahedra have no facet on the hexahedron boundary. They subdivide the hexahedron into two prisms, creating four additional types of triangulations into seven tetrahedra (Figure 7.5).
- *Boundary* tetrahedra connect the four vertices of a facet of the cube. We exclude these tetrahedra from the triangulations we consider because they are not strictly inside the hexahedron. Indeed, the hexahedron facets are bilinear patches strictly inside boundary tetrahedra.

Together with the previous four types of tetrahedra in the cube they constitute the only possible configurations for a tetrahedron in a hexahedron triangulation.

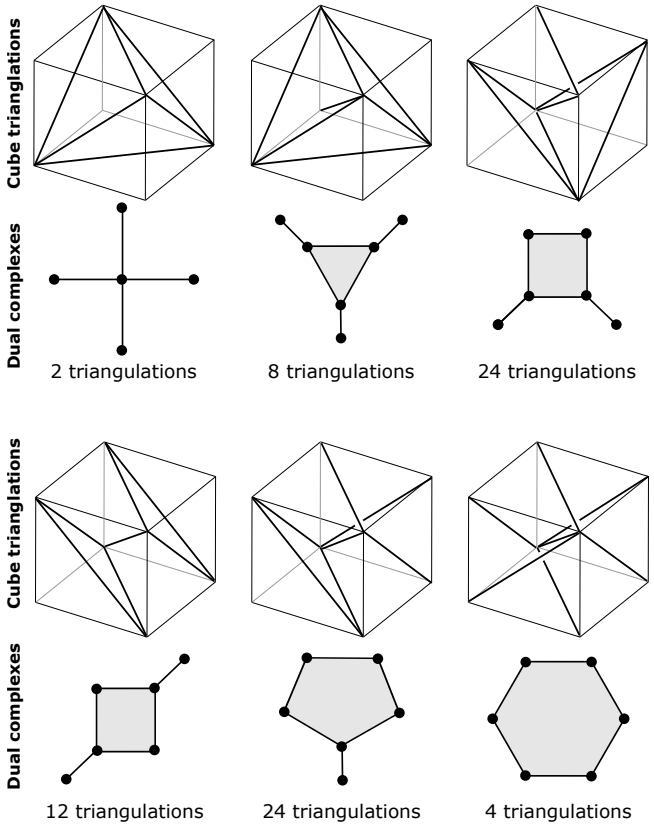


Figure 7.4: The six triangulations of the 3-cube $I^3 = [0,1]^3$ up to isomorphism and their dual complexes. Each class corresponds to 2 to 24 equivalent triangulations.

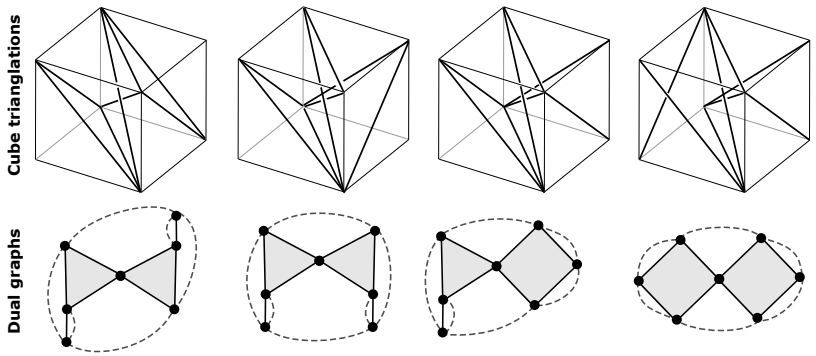


Figure 7.5: The four triangulations of the finite precision cube with seven tetrahedra. Dashed edges connect pairs of tetrahedra incident to the same hexahedron facet.

7.2 Combinatorial Triangulations of the Hexahedron

The geometrical problem of the triangulation of the hexahedron is extremely difficult and we first consider the purely combinatorial side of the question. A set of tetrahedra is a valid combinatorial triangulation of the hexahedron {12345678} (Figure 7.1) if it is a valid combinatorial 3-manifold with 8 vertices and if its boundary contains the 8 vertices, 12 edges {12}, {14}, {15}, {23}, {26}, {34}, {36}, {48}, {56}, {58}, {67}, {78} and 12 triangular facets that can be paired into 6 quadrilaterals {1234}, {1265}, {1485}, {2376}, {3487}, {5678}.

Theorem 7.2.1. *The hexahedron* {12345678} *has 174 combinatorial triangulations up to isomorphism that do not contain any boundary tetrahedra.*

To demonstrate Theorem 7.2.1 we first use a result about the triangulations of the 3-sphere with 9 vertices to compute all triangulations of the 3-ball with 8 vertices. At the next step, we select triangulations whose boundary is the boundary of a triangulated hexahedron. Finally the valid combinations are classified into isomorphism classes.

7.2.1 Triangulations of the 3-Ball

The sphere, or 2-sphere, is the 2-dimensional surface boundary of a 3-dimensional ball. The 3-sphere generalizes this, and is defined as the 3-dimensional boundary of a 4-dimensional ball. There is a direct link between the triangulations of the 2-sphere (3-sphere) and the triangulations of the 2-ball (3-sphere). Indeed, a triangulation of the 2-sphere can be constructed from the triangulation of a 2-ball by building a cone as illustrated on Figure 7.6. The cone is built by adding point v over a polygon and connecting it to the five boundary edges. The inverse transformation, the removal of one point v of the sphere triangulation as well as all triangles incident to v, permits to obtain the triangulation of a ball. (The interested reader is referred to (Ziegler 1995) for more details on this construction).

We enumerate all triangulations of the hexahedron, i.e. 3-dimensional ball with 8 vertices. They can be obtained from the triangulations of the 3-sphere with 9 vertices. The 1296 triangulations of the 3-sphere have been enumerated by (Altshuler and Steinberg 1973; Altshuler and Steinberg 1974; Altshuler and Steinberg 1976; Altshuler, Bokowski, et al. 1980) and are available online (Lutz et al. 2018). Nine triangulations of the 3-ball with eight vertices can be built from each of the 1296 triangulations by removing one of the vertices $v_i, i = 1, \ldots, 9$ and its link, i.e. all tetrahedra incident to v_i.

Could the naive brute force work? Let t be the number of tetrahedra, f the number of facets, e the number of edges and $v = 8$ the number of vertices. The hexahedron boundary is fixed to have 8 vertices, 12 facets and 18 edges. Each tetrahedron has 4 facets, each interior facet is incident to 2 tetrahedra, then $4t = 2f + 12$. Injecting this in the Euler-Poincaré characteristic of the triangulation $\chi = v - e + f - t = 1$, we have $t = e - 13$. There are at least 18 edges and at most $\binom{8}{2} = 28$ edges in the triangulation of a hexahedron, so $18 \leq e \leq 28$, then $5 \leq t \leq 15$. It is then possible to write a brute-force naive algorithm to enumerate the isomorphism-free triangulations of the hexahedron {12345678}. The exponential complexity of such an algorithm make it very expensive, but it ran in several hours in parallel thanks to the implementation of aggressive optimizations.

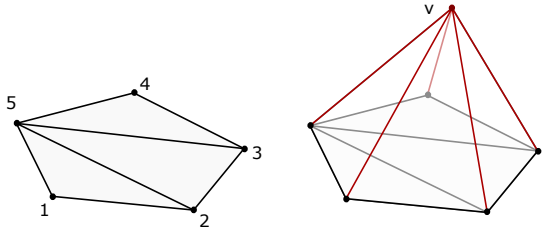


Figure 7.6: Building a cone over a triangulation of a 2-dimensional pentagon into 3 triangles gives a triangulation of the 2-sphere into 8 triangles.

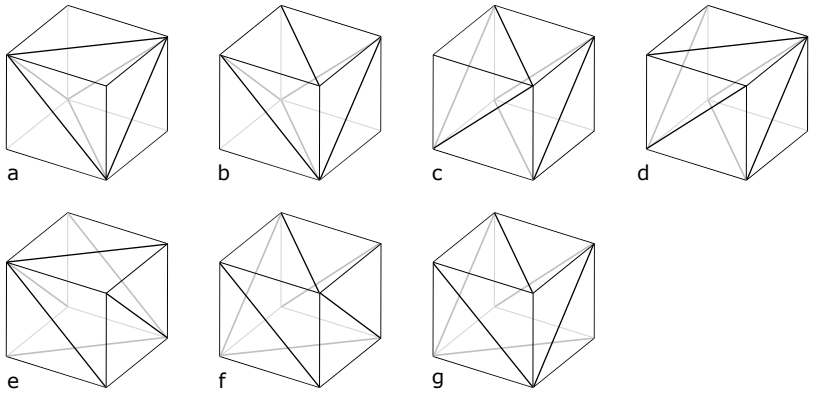


Figure 7.7: The 7 triangulations of a hexahedron boundary up to isomorphism.

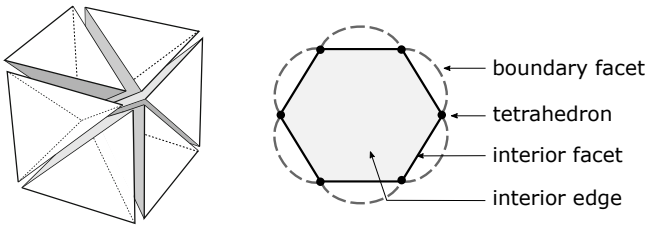


Figure 7.8: Definition of the decomposition graph that is used to represent hexahedron triangulations.

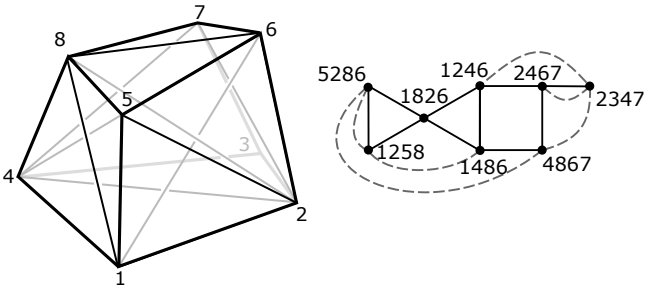


Figure 7.9: The correspondence between a hexahedron triangulations and its decomposition graph for triangulation 8_E.

7.2.2 Triangulations of the Hexahedron

The next step is to test if the obtained 3-ball triangulations are valid hexahedron triangulations. They are if their boundary matches the triangulated boundary of a hexahedron. The triangulation of the boundary of a hexahedron has 8 vertices and 18 edges. Among these, 12 are fixed and there are 2 possibilities to place the remaining 6 diagonals of the quadrilateral facets. We have then $2^6 = 64$ possible triangulations. These triangulations can be classified into 7 equivalence classes, i.e. there are 7 triangulations of the hexahedron boundary up to isomorphism (Figure 7.7).

7.2.3 Comparison of Hexahedron Triangulations

The final step is to compare the obtained triangulations of the hexahedron and determine those that are the same up to isomorphism. We use a graph formalism proposed in (Meshkat et al. 2000) to represent and compare the hexahedron triangulations.

The *decomposition graph* (Figure 7.8) of the triangulation of a hexahedron is an edge-colored graph where there is one vertex per tetrahedron, one black (plain) edge between vertices if the corresponding tetrahedra are adjacent, and one grey (dashed) edge between vertices if the corresponding tetrahedra have triangle faces on the same hexahedron facet. By construction, each vertex is of degree 4 and

simple chordless cycles of plain edges correspond to interior edges of the triangulation.

There is a one-to-one correspondence between triangulations without boundary tetrahedra and decomposition graphs (Figure 7.9). Suppose that we fix the vertex labels of the triangulation such that {12345678} is a hexahedron. Then 12 of its edges are fixed ({12}, {14}, {15}, {23}, {26}, {34}, {36}, {48}, {56}, {58}, {67}, {78}). There is then one choice to subdivide each of the 6 facets into two triangles. Once the 6 boundary diagonal edges are chosen, the remaining degrees of freedom to triangulate the hexahedron are controlled by the interior facets. The plain edges of the decomposition graph represent the interior facets, while the 6 dashed edges represent the diagonal edges.

Two combinatorial triangulations are isomorphic, if and only if their decomposition graphs are isomorphic, i.e. are the same with respect to the relabeling of their vertices. More formally, an isomorphism of graphs G and H is a bijection between the vertex sets of G and H, $f : V(G) \to V(H)$, such that any two vertices u and v of G are adjacent if and only if $f(u)$ and $f(v)$ are adjacent in H.

The decomposition graphs of the triangulations of the hexahedron obtained in the previous section are therefore classified into isomorphism classes to obtain our main combinatorial result. Our reference implementation, attached in the supplementary materials, uses the nauty library (McKay et al. 2014). This concludes the demonstration of Theorem 7.2.1.

7.3 Geometrical Realization

We now tackle the geometrical side of the problem of the triangulations of the hexahedron in $\mathbb{R}^3$. There is indeed no guarantee that a valid geometrical triangulation exists for each of the 174 combinatorial triangulations.

More formally, we are solving a geometrical realization problem for each combinatorial triangulation T, i.e. we are searching positions of the 8 vertices, $\mathcal{A}$ in $\mathbb{R}^3$, such that the geometrical triangulation defined by T is valid. A triangulation is valid if (1) the union of the tetrahedra is the convex hull of the points $\bigcup_{\sigma \in T} conv\,\sigma = conv\,\mathcal{A}$ and (2) there is no unwanted intersection between tetrahedra. $conv\,\sigma \cap conv\,\sigma' = conv(\sigma \cap \sigma')\ \forall \sigma, \sigma' \in T$.

Theorem 7.3.1. *171 of the 174 combinatorial triangulations of the hexahedron have a geometrical realization in* $\mathbb{R}^3$.

We provide the realizations along with the code at in the supplemental material of (Pellerin, Verhetsel, et al. 2018). The 3 triangulations that do not have a realization have 15 tetrahedra and are illustrated on Figure 7.12. This is a particular case of the general problem of finding an embedding of an abstract triangulations of dimension m in $\mathbb{R}^d$. For our case, $m = 3$ and $d = 3$, the problem complexity is itself an open-problem (Matoušek et al. 2009). For some values it is solvable in polynomial time (e.g. $m = 2, d = 2$), for others it is a NP-hard problem (e.g. $m = 2, d = 4$), for others it is algorithmically undecidable (e.g. $m = 4, d = 5$).

Our proof follows the strategy described in (Pfeifle et al. 2003; Schewe 2010; Firsching 2017) which consists in (1) solving the underlying combinatorial problem of the point configurations that can realize a given triangulation T, before (2) finding point coordinates in $\mathbb{R}^3$ corresponding to one configuration. The main steps are described in Algorithm 6.

7.3.1 Useful Definitions

To encode the combinatorial properties of the point sets realizing a given combinatorial triangulation, we use oriented matroids. This is an important topic of discrete mathematics (Björner 1999), here we restrict definitions to the studied problem.

Let E be a finite set, $r \in \mathbb{N}$ with $r \leq |E|$, and a mapping $\chi : E^r \mapsto \{-1, +1\}$. $\mathcal{M} = (E, \chi)$ is a *uniform oriented matroid* of rank r if the following two conditions are satisfied:

1. The mapping χ is alternating, i.e. for all permutations π of $\{1, \ldots, r\}$

$$\chi(e_{\pi(1)}, e_{\pi(2)}, \ldots, e_{\pi(r)}) = sgn(\pi)\chi(e_1, e_2, \ldots, e_r)$$

where $sgn(\pi)$ is the permutation sign, i.e. 1 for even permutations and -1 for odd permutations.

Algorithm 6 Geometrical realization

Input: T combinatorial triangulation
Output: $\mathcal{A}$ a realization

```
1:  clauses = encodeSAT(T)                          ▷ Section 7.3.2
2:  S = solveSAT(clauses)                           ▷ MiniSAT solver (Sorensson et al. 2005)
3:  if S == ∅ then
4:      return ∅                                    ▷ Not realizable
5:  for all χ ∈ S do
6:      inequalities = setDeterminantInequalities(χ)   ▷ Section 7.3.3
7:      A = solveSystem(inequalities)               ▷ SCIP solver (Gleixner et al. 2017)
8:      if A ≠ ∅ then return A                      ▷ Realization is found
9:  undecidedRealizability = false
10: for all χ ∈ S do                                ▷ Check non-realizability
11:     if !FindFinalPolynomial(χ) then
12:         undecidedRealizability = true
```

2. For all $\sigma \in \binom{E}{r-2}$ and all subsets $\{e_1, e_2, e_3, e_4\} \subseteq E \setminus \sigma$, we have that:

$$\begin{aligned}\{-1,+1\} \subseteq \{&\chi(\sigma,e_1,e_2)\chi(\sigma,e_3,e_4),\\ &-\chi(\sigma,e_1,e_3)\chi(\sigma,e_2,e_4),\\ &\chi(\sigma,e_1,e_4)\chi(\sigma,e_2,e_3)\}\end{aligned}$$

The mapping χ is called the *chirotope* of the oriented matroid. Given a set of points $\mathcal{A}$ in $\mathbb{R}^d$, we can build a chirotope for a matroid of rank $r = d+1$ can be constructed by considering the determinant of the points in homogeneous coordinates using $v \mapsto \binom{1}{v}$.

Set for any $(\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_{d+1}}) \in \mathcal{A}$

$$\chi(1,2,\ldots,d+1) = \operatorname{sign}\det\begin{pmatrix}1 & 1 & \ldots & 1\\ \mathbf{a_1} & \mathbf{a_2} & \ldots & \mathbf{a_{d+1}}\end{pmatrix}$$

The chirotope is a sign function which gives the orientation of all possible subsets of $r = d+1$ of points in $\mathcal{A}$. The chirotope of a point set $\mathcal{A} \in \mathbb{R}^2$ (respectively $\mathbb{R}^3$) is the orientation of all the triangles (respectively tetrahedra) that can be built from $\mathcal{A}$ (Figure 7.10). The chirotope encodes the point set structure, and, as we will see in Section 7.3.2, all necessary information to check if $\mathcal{A}$ is a realization of the combinatorial triangulation T.

Note that we only consider point sets $\mathcal{A}$ in general position (no 4-coplanar of 5-copspherical points) restricting ourselves to uniform oriented matroids, χ cannot take the value zero.

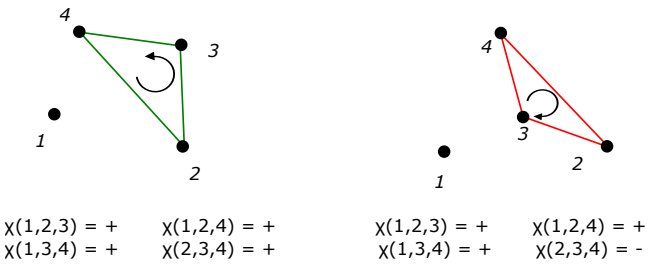


Figure 7.10: The two possible 4-point configurations in $\mathbb{R}^2$ have different chirotopes. The chirotope encodes the orientation of all 4 triangles, only the one whose orientation changes is drawn.

7.3.2 Combinatorial Formulation of Geometrical Realizations

We first review the five constraints that the oriented matroids of point sets realizing a combinatorial triangulation T must enforce, before seeing how the oriented matroids are determined.

Constraints The point set is in general position therefore the corresponding oriented matroid should:

1. be uniform, $\chi \neq 0$ (no 4-coplanar or 5-cospherical points)
2. be acyclic, a property shared by the oriented matroids of all point sets.
3. be valid, an oriented matroid mapping. All $r-2$ subsets should satisfy condition (2) of the definition in Section 7.3.1.

The tetrahedra of the geometrical triangulation are valid, therefore the oriented matroid should:

4. define valid simplices, $\forall \sigma \in T, \chi(\sigma_1, \ldots, \sigma_{d+1}) = +1$. We assume that they are consistently oriented and are all positive in T.

The geometrical triangulation is valid, i.e. all tetrahedra of T intersect properly, i.e. for all pairs of tetrahedra σ and σ' of T we have $conv\,\sigma \cap conv\,\sigma' = conv(\sigma \cap \sigma')$. The intersection property can also be translated to constraints on the oriented matroids, let us see how.

If a point set $\mathcal{A}$ is in general position, any subset of $d+1$ points is affinely independent while any subset of $d+2$ points is affinely dependent, i.e. there exists $\lambda \in \mathbb{R}^{d+2}$ such that:

$$\sum_{i=1}^{d+2} \lambda_i a_i = \mathbf{0} \text{ and } \sum_{i=1}^{d+2} \lambda_i = 0$$

We can then build two subsets of $\mathcal{A}$, $A^+ = \{a_i \mid \lambda_i > 0\}$ and $A^- = \{p_i \mid \lambda_i < 0\}$, whose convex hulls intersect at a unique point (Radon's theorem). If two different simplices σ and σ' of T intersect, then there is a subset of $d+2$ points of $vert(\sigma) \cup vert(\sigma')$ that are affinely dependent and corresponds to a pair (X^+, X^-). In oriented matroid jargon, this pair (X^+, X^-) is called a circuit of $\mathcal{A}$, a fundamental notion related to another definition of oriented matroids (Björner 1999). All $\binom{n}{d+2}$ subsets of $(d+2)$ points of $\mathcal{A}$ define a circuit (X^+, X^-) that can be computed from

$$X^+ = \{i \mid (-1)^i \chi(1, 2, \ldots, i-1, i+1, \ldots, d+2) = +1\}$$
$$X^- = \{i \mid (-1)^i \chi(1, 2, \ldots, i-1, i+1, \ldots, d+2) = -1\}$$

This give us the final constraint on oriented matroids of points sets realizing a combinatorial triangulation T, they should:

5. not admit any circuit (X^+, X^-) such that $X^+ \subseteq \sigma$ and $X^- \subseteq \sigma'$ where σ and σ' are two tetrahedra of T.

Solving The five constraints described above fix partially the values taken by χ for all $\binom{n}{d+1}$ subsets of point labels. To completely determine the admissible oriented matroids for a triangulation T, we follow(Schewe 2010), and solve an instance of a boolean satisfiability problem SAT that is satisfiable, if and only if, an admissible oriented matroid exists for a triangulation T. For each combination $c = (c_1, c_2, \ldots, c_{d+1})$ (where the elements are ordered by increasing index), we define a Boolean variable $[c]$. Setting $[c]$ to **true** (resp. **false**) corresponds to setting $\chi(c)$ to 1 (resp. -1). We express the remaining axioms and conditions using those $\binom{n}{d+1}$ Boolean variables, logical negations ($\neg[c]$), logical conjunctions ($[c_1] \wedge [c_2]$ is **true** iff both $[c_1]$ and $[c_2]$ are **true**) and logical disjunctions ($[c_1] \vee [c_2]$ is **true** iff at least one of $[c_1]$ and $[c_2]$ is **true**).

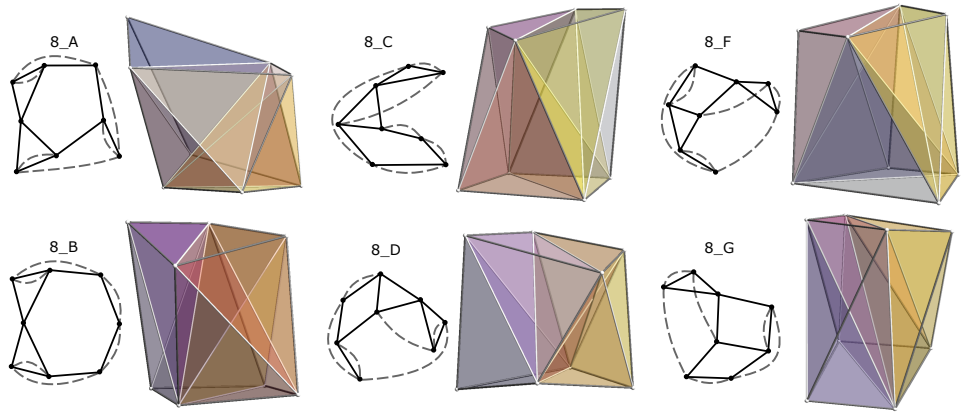


Figure 7.11: Six triangulations of the hexahedon into 8 tetrahedra and their geometrical realizations. Colors correspond to tetrahedron indices. Hexahedron edges are grey, while tetrahedron edges are white. The seventh triangulation with 8 tetrahedra, 8_E, is presented on Figure 7.9.

The chirotope must satisfy :

1. the oriented matroid definition, that gives a constraints on all $r = 4$ subsets of elements. We use the formulation as a logical formula given by (Schewe 2010).
2. positivity. For any $(d+1)$-simplex $t \in T$, $[t]$ must be **true**.
3. the intersection property. For any signed circuit (X^+, X^-) with $X = \{p_1, p_2, \ldots, p_{d+2}\}$ that the oriented matroid cannot accept, we write

$$\left(\bigvee_{p_i \in I} [p_1, p_2, \ldots, p_{i-1}, p_{i+1}, \ldots, p_{d+2}]\right) \vee \left(\bigvee_{p_j \in J} \neg[p_1, p_2, \ldots, p_{j-1}, p_{j+1}, \ldots, p_{d+2}]\right)$$

where

$$I = \{p_i \in X^+ \mid (-1)^i = -1\} \cup \{p_i \in X^- \mid (-1)^i = +1\}$$
$$J = \{p_i \in X^+ \mid (-1)^i = +1\} \cup \{p_i \in X^- \mid (-1)^i = -1\}$$

From the values of the $\binom{n}{d+1}$ Boolean variables, we can determine the chirotope of an oriented matroid admissible for the abstract triangulation T. We must now find a point set, the chirotope of which matches.

We used the MiniSAT (Sorensson et al. 2005) solver to find all admissible chirotopes for 174 combinatorial triangulations of the hexahedron. Our implementation is provided in the supplemental material.

Result Three combinatorial triangulations 15_A, 15_B, 15_J do not admit an oriented matroid that enforce the five constraints and are therefore not realizable.

7.3.3 Geometric Realizations

Once the oriented matroids fulfilling all the constraints related to a given combinatorial triangulation are computed, the final step is to find real coordinates for the 8 vertices realizing one of these oriented matroids. We follow the approach of (Firsching 2017) who proposes to solve the system of $\binom{n}{d+1}$ polynomial inequalities of the form:

$$\chi(1, 2, \ldots, d+1) \det \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \mathbf{p}_1 & \mathbf{p}_2 & \cdots & \mathbf{p}_{d+1} \end{pmatrix} > 0$$

where the possible values of $\chi(1, 2, \ldots, d+1)$ have been computed in Section 7.3.2.

Solving In order to find feasible solutions we used SCIP, a solver that relies on branch and bound techniques (Gleixner et al. 2017). To encode the strict inequalities numerically, we rewrite the strictly positive constraint as superior or equal to an ϵ value (we used $\epsilon = 10^{-4}$). When a solution exists, SCIP quickly converges and we obtain a realization of the combinatorial triangulation T.

Result All the 171 triangulations that admit an oriented matroid have a realization. All realizations are provided along our implementation in supplemental material. This completes the demonstration of Theorem 7.3.1. Figure 7.11 gives realizations for the configurations with 8 tetrahedra.

7.3.4 Convex Geometric Realization

We furthermore studied the realization problem with an additional constraint on the 8 point positions to ensure that they are in convex position, i.e. all vertices are on their convex hull (none is in its interior). This is a reasonable condition for hexahedra to be valid for finite element computations. The convexity condition can be directly taken into account when testing admissible chirotopes by excluding all configurations in which a vertex is inside a tetrahedron defined by four other vertices.

Result Over the 171 realizable triangulations of the hexahedron, 13 do not have a realization such that their vertices are in convex position. 12_U, 12_Y, 13_T, 13_W, 14_A, 14_N, 14_O, 14_P, 14_R, 15_F, 15_H, 15_I do not admit a uniform oriented matroid and are therefore not realizable. Triangulation 15_G admits an oriented matroid, but SCIP does not terminate when searching for a realization (Figure 7.12). To prove that there is none, a different approach is necessary. A property of non-realizable matroids is that they admit a final polynomial, which can be found, if bi-quadratic, using the practical method of (Bokowski et al. 1990). It consists in proving the infeasibility of a linear program that encodes the second property of oriented matroid definition. 15_G oriented matroid has a final polynomial and is therefore not realizable with points in convex position.

7.4 Discussion

We demonstrated that there are 174 combinatorial triangulations of the hexahedron up to isomorphism and that 171 of them have a geometric realization.

This result is consistent with the previous results on the triangulations of the 3-cube presented in (De Loera et al. 2010). However, Theorem 4 in (Sokolov, Ray, Untereiner, and Levy 2016), that states that there are only 10 triangulations of the hexahedron with 5, 6, or 7 tetrahedra, is incorrect. We described 161 additional geometrical triangulations of the hexahedron. The smallest triangulations for which no valid hexahedron may exist have 12 tetrahedra. The largest triangulations for which a valid hexahedron exists have 15 tetrahedra, by valid we mean a a strictly positive Jacobian (Figure 7.12). The existence of these triangulations contradicts the belief that there may only be subdivisions of the hexahedron with up to 13 tetrahedra. The practical consequence is that the methods solely based on the 10 triangulations of the cube to identify groups of tetrahedra that can be merged into a hexahedron do not compute all the possible hexahedra as claimed (Meshkat et al. 2000; Yamakawa et al. 2003; Botella et al. 2016; Sokolov, Ray, Untereiner, and Levy 2016). Adapting these methods to take into account dozens more patterns is theoretically possible, but consequences on performances would be catastrophic.

Occurrences in real meshes The theoretical problem of the triangulation of the hexahedron is an important step for a better understanding of the methods combining elements of a tetrahedral mesh to build hexahedra. Indeed, in practice, when combining the element of a tetrahedral mesh, many of the additional patterns we identified occur Figure 7.13.

To identify combinations of tetrahedra into hexahedra in a input tetrahedral mesh, we used the algorithm presented in Chapter 6. This algorithm provably computes all the possible combinations of eight vertices whose connectivity matches the one of a hexahedron. It further guarantees that all the identified hexahedra are valid for finite element computations using the exact test of (Johnen, J.-F. Remacle, et al. 2013). We compute all valid hexahedra for the input tetrahedral meshes provided

Model	Nb V	5	6	7	8	9	10	11	12	13	14	15	Total
Cube	127	1	5	2	1	0	0	0	0	0	0	0	9
Fusee	11,975	1	5	5	7	13	9	4	0	0	0	0	44
CShaft	23,245	1	5	5	7	13	17	6	0	0	0	0	54
Fusee_1	71,947	1	5	5	7	7	1	0	0	0	0	0	26
Caliper	130,572	1	5	5	7	7	1	0	0	0	0	0	26
CShaft2	140,985	1	5	5	7	8	0	1	0	0	0	0	27
Fusee_2	161,888	1	5	5	7	7	1	0	0	0	0	0	26
FT47_b	221,780	1	5	5	7	8	2	0	0	0	0	0	28
FT47	370,401	1	5	5	7	7	2	0	0	0	0	0	27
Fusee_3	501,021	1	5	5	7	8	4	0	0	0	0	0	30
Los1	583,561	1	5	5	7	7	2	0	0	0	0	0	27
Knuckle	3,058,481	1	5	5	7	8	2	0	0	0	0	0	28

Table 7.2: Number of triangulations patterns per number of tetrahedra counted in the available input data of (Pellerin, Johnen, Verhetsel, et al. 2018).

#points	5	6	7	8	9	10	11	12	13	14	15	Total
3,000	1	5	2	7	13	16	4	0	0	0	0	51
10,000	1	5	5	7	13	19	10	2	0	0	0	62
20,000	1	5	5	7	13	19	15	2	0	0	0	67
100,000	1	5	5	7	13	20	24	5	0	0	0	80
500,000	1	5	5	7	13	20	28	12	1	0	0	92
1,000,000	1	5	5	7	13	20	30	14	0	0	0	95
2,000,000	1	5	5	7	13	20	30	15	4	1	0	101
5,000,000	1	5	5	7	13	20	30	16	4	1	0	102
10,000,000	1	5	5	7	13	20	31	16	6	1	0	105

Table 7.3: Number of triangulations patterns per number of tetrahedra counted in the Delaunay triangulations of random point sets.

with that paper (Table 7.2) as well as for the Delaunay triangulations of random point sets (Table 7.3). The decomposition graphs of all identified hexahedra are compared to the 171 possible triangulations and classified.

There is a clear link between the quality of the point set for hexahedral mesh generation and the number of patterns of combinations of tetrahedra into hexahedra. In random point sets, all existing realizable patterns up to 9 tetrahedra are found with as few as 3,000 randomly distributed vertices. In point sets generated for hexahedral meshing with the placement strategy described in (Baudouin et al. 2014), the number of patterns is smaller and we note that the number of combination patterns is relatively independent of both the model and the number of vertices. Two models stand out and the high number of patterns is correlated to a lower number of hexahedra of worse quality in the final mesh. We further observed that occurrences of triangulations with more than 11 tetrahedra are rare. Since only the meshes are based on Delaunay triangulations, it is also probable that some combinatorial triangulations only have realizations that are not Delaunay.

Challenges Using our results, we would be able to decide on the existence of a triangulation of eight vertices whose boundary matches a hexahedron. We could also produce all existing triangulations of 8 given points. Since the principle of our demonstration is general, it could be extended to enumerate the triangulations of any polyhedron with quadrilateral and triangular faces and could be used to characterize non-triangulable polyhedra, a major difficulty in many geometric and combinatorial problems about which little is known theoretically (Rambau 2003). We expect the theoretical advance of this chapter to be an important step toward a more complete study of the geometrical properties of hexahedral cells and will help develop robust indirect hexahedral mesh generation algorithms.

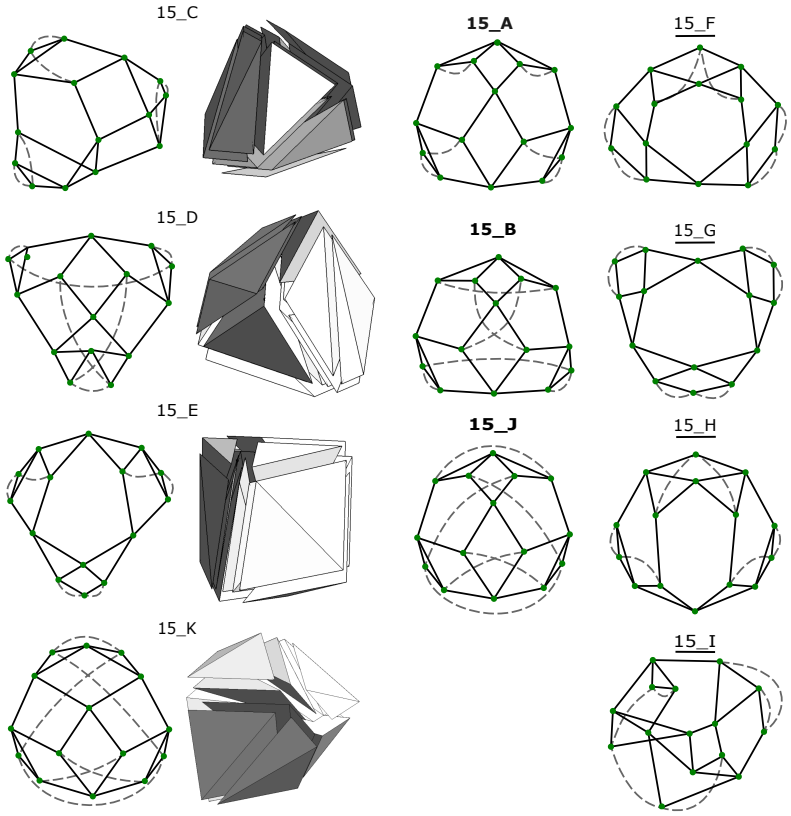


Figure 7.12: The 11 combinatorial triangulations of the hexahedron into 15 tetrahedra. Four are realizable with points in convex position. The hexahedra are valid, their Jacobian is strictly positive (left). Three are not realizable (center). Four are realizable but with points in non-convex position (right).

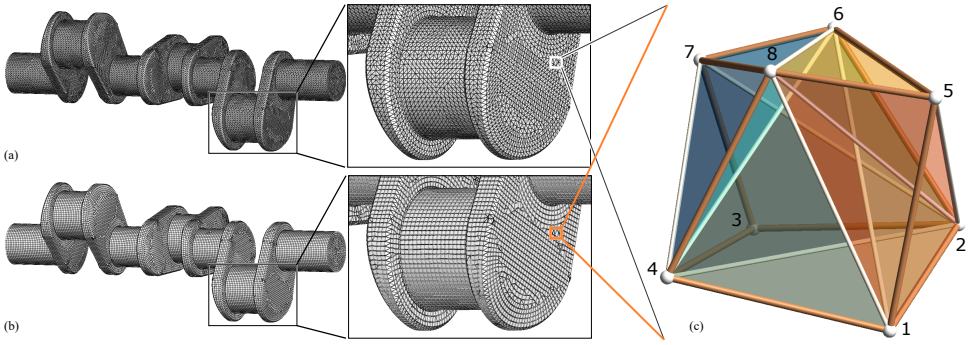


Figure 7.13: The link between tetrahedral meshes (a) and hexahedral meshes (b) depends on hexahedron triangulations. (c) One of the 174 triangulations of hexahedron {12345678} has 8 tetrahedra {1258}, {5286}, {1826}, {1246}, 1486}, {2467}, {4867}, {2347} .

Chapter 8

Theoretical Hexahedral Meshing

Motivations One fundamental problem in hexahedral mesh generation is that there is no algorithm to generate a hexahedral mesh conformal to a given quadrilateral boundary. The existence of hexahedral meshes for all even quadrangulations of the topological sphere has been proven by (Mitchell 1996), yet seemingly innocuous quadrangulations such as the 16-quadrangle pyramid (Schneider's pyramid) or the 8-quadrangle tetragonal trapezohedron, also called octogonal spindle, (Figure 8.1) are notorious failure cases of general purpose meshing methods. (Eppstein 1999) shows how the interior of a quadrangulated sphere can be meshed with a linear number of hexahedra; this construction was later generalized to all domains which admit hexahedral meshes (Erickson 2014). Both methods reduce the problem to meshing a few quadrangulated spheres, but neither provide explicit hex meshes for these cases. One method only is able to generate hexahedral meshes for these templates (Carbonera et al. 2010). The huge drawback is that it requires 5396 n hexahedra to construct a non-degenerate hexahedral mesh of a ball bounded by n quadrangles, and is not directly applicable to arbitrary domains.

How do we generate combinatorial hexahedral meshes? In this chapter we are interested in the combinatorial constrained hexahedral meshing problem. The algorithms described are dedicated to building combinatorial hexahedral meshes and the geometrical problem is ignored for the most part of this work. Given a quadrangulation of the topological sphere Q, the objective is to determines a set of combinatorial cubes H such that:

1. the intersection of any two hexahedra $h_1, h_2 \in H$ is a combinatorial face shared by h_1 and h_2 (*i.e.* the empty set, a vertex, an edge, or a quadrangle);
2. all quadrangular faces are shared by at most two hexahedra;
3. the set of boundary facets (adjacent to exactly one hexahedron) exactly matches Q.

This is an extremely challenging problem, even when the subsequent problem of finding a geometrical embedding is ignored, and for which no practical method exists.

Contributions This chapter presents results of my collaboration with Kilian Verhetsel, a brilliant PhD student I co-advised during my post-doc in Belgium. By applying constrained programming techniques we successfully solved problems that we only dreamt solving, such as the Schneider's pyramid hexahedral meshsing challenge. This project was funded by European Research Council project HEXTREME, ERC-2015-AdG-69402 awarded to Jean-François Remacle.

In this work with Kilian we first focused on the the hexahedral meshing of Schneider's pyramid. The goal was to determine the smallest hexahedral mesh of the pyramid. Following the idea explored to enumerate triangulations Chapter 7, we considered all possible groups of 8 vertices, all hexahedra, that can be built without creating an invalid mesh. The algorithm of this vertex-based enumeration of hexahedral meshes permit to compute lower bounds on the number of hexahedra and vertices required to mesh a given boundary Section 8.2. For small boundaries, that same algorithm can compute the smallest hexahedral mesh and we propose to locally remesh groups of adjacent hexahedra Section 8.3.

We compute a 44 hexahedra mesh of Schneider's pyramid by remeshing the 88 hex mesh of (Yamakawa et al. 2010).

The major contribution is the first practical algorithm to build constrained combinatorial hexahedral meshes (Section 8.4). The key idea of the method is to exploit the equivalence between quad flips in the boundary and the insertion of hexahedra glued to this boundary. The tree of all sequences of flipping operations is explored, searching for a path that transforms the input quadrangulation Q to the boundary of a cube. For larger meshes the search may stop when a sequence transforming Q to a set of pre-computed hexahedral meshes is found Section 8.5. Combined with an efficient backtracking search, it allows small shellable hexahedral meshes to be found for all even quadrangulations with up to 20 quadrangles. The 54, 943 such quadrangulations were meshed using no more than 72 hexahedra. This algorithm is able to compute small hexahedral meshes of quadrangulations for which the previously known best solutions could only be built by hand or contained thousands of hexahedra.

We finally prove that an arbitrary ball bounded by n quadrangles can be meshed using only 78 n hexahedra Section 8.6.1. This very significantly lowers the previous upper bound of 5396 n. The construction of Erickson is made fully explicit by computing hexahedral meshes for its two quadrangulated templates.

- Verhetsel, K., J. Pellerin, and J.-F. Remacle (2018). "A 44-element mesh of Schneiders' pyramid: Bounding the difficulty of hex-meshing problems". In: *Proceedings of the 27th International Meshing Roundtable*. Springer **Best paper award**
- Verhetsel, K., J. Pellerin, and J.-F. Remacle (2019a). "A 44-element mesh of Schneiders' pyramid: Bounding the difficulty of hex-meshing problems". en. In: *Computer-Aided Design* 116, p. 102735
- Verhetsel, K., J. Pellerin, and J.-F. Remacle (2019b). "Finding hexahedrizations for small quadrangulations of the sphere". en. In: *ACM Transactions on Graphics* 38.4, pp. 1–13

Reference implementation of all algorithms presented in this chapter, as well as all result meshes are available at `https://www.hextreme.eu`.

8.1 State of the Art

Hexahedral mesh generation is a thriving field of research, with a variety of proposed methods. These include multi-block decomposition methods using frame-field parametrizations (Kowalski et al. 2014; Nieser et al. 2011; H. Liu et al. 2018; Lyon et al. 2016), hex-dominant meshing methods (Yamakawa et al. 2003; Gao et al. 2017; Baudouin et al. 2014; Sokolov, Ray, Untereiner, and Lévy 2017; Pellerin, Johnen, Verhetsel, et al. 2018), octree-based methods (Maréchal 2009; Ito et al. 2009; Zhang et al. 2012; Qian et al. 2010), and polycube-based methods (Gregson et al. 2011; Han et al. 2011; Yu et al. 2014; Fang et al. 2016). Most methods do not address the problem of generating meshes with a given boundary quadrangular mesh. The rest of this section focuses on methods tackling the constrained problem.

8.1.1 Existence Proofs

Existence theorem for ball inputs (Thurston 1993) and (Mitchell 1996) independently showed that a ball bounded by a quadrangulated sphere can be meshed with hexahedra, if and only if, the number of quadrangles on the boundary, n, is even. The proof is based on the dual cell complex of quadrangular and hexahedral meshes. The dual complex of a quadrangular mesh is obtained by placing a vertex at the center of each quadrangle and adding edges between the vertices corresponding to adjacent quadrangles. Grouping edges traversing opposite edges of a same quadrangle, the dual complex is interpreted as an arrangement of curves (Figure 8.2). Similarly, the dual of a hexahedral mesh can be interpreted as an arrangement of surfaces (Murdoch et al. 1997). In Mitchell proof, an arrangement of surfaces bounded by the dual arrangement of curves of the input quadrangulation is first constructed. For curves with an even number of self-intersections (including curves with no self-intersections), a disk is constructed inside the domain and a regular homotopy between a circle

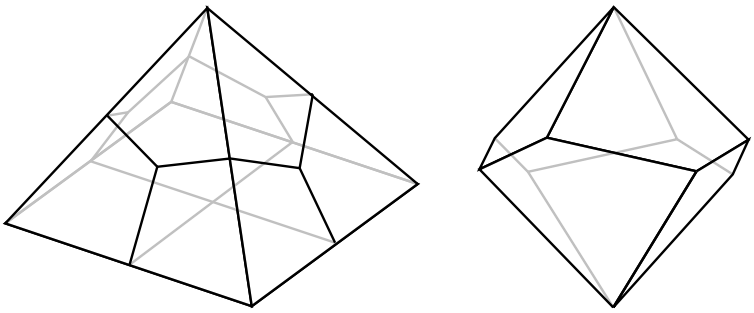


Figure 8.1: Two famous examples of boundary quadrangulation for which no hexahedral mesh was known in 1996. Left is Schneiders'pyramid (Schneiders 1996) and right the octogonal Spindle

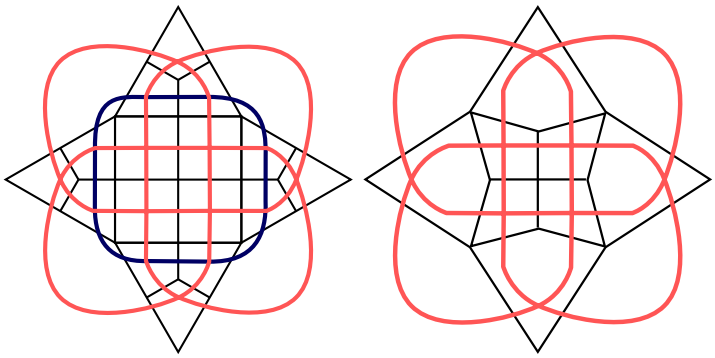


Figure 8.2: The dual of Schneider's pyramid (left) and the octogonal spindle (right) are are arrangement of curves. Adding the simple dark blue curve (i.e. one layer of quadrangles) to the self-intersecting red line permit to obtain Schneiders pyramid from the octogonal spindle.

and the curve can be used to create a manifold bounded by that curve. Curves with an odd number of self-intersections are paired up arbitrarily. For each pair, a manifold bounded by the two curves is constructed by computing a regular homotopy between the two of them. This arrangement is not in general the dual of a hexahedral mesh, so the next step of the construction is to add new surfaces completely inside the ball until all connectivity requirements of a hexahedral mesh are met.

Linear-complexity meshing The construction of Mitchell can necessitate up to $\Omega(n^2)$ hexahedra where n is the number quadrangles. Eppstein showed this was the case and proposed a different construction which guarantees the use of $O(n)$ hexahedra (Eppstein 1999). His algorithm first subdivides each quadrangle into two triangles, so that a tetrahedral mesh of the interior can be computed. After subdividing each tetrahedron into four hexahedra, a hexahedral mesh is obtained. However, its boundary does not match the initial input quadrangulation. This is solved by inserting *buffer cells*: for each quadrangle, add a cube, and glue one of its face to the original quadrangle; then, subdivide the opposite face into six quadrangles. The six new quadrangles are matched with those obtained from subdividing the original quadrangles during the previous step. The four remaining sides of the buffer cells are carefully subdivided into either two or three quadrangles, so that each buffer cell is bounded by an even number of quadrangles. Mitchell proof can then be invoked to show that each buffer cell can be subdivided into a finite number of hexahedra.

Generalization to other inputs Generalizing the previous results, (Erickson 2014) gives necessary and sufficient conditions for the existence of a hexahedral mesh of a domain Ω bounded by a quadrangulation Q. The requirement is that every null-homologous subgraph of the input quadrangulation (*i.e.* every subgraph which bounds an embedded surface of Ω) contain an even number of edges. The construction of the hexahedral mesh is similar to the one proposed by Eppstein, and also starts by computing a tetrahedral mesh of the domain, subdividing it into a hexahedral mesh, and inserting buffer cells to get a complete mesh with the correct boundary. The last step is to subdivide the buffer cells of two different types (Figure 8.3) into hexahedra, which is again shown to be possible Mitchell

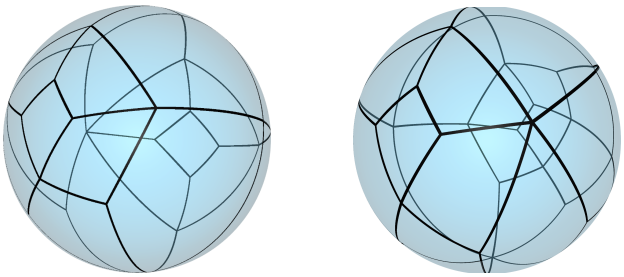


Figure 8.3: If hexahedrizations of these two quadrangulated spheres exists, then it is possible to construct hexahedrizations for all other quadrangulated surface (Erickson 2014). But this was left to the reader to build by hand as an exercise.

1996. Like Eppstein, Erickson does not give an explicit construction of the hexahedral meshes of the buffer cells that are the base of its proofs.

A constructive method (Carbonera et al. 2010) give the first completely explicit construction. Their algorithm first adds hexahedra inside the domain, guaranteeing that the dual arrangement of the boundary of the remaining region contain no self-intersecting curve. Buffer cells are then inserted to transition to a mesh where each quadrangle has been subdivided into four quadrangles. The rest of the domain is then filled using pyramids. A complete hexahedral mesh is obtained after subdividing the pyramids into hexahedra. Given a topological ball bounded by n quadrangles, their construction produces a mesh of 76 n hexahedra. This mesh is degenerate: it contains quadrangles sharing multiple edges and hexahedra sharing multiple faces. A combinatorially valid mesh can be obtained by further refining the mesh (Mitchell and Tautges 1995). This method, however, requires as many as 5396 n hexahedra to build a hexhahedral mesh bounded by quadrangulation of size n.

8.1.2 Constrained Hexahedral Meshing in Practice

The methods presented in the previous section are impractical, they create far more hexahedra than necessary. In practice, less general methods have been used to obtain smaller meshes.

Whisker Weaving Whisker Weaving has been proposed by (Tautges, Blacker, et al. 1996). The idea is to use a topological advancing front to construct the dual of a hexahedral mesh. The algorithm initially assumes that the final mesh will contain one dual surface for each dual curve of the input quadrangulation. Hexahedra are created inside the domain by creating intersections between three of these sheets, until the entire domain is filled. To choose between the multiple possible operations, heuristics based on geometric information such as the dihedral angle of faces are used. These heuristics are often not enough to completely fill the domain.

Dual cycle elimination (Müller–Hannemann 1999) proposed a method based on *dual cycle eliminations*. At each step of the algorithm, one of the curves of the dual mesh is removed, matching this elimination as the insertion of a layer of hexahedra. The new boundary after removing this cycle bounds the part of the input domain which has not been meshed yet. This process is repeated until the boundary matches that of a single cube. This method succeeds for certain classes of input quadrangulations, but fails for the common cases where the dual contains self-intersecting curves (e.g. Figure 8.2).

Searching for hexahedral meshes Some specific cases have particularly attracted the attention of the research community: Schneiders' pyramid (Schneiders 1995) and the octogonal spindle (Figure 8.1). In 2002, (Yamakawa et al. 2002) introduced the HEXHOOP template family and constructed by hand a hexahedral mesh of Schneiders' pyramid with 118 hexahedra. Later on, they improved their solution, building an 88-element mesh (Yamakawa et al. 2010). These *ad-hoc* constructions have long been the smallest known solutions for these two polyhedra.

8.2 Vertex-based Enumeration of Hexahedral Meshes

Following the work on triangulations Chapter 7, the idea we explored first was to consider all possible groups of 8 vertices that can be built without creating an invalid mesh. This means using computer search to exhaustively explore the space of all possible hexahedral meshes matching an input quadrangulation up to a given number of vertices. The search space is usually too large for a solution to be found except in small configurations. This allows to define cavity operations to locally modify the mesh and reduce the number of hexahedra Section 8.3.

The algorithm is a backtracking algorithm that lists all possible combinatorial hexahedral meshes with a prescribed boundary. It is used to prove that there is no hexahedral mesh of Schneiders' pyramid with strictly fewer than 12 interior vertices. Using the same approach, we also prove that there is no hexahedral mesh of the octagonal spindle with strictly fewer than 21 interior vertices.

8.2.1 Backtrack Search Algorithm

Given ∂H, a combinatorial quad-mesh of a sphere or a handlebody, H_{max} a maximum number of hexahedra, and V_{max} a maximum number of vertices, our algorithm lists all combinatorial hexahedral meshes H such that:

- the boundary of H is ∂H,
- the number of hexahedra $|H|$ is at most $H_{\max}$,
- the total number of vertices in H is at most $V_{\max}$.

This problem has similarities with problems commonly encountered in *constraint programming*: (i) efficiently filtering a large set of potential solutions and (ii) managing solutions having multiple equivalent representations. Our implementation adopts concepts and strategies from this field. For a more general study of these problems, we refer the reader to (Rossi et al. 2006).

In Algorithm 7, the hexahedra are built one at a time by choosing a sequence of 8 vertices. At each step, all possible candidates for one of the 8 vertices are considered and the algorithm branches for each possibility. Each branch corresponds to the addition of a vertex to the current hexahedron. When a complete solution is determined, or when the search fails (no available candidates to complete a hexahedron), the algorithm backtracks to the previous choice. This process is repeated until all possibilities have been explored. This algorithm corresponds to the exploration of a search tree (Figure 8.4) where each branching node represents the choice of a vertex, and the leaves represent either solutions or failure points where the algorithm backtracks. The search tree has an exponential size in the maximum number of hexahedra in a solution. This high complexity is managed by pruning branches that cannot contain a solution and by using efficient implementations of all performed operations.

8.2.2 Search Space Reduction Strategies

In this section, we describe the key points of our implementation of Algorithm 7, all of which aim at reducing the search space explored by the algorithm:

- the order in which the hexahedra are constructed is crucial — we use an advancing-front strategy and start the construction of hexahedra from the boundary;
- an efficient filtering algorithm that eliminates candidate vertices that would create incompatible combinatorial hexahedra in the solution;
- a method to manage the high number of symmetries of this problem;
- the order in which the current hexahedron vertices are selected;
- a strategy to use topological invariants in order to filter out branches that do not contain any solutions.

Algorithm 7 Recursive enumeration of the hexahedral meshes of the interior of ∂H

Input: ∂H, the boundary
S, a partial solution
$C = (C_1, \ldots, C_8)$, the sets of candidate vertices for the current hexahedron
if boundary of S == ∂H **then**
    Print solution S
else if $|S| = H_{\max}$ **then**
    Backtrack
else
    $C \leftarrow$ Filter-Candidates$(\partial H, S, C)$
    if $|C_1| = \cdots = |C_8| = 1$ **then**
        $S' \leftarrow S \cup \{(v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)\}$
        Search $(\partial H, S',$ Initialize-Candidates$(S'))$
    else if $\min_{i \in \{1,\ldots,8\}} |C_i| = 0$ **then**
        Backtrack
    else
        $i \leftarrow$ Pick-Hex-Vertex(C)
        for all $v \in C_i$ **do**
            $C' \leftarrow C$
            $C'_i \leftarrow \{v\}$
            Search$(\partial H, S, C')$

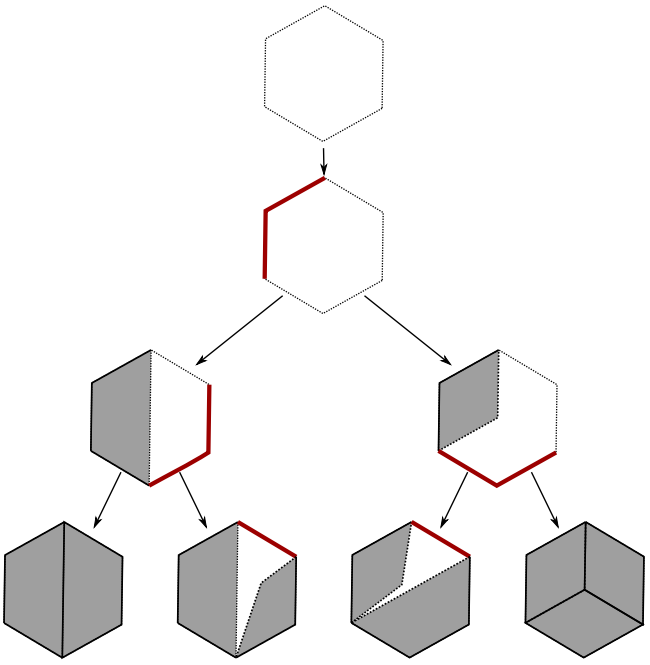


Figure 8.4: Searching all quadrilateral meshes of a polygon with up to one interior point. The search tree leaves are either valid solutions, or correspond to detected failure points where Algorithm 7 backtracks.

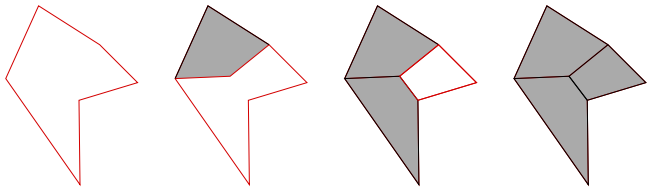


Figure 8.5: Each new element must share a face with the front of boundary faces (red).

Algorithm 8 INITIALIZE-CANDIDATES(S): Compute the sets candidate vertices

Input: S, a set of hexahedra
return $C = (C_1, \ldots, C_8)$, the sets of candidate vertices for the next hexahedron.
(v_1, v_2, v_3, v_4) is a quadrangle that is an output mesh facet.
for each $i \in \{1, \ldots, 4\}$ **do**
    $C_i \leftarrow \{v_i\}$
for each $i \in \{1, \ldots, 4\}$ **do**
    $C_{4+i} \leftarrow \{0, 1, \ldots, V_{\max} - 1\} \setminus \{v_1, v_2, v_3, v_4\} \setminus$ INTERIOR-DIAGONALS(v_i)
    for each $j \in \{1, \ldots, 4\}$ **do**
        if $i \neq j$ **then**
            $C_{4+i} \leftarrow C_{4+i} \setminus$ INTERIOR-DIAGONALS(v_j) $\setminus$ EDGES(v_j)
        if $i = j + 2 \mod 4$ **then**
            $C_{4+i} \leftarrow C_{4+i} \setminus$ QUAD-DIAGONALS(v_j)
return $(C_1, \ldots, C_8)$

Advancing Front Construction While the hexahedra of a combinatorial mesh can be arbitrarily reordered, constructing them in a specific order makes the algorithm significantly faster. We use a classical advancing front generation strategy and require the hexahedron under construction to share a face with a front of quadrangles. There are then only four vertices needed to complete a hexahedron. The quadrangle front is constituted of the interior facets that are in only one hexahedron, or of boundary facets that are in no hexahedra. At the root of the search tree, it is set to be the boundary ∂H. An interior facet is added to the front after its first appearance in the mesh. The facet is removed from the front when it is added to the partial solution. When the front becomes empty, the boundary of the solution matches the input (Figure 8.5).

Filtering Candidate Vertices For each of the eight vertices of the hexahedron under construction, we store a set of candidate vertices that could be part of the solution. Some of these vertices would make the current hexahedron incompatible with some already existing hexahedra. Therefore when initiating the construction of a hexahedron, or when adding a vertex to a hexahedron, vertices that cannot be added without creating incompatibilities between the current hexahedron and the already built hexahedra are filtered out. The following rules are used to eliminate candidates:

1. the sets of edges, quadrangle diagonals, and interior diagonal of hexahedra are disjoint;
2. no two hexahedra may share an interior diagonal;
3. if one facet diagonal matches an existing quadrangle diagonal, so must the second one;
4. all eight vertices must be different.

To enforce these rules, our implementation tracks three sets of vertices for each vertex v: the sets of vertices u such that (u, v) is an edge, the diagonal of a quadrangle, or the diagonal of a hexahedron. These sets are updated whenever a new hexahedron is created. Because the execution time of the search algorithm blows up as the number of vertices increases, the number of vertices each set contains is always small, making them good candidates for being represented as bit-sets.

Symmetry breaking Combinatorial meshes are characterized by their large number of symmetries, a major challenge when operating on combinatorial hexahedral meshes. Indeed, a combinatorial hexahedral mesh has many equivalent representations:

1. n interior vertices can be labelled in $n!$ different ways (boundary vertices labels are in input)
2. m hexahedra can be constructed therefore labelled in $m!$ different ways;
3. each hexahedron, written as an ordered sequence of 8 vertices, has $1680 = 8!/24$ equivalent vertex labelling.

The advancing front strategy defines the order in which the solution hexahedra are constructed (symmetry 2). This also uniquely determines the order of vertices in a hexahedron (symmetry 3). To prevent the relabelling of interior vertices (symmetry 1), we add *value precedence* constraints to our problem (Law et al. 2004). A solution H found by the algorithm can be written as an array of $8|H|$ integers, writing down the vertices of each hexahedron in the order in which they were constructed by the algorithm. In an array, x *precedes* y when the first occurrence of x is before the first occurrence of y. Enforcing a total precedence order on interior vertices, we guarantee that only one of their permutations is a solution.

Optimization of hexahedron construction The efficiency of the backtracking search depends on the size of the search tree needed to explore all possibilities. A cheap approach to reduce the number of nodes in the search tree is to choose the vertex with the smallest set of candidate vertices when deciding which vertex to branch on (Beck et al. 2004). This does not affect the correctness of the algorithm, as long as a vertex with more than one candidate is selected.

A similar heuristic is used to select the quadrangle on top of which the next hexahedron should be built: for each quadrangle on top of which a hexahedron may be built, the sets of candidates for the four remaining vertices are computed. The quadrangle which minimizes the product of the sizes of these four sets is then selected. This allows the search to detect failures early when it is not possible to build any hexahedron on top of an existing quadrangle.

Using topological properties during the search The meshes that we are searching for, in addition to being valid combinatorial meshes, need to be meshes of the interior of the input surface and some some topological requirements must be met by any solution. These requirements are used not only to filter out branches of the search tree that do not contain any valid meshes, but also to reject combinatorial meshes of different 3-manifolds with the same boundary. Only topological invariants that can efficiently be computed are considered during the search, since no efficient algorithms to recognize 3-manifolds are known, even for the 3-sphere (Schleimer 2011).

At any point, the partial mesh is maintained to be *oriented*: every quadrangle that appears in two hexahedra must have an opposite orientation in each of them. Whenever one of the faces of the hexahedron under construction is identified with an existing quadrangle because they share a diagonal, the other two vertices are selected so that the two quadrangles have opposite orientation.

Meshes of a topological ball, or any hexahedral mesh that could be used as a subset of a mesh of the ball, have a bipartite graph (Eppstein 1999). Let A and B be the two parts of the partition. The sets of candidate vertices are reduced so that edges between two elements of A or of B are never created. The partition is computed initially based on the input quadrangulation, and updated every time a hexahedron is added to the partial mesh.

The last topological property that we use is related to *homology groups*, computed over $\mathbb{Z}_2$. A detailed introduction to homology computations can be found in (Hatcher 2002). We define a k-chain as a set of elements of dimension k — a 1-chain is a set of edges, a 2-chain a set of quadrangles, and a 3-chain a set of hexahedra. The boundary of a k-chain C is the $(k-1)$-chain ∂C whose elements are the faces contained in an odd number of elements of C. A k-chain whose boundary is empty is referred to as a cycle. A k-cycle which is the boundary of a $(k+1)$-chain is called *null-homologous* (Figure 8.6).

For the domains that we consider, all 2-cycles are null-homologous. Since hexahedra have an even number of faces, any set of hexahedra is bounded by an even number of quadrangles. Therefore, if a 2-cycle containing an odd number of quadrangles is ever created, the search can backtrack, as it will never be possible to create a set of hexahedra bounded by this cycle. We therefore compute a basis for

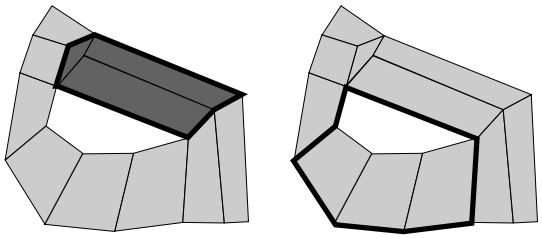


Figure 8.6: (left) A null-homologous 1-cycle in a quad-mesh, bounding 3 quadrangles; (right) a cycle which is not null-homologous, because it surrounds a hole within the mesh.

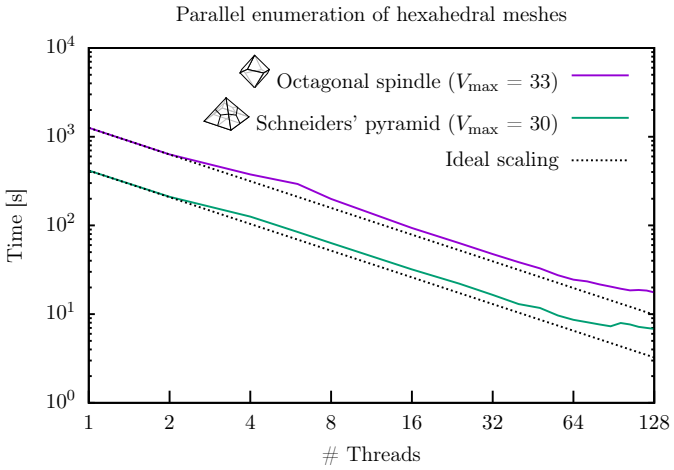


Figure 8.7: Time to explore a search tree in parallel on a machine with two AMD EPYC 7551 CPUs (32 cores each, 2 threads per core). Using 64 threads, the speed-up is of 48 for Schneiders' pyramid and 52 for the octagonal spindle.

the space of 2-cycles using Gaussian elimination, and verify that every element of the basis contains an even number of quadrangles.

Parallel search implemntation The exploration a search tree can be parallelized in a natural way by exploring different subtrees in parallel, making the algorithm much faster on parallel architectures (Figure 8.7). We use an approach similar to the embarrassingly parallel search of (Régin et al. 2013). The main challenge to overcome is that some subtrees are multiple orders of magnitude larger than other ones without any possibility to determine it ahead of time.

We solve this issue by attributing many subtrees to each worker thread, so that all threads must on average perform the same amount of work (we used 4096 subtrees per thread). At the start of the search, the tree is explored in a breadth-first manner until a layer with enough subproblems is reached. The nodes of this layer are then explored in parallel by independent worker threads using Algorithm 7.

8.2.3 Lower Bounds for Schneiders Pyramid and the Octagonal Spindle

Using Algorithm 7, we computed lower bounds for the number of vertices and hexahedra required to mesh Schneiders' pyramid and the octagonal spindle (Figure 8.1). The algorithm is run multiple times, and we increment either V_{max} or H_{max} between each run. At each step, we verify that no solution was found by the algorithm. The time required to compute these bounds increases exponentially as the bounds become tighter (Figure 8.8).

Theorem 8.2.1. *Any hexahedral mesh of Schneiders' pyramid has at least 18 interior vertices and 17 hexahedra.*

Theorem 8.2.2. *Any hexahedral mesh of the octagonal spindle has at least 29 interior vertices and 21 hexahedra.*

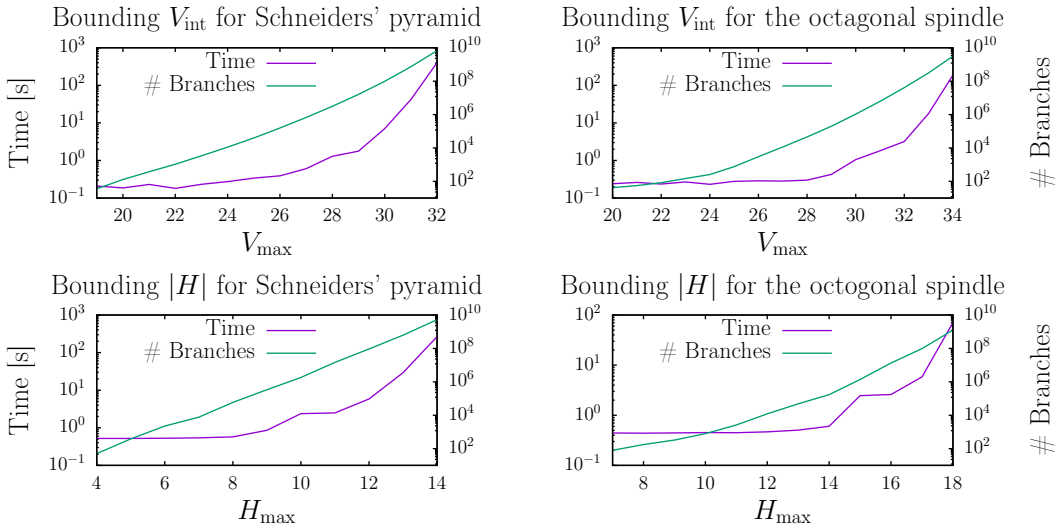


Figure 8.8: The time to prove lower bounds for the number of interior vertices V_{int} and the number of hexahedra H required to mesh a polyhedron increases exponentially. This is due to the exponential size of the search tree explored by the algorithm.

Algorithm 9 Cavity selection algorithm

Input: H, the mesh; n, the size of the cavity
Output: A cavity C of n elements

1: $h \leftarrow$ a random element of H
2: $C \leftarrow \{h\}$
3: **while** $|C| \neq n$ **do**
4: $\quad h \leftarrow$ a random element of $H \setminus C$ sharing a facet with a hexahedron in C
5: $\quad C \leftarrow C \cup \{h\}$
return C

8.3 Cavity Based Hexahedral Remeshing Algorithm

The algorithm described in Section 8.2 can be used to find the smallest hexahedral mesh with a given boundary. In this section, we use this algorithm to compute upper bounds for the number of hexahedra required to mesh Schneiders' pyramid. From the 88-element solution of (Yamakawa et al. 2010), we locally simplify the mesh. By simplification we mean decreasing the number of hexahedra (Figure 8.9). The realized operations may be viewed as a generalized form of cube flips (Bern et al. 2002) that substitute a set of hexahedra by another set without changing their boundary. However, instead of having a predefined set of operations, as e.g (Tautges and Knoop 2003), the algorithm proposed in this section is able to operate on any group of hexahedra and automatically determines them at execution time.

Globally minimizing the number of hexahedra in the mesh is a computationally demanding task. Our algorithm therefore selects a small subset of the mesh, or *cavity*, and focuses on modifying the connectivity of the mesh only within this cavity. Our hexahedral mesh simplification algorithm is based on Algorithm 7. From a geometric hexahedral mesh it outputs a geometric hexahedral mesh whose boundary is strictly identical and which has fewer elements.

The mesh simplification procedure has three main steps:

1. the selection of a cavity, the group of hexahedra to simplify, C;
2. finding the smallest hexahedral mesh C_{min} compatible with the cavity boundary ∂C and replacing the cavity with this smaller mesh;
3. untangling the hexahedra to determine valid coordinates for the mesh vertices.

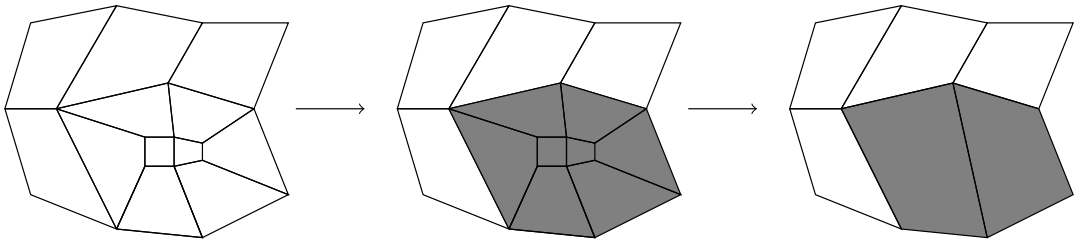


Figure 8.9: The number of elements in a mesh can be reduced by operating locally on a cavity.

Cavity selection The cavity selection algorithm is a greedy algorithm that starts from a random element of the input hexahedral mesh (Algorithm 9). When the target size, in terms of number of hexahedra, is reached, this process stops. The choice of a target cavity size is a trade-off between the cost of finding the hexahedral meshes of the cavity and the likelihood that the mesh can be simplified by remeshing the cavity. Cavities with many hexahedra are more likely to accept smaller meshes, but the cost of finding the smallest hexahedral mesh $C_{\min}$ increases exponentially with the number of hexahedra in the cavity. In practice, we start by considering relatively small cavities containing up to 10 hexahedra, and increase this limit when no improvement is possible. We require the cavity to contain at least 4 interior vertices. Indeed, when there are no interior vertices (e.g. with a stack of hexahedra), it is not possible to remove any hexahedra. As the number of interior vertices increases, so does the likelihood that the cavity can be simplified.

Cavity remeshing To find a smaller mesh of the boundary of a cavity C, we first solve the combinatorial problem, i.e. we find the smallest combinatorial hexahedral mesh of ∂C, and then solve the geometric problem of finding valid coordinates for the modified mesh vertices.

The combinatorial problem of finding the smallest mesh of ∂C is a direct application of Algorithm 7 which enumerates all combinatorial meshes of a given surface. The maximum number of hexahedra H_{max} of the solution is set to a smaller value than $|C|$. Changing the parity of a hexahedral mesh is known to be a difficult operation (Schwartz et al. 2004), so we set H_{max} to $|C| - 2$. We also set the limit to the number of interior vertices V_{max} to one less than the number of interior vertices in C to accelerate the search.

There is a subtle but important difference between meshing a cavity in an existing mesh and meshing a stand-alone polyhedron: the hexahedra inside the cavity must be compatible with the other elements of the input mesh. An example where new elements from a cavity are not compatible with elements adjacent to the cavity is given in Figure 8.10. A 3-element cavity is replaced by 2 quadrangles, but one of these two quadrangles shares two edges with an existing element, which is an invalid configuration. To guarantee that the algorithm does not break the mesh validity, the data structures used to filter out inadequate vertex candidates (Section 8.2.2) are modified to take into account the hexahedra that are not part of the cavity.

Geometrical untangling The previous step of the algorithm found a new connectivity for the mesh. The simplified mesh obtained by using this result is not valid in general because the interiors of hexahedra may intersect (Figure 8.11). To obtain a valid geometric mesh we use the untangling algorithm described in (Toulorge et al. 2013). The vertices are iteratively moved until all hexahedra in the mesh are valid. If the untangling fails, connectivity changes are undone. The validity of the final mesh is evaluated with the method proposed by (Johnen, Weill, et al. 2017).

8.3.1 Remeshing Schneiders' Pyramid and the Octagonal Spindle

A 66-element mesh of Schneiders' pyramid In this section, we build a hexahedral mesh of Schneiders' pyramid and the smallest known mesh of the octagonal spindle (Figure 8.13) using the remeshing algorithm. Starting from the 88-element solution of (Yamakawa et al. 2010), the number of hexahedra is reduced to 66 by locally simplifying groups of hexahedra. Table 8.1 shows the sizes of the different cavities simplified by our algorithm. It takes a few minutes for our algorithm to reduce the number of hexahedra in the mesh from 88 down to 66 in our final mesh. Figure 8.12 shows the changes to the

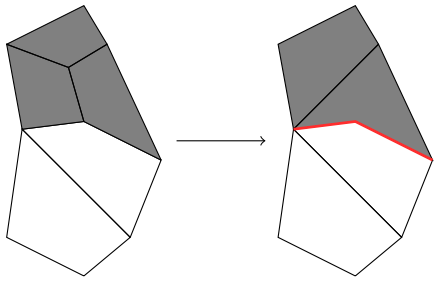


Figure 8.10: Replacing the cavity with a valid mesh sharing the same boundary still produces an invalid mesh by creating two quadrangles sharing two edges.

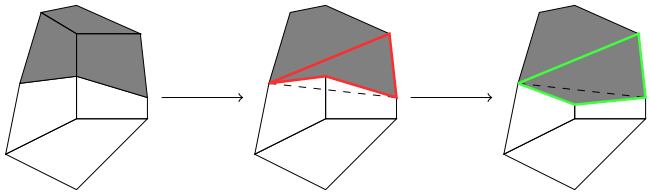


Figure 8.11: A valid change to the connectivity of the mesh can create a geometrically invalid mesh, fixed by moving the vertices.

Initial mesh		Initial cavity			Remeshed cavity		New mesh	
#hex	#vert.	#hex	#vert.	#bd. facets	#hex	#vert.	#hex	#vert.
88	105	8	23	18	6	21	86	103
86	103	8	23	18	6	21	84	101
84	101	8	23	18	6	21	82	99
82	99	14	33	24	8	27	76	93
76	93	6	16	10	2	12	72	89
72	89	18	40	30	12	32	66	81

Table 8.1: Cavity remeshing operations performed by our hex-mesh simplification algorithm on Yamakawa's 88-element mesh of Schneiders' pyramid (Yamakawa et al. 2010).

connectivity of the mesh performed in two different iterations of the algorithm. The vertices had to be moved to obtain a valid mesh, but the combinatorial boundary remains the same. For example, for the second pair of cavities in the figure, the same 30 facets can be seen before and after the remeshing operation: there is a central facet, surrounded by a ring of five quadrangles, followed by three rings of six quadrangles, followed by one more ring of five quadrangles surrounding a single face. We also determined a combinatorial mesh with 64 hexahedra and 59 interior vertices, on which the untangling failed.

Shortly after the first version of our conference paper was published on ArXiv an alternative algorithm was proposed by (Xiang et al. 2018). Starting from a single cube, the algorithm of considers all possible ways to add one hexahedron while maintaining a mesh which is both valid and combinatorially equivalent to a topological ball. The process is stopped when the boundary of the mesh matches the target quadrangulation. The search space considered is smaller, and they however build a mesh of the pyramid with 36 hexahedra (Figure 8.13).

A 40-element mesh of the octogonal spindle We used the 72-element mesh constructed in one of the intermediate steps detailed in Table 8.1 to create a 40-element mesh of the octagonal spindle. Indeed, Schneiders' pyramid and the octagonal spindle are related by their dual graphs: replace each quadrangle by a vertex, and create an edge between each pair of adjacent quadrangles. If the dual edges that traverse opposite edges of a quadrangle in the primal are grouped together, a simple arrangement of curve is obtained. The dual of Schneiders' pyramid is obtained by adding one curve to the dual of the octagonal spindle (Figure 8.2).

This relationship determines a method to create hexahedral meshes of the octagonal spindle from

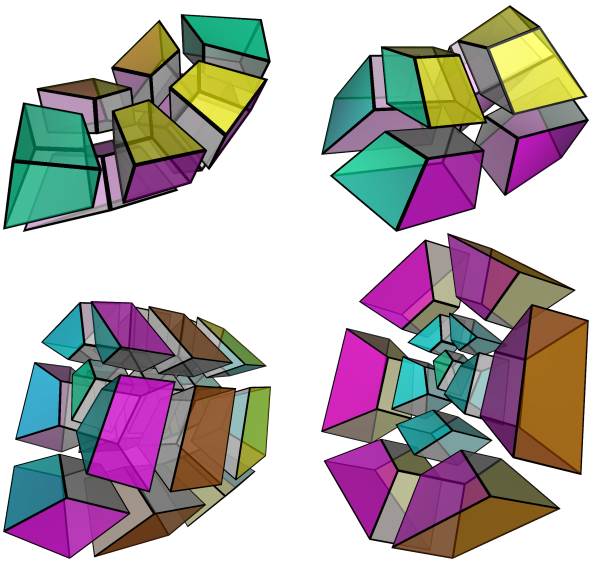


Figure 8.12: (top) Removal of two hexahedra from Schneiders' pyramid; (bottom) removal of six hexahedra. The initial cavity (left) and the remeshed cavity (right) have the same combinatorial boundary (top: 18 facets; bottom: 30 facets). Colors highlight the correspondence between faces.

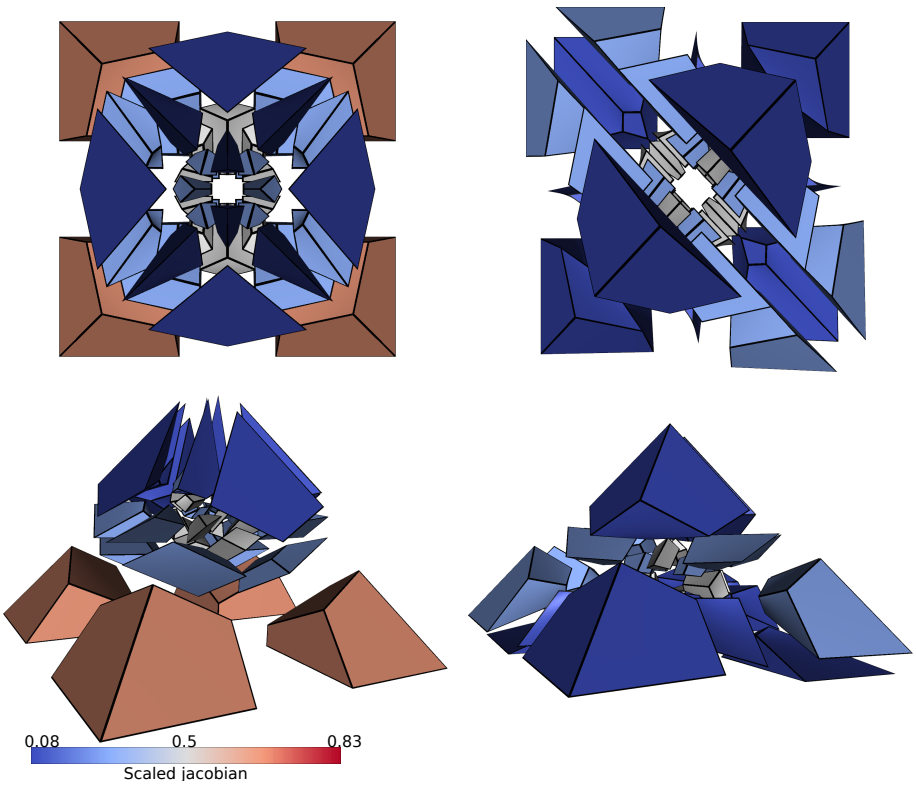


Figure 8.13: Comparison of our 44-element mesh of Schneiders' pyramid (left) with the smallest known 36-element solution (right). Both admit two planar symmetries.

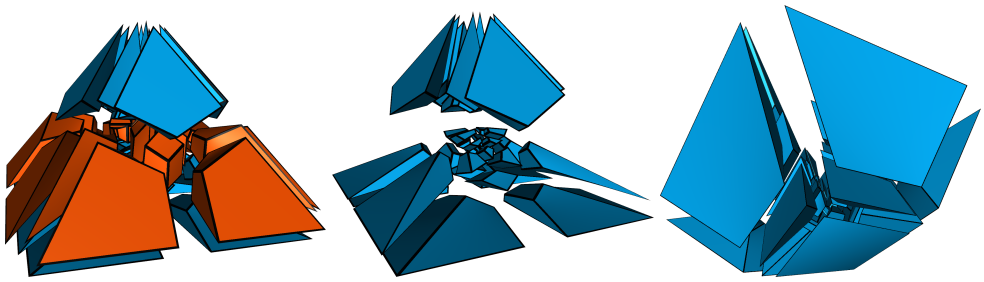


Figure 8.14: A layer of hexahedra in a 72-element mesh of Schneiders' pyramid (left) is removed (center). After merging the endpoints of each removed edge, a mesh of the octagonal spindle is obtained (right).

certain meshes of Schneiders' pyramid. The dual of a hexahedral mesh is a simple arrangement of surfaces (Murdoch et al. 1997), where each surface is bounded by zero, one, or multiple curves of the dual arrangement of the boundary quad mesh (Figure 8.2). If the dual of a mesh of Schneiders' pyramid contains a surface which is bounded only by the curve present in Schneiders' pyramid but not in the octagonal spindle, that surface can be removed by collapsing all the edges that it traverses (Borden et al. 2002). The resulting mesh is a hexahedral mesh of the octagonal spindle.

In general, this operation may produce a degenerate mesh, with hexahedra sharing multiple quadrangles or quadrangles sharing multiple edges. This is what prevented the construction of a mesh of the octagonal spindle from the 88-element mesh of (Yamakawa et al. 2010). Applying this construction to our 72-element mesh, however, we obtain a new mesh the octagonal spindle, with 40 hexahedra and 42 interior vertices (Figure 8.14). This is the smallest known mesh of the octagonal spindle.

A 44-element mesh of Schneiders pyramid Our 44-element mesh of the pyramid was obtained by adding 4 hexahedra to our mesh of the spindle (Figure 8.13). One limitation of the hex-mesh simplification algorithm described in this section is that its execution becomes expensive, because finding the smallest hexahedral mesh of a cavity becomes exponentially more time consuming as its size increases. A cheaper algorithm could be designed by finding a small set of local operations and an algorithm to choose which of them to perform in order to reduce the size of the mesh.

8.4 Quad Flip Based Enumeration of Hexahedral Meshes

Enumerating hexahedral meshes from their vertices explore the space of all hexahedral meshes, but is extremely expensive, because of the very strong topological constraints. In this section we restrict the search space to a subset of all hexahedral meshes, the space of so-called *shellable* meshes. These hexahedral meshes can be explored efficiently using quad flips (Section 8.5.1). These *quad flips* are a set of operations to modify quadrilateral meshes and whose application can be interpreted as the construction of a hexahedron. They are the base of the practical algorithm we propose. Given a quadrangulated sphere Q, a hexahedral mesh bounded by Q is built by exploring the space of flipping operations that can be applied to Q. A solution is then obtained by finding a sequence of operations that transforms Q into the boundary of a cube (Section 8.4). When this search space is too large, the algorithm instead searches for a sequence of operations transforming Q into the boundary of any mesh within a library of pre-computed hexahedral meshes (Section 8.5). The shellable meshes space is explored in its entirety by considering all possible sequences of quad flips that correspond to valid hexahedral meshes (Section 8.4.1). Because many different sequences of flipping operations represent the same mesh, most of this work focuses on how to account for the symmetries of the input quadrangulation in order to avoid generating different sequences of quad flips corresponding to isomorphic hexahedral meshes (Section 8.4.2).

Algorithm 10 SEARCH: Enumerate shellable hexahedral meshes

Input: Q: A quadrangulation of the sphere;
H: A partial mesh;
$H_{\max}$: maximum number of hexahedra in a solution;
$V_{\max}$: maximum number of vertices in a solution.

```
1:  if H_max = |H| then
2:      return
3:  else if VISITED-SYMMETRIC-COUNTERPART(H) then
4:      return                                              ▷ Section 8.4.2
5:  else if Q ≈ a cube then
6:      h ← the hexahedron bounded by Q
7:      if IS-COMPATIBLE(H, h) then
8:          OUTPUT-SOLUTION(H ∪ {h})
9:  for all quad flip F do
10:     (Q', h) ← PERFORM-FLIP(F, Q)                        ▷ Section 8.4.1
11:     if NUM-VERTICES(H ∪ {h}) ≥ V_max then
12:         continue
13:     if IS-COMPATIBLE(H, h) then
14:         SEARCH(Q', H ∪ {h}, H_max, V_max)
```

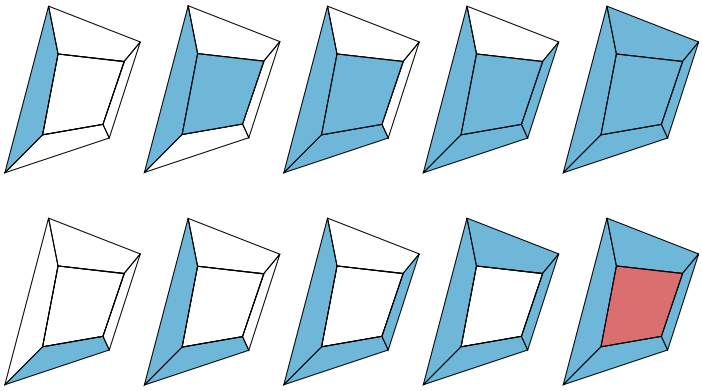


Figure 8.15: A shelling of a quadrangulation on the top. Not a shelling on the bottom because a hole appears after inserting the first 4 quadrangles.

8.4.1 Construction of Shellable Hexahedral Meshes Using Quad Flips

Shellable hexahedral mesh The method only considers a specific class of meshes: *shellable hexahedral meshes*. *Shellability* is an important and useful combinatorial concept in the study of polytopes and cell complexes (Ziegler 1995). Slightly different notions of shellability are found in the literature. We use that of *pseudo-shellings* (Bern et al. 2002) or *topology-preserving shellings* (Müller–Hannemann 1999). This type of shelling is an ordering of the hexahedra $(H_1, H_2, \ldots, H_n)$ of a hexahedral mesh such that any prefix $\bigcup_{0 \le i < k} H_i$ is homeomorphic to a ball (Figure 8.15). This definition implies that any hexahedron H_k must intersect the union of the previous hexahedra in one of six possible patterns.

The hex mesh - quad flips correspondence Gluing a hexahedron to one of these patterns modifies the boundary of the mesh locally (Figure 8.16). The transitions between these patterns are known as *quad flips* or *bubble moves* (Funar 1999). These flipping operations are therefore a valuable building block to explore the space of shellable meshes.

Note that not all hexahedral meshes admit a shelling order — see for example Furch's ball (Furch 1924). Hence, by relying on these flipping operations to build hexahedral meshes, our method is inherently unable to construct certain meshes. Nonetheless, we can guarantee that a solution still exists: all quadrangulations of the sphere with an even number of quadrangles admit a shellable hexahedral mesh (Bern et al. 2002).

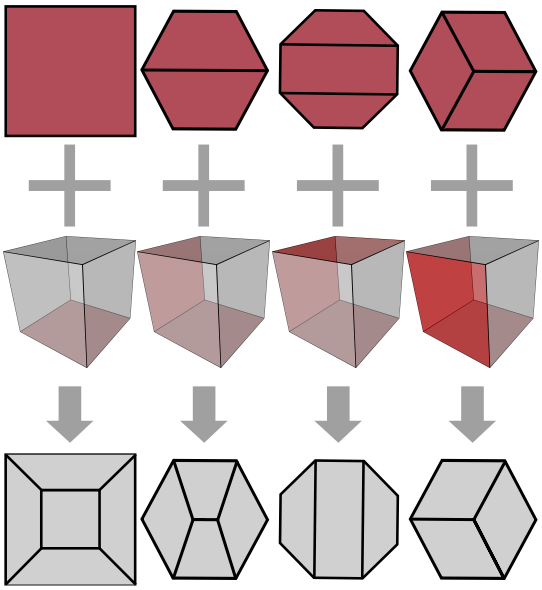


Figure 8.16: Equivalence between quad flips and hex creation. Adding one hexahedron to the the top quadrangles, correspond to a flip in the quadrangulation resulting in the bottom grey quadrangles. Six patterns are obtained by taking the symmetric counterpart of the two flips on the left.

Identifying possible flips For each quadrangulation Q visited during the search, all possible quad flips need to be identified. Each flip corresponds to a different hexahedron that can be inserted in the mesh. The algorithm successively tries adding all of them to the current mesh. Because flips are performed by starting from the target boundary, the hexahedra that are constructed during this process form the reverse of a shelling order (this is one difference with the search method of (Xiang et al. 2018)). (Müller–Hannemann 1999) construct the hexahedra in the same order, but our method, instead of only considering one mesh, explores the entire tree of possible sequences of quad flips.

The identification of all possible flips is split into two steps: first, the boundary Q is inspected to identify all occurrences of the 6 patterns from Figure 8.16. Second, those flips that correspond to the insertion of hexahedra that would make the mesh invalid are filtered out.

Checking the validity of the result mesh The hexahedron inserted by performing a flip is obtained by computing the union of the pattern before and after the flip. To determine whether or not this hexahedron is compatible with the mesh constructed so far, an efficient test is devised by considering three relations between the vertices of the mesh:

1. E, the edges of the mesh;
2. D_Q, the diagonals of the quadrangles in the mesh;
3. D_H, the interior diagonals of the hexahedra in the mesh.

These relations are disjoint in any combinatorial hexahedral mesh. For example, if a pair (u, v) is an edge, it is not the diagonal of any quadrangles or hexahedra. This leads to an efficient implementation of the test: simply maintain the three sets E, D_Q, and D_H, and verify that, after adding a new hexahedron:

1. the sets E, D_Q, and D_H remain disjoint;
2. new quadrangle facets in the hexahedron share no diagonals with any other quadrangle in the mesh;
3. none of the four interior diagonals of the new hexahedron are an interior diagonal of angleother hexahedron.

It is easy to verify that when any two hexahedra share only a vertex, an edge, or a quadrangle, these conditions are met. To verify their sufficiency, consider two hexahedra with an invalid intersection pattern. If an interior diagonal of one hexahedron is contained in the other hexahedron, one of the rules is always violated: rule 3 is violated if it is also an interior diagonal of the second hexahedron, and rule 1 is violated if it is an edge or the diagonal of a quadrangle. The only remaining cases to consider are those where the shared vertices are part of two distinct quadrangles. In all of those cases, the diagonal of one of those quadrangles appears in the other one. If it appears as an edge, rule 1 is violated; if not, both quadrangles have a shared diagonal, violating rule 2.

8.4.2 Symmetry Breaking

There are many distinct sequences of quad flips which represent identical hexahedral meshes. It is thus important to only consider a single representation for each hexahedral mesh constructed during the search, otherwise most of the computation time would be spent generating different representations of equivalent solutions.

A technique commonly used to deal with this type of issue is to define a canonical representation for objects under constructions, so that all those that belong to a given isomorphism class are transformed into the same representative element (Burton 2011; Brinkmann and McKay 2007). A significant portion of the execution time is then spent computing the canonical representations of partial solutions, which may completely change after every operation (Jordan et al. 2018). The symmetry breaking method used within our algorithm instead compares partial solutions directly, and exploits the tree-shaped structure of the search in order to reuse results from previous computations.

The strategy described in this section is based on *Symmetry Breaking via Dominance Detection* (SBDD) (Fahle et al. 2001). Consider the search tree explored by the algorithm: its nodes are partial meshes constructed during the search, and edges correspond to the insertion of new hexahedra through quad flips. The objective is to prune from this search tree nodes that correspond to meshes that have already been explored (up to symmetry). This is accomplished using the following steps:

1. The automorphism group of the input quadrangulation is pre-computed;
2. Traverse the search tree and encode fully explored subtrees into a sequence S;
3. Determine if each new node should be pruned or notion by comparing it against the nodes stored in S.

Computing the automorphism group Given a quadrangulation Q, we compute the set of its symmetries, known as its automorphism group. A permutation σ of the vertices of Q is a symmetry if it *preserves* the set of quadrangles: the image of any quadrangle (a, b, c, d), $(\sigma(a), \sigma(b), \sigma(c), \sigma(d))$ is a quadrangle of Q, and any quadrangle (a, b, c, d) is the image of a quadrangle $(\sigma^{-1}(a), \sigma^{-1}(b), \sigma^{-1}(c), \sigma^{-1}(d))$. Note that the orientations of the quadrangles may be reversed by σ.

Symmetries are computed one after another, by fixing some quadrangle $q_A \in Q$ and assuming that its image under a symmetry σ is known to be $q_B \in Q$. There are 8 different ways to map the vertices of q_A onto the vertices of q_B, corresponding to the 8 symmetries of a quadrangle. The entire permutation σ is uniquely determined by this part of the map: the quadrangles adjacent to q_A must be the images of the quadrangles adjacent to q_B under σ, and the quadrangles adjacent to those must also be images of each other, and so on, until the whole quadrangulation has been traversed. This process is well-defined because each edge is in at most two quadrangles.

The entire set of symmetries is computed by considering all $8|Q|$ possible ways to map an arbitrary quadrangle q_A to any other quadrangle of Q. If an assumption is correct, a symmetry σ is obtained; if not, a contradiction will be reached when trying to construct the symmetry (two vertices mapping onto the same target vertex, or a single vertex with two images under σ). Because q_A must be the image of some quadrangle under any symmetry σ, this process yields the entire automorphism group.

In the worst case, the entire automorphism group is determined in $O(|Q|^2)$ operations. In practice, this quadratic time algorithm outperforms more complex linear time algorithms designed for planar graph isomorphism (Eppstein 1999; Colbourn et al. 1981) when applied to small quadrangulations, thanks to well-tuned heuristics. In particular, our implementation stops the algorithm as soon as

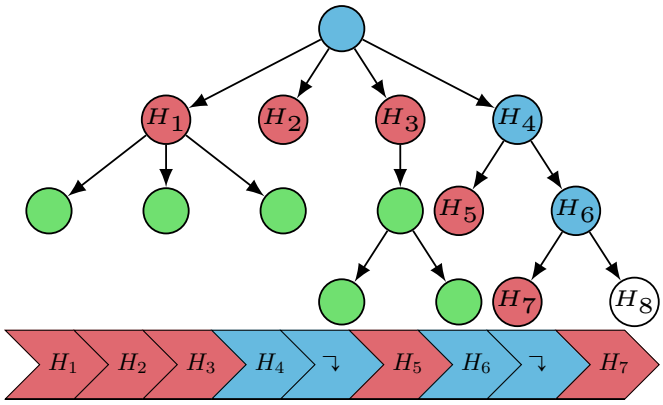


Figure 8.17: A partially explored search tree and the sequence used to compare the current node (in white) against the previously explored part of the tree. No-goods are shown in red, and partially explored subtrees in blue.

two vertices of different degree are mapped onto one another by the permutation under construction (Brinkmann and McKay 2007).

Moreover, because this method does not use the planarity of the graph, it is also more general. The only requirement is that the input be a *pseudomanifold*: a combinatorial cell complex in which every facet is contained in at most two distinct cells. Indeed, a variant of this method will be used to compare hexahedral meshes in Section 8.4.2, by having quadrangles take over the role of edges.

Encoding the already explored partial meshes An efficient traversal of the search tree requires the search to stop as soon as the mesh under construction is the symmetric counterpart of a mesh that has previously been constructed. In the previous section, the set of symmetries that need to be considered was determined. This section now focuses on efficiently encoding the set of hexahedral meshes that have been constructed during the search.

Of course, the search tree is exponentially large, making it impossible to store every single mesh that is constructed during the search. Instead, SBDD only stores information about the roots of maximal fully explored subtrees, known as *no-goods* (Gent et al. 2006). The current node should then be pruned if and only if the mesh under construction is the symmetric counterpart of one of the children of one of the no-goods. Note that a no-good is referred as such even if some of its children are solutions, since it is not desirable to compute the symmetric counterparts of those solutions.

No-goods can be stored efficiently thanks to the structure of the search tree. Recall that each node within the search tree corresponds to a partial hexahedral mesh, and each edge corresponds to the insertion of a hexahedron. Nodes with a common ancestor in the tree then share a common set of hexahedra as a prefix, and this prefix only needs to be stored once Figure 8.17. Upon visiting a new node, the most recently added hexahedron is inserted in the sequence, followed by a special branching symbol, indicating that the rest of the sequence will encode the children of this node. Upon backtracking, everything up to and including the last branching symbol of the sequence is removed.

Dominance detection and pruning The last part of our symmetry breaking method is the test used to prune nodes of the search tree that do not need to be explored because any solution that could be found by doing so has already been found. These nodes are said to be *dominated* by one of the no-goods, *i.e.* they are the symmetric counterpart of one of the children of one of the nodes that have been previously explored and stored in the sequence Figure 8.17.

Since the search involves exploring exponentially many nodes, this dominance test must be implemented without explicitly comparing the current node against all previously explored nodes. Instead, this test is broken down into two steps:

1. Find a no-good such that all its hexahedra are contained in the current partial solution
2. Determine if the hexahedra that are in the partial solution but missing from the no-good could be inserted using flipping operations.

The sequence S constructed in the previous section is very valuable for this: not only does it save space by factoring out a common prefix, but it also saves time by allowing this prefix to be processed only once.

Let H be the current partial solution. The first step is to search within S for a partial mesh whose hexahedra are a subset of H. The process to find such a partial mesh is similar to the algorithm used to compute the automorphism group initially (Section 8.4.2). The goal is to construct σ, which maps the vertices of some partial mesh encoded in S to vertices of the current solution H, such that all hexahedra in the no-good are preserved by the map σ. The construction of σ again begins from an initial assumption, namely that the images of all boundary vertices through σ are known. Because boundary quadrangles must be preserved by σ, the set of possible initial assumptions is precisely the automorphism group that was previously computed.

The dominance test algorithm is executed once for each element of the automorphism group and consists in a traversal of S during which the map σ is extended. The process ends either upon finding a partial mesh contained in H or upon reaching a contradiction. For each hexahedron h found in S, we attempt to extend σ such that h maps to some hexahedron of the current solution H. Each hexahedron created by a quad flip shares at least one quadrangle with the boundary or with a previously created hexahedron. Because of this, each hexahedron in S has at least one quadrangle whose symmetric counterpart is known. It is therefore possible to search for the hexahedron h' within the current solution H that contains this quadrangle (and has not already been determined to be the symmetric counterpart of another hexahedron).

If such a hexahedron h' exists, it must be the symmetric counterpart of the hexahedron h encoded in S, and the map σ is extended accordingly. If h corresponds to a fully explored node (shown in red in Figure 8.17), the current partial solution H contains the symmetric counterparts of all hexahedra of the corresponding no-good. If, however, h corresponds to a partially explored node (shown in blue in Figure 8.17), and its symmetric counterpart cannot be found, the traversal ends early because all subsequent partial meshes encoded in S contain h, which is not in the current solution. In all other cases, the traversal of the sequence continues.

Clearly, containing all hexahedra from some no-good is a requirement for a node being dominated — all children of the no-good share this common prefix. There could still be cases where none of the children of this no-good contain all the hexahedra that are in the current node. In other words, it may be impossible to find a sequence of quad flips which inserts the missing hexahedra when starting from the no-good. Testing for the existence of such a sequence may appear intractable at first, because shellability is an NP-Complete property (Goaoc et al. 2018). Thankfully, a correct test only needs not to produce any false positives, since false positives are the only reason a part of the search tree would incorrectly get pruned, causing solutions to be missed. Furthermore, because shellable meshes tend to accept many different shelling orders, there is a straightforward algorithm meeting this requirement and which very often computes the correct result: try a small number of permutations (say 10), then give up if no reverse shelling order was found.

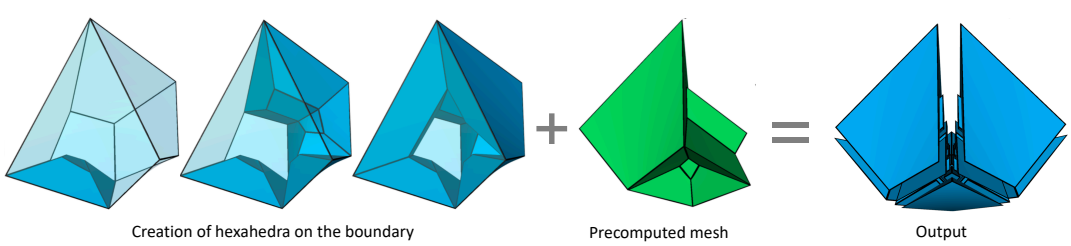


Figure 8.18: Given a quadrangulation of the topological sphere, our algorithm creates hexahedra on the boundary until the unmeshed cavity matches the boundary of a pre-computed hex mesh that is merged to obtain the final combinatorial hexahedral mesh.

8.5 A Practical Algorithm to Compute Hexahedrization of Small Quadrangulations

The exhaustive search described in the previous section can only be used within small limits on the maximum number of hexahedra, because of its exponential execution time. In many cases, finding a complete shelling by searching exhaustively is too difficult: the sequence of flips to construct the smallest solution is too long, and the search tree contains many paths which transform the initial boundary into one which is more difficult to mesh, instead of being closer to a solution.

Instead of searching for a sequence of quad flips that transforms the initial boundary Q into a cube (see line 5 in Algorithm 10), the key idea for solving larger cases is to stop the algorithm when a known configuration is found. For that purpose, we compute all boundaries that can be shelled with at most n hexahedra (say $n \leq 11$). Using a list of all such boundaries and one of their shellings (Section 8.5.1), this variant of the algorithm can efficiently look up boundaries in the list during the search. This allows complete solutions to be constructed from any sequence of flips leading to any of the boundaries in the pre-computed set.

8.5.1 Pre-computing Small Shellable Meshes

Consider the flip graph for quadrangulations of the sphere: its nodes represent quadrangulations of the sphere, and arcs between these nodes represent a flip between two quadrangulations. A breadth-first traversal of this graph starting from the cube and stopped at depth n generates all quadrangulations that can be obtained using a sequence of up to n flips. To deal with cycles in this graph, previously visited quadrangulations are stored in a hash table. The hash value for quadrangulations is constructed from a signature based on the valence of vertices, and the isomorphism test two quadrangulations is performed using a variation on the symmetry breaking algorithm where the two starting quadrangles are part of different quadrangulations.

The signature used by our algorithm is a histogram of the valences of each vertex, followed by the number of edges connecting vertices of valence v_a and valence v_b, for any v_a and v_b where this number is non-zero. While this choice of signature causes collisions, it can be computed quickly and new entries can be inserted without necessarily needing a slower computation to find a unique canonical representation.

Not all quadrangulations generated in this breadth first search admit a shelling with up to n hexahedra: interpreting the flips performed during the traversal of the graph as the insertion of hexahedra, these hexahedra may not all be compatible. By explicitly testing for compatibility while performing the breadth first search (Algorithm 11), we obtain a greedy construction similar to the procedure outlined by (Xiang et al. 2018): the hexahedra that are found are those which admit a shelling such that any prefix is the smallest shellable hexahedral mesh for the corresponding boundary. For large values of n, it is not clear that such a shelling should always exist, but we can verify this property for small values of n. For every quadrangulation found during the breadth-first search but without a hexahedral mesh found by Algorithm 11, Algorithm 10 is used to verify that there is indeed no shellable hexahedral mesh with at most n hexahedra. This test was performed for $n \leq 10$, and no counter-examples were found.

From Algorithm 11, a table of $69,043,690$ boundaries that can be meshed with up to 11 hexahedra is constructed (Table 8.2).

8.5.2 Using the Pre-computed Table

If at any point during the search, the boundary of the unmeshed region matches one of the pre-computed quadrangulations, the shelling of that quadrangulation is used to finish the meshing of that region.

The idea is to use the shelling computed in the previous section to fill the unmeshed region. Simply combining the two solutions is not always possible: this may produce an invalid mesh where, for example, two hexahedra share multiple quadrangles (Figure 8.19). Algorithm 10 could be used to compute all shellings of the unmeshed region with up to n hexahedra. If this search finds a shelling compatible with the partial solution constructed so far, a solution can be generated, at the cost of an additional computation.

Algorithm 11 GENERATE-SHELLINGS: Generate small shellable hexahedral meshes

Input n: maximum size for the generated hexahedral meshes
Output $\mathcal{H}$: a set of hexahedral shellings with up to n hexahedra in each mesh.

```
1:  S ← ∅
2:  H ← ∅
3:  Q ← NEW-QUEUE()
4:  ENQUEUE(Q, CUBE)
5:  while Q is not empty do
6:      H ← DEQUEUE(Q)
7:      H ← H ∪ {H}
8:      if |H| = n then
9:          continue
10:     for all quad flip F do
11:         (B, h) ← PERFORM-FLIP(F, ∂H)
12:         if IS-COMPATIBLE(H, h) ∧ B ∉ S then
13:             S ← S ∪ {B}
14:             ENQUEUE(Q, H ∪ {h})
15: return H
```

$H_{\max}$	# quad meshes	timing
1	1	< 0.1s
2	2	< 0.1s
3	5	< 0.1s
4	17	< 0.1s
5	74	< 0.1s
6	489	< 0.1s
7	4,192	0.12s
8	42,676	1.78s
9	476,520	34.418s
10	5,632,488	14min 55s
11	69,043,690	6h 41min

Table 8.2: Number of combinatorial quadrangulated boundaries that can be shelled with up to $H_{\max}$ hexahedra. Timings are given for a single thread on an Intel® Core™ i7-7700HQ CPU.

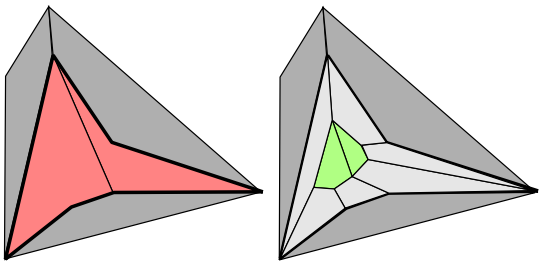


Figure 8.19: Insertion of a buffer layer (light gray) to guarantee final mesh validity when using precomputed cavity meshes.

Even if no such shelling was found, a solution can be constructed from any shelling of the unmeshed region, without performing an additional search or storing multiple hexahedrizations for each boundary: first construct a copy of the boundary of the unmeshed region, then, for each quadrangle of this boundary, create a hexahedron to connect each quadrangle to its copy. The hexahedra that have been inserted in this manner are guaranteed to be compatible with any hexahedrization of the unmeshed region, allowing a complete mesh to be constructed. When this approach is used, the first solution found by the algorithm is not in general the smallest. However, when the smallest solution contains a large number of hexahedra, this approach can construct solutions in many cases where methods with stronger guarantees fail to find any, because it adds several hexahedra without branching.

Efficient access to the pre-computed table is performed using a binary search. We create an array of all the quadrangulations we found, sorted by their signatures. To find the hexahedral mesh corresponding to a given quadrangulation, its signature is computed and an isomorphism test is performed on all quadrangulations in the table that have the same signature.

8.6 Results

8.6.1 A New Bound on Hexahedral Mesh Size

Only one previous solution to the constrained hexahedral meshing problem gives a completely explicit construction (Carbonera et al. 2010). This method requires 5396 n hexahedra to construct a valid mesh bounded by n quadrangles. In the following, we prove that this bound can be lowered to 78 n using the construction proposed by (Erickson 2014).

Using our search algorithm (Section 8.5), we found hexahedral meshes for both types of buffer cells that Erickson's construction needs, along with geometric realizations using linear hexahedra (Figure 8.20) obtained by applying existing mesh untangling techniques (Toulorge et al. 2013; Livesu et al. 2015), although the solutions have a very low minimum scaled Jacobian (Table 8.3). The meshes that we found contain 37 and 40 hexahedra. Because gluing multiple buffer cells together as needed by the construction would create a degenerate mesh, a hexahedron is added on each boundary quadrangle. The resulting meshes of the buffer cells have 57 and 62 hexahedra respectively, giving the following result:

Theorem 8.6.1. *Let Ω be a compact and connected subset of $\mathbb{R}^3$ bounded by a 2-manifold $\partial\Omega$. Given a quadrangulation Q of $\partial\Omega$, each component of Q containing an even number of quadrangles, and a triangulation T of Ω (splitting each quadrangle of Q into two triangles), if there is a combinatorial hexahedral mesh of Ω bounded by Q, then there is one with no more than $62|Q| + 8|T|$ hexahedra. In particular, if Ω is a ball (hence $\partial\Omega$ is a sphere) and $|Q|$ is even, there is a combinatorial hexahedral mesh bounded by Q with no more than $78|Q|$ hexahedra.*

Proof. Follow the construction of (Erickson 2014) using the templates that we computed. There is one buffer cell for each boundary quadrangle, and each tetrahedron of the triangulation T is split into 4, 7, or 8 hexahedra. In the worst case, each buffer cell will be meshed with 62 hexahedra, and each tetrahedron will be split into 8 hexahedra.

If Ω is a ball, there is always a triangulation T with $2|Q|$ tetrahedra, obtained by arbitrarily splitting each quadrangle into two triangles, adding a vertex inside the domain, and joining each triangle to this new vertex by a tetrahedron. The bound for this special case is therefore $62|Q| + 8 \times 2|Q| = 78|Q|$. □

A similar bound can be obtained for quadrangulations with an odd-number of quadrangles in some of their components. In that case, hexahedra are added to connect pairs of odd components, and 8.6.1 is used to compute the number of hexahedra to mesh the rest of the domain.

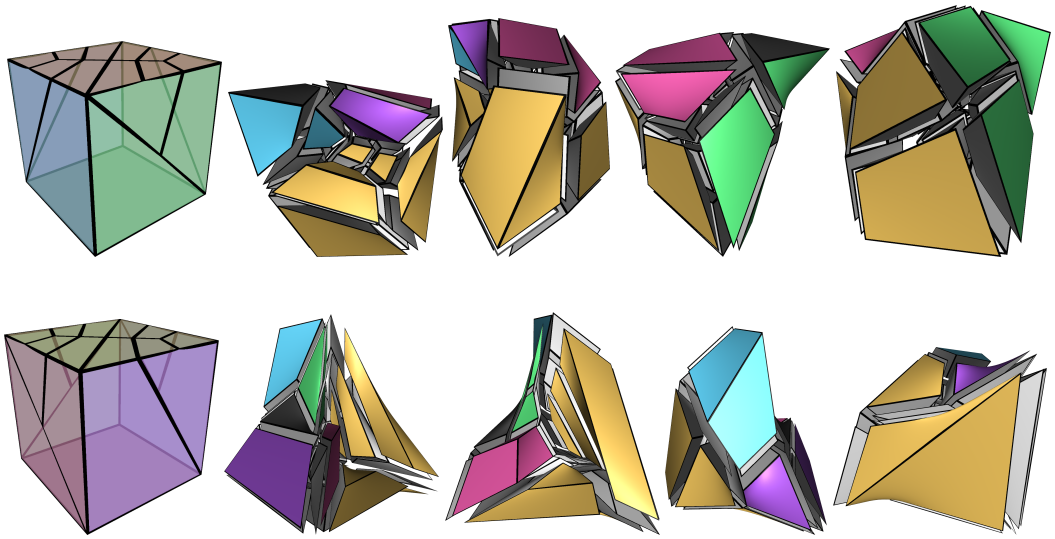


Figure 8.20: Hexahedrizations of the two types of buffer cubes used to mesh arbitrary domains in the algorithm of (Erickson 2014). (top) 37 hexahedra to mesh the 20-quadrangle cell; (bottom) 40 hexahedra to mesh the 22-quadrangle cell. Colors correspond to the different sides of the original cubes (shown on the left).

Template	$\|Q\|$	$\|V_{\text{bnd}}\|$	$\|H\|$	$\|V_{\text{total}}\|$	$\|E\|$ per valence			sc. jacobian	
					3	4	5	min	max
Tetragonal trapezohedron	8	10	40	52	40	75	4	0.35	0.42
Schneiders' pyramid	16	18	36	51	32	62	4	0.12	0.49
Erickson's buffer cell (1)	20	22	37	53	43	49	4	0.31	0.63
Erickson's buffer cell (2)	22	24	40	55	44	48	9	0.031	0.45

Table 8.3: Statistics for the geometric meshes computed for the test cases of Figures 8.1 and 8.3

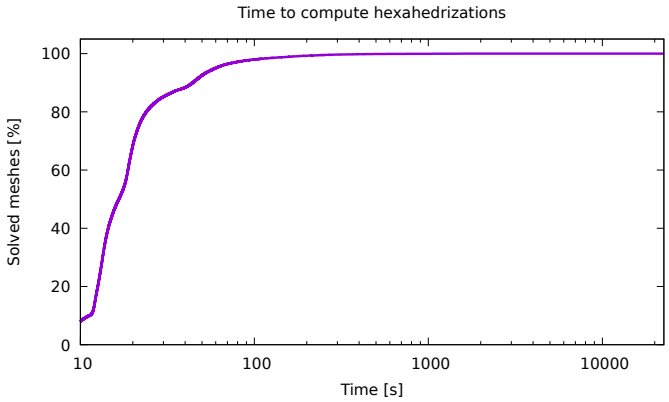


Figure 8.21: Time to compute hexahedrizations for the quadrangulations of the sphere with up to 20 quadrangles. Run on a machine with two AMD EPYC 7551 CPUs (32 cores per CPU).

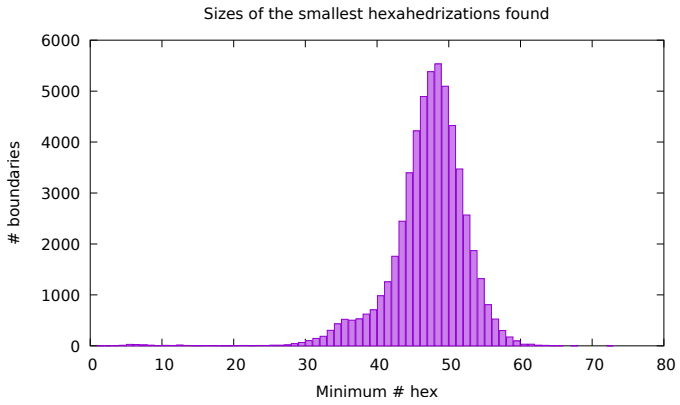


Figure 8.22: Sizes of the smallest hexahedrizations found for the quadrangulations of the sphere with up to 20 quadrangles.

8.6.2 One Hexahedrization for each Quandrangulation of the Sphere

We used the algorithm described in Section 8.5 to compute hexahedrizations for all even quadrangulations of the sphere containing up to 20 quadrangles (Table 8.4). The 54,943 input quadrangulations were generated using `plantri` (Brinkmann, Greenberg, et al. 2005). We pre-computed shellable hexahedral meshes with up to 11 hexahedra. Of the 69,043,690 boundaries that were pre-computed, only 130 are included in the list of inputs. Nonetheless, in about 20% of all instances, the search for a solution terminates almost immediately after loading the set of pre-computed solutions (Figure 8.21). Only a few additional seconds are enough to find hexahedrizations bounded by most quadrangulations of the sphere.

Which are the most challenging boundaries? There are however some more difficult cases, requiring over an hour of computation time (Figure 8.23). The trapezohedron bounded by n faces, obtained by generalizing the tetragonal trapezohedron of Figure 8.1, is usually among the most difficult cases of a given size, requiring meshes with an intricate internal structure in order to be filled. For example, the smallest solution found for the 20-face decagonal trapezohedron contained 72 hexahedra, strictly more than any of the other boundaries (Figure 8.22). Similarly, the 16-face octagonal trapezohedron required 67 hexahedra, with the decagonal trapezohedron being the only boundary for which all solutions found were larger. The trapezohedra are also among the boundaries that require the most time before any solution could be found. The 14-face heptagonal trapezohedron is the second most time consuming input, requiring 2h 50min, and the 20-face decagonal trapezohedron is the third, requiring 2h 43min. In the worst case, shown on Figure 8.23, it took 6h 15min before a 58-element mesh was found.

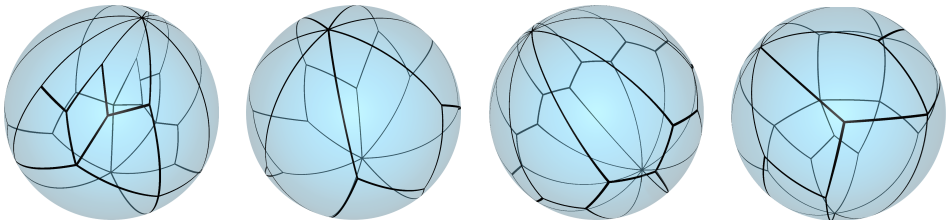


Figure 8.23: The four most time-consuming quadrangulated spheres to mesh using our method. Each required over an hour of computation time on 64 cores.

Q	$\lvert H\rvert$			$\lvert V_{\text{total}}\rvert$			%edges by valence			
	min	max	med	min	max	med	3	4	5	6
6		1			8			(none)		
8		44			56		48	34	16	2
10	2	58	36	12	64	48	39	45	15	1
12	3	47	43	14	57	52	38	50	11	1
14	3	59	44	16	73	55	38	48	12	1
16	4	67	45	18	77	56	37	51	11	1
18	4	67	46	20	79	58	38	49	12	1
20	5	72	47	22	81	59	38	49	12	1

Table 8.4: Statistics on the number of hexahedra, number of vertices, and edge valences for the combinatorial meshes we computed for the even quadrangulations of the sphere with up to 20 quadrangles.

Schneiders' pyramid and the octogonal spindle The quadrangulations of Figure 8.1 are not particularly difficult to mesh using our method. Indeed, most quadrangulations of up to 18 quadrangles require more computation time than these two cases — and even finding the smallest known solutions is orders of magnitude easier than finding any solutions for the cases shown on Figure 8.23. On a 4-core Intel® Core™ i7-7700HQ CPU, after pre-computing a list of shellable meshes with up to 10 hexahedra, the 36-element mesh of Schneiders' pyramid originally found by (Xiang et al. 2018) is found within 3 seconds of search. A 44-element mesh of the tetragonal trapezohedron is found within 4 seconds and it takes 31 seconds to find the smallest known 40-element mesh constructed in (Verhetsel, Pellerin, and J.-F. Remacle 2019a). Statistics for the geometric realizations found for both meshes are shown on Table 8.3.

This research, by allowing hexahedral meshes to be computed for any small quadrangulation of the sphere, opens up a wide array of new possibilities. It is an important step for hex-dominant meshing (Yamakawa et al. 2003), as this solves the combinatorial aspect of the problem of filling the cavities that those methods leave. This method also offers new insights into the structure of block decompositions for configurations for which state-of-the-art techniques such as (H. Liu et al. 2018) fail to generate a valid block structure, by allowing combinatorial meshes to be computed in some of those cases.

Chapter 9

Perspectives

The best meshes are invisible. When you are watching the latest 3D animation movie or playing your favorite video game, you are not thinking (at all) about the meshes defining the geometry of all the 3D shapes around. If you are running numerical simulations, you wish you should not have to see the mesh, but you have to. If the mesh is not good enough the simulation will not run or will give incorrect results. You may dream about a one-click meshing method that builds the mesh you exactly want (without even knowing the criteria yourself) completely automatically at the right level of detail, fully adapted to the simulations you will be running next. Such a general automatic method might never exist and mesh generation will probably remain an expensive step in engineering analysis for many years.

The research contributions presented in this manuscript open promising perspectives to improve our understanding of the underlying combinatorial problems of mesh generation, and more particularly to develop algorithms for the generation and optimization of hexahedral and hybrid meshes. Chapter 4 presented a parallel algorithm to generate automatically tetrahedral meshes for general domains. Up to this date, there is no equivalent hexahedral meshing algorithm. My point of view is that the main difference is theoretical. Major theoretical results on triangulations lead to robust provable algorithms on one hand and to more practical algorithms used for numerical simulations. For hexahedral meshes, theoretical results are scarce and mostly topological. The contributions presented in Chapter 8 are important but remain topological. One main perspective is to study the geometrical counterparts of these results.

9.1 Subdividing and Combining Polyhedra

Studying the underlying theoretical problems is one way forward to new algorithms in mesh generation. My theoretical contributions Chapter 7 and Chapter 8 show that an adequate strategy is to first solve the combinatorial problems before taking into account geometry. Both the combination of tetrahedra (or other polyhedra) and subdivision into tetrahedra (or other polyhedra) are two basic problems on which we have but a limited knowledge and theoretical understanding. Exploring and understanding the subdivision (that is but the meshing) of simple polyhedra is the key to the development of new, potentially disruptive meshing algorithms. The objective would be to constitute a database of subdividing patterns of polyhedra into tetrahedra, hexahedra, or other polyhedra. In meshes, to the contrary of the usually admitted definition of polyhedra, the cell faces are not necessarily planar. Quadrilateral facets of finite elements are bilinear. The range of admissible geometries is extremely large and the only constraint to enforce is that the mapping must be injective, therefore the determinant of the Jacobian cannot take the zero value. The bilinear faces can be approximated by two triangles, defining this way, polyhedral with triangular facets.

Subdivision in tetrahedra Enumerating the triangulations of polyhedral cells with triangular of quadrilateral facets is related to two fundamental problems: the existence of a triangulation of a given polyhedron and the computation of the minimal and maximal subdivision of a polyhedron. The first one is an NP-complete problem (Ruppert et al. 1992) and the second is NP-hard (Below et al. 2004).

The solution to this problem is related to the vanishing not well understood, nor well characterized boundary between triangulable and non-triangulable polyhedra. This is a central problem for the boundary recovery step in tetrahedral meshing. Discrete geometry results and methods can be used to enumerate combinatorial triangulation as well as discriminate between realizable and non-realizable triangulations for a small number of vertices. Additional work is necessary to extend these results to other polyhedra than the hexahedron.

Subdivision in hexahedra The most important perspective is to explore the subdivision of general polyhedra into hexahedra. Prisms, hexahedra themselves, and tetrahedra can very easily be subdivided into respectively 4, 6, and 8 hexahedra. As we have seen, there is no simple solution for the square based pyramid. This problem was described initially by (Schneiders 1996). We know that finding a good quality solution with a small number of hexahedra is not possible, since we proved that any solution must at least contain 17 hexahedra and we suspect that the smallest solution has 36 hexahedra (Chapter 8). The method developed to prove this result is particularly promising. The combinatorial meshing problem (in short, geometry is ignored) is extremely constrained and particularly difficult for hexahedral meshes. To ever be able to take the results we presented into account in meshing algorithms, we need to better understand which are the base operations and to take into account geometry. The method could be used for remeshing hexahedral meshes while ensuring the elements to be valid for computations.

Remark The meshing problem is an ill-posed problem, in most cases there are an infinite number of meshes of a given domain because there is an infinite number of positions for the vertices of the mesh. When the vertices are fixed, we know that the set of all the meshes that can be built from the points is finite, but we also know that we cannot compute that set in the general case in 3D. There is however a potentially huge impact in the exploration of all the possible meshes that can be generated from a given set of points.

9.2 Practical Meshing

Significant improvements of the main steps of the indirect meshing method (Section 6.3) could permit to build an algorithm that can be adapted to the needs of multiple applications.

Mesh vertices sampling Geometrical features (low angles, thin layers, high curvature) must be taken into account when sampling the mesh vertices because these geometrical characteristics are often in conflict with the carateristics desired for the cells (type of cells, minimal angle, edge size). 3D models, whatever the application, geological modeling, aeronautics, etc., include thin layers that are a challenges for meshing software because either a high number of points are added, or the points should be aligned to build prisms and hexahedra. No existing method is able to build these cells robustly. The measures of the difficulty to locally mesh a given geometrical model in Section 2.3 could be extended to locally determine the type of cell to build, i.e. to design measures that caraterise the size, the orientation, and the type of cells to generate.

Tetrahedral mesh generation Each step of mesh generation raises questions, and one important for indirect mesh generation is which tetrahedral mesh is adequate to combine tetrahedra. This is an open question and the geometrical and topological properties of the triangulations from a point of view of their ultimate combination in hexahedra are unknown. The answer is most probably not the Delaunay triangulation that is the base of most algorithms. It maximizes the smallest angle in the mesh, but shows not particular qualities to build tetrahedra with angles closer to 90 degrees.

Combine tetrahedra in hexahedra, prisms and pyramids Building all possible cells by combining tetrahedra into other cells is a now a solved problem (Section 6.3). But choosing the cells of the final mesh is much more difficult problem and one main obstacle to the usability of the method. From my point of view, the abstract formulation of the problem as a graph problem is not fully adapted to the meshing problem because meshing is a local geometrical problem.

Mesh improvement The indirect method is very robust but will probably never be able to generate full-hexahedral meshes. Let us suppose that the obtained hex-dominant mesh could be modified so that all pyramids, prisms and tetrahedra are replaced by hexahedra. The real question is then: Do local operations that improve a hybrid mesh in a provable way exist? If they do exist, they are the key to adapt a hybrid mesh to particular application needs. Local operations, also called cavity operations, are the operations that allow the automatization of tetrahedral meshing algorithms, as we have seen in Section 4.1.1 and Section 8.3. A cavity operation takes as input a set of neighboring mesh cells that are deleted (the cavity) and replaced by another set of cells that have exactly the same boundary. For hexahedral meshes the base operations are not all known (Ledoux et al. 2010) and the status for hybrid meshes is worse. There might not even be a tentative list of the operations where pyramids, tetrahedra, hexahedra, prisms could be involved. Such operations would be extremely useful for robust and efficient meshing of complex geometries and mesh adaptation of hybrid meshes.

9.3 Final Words

Can you mesh my model? Having been working on mesh generation algorithm for some years, I have been asked this question many times. Answering right away is often not a good idea because tight definitions of the input and output are required to give a reliable answer. What is the input (exact file format, geometry, characteristics)? Is the input a valid solid model (watertight, tagged components, no holes, no gaps, no intersections, no hidden defects)? What is the desired output mesh (cell type, size, quality measure, level of detail, angles)? What is the mesh purpose? Depending on the exact answers, generating the desired mesh can be trivial, easy, difficult, extremely difficult, or simply cannot be done. Relatively easy in two dimensions, the difficulty of mesh generation is widely underestimated in three dimensions and mesh generation is a fascinating problem where one can always be surprised.

Could a software answer that question? In other words, can we predict the success or failure of a meshing algorithm on a given input geometry? This is one of the key of automatic mesh generation. The main problem are the geometrical features of 3D models that are (always) in conflict with the desired mesh properties that have been for decades an incredible challenge in mesh generation. The increasing size and complexity of models emphasizes this challenge. Whatever the type of mesh, whether tetrahedral, hexahedral, or hybrid, evaluating efficiently what can be done and cannot be done would be a great help to all users. Different constraints should be accounted for, constraints on cells (points, segments, faces, cells) that must be part of the final mesh, constraints on the number of points or cells inside a volume, on the distance between the mesh and the input model and so on and so forth.

A meshing dream Imagine we can compute all the subdivisions of a 3D polyhedron in tetrahedra, hexahedra, prisms, pyramids, or combinations of these. Imagine that these computations are extremely fast and that the cell qualities can be evaluated as fast. I believe then than most of the algorithmic challenges in mesh generation would be over. This is however a dream. For hexahedra, we have been stuck for a long time at trying to compute one subdivision. For tetrahedra, we can in most cases compute one subdivision in tetrahedra, but computing all of them to choose the best remains out of reach. I believe that studying those theoretical questions could be one of the key to find really new algorithms for 3D meshing, even if they do not appear at first relevant in practice. I believe that understanding in depth these problems that are encountered by our current algorithms is one of the key to more robust and more efficient algorithms.

Bibliography

Abel, D. J. and D. M. Mark (1990). "A Comparative Analysis of some 2-Dimensional Orderings". In: *International Journal of Geographical Information Systems* 4.1, pp. 21–31.

Alliez, P., D. Cohen-Steiner, M. Yvinec, and M. Desbrun (2005). "Variational tetrahedral meshing". In: *ACM Transactions on Graphics* 24.3, pp. 617–625.

Alliez, P., E. Colin de Verdiere, O. Devillers, and M. Isenburg (2005). "Centroidal Voronoi diagrams for isotropic surface remeshing". In: *Graphical Models* 67.3, pp. 204–231.

Altshuler, A., J. Bokowski, and L. Steinberg (1980). "The classification of simplicial 3-spheres with nine vertices into polytopes and nonpolytopes". en. In: *Discrete Mathematics* 31.2, pp. 115–124.

Altshuler, A. and L. Steinberg (1976). "An enumeration of combinatorial 3-manifolds with nine vertices". en. In: *Discrete Mathematics* 16.2, pp. 91–108.

Altshuler, A. and L. Steinberg (1973). "Neighborly 4-polytopes with 9 vertices". en. In: *Journal of Combinatorial Theory, Series A* 15.3, pp. 270–287.

— (1974). "Neighborly combinatorial 3-manifolds with 9 vertices". en. In: *Discrete Mathematics* 8.2, pp. 113–137.

Aluru, S. and F. E. Sevilgen (1997). "Parallel domain decomposition and load balancing using space-filling curves". In: *Proceedings of the Fourth International on High-Performance Computing, HiPC 1997, Bangalore, India, 18-21 December, 1997*, pp. 230–235.

Amenta, N. and M. Bern (1999). "Surface reconstruction by Voronoi filtering". In: *Discrete and Computational Geometry* 22.4, pp. 481–504.

Amenta, N., S. Choi, and G. Rote (2003). "Incremental constructions con BRIO". In: *Proceedings of the 19th ACM Symposium on Computational Geometry, San Diego, CA, USA, June 8-10, 2003*. ACM, pp. 211–219.

Andrle, R. (1996). "Complexity and scale in geomorphology: Statistical self-similarity vs. characteristic scales". English. In: *Mathematical Geology* 28.3, pp. 275–293.

Andujar, C., P. Brunet, and D. Ayala (2002). "Topology-reducing surface simplification using a discrete solid representation". In: *ACM Transactions on Graphics* 21.2, pp. 88–105.

Anquez, P. (2019). "Correction et simplification de modèles géologiques par frontières : impact sur le maillage et la simulation numérique en sismologie et hydrodynamique". PhD Thesis.

Anquez, P., J. Pellerin, M. Irakarama, P. Cupillard, B. Lévy, and G. Caumon (2019). "Automatic correction and simplification of geological maps and cross-sections for numerical simulations". en. In: *Comptes Rendus Geoscience* 351.1, pp. 48–58.

Aurenhammer, F. (1991). "Voronoi diagrams—a survey of a fundamental geometric data structure". en. In: *ACM Computing Surveys* 23.3, pp. 345–405.

Batista, V. H., D. L. Millman, S. Pion, and J. Singler (2010). "Parallel geometric algorithms for multi-core computers". en. In: *Computational Geometry* 43.8, pp. 663–677.

Baudouin, T. C., J.-F. Remacle, E. Marchandise, F. Henrotte, and C. Geuzaine (2014). "A frontal approach to hex-dominant mesh generation". In: *Advanced Modeling and Simulation in Engineering Sciences* 1.1, p. 1.

Beck, J. C., P. Prosser, and R. J. Wallace (2004). "Trying again to fail-first". In: *International Workshop on Constraint Solving and Constraint Logic Programming*. Springer, pp. 41–55.

Bell, N. and J. Hoberock (2011). "Thrust: A productivity-oriented library for CUDA". In: *GPU computing gems Jade edition*. Elsevier, pp. 359–371.

Below, A., J. A. De Loera, and J. Richter-Gebert (2004). "The complexity of finding small triangulations of convex 3-polytopes". In: *Journal of Algorithms* 50.2, pp. 134–167.

Benzley, S. E., E. Perry, K. Merkley, B. Clark, and G. Sjaardema (1995). "A comparison of all hexagonal and all tetrahedral finite element meshes for elastic and elasto-plastic analysis". In: *In Proceedings, 4th International Meshing Roundtable*, pp. 179–191.

Bern, M., D. Eppstein, and J. Erickson (2002). "Flipping cubical meshes". In: *Engineering with Computers* 18.3. Publisher: Springer, pp. 173–187.

Björner, A., ed. (1999). *Oriented matroids*. 2nd ed. Encyclopedia of mathematics and its applications v. 46. Cambridge ; New York: Cambridge University Press.

Blandford, D. K., G. E. Blelloch, and C. Kadow (2006). "Engineering a compact parallel delaunay algorithm in 3D". en. In: ACM Press, p. 292.

Blelloch, G. E., G. L. Miller, J. C. Hardwick, and D. Talmor (1999). "Design and Implementation of a Practical Parallel Delaunay Algorithm". en. In: *Algorithmica* 24.3-4, pp. 243–269.

Boissonnat, J.-D., O. Devillers, and S. Hornus (2009). "Incremental Construction of the Delaunay Triangulation and the Delaunay Graph in Medium Dimension". In: *Proceedings of the Twenty-fifth Annual Symposium on Computational Geometry*. SCG '09. event-place: Aarhus, Denmark. New York, NY, USA: ACM, pp. 208–216.

Boissonnat, J.-D., O. Devillers, M. Teillaud, and M. Yvinec (2000). "Triangulations in CGAL". In: *Proceedings of the sixteenth annual symposium on Computational geometry*. ACM, pp. 11–18.

Bokowski, J. and J. Richter (1990). "On the finding of final polynomials". In: *European Journal of Combinatorics* 11.1. Publisher: Elsevier, pp. 21–34.

Borden, M. J., S. E. Benzley, and J. F. Shepherd (2002). "Hexahedral Sheet Extraction." In: *IMR*, pp. 147–152.

Botella, A., B. Levy, and G. Caumon (2016). "Indirect unstructured hex-dominant mesh generation using tetrahedra recombination". en. In: *Computational Geosciences* 20.3, pp. 437–451.

Bowyer, A. (1981). "Computing Dirichlet tessellations". en. In: *The Computer Journal* 24.2, pp. 162–166.

Brinkmann, G., S. Greenberg, C. S. Greenhill, B. D. McKay, R. Thomas, and P. Wollan (2005). "Generation of simple quadrangulations of the sphere". In: *Discrete Mathematics* 305.1-3, pp. 33–54.

Brinkmann, G. and B. D. McKay (2007). "Fast generation of planar graphs". In: *MATCH Commun. Math. Comput. Chem* 58.2, pp. 323–357.

Burton, B. A. (2011). "The pachner graph and the simplification of 3-sphere triangulations". In: *Proceedings of the 27th Symposium on Computational Geometry*, pp. 153–162.

Carbonera, C. D. and J. F. Shepherd (2010). "A constructive approach to constrained hexahedral mesh generation". In: *Eng. Comput. (Lond.)* 26.4, pp. 341–350.

Caumon, G., P. Collon-Drouaillet, C. Le Carlier de Veslud, J. Sausse, and S. Viseur (2009). "Surface-based 3D modeling of geological structures". In: *Mathematical Geosciences* 41.9, pp. 927–945.

Caumon, G., F. Lepage, C. H. Sword, and J.-L. Mallet (2004). "Building and editing a sealed geological model". In: *Mathematical Geology* 36.4. Publisher: Springer, pp. 405–424.

Chandrupatla, T. R. and A. D. Belegundu (2011). *Introduction to finite elements in engineering*. 4th ed. Upper Saddle River,NJ: Prentice Hall.

Chauvin, B., J. Stockmeyer, J. H. Shaw, A. Plesch, J. Herbert, P. J. Lovely, et al. (2016). "Defining Proper Boundary Conditions in 3-D Structural Restoration: A Case Study Restoring a 3-D Forward Model of Suprasalt Extensional Structures". In: *AAPG Annual Convention and Exhibition*. Calgary, Canada: AAPG.

Chen, M. (2010). "The Merge Phase of Parallel Divide-and-Conquer Scheme for 3D Delaunay Triangulation". In: *International Symposium on Parallel and Distributed Processing with Applications*, pp. 224–230.

Chen, Z., J. Cao, and W. Wang (2012). "Isotropic Surface Remeshing Using Constrained Centroidal Delaunay Mesh". In: *Computer Graphics Forum* 31.7pt1, pp. 2077–2085.

Chrisochoides, N. and D. Nave (2003). "Parallel Delaunay mesh generation kernel". en. In: *International Journal for Numerical Methods in Engineering* 58.2, pp. 161–176.

Cignoni, P., C. Montani, R. Perego, and R. Scopigno (1993). "Parallel 3D Delaunay Triangulation". en. In: *Computer Graphics Forum* 12.3, pp. 129–142.

Colbourn, C. J. and K. S. Booth (1981). "Linear time automorphism algorithms for trees, interval graphs, and planar graphs". In: *SIAM J. Comput.* 10.1, pp. 203–225.

Colin de Verdière, E., G. Ginot, and X. Goaoc (2012). "Multinerves and helly numbers of acyclic families". In: *Proceedings of the 2012 symposuim on Computational Geometry*. ACM, pp. 209–218.

Colletta, B., J. Letouzey, R. Pinedo, J. F. Ballard, and P. Balé (1991). "Computerized X-ray tomography analysis of sandbox models: Examples of thin-skinned thrust systems". In: *Geology* 19.11, pp. 1063–1067.

Collon, P., W. Steckiewicz-Laurent, J. Pellerin, G. Laurent, G. Caumon, G. Reichart, et al. (2015). "3D geomodelling combining implicit surfaces and Voronoi-based remeshing: A case study in the Lorraine Coal Basin (France)". en. In: *Computers & Geosciences* 77, pp. 29–43.

Cupillard, P., A. Botella, and Y. Capdeville (2015). "Homogenization of 3d geological models for seismic wave propagation". In: *SEG Technical Program Expanded Abstracts*, pp. 3656–3660.

Dardenne, J., S. Valette, N. Siauve, N. Burais, and R. Prost (2009). "Variational tetrahedral mesh generation from discrete volume data". In: *The Visual Computer* 25.5-7, pp. 401–410.

De Floriani, L., B. Falcidieno, G. Nagy, and C. Pienovi (1991). "On sorting triangles in a delaunay tessellation". In: *Algorithmica* 6.1, pp. 522–532.

De Loera, J. A., J. Rambau, and F. Santos (2010). *Triangulations: structures for algorithms and applications*. Algorithms and computation in mathematics v. 25. OCLC: ocn646114288. Berlin ; New York: Springer.

Delaunay, B. (1934). "Sur la sphere vide". In: *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk* 7.793-800, pp. 1–2.

Devine, K. D., E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, et al. (2005). "New challenges in dynamic load balancing". In: *Applied Numerical Mathematics* 52.2-3. Publisher: Elsevier, pp. 133–152.

Dey, T. K. and J. A. Levine (2009). "Delaunay Meshing of Piecewise Smooth Complexes without Expensive Predicates". In: *Algorithms* 2.4, pp. 1327–1349.

Du, Q., V. Faber, and M. Gunzburger (1999). "Centroidal Voronoi Tesselations: Applications and Algorithms". In: *SIAM Review* 41.4, pp. 637–676.

Du, Q., M. Gunzburger, and L. Ju (2003). "Constrained centroidal Voronoi tessellations for surfaces". In: *SIAM Journal on Scientific Computing* 24.5, pp. 1488–1506.

Edelsbrunner, H., F. Preparata, and D. West (1990). "Tetrahedrizing point sets in three dimensions". en. In: *Journal of Symbolic Computation* 10.3-4, pp. 335–347.

Edelsbrunner, H. and N. Shah (1997). "Triangulating topological spaces". In: *International Journal of Computational Geometry & Applications* 7.4, pp. 365–378.

Edelsbrunner, H. and E. P. Mücke (1990). "Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms". In: *ACM Transactions on Graphics* 9.1, pp. 66–104.

Eppstein, D. (1999). "Linear complexity hexahedral mesh generation". In: *Computational Geometry* 12.1-2. Publisher: Elsevier, pp. 3–16.

Erickson, J. (2014). "Efficiently hex-meshing things with topology". In: *Discrete & Computational Geometry* 52.3, pp. 427–449.

Euler, N., C. H. Sword Jr, and J. C. Dulac (1998). "A new tool to seal a 3D earth model: a cut with constraints". In: *SEG Technical Program Expanded Abstracts 1998*. Society of Exploration Geophysicists, pp. 710–713.

Fahle, T., S. Schamberger, and M. Sellmann (2001). "Symmetry Breaking". In: *Principles and Practice of Constraint Programming*, pp. 93–107.

Fang, X., W. Xu, H. Bao, and J. Huang (2016). "All-hex meshing using closed-form induced polycube". en. In: *ACM Transactions on Graphics* 35.4, pp. 1–9.

Firsching, M. (2017). "Realizability and inscribability for simplicial polytopes via nonlinear optimization". In: *Mathematical Programming* 166.1-2, pp. 273–295.

Foteinos, P. and N. Chrisochoides (2012). "Dynamic Parallel 3D Delaunay Triangulation". en. In: *Proceedings of the 20th International Meshing Roundtable*. Ed. by W. R. Quadros. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 3–20.

Frey, P. J. and H. Borouchaki (1999). "Surface mesh quality evaluation". In: *International Journal for Numerical Methods in Engineering* 45.1, pp. 101–118.

Frey, P. J. and P.-L. George (2007). *Mesh Generation: Application to Finite Elements*. ISTE.

Fu, H., J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, et al. (2016). "The Sunway TaihuLight supercomputer: system and applications". In: *Science China Information Sciences* 59.7. Publisher: Springer, p. 072001.

Funar, L. (1999). “Cubulations mod bubble moves”. In: *Contemporary Mathematics* 233. Publisher: Providence, RI: American Mathematical Society, pp. 29–44.

Funke, D. and P. Sanders (2017). “Parallel *d* -D Delaunay Triangulations in Shared and Distributed Memory”. en. In: *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, pp. 207–217.

Furch, R. (1924). “Zur grundlegung der kombinatorischen topologie”. In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 3.1, pp. 69–88.

Gao, X., W. Jakob, M. Tarini, and D. Panozzo (2017). “Robust hex-dominant mesh generation using field-guided polyhedral agglomeration”. en. In: *ACM Transactions on Graphics* 36.4, pp. 1–13.

Garland, M. and P. S. Heckbert (1997). “Surface simplification using quadric error metrics”. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press, pp. 209–216.

Gent, I. P., K. E. Petrie, and J.-F. Puget (2006). “Symmetry in Constraint Programming”. In: *Handbook of Constraint Programming*, pp. 329–376.

Geuzaine, C. and J.-F. Remacle (2009). “Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities”. en. In: *International Journal for Numerical Methods in Engineering* 79.11, pp. 1309–1331.

Gilbert, E. G. and C.-P. Foo (1990). “Computing the distance between general convex objects in three-dimensional space”. In: *IEEE Transactions on Robotics and Automation* 6.1. Publisher: IEEE, pp. 53–61.

Gleixner, A., L. Eifler, T. Gally, G. Gamrath, P. Gemander, R. L. Gottwald, et al. (2017). *The SCIP Optimization Suite 5.0*. eng. Tech. rep. 17-61. Takustr.7, 14195 Berlin: ZIB.

Goaoc, X., P. Paták, Z. Patáková, M. Tancer, and U. Wagner (2018). “Shellability is NP-Complete”. In: *Proceedings of the 34th Symposium on Computational Geometry*, 41:1–41:15.

Gregson, J., A. Sheffer, and E. Zhang (2011). “All-hex mesh generation via volumetric polycube deformation”. In: *Comput. Graph. Forum* 30.5. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2011.02015.x, pp. 1407–1416.

Guzofski, C. A., J. P. Mueller, J. H. Shaw, P. Muron, D. A. Medwedeff, F. Bilotti, et al. (2009). “Insights into the mechanisms of fault-related folding provided by volumetric structural restorations using spatially varying mechanical constraints”. In: *AAPG Bulletin* 93.4, pp. 479–502.

Hamilton, C. H. and A. Rau-Chaplin (2008). “Compact Hilbert indices: Space-filling curves for domains with unequal side lengths”. In: *Information Processing Letters* 105.5, pp. 155–163.

Han, S., J. Xia, and Y. He (2011). “Constructing hexahedral shell meshes via volumetric polycube maps”. In: *Computer-Aided Design* 43.10, pp. 1222–1233.

Hatcher, A. (2002). *Algebraic topology*. Cambridge University Press, Cambridge.

Haverkort, H. J. and F. v. Walderveen (2010). “Locality and bounding-box quality of two-dimensional space-filling curves”. In: *Comput. Geom.* 43.2, pp. 131–147.

Huang, J., Y. Tong, H. Wei, and H. Bao (2011). “Boundary aligned smooth 3D cross-frame field”. en. In: ACM Press, p. 1.

Ibanez, D. A., E. S. Seol, C. W. Smith, and M. S. Shephard (2016). “PUMI: Parallel Unstructured Mesh Infrastructure”. en. In: *ACM Transactions on Mathematical Software* 42.3, pp. 1–28.

Ito, Y., A. M. Shih, and B. K. Soni (2009). “Octree-based reasonable-quality hexahedral mesh generation using a new set of refinement templates”. In: *Internat. J. Numer. Methods Engrg.* 77.13. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.2470, pp. 1809–1833.

Johnen, A., J.-F. Remacle, and C. Geuzaine (2013). “Geometrical validity of curvilinear finite elements”. en. In: *Journal of Computational Physics* 233, pp. 359–372.

Johnen, A., J.-C. Weill, and J.-F. Remacle (2017). “Robust and efficient validation of the linear hexahedral element”. In: *Procedia Engineering* 203. Publisher: Elsevier, pp. 271–283.

Jolley, S. J., D. Barr, J. J. Walsh, and R. J. Knipe (2007). “Structurally complex reservoirs: an introduction”. In: *Geological Society, London, Special Publications* 292.1, pp. 1–24.

Jordan, C., M. Joswig, and L. Kastner (2018). “Parallel Enumeration of Triangulations”. In: *Electr. J. Comb.* 25.3, P3.6.

Kohout, J., I. Kolingerová, and J. Žára (2005). “Parallel Delaunay triangulation in E2 and E3 for computers with shared memory”. en. In: *Parallel Computing* 31.5, pp. 491–522.

Kowalski, N., F. Ledoux, and P. Frey (2014). “Block-structured Hexahedral Meshes for CAD Models Using 3D Frame Fields”. In: *Procedia Engineering* 82, pp. 59–71.

Kruhl, J. H. (2013). “Fractal-geometry techniques in the quantification of complex rock structures: A special view on scaling regimes, inhomogeneity and anisotropy”. In: *Journal of Structural Geology* 46.0, pp. 2–21.

Lachat, C., C. Dobrzynski, and F. Pellegrini (2014). “Parallel mesh adaptation using parallel graph partitioning”. en. In: p. 13.

Laurent, G. (2013). “Prise en compte de l’histoire geologique des structures dans la creation de modeles numeriques 3D compatibles”. PhD Thesis. Universite de Lorraine, France.

Law, Y. C. and J. H. Lee (2004). “Global constraints for integer and set value precedence”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer, pp. 362–376.

Ledoux, F. and J. Shepherd (2010). “Topological modifications of hexahedral meshes via sheet operations: a theoretical study”. In: *Engineering with Computers* 26.4. Publisher: Springer, pp. 433–447.

Levy, B. (2016). *Geogram*.

Lévy, B. (2015). “Robustness and Efficiency of Geometric Programs The Predicate Construction Kit (PCK)”. In: *Computer-Aided Design*. Publisher: Elsevier.

Lévy, B. and N. Bonneel (2013). “Variational Anisotropic Surface Meshing with Voronoi Parallel Linear Enumeration”. In: *Proceedings of the 21st International Meshing Roundtable*. Ed. by X. Jiao and J.-C. Weill. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 349–366.

Levy, B. and Y. Liu (2010a). “*L* $_\textrm *p* $ Centroidal Voronoi Tessellation and its applications”. en. In: *ACM Transactions on Graphics* 29.4, p. 1.

— (2010b). “Lp Centroidal Voronoi Tesselation and its Applications”. In: *ACM Transactions on Graphics (SIGGRAPH conference proceedings)*.

Lieber, M., V. Grützun, R. Wolke, M. S. Müller, and W. E. Nagel (2010). “FD4: A Framework for Highly Scalable Load Balancing and Coupling of Multiphase Models”. In: *AIP Conference Proceedings* 1281.1. _eprint: https://aip.scitation.org/doi/pdf/10.1063/1.3498143, pp. 1639–1642.

Lindsay, M. D., M. W. Jessell, L. Ailleres, S. Perrouty, E. d. Kemp, and P. G. Betts (2013). “Geodiversity: Exploration of 3D geological model space”. In: *Tectonophysics* 594.0, pp. 27–37.

Liu, H., P. Zhang, E. Chien, J. Solomon, and D. Bommes (2018). “Singularity-constrained octahedral fields for hexahedral meshing”. In: *ACM Trans. Graph.* 37.4, 93:1–93:17.

Liu, Y., W. Wang, B. Lévy, F. Sun, and D.-M. Yan (2009). “On Centroidal Voronoi Tessellation - Energy Smoothness and Fast Computation.” In: *ACM Transactions Graphics*. 30p, pp. 1–17.

Livesu, M., A. Sheffer, N. Vining, and M. Tarini (2015). “Practical hex-mesh optimization via edge-cone rectification”. In: *ACM Trans. Graph.* 34.4, 141:1–141:11.

Lloyd, S. (1982). “Least squares quantization in PCM”. In: *Information Theory, IEEE Transactions on* 28.2, pp. 129–137.

Lo, S. (2012). “Parallel Delaunay triangulation in three dimensions”. en. In: *Computer Methods in Applied Mechanics and Engineering* 237-240, pp. 88–106.

Loseille, A., F. Alauzet, and V. Menier (2017). “Unique cavity-based operator and hierarchical domain partitioning for fast parallel generation of anisotropic meshes”. en. In: *Computer-Aided Design* 85, pp. 53–67.

Lutz, F. H. and T. Sulanke (2018). *The Manifold Page*.

Lyon, M., D. Bommes, and L. Kobbelt (2016). “HexEx: robust hexahedral mesh extraction”. en. In: *ACM Transactions on Graphics* 35.4, pp. 1–11.

Maréchal, L. (2009). “Advances in octree-based all-hexahedral mesh generation: Handling sharp features”. In: *Proceedings of the 18th International Meshing Roundtable*, pp. 65–84.

Marot, C., J. Pellerin, J. Lambrechts, and J.-F. Remacle (2017). “Toward one billion tetrahedra per minute”. In: *26th International Meshing Roundtable, Research Notes*. Barcelona, Spain.

Marot, C., J. Pellerin, and J. Remacle (2018). “One machine, one minute, three billion tetrahedra”. en. In: *International Journal for Numerical Methods in Engineering* 117.9, pp. 967–990.

Martinez-Rubi, O., S. Verhoeven, M. van Meersbergen, and P. van Oosterom (2016). “Taming the beast: Free and open-source massive point cloud web visualization”. en. In: p. 12.

Matoušek, J., M. Tancer, and U. Wagner (2009). “Hardness of embedding simplicial complexes in Rd”. In: *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, pp. 855–864.

Mazuyer, A., R. Giot, P. Cupillard, M. Conin, and P. Thore (2016). "Stress estimation in reservoirs by a stochastic inverse approach". In: *7th International Symposium on In-Situ Rock Stress*. International Society for Rock Mechanics.

McKay, B. D. and A. Piperno (2014). "Practical graph isomorphism, II". en. In: *Journal of Symbolic Computation* 60, pp. 94–112.

Meshkat, S. and D. Talmor (2000). "Generating a mixed mesh of hexahedra, pentahedra and tetrahedra from an underlying tetrahedral mesh". en. In: *International Journal for Numerical Methods in Engineering* 49.1-2, pp. 17–30.

Mitchell, S. A. (1996). "A characterization of the quadrilateral meshes of a surface which admit a compatible hexahedral mesh of the enclosed volume". In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, pp. 465–476.

Mitchell, S. A. and T. J. Tautges (1995). "Pillowing doublets: refining a mesh to ensure that faces share at most one edge". In: *4th International Meshing Roundtable*. Citeseer, pp. 231–240.

Müller–Hannemann, M. (1999). "Hexahedral mesh generation by successive dual cycle elimination". In: *Eng. Comput. (Lond.)* 15.3. Publisher: Springer, pp. 269–279.

Murdoch, P., S. Benzley, T. Blacker, and S. A. Mitchell (1997). "The spatial twist continuum: A connectivity based method for representing all-hexahedral finite element meshes". In: *Finite elements in analysis and design* 28.2. Publisher: Elsevier, pp. 137–149.

Mustapha, H., R. Dimitrakopoulos, T. Graf, and A. Firoozabadi (2011). "An efficient method for discretizing 3D fractured media for subsurface flow and transport simulations". In: *International Journal for Numerical Methods in Fluids* 67.5, pp. 651–670.

Nieser, M., U. Reitebuch, and K. Polthier (2011). "CubeCover- parameterization of 3D volumes". In: *Comput. Graph. Forum* 30.5, pp. 1397–1406.

Nocedal, J. (1980). "Updating quasi-Newton matrices with limited storage". In: *Mathematics of computation* 35.151, pp. 773–782.

Nystrom, N. A., M. J. Levine, R. Z. Roskies, and J. R. Scott (2015). "Bridges: A Uniquely Flexible HPC Resource for New Communities and Data Analytics". In: *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*. XSEDE '15. event-place: St. Louis, Missouri. New York, NY, USA: ACM, 30:1–30:8.

Okabe, A., B. Boots, K. Sugihara, and S. N. Chiu (2009). *Spatial tessellations: concepts and applications of Voronoi diagrams*. Vol. 501. Wiley.

Okusanya, T. and J. Peraire (1996). *3D PARALLEL UNSTRUCTURED MESH GENERATION*. en. Citeseer.

Owen, S. J. and S. Saigal (2000). "H-Morph: an indirect approach to advancing front hex meshing". en. In: *International Journal for Numerical Methods in Engineering* 49.1-2, pp. 289–312.

Owen, S. J., S. A. Canann, and S. Saigal (1997). "Pyramid elements for maintaining tetrahedra to hexahedra conformability". In: *ASME Applied Mechanics Division-Publications-AMD* 220, pp. 123–130.

Paradigm (2016). *SKUA-GOCAD*.

Pellerin, J., Arnaud Botella, A. Mazuyer, B. Chauvin, B. Levy, and G. Caumon (2015). "RINGMesh A programming library for geological model meshes". In: *Proc. 17th IAMG conference*. Freiberg, Germany.

Pellerin, J., A. Botella, F. Bonneau, A. Mazuyer, B. Chauvin, B. Lévy, et al. (2017). "RINGMesh: A programming library for developing mesh-based geomodeling applications". en. In: *Computers & Geosciences* 104, pp. 93–100.

Pellerin, J., G. Caumon, C. Julio, P. Mejia-Herrera, and A. Botella (2015). "Elements for measuring the complexity of 3D structural models: Connectivity and geometry". In: *Computers & Geosciences* 76. Publisher: Elsevier, pp. 130–140.

Pellerin, J., A. Johnen, and J.-F. Remacle (2017). "Identifying combinations of tetrahedra into hexahedra: a vertex based strategy". en. In: *Procedia Engineering* 203. Proceedings of the 26th International Meshing Roundtable, Barcelona, Spain, pp. 2–13.

Pellerin, J., A. Johnen, K. Verhetsel, and J.-F. Remacle (2018). "Identifying combinations of tetrahedra into hexahedra: A vertex based strategy". en. In: *Computer-Aided Design* 105, pp. 1–10.

Pellerin, J., B. Lévy, and G. Caumon (2011). "Topological control for isotropic remeshing of nonmanifold surfaces with varying resolution: application to 3D structural models". In: *Proc. 13th IAMG conference*. Salzburg, Austria.

— (2012). “A Voronoi-Based Hybrid Meshing Method”. In: *21st International Meshing Roundtable, Research Notes*. San Jose, CA, USA.
— (2014). “Toward Mixed-element Meshing based on Restricted Voronoi Diagrams”. en. In: *Procedia Engineering* 82. Proceedings of the 23rd International Meshing Roundtable, London, UK, pp. 279–290.
Pellerin, J., B. Lévy, G. Caumon, and A. Botella (2014). “Automatic surface remeshing of 3D structural models at specified resolution: A method based on Voronoi diagrams”. In: *Computers & Geosciences* 62.0, pp. 103–116.
Pellerin, J., K. Verhetsel, and J.-F. Remacle (2018). “There are 174 subdivisions of the hexahedron into tetrahedra”. en. In: *ACM Transactions on Graphics* 37.6, pp. 1–9.
Peyrusse, F., N. Glinsky, C. Gélis, and S. Lanteri (2014). “A nodal discontinuous Galerkin method for site effects assessment in viscoelastic media—verification and validation in the Nice basin”. en. In: *Geophysical Journal International* 199.1, pp. 315–334.
Pfeifle, J. and J. Rambau (2003). “Computing Triangulations Using Oriented Matroids”. en. In: *Algebra, Geometry and Software Systems*. Ed. by M. Joswig and N. Takayama. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 49–75.
Polychroniou, O. and K. A. Ross (2014). “A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort”. In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pp. 755–766.
Qian, J. and Y. Zhang (2010). “Sharp feature preservation in octree-based hexahedral mesh generation for CAD assembly models”. In: *Proceedings of the 19th International Meshing Roundtable*, pp. 243–262.
Quadros, W. R., S. J. Owen, M. L. Brewer, and K. Shimada (2004). “Finite Element Mesh Sizing for Surfaces Using Skeleton.” In: *Proc. 13th International Meshing Roundtable*, pp. 389–400.
Rambau, J. (2003). “On a generalization of Schönhardt’s polyhedron”. In: *Combinatorial and computational geometry* 52, pp. 510–516.
Rasquin, M., C. Smith, K. Chitale, S. Seol, B. A. Matthews, J. L. Martin, et al. (2014). “Scalable fully implicit finite element flow solver with application to high-fidelity flow control simulations on a realistic wing design”. en. In: *Computing in Science and Engineering* 16, p. 7.
Régin, J.-C., M. Rezgui, and A. Malapert (2013). “Embarrassingly parallel search”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer, pp. 596–610.
Remacle, J.-F., J. Lambrechts, B. Seny, E. Marchandise, A. Johnen, and C. Geuzainet (2012). “Blossom-Quad: A non-uniform quadrilateral mesh generator using a minimum-cost perfect-matching algorithm”. en. In: *International Journal for Numerical Methods in Engineering* 89.9, pp. 1102–1119.
Remacle, J.-F. (2017). “A two-level multithreaded Delaunay kernel”. en. In: *Computer-Aided Design* 85, pp. 2–9.
Remacle, J.-F., R. Gandham, and T. Warburton (2016). “GPU accelerated spectral finite elements on all-hex meshes”. In: *Journal of Computational Physics* 324, pp. 246–257.
Rohmer, O., E. Bertrand, E. Mercerat, J. Régnier, M. Pernoud, P. Langlaude, et al. (2020). “Combining borehole log-stratigraphies and ambient vibration data to build a 3D Model of the Lower Var Valley, Nice (France)”. en. In: *Engineering Geology* 270, p. 105588.
Rossi, F., P. v. Beek, and T. Walsh, eds. (2006). *Handbook of Constraint Programming*. Vol. 2. Foundations of Artificial Intelligence. Elsevier.
Rossignac, J. (2005). “Shape complexity”. In: *The visual computer* 21.12, pp. 985–996.
Rossignac, J. R. and P. Borrel (1993). “Multi-resolution 3D approximations for rendering complex scenes”. In: *Geometric Modeling in Computer Graphics*. Ed. by B. Falcidiendo and T. L. Kunii. Springer, pp. 455–465.
Ruppert, J. and R. Seidel (1992). “On the difficulty of triangulating three-dimensional Nonconvex Polyhedra”. en. In: *Discrete & Computational Geometry* 7.3, pp. 227–253.
Salles, L., M. Ford, P. Joseph, C. De Veslud, and A. Le Solleuz (2011). “Migration of a synclinal depocentre from turbidite growth strata: the Annot syncline, SE France”. In: *Bulletin de la Sociéte Géologique de France* 182.3, pp. 199–220.
Satish, N., M. J. Harris, and M. Garland (2009). “Designing efficient sorting algorithms for manycore GPUs”. In: *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pp. 1–10.

Satish, N., C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, et al. (2010). "Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. event-place: Indianapolis, Indiana, USA. ACM, pp. 351–362.

Schewe, L. (2010). "Nonrealizable minimal vertex triangulations of surfaces: showing nonrealizability using oriented matroids and satisfiability solvers". In: *Discrete & Computational Geometry* 43.2. Publisher: Springer, pp. 289–302.

Schleimer, S. (2011). "Sphere recognition lies in NP". In: *Low-dimensional and symplectic topology*. Vol. 82. Proc. Sympos. Pure Math. Amer. Math. Soc., Providence, RI, pp. 183–213.

Schneiders, R. (1996). "A grid-based algorithm for the generation of hexahedral element meshes". en. In: *Engineering with Computers* 12.3-4, pp. 168–177.

Schneiders, R. (1995). *Open problem*.

Schwartz, A. and G. M. Ziegler (2004). "Construction techniques for cubical complexes, odd cubical 4-polytopes, and prescribed dual manifolds". In: *Experimental Mathematics* 13.4. Publisher: Taylor & Francis, pp. 385–413.

Sengupta, S., M. J. Harris, Y. Zhang, and J. D. Owens (2007). "Scan primitives for GPU computing". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2007, San Diego, California, USA, August 4-5, 2007*, pp. 97–106.

Shepherd, J. F. and C. R. Johnson (2008). "Hexahedral mesh generation constraints". en. In: *Engineering with Computers* 24.3, pp. 195–213.

Shewchuk, J. (1997). "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates". en. In: *Discrete & Computational Geometry* 18.3, pp. 305–363.

Shewchuk, J. R. (1997). *Delaunay refinement mesh generation*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE.

Shewchuk, J. R. (1996). "Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator". In: *Applied computational geometry towards geometric engineering*. Springer, pp. 203–222.

Si, H. (2015). "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator". en. In: *ACM Transactions on Mathematical Software* 41.2, pp. 1–36.

Slotnick, J., A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie, et al. (2014). "CFD vision 2030 study: a path to revolutionary computational aerosciences". In:

Sohn, A. and Y. Kodama (1998). "Load balanced parallel radix sort". In: *Proceedings of the 12th international conference on Supercomputing*. ACM, pp. 305–312.

Sokolov, D., N. Ray, L. Untereiner, and B. Levy (2016). "Hexahedral-Dominant Meshing". en. In: *ACM Transactions on Graphics* 35.5, pp. 1–23.

Sokolov, D., N. Ray, L. Untereiner, and B. Lévy (2017). "Hexahedral-dominant meshing". In: *ACM Trans. Graph.* 36.4.

Solomon, J., A. Vaxman, and D. Bommes (2017). "Boundary Element Octahedral Fields in Volumes". en. In: *ACM Transactions on Graphics* 36.3, pp. 1–16.

Sorensson, N. and N. Een (2005). "Minisat v1. 13-a sat solver with conflict-clause minimization". In: *SAT* 2005.53, pp. 1–2.

Su, T., W. Wang, Z. Lv, W. Wu, and X. Li (2016). "Rapid Delaunay triangulation for randomly distributed point cloud data using adaptive Hilbert curve". en. In: *Computers & Graphics* 54, pp. 65–74.

Sukumar, S. R., D. L. Page, A. F. Koschan, and M. A. Abidi (2008). "Towards understanding what makes 3D objects appear simple or complex". In: IEEE, pp. 1–8.

Tautges, T. J., T. Blacker, and S. A. Mitchell (1996). "The whisker weaving algorithm: a connectivity-based method for constructing all-hexahedral finite element meshes". In: *International Journal for Numerical Methods in Engineering* 39.19, pp. 3327–3349.

Tautges, T. J. and S. E. Knoop (2003). "Topology modification of hexahedral meshes using atomic dual-based operations". In: *algorithms* 11, p. 12.

The CGAL Project (2016). *CGAL*.

Thurston, W. P. (1993). *Hexahedral decomposition of polyhedra*. Published: Posting to sci.math.

Toulorge, T., C. Geuzaine, J.-F. Remacle, and J. Lambrechts (2013). "Robust untangling of curvilinear meshes". In: *J. Comput. Physics* 254, pp. 8–26.

Tournois, J. (2009). "Optimisation de maillages". Anglais. Thèse. Université Nice Sophia Antipolis.

Tournois, J., C. Wormser, P. Alliez, and M. Desbrun (2009). "Interleaving Delaunay refinement and optimization for practical isotropic tetrahedron mesh generation". In: *ACM Transactions on Graphics* 28.3, p. 1.

Valette, S., J.-M. Chassery, and R. Prost (2008). "Generic remeshing of 3D triangular meshes with metric-dependent discrete voronoi diagrams". In: *IEEE Transactions on Visualization and Computer Graphics* 14.2, pp. 369–381.

Vázquez, M., G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Arís, et al. (2016). "Alya: Multiphysics engineering simulation toward exascale". In: *Journal of Computational Science*. The Route to Exascale: Novel Mathematical Methods, Scalable Algorithms and Computational Science Skills 14, pp. 15–27.

Verhetsel, K., J. Pellerin, A. Johnen, and J.-F. Remacle (2017). "Solving the Maximum Weight Independent Set Problem: Application to Indirect Hexahedral Mesh Generation". In: *26th International Meshing Roundtable, Research Notes*. Barcelona, Spain.

Verhetsel, K., J. Pellerin, and J.-F. Remacle (2018). "A 44-element mesh of Schneiders' pyramid: Bounding the difficulty of hex-meshing problems". In: *Proceedings of the 27th International Meshing Roundtable*. Springer.

— (2019a). "A 44-element mesh of Schneiders' pyramid: Bounding the difficulty of hex-meshing problems". en. In: *Computer-Aided Design* 116, p. 102735.

— (2019b). "Finding hexahedrizations for small quadrangulations of the sphere". en. In: *ACM Transactions on Graphics* 38.4, pp. 1–13.

Voronoï, G. (1908). "Nouvelles applications des paramètres continus à la théorie des formes quadratiques. {D}euxième mémoire. {R}echerches sur les parallélloèdres primitifs." In: *Journal für die reine und angewandte Mathematik* 134, pp. 198–287.

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

Wassenberg, J. and P. Sanders (2011). "Engineering a Multi-core Radix Sort". In: *Euro-Par 2011 Parallel Processing*. Ed. by E. Jeannot, R. Namyst, and J. Roman. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 160–169.

Watson, D. F. (1981). "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes". en. In: *The Computer Journal* 24.2, pp. 167–172.

White, D. R., S. Saigal, and S. J. Owen (2005). "Meshing complexity: predicting meshing difficulty for single part CAD models". In: *Engineering with Computers* 21.1, pp. 76–90.

Wu, Q. and J.-K. Hao (2015). "A review on algorithms for maximum clique problems". en. In: *European Journal of Operational Research* 242.3, pp. 693–709.

Xiang, S. and J. Liu (2018). "A 36-Element Solution To Schneiders' Pyramid Hex-Meshing Problem And A Parity-Changing Template For Hex-Mesh Revision". In: *arXiv preprint arXiv:1807.09415*.

Yamakawa, S. and K. Shimada (2002). "Hexhoop: modular templates for converting a hex-dominant mesh to an all-hex mesh". In: *Engineering with computers* 18.3. Publisher: Springer, pp. 211–228.

— (2003). "Fully-automated hex-dominant mesh generation with directionality control via packing rectangular solid cells". In: *International journal for numerical methods in engineering* 57.15, pp. 2099–2129.

— (2010). "88-Element solution to Schneiders' pyramid hex-meshing problem". en. In: *International Journal for Numerical Methods in Biomedical Engineering* 26, pp. 1700–1712.

Yan, D.-M., B. Lévy, Y. Liu, F. Sun, and W. Wang (2009). "Isotropic Remeshing with Fast and Exact Computation of Restricted Voronoi Diagram". In: *ACM/EG Symposium on Geometry Processing / Computer Graphics Forum* 28.5, pp. 1145–1454.

Yu, W., K. Zhang, S. Wan, and X. Li (2014). "Optimizing polycube domain construction for hexahedral remeshing". In: *Computer-Aided Design* 46, pp. 58–68.

Zagha, M. and G. E. Blelloch (1991). "Radix sort for vector multiprocessors". In: *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*, pp. 712–721.

Zhang, Y., X. Liang, and G. Xu (2012). "A robust 2-refinement algorithm in octree and rhombic dodecahedral tree based all-hexahedral mesh generation". In: *Proceedings of the 21st International Meshing Roundtable*, pp. 155–172.

Ziegler, G. M. (1995). *Lectures on Polytopes*. Vol. 152. Graduate Texts in Mathematics. New York, NY: Springer New York.