PhD Thesis

Robust volume mesh generation for non-watertight geometries

Abel Coll Sans

Supervisors: Pooyan Dadvand
Eugenio Oñate Ibáñez de Navarra

May 2014

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Acta de qualificació de tesi doctoral

Curs acadèmic:

Nom i cognoms

Programa de doctorat

Unitat estructural responsable del programa

Resolució del Tribunal

Reunit el Tribunal designat a l'efecte, el doctorand / la doctoranda exposa el tema de la seva tesi doctoral titulada ____
____.

Acabada la lectura i després de donar resposta a les qüestions formulades pels membres titulars del tribunal, aquest atorga la qualificació:

☐ NO APTE ☐ APROVAT ☐ NOTABLE ☐ EXCEL·LENT

(Nom, cognoms i signatura) President/a	(Nom, cognoms i signatura) Secretari/ària	
(Nom, cognoms i signatura) Vocal	(Nom, cognoms i signatura) Vocal	(Nom, cognoms i signatura) Vocal

______________________, _______ d'/de __________________ de _______________

El resultat de l'escrutini dels vots emesos pels membres titulars del tribunal, efectuat per l'Escola de Doctorat, a instància de la Comissió de Doctorat de la UPC, atorga la MENCIÓ CUM LAUDE:

☐ SÍ ☐ NO

(Nom, cognoms i signatura) President de la Comissió Permanent de l'Escola de Doctorat	(Nom, cognoms i signatura) Secretària de la Comissió Permanent de l'Escola de Doctorat

Barcelona, _______ d'/de ____________________ de _________

A mi padre.

Agradecimientos

Con estas líneas quiero agradecer a todos aquellos que, de alguna manera, han contribuido a la realización de este trabajo.

En primer lugar, a los tutores de esta Tesis. A Pooyan por su paciencia y su tiempo a la hora de hacer el seguimiento de la Tesis. He tenido como tutor a un gran ingeniero y compañero de trabajo, pero ante todo, a un gran amigo. A Eugenio por su apoyo incondicional y su tenacidad a la hora de animarme a hacer la Tesis. Es una suerte tener un jefe que te dé los niveles de libertad, confianza y responsabilidad que él me ha dado.

A Riccardo, le agradezco su atención y discusiones. Me ayudó a superar uno de los últimos embrollos técnicos que estancaron el avance de la Tesis.

Algunos de los ejemplos de validación utilizados en este trabajo han sido facilitados por Pere-Andreu, Quantech y Barcelona Media. Muchas gracias a todos ellos por su colaboración.

También agradezco la ayuda incondicional de Rosa y Mercè en todos los temas administrativos con los que he tenido que lidiar. Este siempre es un trabajo tedioso y gracias a ellas ha sido mucho más fácil.

Hace ya casi 10 años que empecé mi etapa en CIMNE dentro del *GiD Team*. Esta Tesis no hubiera sido posible sin la experiencia adquirida en este brillante equipo. A sus miembros va mi más sincero agradecimiento, en especial a Quique por sus fructíferas discusiones en temas de geometría y mallado, a Miguel por su paciencia a la hora de adentrarme en el mundo de la programación, y a Ramón por su forma de hacerme ver algunos aspectos desde otro punto de vista y sus interesantes opiniones. Por encima de las cuestiones técnicas, lo que más agradezco del equipo de GiD es su calidad humana y el fantástico ambiente de trabajo.

Dicho entorno no podría entenderse fuera de CIMNE. Es muy larga la lista de compañeros y amigos que hacen (o han hecho) que el ambiente en CIMNE sea algo especial: Pooyan, Riccardo, Quique, Miguel, Anna, Adrià, Miquel Àngel, Aleix, Antonia, Temo, Edu, Javier Mora, Pablo, Jordi Cotela, Pere-Andreu, Salva, Guillermo, Jorge, Roberto, Carlos Labra y Hiram entre otros. Juegan un papel especial los cafés y las cervezas en el bar de Caminos (muchas gracias Toni, Jordi, Paco y Manoli), y sobretodo el momento de *la última* en el Tritón. Grandes tardes y noches rodeados de grandes amigos.

Tampoco puedo olvidarme de las actividades fuera del trabajo que he compartido con ellos estos años: las timbas de poker, las inolvidables bravas y *temos* de los jueves en bici al Tibidabo, los mediodías de fin de semana en el frontón, las bici+paellas, las Crossing Teams y los asados y calotadas.

Termino estos agradecimientos con unas palabras para mi familia, que siempre ha estado a mi lado: mis padres, mi hermana, Núria y Ferran. En especial a mi padre, que me inculcó su punto de vista ingenieril de las cosas: seguro que le hubiera gustado ver este trabajo terminado.

A todos ellos, muchas gracias.

Abstract

Nowadays large part of the time needed to perform a numerical simulation is spent in preprocessing, especially in the geometry cleaning operations and mesh generation. Furthermore, these operations are not easy to automatize because they depend strongly on each geometrical model and they often need human interaction. Many of these operations are needed to obtain a watertight geometry. Even with a clean geometry, classical unstructured meshing methods (like Delaunay or Advancing Front based ones) present critical weak points like the need of a given quality in the boundary mesh or a relatively smooth size transition. These aspects decrease their robustness and imply an extra effort in order to reach the final mesh. Octree based meshers try to relax some of these requirements.

In the present work an octree based mesher for unstructured tetrahedra is presented. The proposed mesher ensures the mesh generation avoiding most of the geometry cleaning operations. It is based in the following steps: fit an octree onto the model, refine it following given criteria, apply a tetrahedra pattern to the octree cells and adapt the tetrahedra close to the contours in order to represent accurately the boundary shape. An important and innovative aspect of the proposed algorithm is it ensures the final mesh preserves the topology and the geometric features of the original model.

The method uses a Ray Casting based algorithm for the identification of the inner and outer parts of the volumes involved in the model. This technique allows the mesh generation of volumes even with non-watertight boundaries, and also opens the use of the mesher for immersed methods only applying slight modifications to the algorithm.

The main advantages of the presented mesher are: robustness, no need for watertight boundaries, independent on the contour mesh quality, preservation of geometrical features (corners and ridges), original geometric topology guaranteed, accurate representation of the contours, valid for immersed methods, and fast performance. A lot of time in the preprocessing part of the numerical simulation is saved thanks to the robustness of the mesher, which allows skipping most of the geometry cleaning operations.

A shared memory parallel implementation of the algorithm has been done. The effectiveness of the algorithm and its implementation has been verified by some validation examples.

Resum

En l'actualitat gran part del temps emprat per córrer una simulació numèrica està dedicat al preprocés, especialment a les operacions de neteja de geometria i generació de malla. A més, aquestes operacions no són fàcils d'automatitzar degut a la seva forta dependència del model geomètric i sovint necessiten d'interacció humana. Moltes d'aquestes operacions són necessàries per aconseguir una definició topològicament hermètica de la geometria. Inclús amb una geometria neta, els mètodes clàssics de mallat (com els basats en Delaunay o avançament frontal) presenten punts febles crítics com la necessitat d'una certa qualitat de les malles de contorn o una transició de mides relativament suau. Aquests aspectes disminueixen la seva robustesa i impliquen un esforç extra a l'hora d'obtenir la malla final. Els mètodes de mallat basats en estructures *octree* relaxen alguns d'aquests requeriments.

En aquest treball es presenta un mallador basat en octree per tetraedres no estructurats. Un dels aspectes claus d'aquest mallador és que garanteix la generació de malla evitant moltes de les operacions de neteja de geometria. Es basa en els següents passos: encaixar un octree al model, refinar-lo seguint certs criteris, aplicar un patró de tetraedres a les cel·les de l'octree i adaptar-los a les zones properes als contorns a fi i efecte de representar acuradament la forma del domini. Un aspecte important i innovador de l'algorisme proposat és que manté la topologia del model a la malla final i preserva les seves característiques geomètriques.

El mètode presentat utilitza un algorisme basat en la tècnica *Ray Casting* per la identificació de les parts interiors i exteriors dels volums del model. Aquesta tècnica permet la generació de malla de volums inclús amb contorns que no tanquen hermèticament, i també obre l'ús del mallador a mètodes *immersed* aplicant només petites modificacions a l'algorisme.

Els principals avantatges del mallador presentat són: robustesa, no necessitat de definicions hermètiques dels contorns, independent de la qualitat de la malla de contorn, preservació de característiques geomètriques (cantonades i arestes abruptes), topologia original de la geometria garantida, representació precisa dels contorns, vàlid per mètodes *immersed* i ràpid rendiment. L'ús del mallador estalvia molt de temps en la part del preprocés de la simulació numèrica gràcies a la seva robustesa que permet obviar la majoria d'operacions de neteja de geometria.

S'ha dut a terme una implementació paral·lela amb memòria compartida de l'algorisme. L'efectivitat del mateix i la seva implementació ha estat verificada mitjançant exemples de validació.

Contents

1 Introduction **1**

1.1 Motivation 4

1.2 Objectives 6

1.2.1 Mesher requirements 6

1.2.2 Input data requirements 7

1.2.3 Final mesh requirements 9

1.2.4 Surface meshing 15

1.3 Structure of the thesis 17

2 State of the art on mesh generation **21**

2.1 Mesh types 21

2.2 Structured meshers 22

2.3 Advancing front method 24

2.4 Delaunay method 26

2.5 Space decomposition methods 28

2.6 Proposed solution 33

3 Basic concepts of the new mesher **35**

3.1 Definition of input data 35

3.1.1 Integration with CAD data 38

3.2 Octree structure 39

3.3 Specific octree properties for mesh generation 41

3.3.1 Octree cell types 42

3.3.2 Octree positions and nodes 43

3.3.3 Tetrahedra patterns 44
3.4 Geometrical intersections 47
3.4.1 Non-watertight geometries 50

4 Coloring algorithm **53**
4.1 Coloring strategies 54
4.1.1 Ray casting 56
4.2 Ray casting-based proposed technique 57
4.2.1 Pathological configurations 58
4.2.2 Adaptations of the method 60
4.3 Implementation of nodes coloring algorithm 62
4.3.1 Pathological situations 64
4.3.2 Computation of distances 69

5 Octree based mesher **73**
5.1 Introduction 73
5.2 Embedded mesher 75
5.2.1 Computation of distances 75
5.2.2 Octree refinement criteria for embedded meshes 76
5.2.3 Octree root 79
5.2.4 Meshing algorithm 80
5.3 Body-fitted mesher 83
5.3.1 Forced edges 84
5.3.2 Forced nodes 90
5.3.3 Octree refinement criteria for body-fitted meshes 92
5.3.4 Meshing algorithm 96
5.3.5 Preserve geometric features 101
5.3.6 Surface fitting 106
5.3.7 Tetrahedra coloring 111
5.3.8 Make-up and smoothing 119
5.3.9 Mesh quality 123
5.3.10 Extension for surface meshing 124

6 Implementation aspects 129
6.1 General aspects 129
6.2 Octree implementation 132
6.3 Generalized mesh size points 134
6.4 Sizes transition function 137
6.5 Body-fitted mesher 142
6.5.1 Geometrical transformations to the input data 145
6.5.2 Octree refinement 147
6.5.3 Tetrahedra coloring 147
6.5.4 Inner surface meshing 149
6.6 Parallel processing 150
6.7 Parameters used 151

7 Examples 155
7.1 Validation examples 159
7.1.1 Preserving topology 160
7.1.2 Preserving geometrical features 169
7.1.3 Non-watertight boundaries 174
7.1.4 Coloring pathological situations 179
7.1.5 Surface mesh inner to a volume 187
7.1.6 Embedded mesh 189
7.1.7 Performance 192
7.2 Racing car example 200
7.3 Barcelona city model 206
7.4 General overview 218
7.4.1 Comparison with other methods 221

8 Conclusions and future lines 223
8.1 Conclusions 223
8.2 Future research lines 226

A Profiling tables and complete data of examples 237
A.1 Validation example $VE - T1$ 237

A.2 Validation example $VE - T2$ 240
A.3 Validation example $VE - T3$ 242
A.4 Validation example $VE - F1$ 244
A.5 Validation example $VE - F2$ 246
A.6 Validation example $VE - W1$ 248
A.7 Validation example $VE - W2$ 250
A.8 Validation example $VE - W3$ 252
A.9 Validation example $VE - C1$ 254
A.10 Validation example $VE - C2$ 256
A.11 Validation example $VE - C3$ 258
A.12 Validation example $VE - I1$ 260
A.13 Validation example $VE - E1$ 262
A.14 Validation example $VE - S1$ 263
A.15 Validation example $VE - S2$ 265
A.16 Racing car 267
A.17 Barcelona model 269

Chapter 1

Introduction

Numerical simulations try to reproduce virtually a physical behavior by solving given equations in a specific domain. They are nowadays essential to understand some complex physical problems in scientific and engineering field. Although experimental setups can be build to study the specific behavior of a given phenomena, sometimes it is hard for these experiments (or even impossible depending on the scale of the tackled problem) to represent it accurately. The increasing advances in terms of computer science technology allow to treat larger and larger problems virtually (in the computer), so each time more and more numerical methods have been developed in the scientific field in order to capture the physics of complex problems.

Together with these developments, the adoption of numerical simulations in industrial processes has became a reality, as it can save a lot of time and effort when evaluating possible solutions for a given problem. The use of numerical simulation tools by the industry requires a software with very high level of robustness, efficiency and performance.

The process to run a numerical simulation involves three main parts: pre-processing, calculation and post-processing. The pre-processing part includes all the operations needed to define and discretize the geometrical domain, assign the required data to it so that the solver can solve the corresponding equations representing a given physical problem (in the calculation part). The post-processing part tries to analyze and visualize the results from the solver in a smart way so that they can be correctly interpreted. This thesis is focused on the pre-processing part of the numerical simulations, and specifically on the discretization of the geometrical domain.

The pre-processing operations can be summarized as follows:

- *Geometrical definition of the domain.* This part of the process is not always performed by the person responsible to run the simulation, and often a third part (a designer) creates the CAD definition of the geometry. Most of the times, this geometry creation does not take into account the requirements of numerical simulation, so a *geometry cleaning* process is needed in order to be able to perform the discretization of the domain.

- *Discretization of the domain.* Depending on the nature of the numerical method used for the simulation, different kinds of discretization of the geometrical domain are required. Meshless methods use a collection of points (nodes) as a discretization of the domain. Discrete Element Methods (DEM) use a collection of topologically unconnected objects for this purpose. In many occasions (like in the Finite Element Method (FEM) [ZTZ05, Oña09], Finite Volumes (FV) or Finite Differences (FD)) the result of the discretization is a mesh: a collection of polygons (in 2D) or polyhedra (in 3D) occupying the space where the domain is. This work focuses on this type of discretization, so in this document the discretization part of the pre-processing operations will be referred as *mesh generation.*

- *Assignation of the data needed to run the simulation.* This data can be of different nature depending on the kind of numerical simulation to be run, but most of the times includes the material properties of the different parts of the domain, the initial and boundary conditions for the equations to be solved, and some general parameters for the simulation.

The presented work proposes a new algorithm for mesh generation: a *mesh generator* (or simply a *mesher*). For designing a mesh generator, the basic requirements to be covered must be very clear. Missing the right requirements for the numerical simulation may lead to several limitations in the use of the mesher. There are three basic requirements to be covered by any mesh generator:

- *Input data requirements*: these requirements fix the characteristics of the input data able to be processed by the mesher. In this sense the input data is understood as the geometrical definition of the domain into which the simulation will be run. Typical requirements of this nature are: allow only a given mathematical definition of the geometry (NURBS, mesh, etc.), a specific topology of the input geometrical entities (watertight geometries, contact entities, etc.) or sizes of input entities inside a given range, among others. Considering the large number of CAD systems available and their different ways for geometrical definition of the domain, these kinds of requirements are in practice really important. Many times the person in charge of the numerical simulation may have no control on the geometrical definition. The situation where the mesher is very restrictive with the quality or the topology of the geometrical definition of the domain, often leads to a considerable increase of the effort in the geometry cleaning operations before the meshing itself. These requirements are not theoretically taken into account to evaluate if a mesher is suitable to generate meshes for given kinds of simulations or not, but in practice they can limit the use of the mesher as the effort dedicated to geometry cleaning operations can grow exponentially when the input geometries become more complex.

- *Final mesh requirements*: these requirements focus on the specific characteristics of the mesh required by the simulation to be run successfully. Typical requirements of this nature are: uniform elements sizes distribution, control of the mesh sizes in different zones of the domain or precise representation of the contours among others. Each kind of numerical simulation (structural dynamics, computational fluid mechanics (CFD), electromagnetism, etc.) has specific requirements for the mesh to be used. If a numerical simulation has one requirement of this nature for the final mesh and it is not covered by a mesher, this mesh generator is not suitable for the simulation.
- *Mesher requirements*: these requirements deal with the mesher behavior itself. The mesher can cover the two kinds of requirements presented before in several ways (with different implementations). The use of the mesher in practice often fixes some requirements in terms of speed, usability or memory availability which are also important.

Often, a mesh generator is focused in one type of mesh. Different kinds of meshes can be identified depending on their nature:

- 1D, 2D or 3D; depending on the dimensions of the space used to define the geometry.
- Line, surface or volume meshes; depending on the hierarchy of the geometrical entity to be meshed.
- Structured, semi-structured, unstructured or cartesian; depending on the nature of the topology of the mesh.
- Isotropic or anisotropic; depending on the aspect ratio of the elements to be generated following specific directions.

- Embedded or body-fitted; depending on how they fit or not the boundaries of the domain to be meshed. Body-fitted meshes (used, for instance, in FEM or FV simulations) match perfectly the boundaries of the domain, as embedded ones (used, for instance, in FD, immersed or embedded methods) have faces crossing that boundaries.

- Conformal or non-conformal; depending on the continuity between neighbor elements across edges or faces. In conformal meshes edges and faces match perfectly between neighbor elements, as oposed to non-conformal ones.

- Depending on the element type to be used and the quadratic type of the element (linear or different degrees of quadratic types). Although there are methods using arbitrary polyhedral shapes for the mesh elements [CIO03], it is common to use the same simple polyhedral element for all the domain (triangle, quadrilateral, tetrahedral, etc.).

In this thesis 3D unstructured isotropic conformal tetrahedral meshes are considered. Both embedded and body-fitted cases will be covered.

1.1 Motivation

In industrial simulations, the pre-processing operations represent the most time consuming part of the whole process. Among the pre-processing operations, the geometry cleaning and mesh generation parts are the ones which consume more time, due to their specific characteristics:

- The geometry cleaning process implies all the operations needed to allow the mesher the correct discretization of the domain. It often needs much human interaction, as it is really difficult to automatize it following general criteria for all kinds of simulations. Most of the meshers need watertight geometries to be able to generate the mesh. In this context, the concept of *watertight* (for volumes) is used to define a closed collection of surfaces sharing their contour lines. Make watertight an input geometry implies to avoid surface overlapping and gaps in the geometrical definition of the volumes. For many complex 3D geometries (like the one shown in Figure 1.1), to reach this goal requires a huge effort. Furthermore, classical meshers are not totally independent on the mathematical definition of the geometry. Often the characteristic size of the geometrical entities defining the boundary must be similar to the final mesh size desired for the simulation. This aspect enforces a modification on the definition of the input

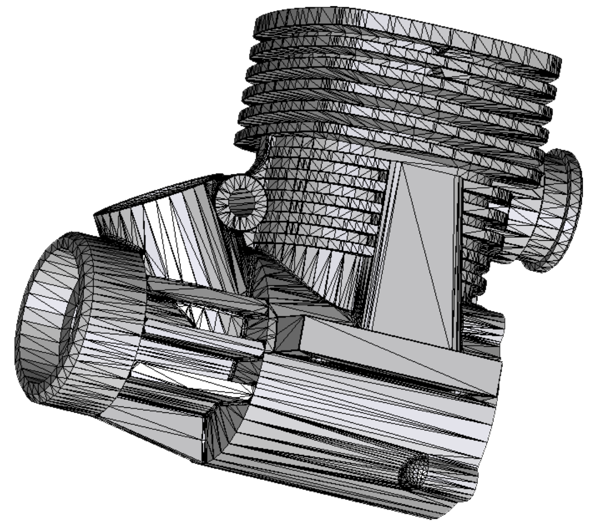


Figure 1.1: Example of a 3D input boundary (represented with a mesh) where the quality of the triangles is very bad.

boundaries depending on the final mesh desired, although it may represent well the shape of the domain.

- Once the geometry is clean enough to generate the mesh, the mesh generation itself can consume a considerable amount of time. Furthermore, the mesh generation process often involves an iterative loop where the person in charge of the simulation defines the desired mesh sizes in the different parts of the domain, and several meshes have to be generated, as the mesher parameters are tuned in order to get an optimum mesh. It also has to be taken into account that nowadays the volume unstructured meshers are not fully robust in the sense that, when dealing with complex geometries, it is not easy to tune its parameters and prepare the input geometry to generate the mesh successfully: often the user must try several configurations (geometry and parameters) until a mesh is generated. This relativizes the speed in the mesh generation, as the mesher itself can be very fast, but the user may need several tries to get a suitable mesh for the simulation.

Much effort has to be spent in order to generate a volume mesh of a complex geometry considering the existing meshing algorithms. Some of them are really fast and robust, but they require several geometry cleaning operations. If the time needed for them is added to the total meshing time, they became not so fast in practice. Contribute to reduce this extra effort is the motivation of this thesis.

1.2 Objectives

The main objective of this thesis is to develop an algorithm for isotropic unstructured volume mesh generation robust enough to be able to generate a mesh from non-cleaned input geometries, using as less input data as possible. This will lead to a drastic reduction of the time consumed in the pre-processing part, and will overcome the actual bottleneck in the whole simulation process.

The mesh generator must be flexible enough in terms of mesh adaptation to the solver requirements considering specific input data. The idea in this work is that the algorithm should be able to generate always a mesh from a given geometrical domain, almost without any specific meshing property assigned to it.

The meshing algorithm presented in this thesis has been designed with the following objectives in mind:

- It should be prepared to use the more common geometrical definitions as input data reducing as much as possible its preparation (CAD cleaning operations).
- It must be able to run on a simple PC, but also it should be able to take profit from more complex computer architectures, which enable the use of parallel processing.
- It should be useful for a wide range of application fields. The main characteristics for the mesh to be generated are taken from the requirements for the FEM, but they are general enough to be applied to other numerical methods.

These characteristics lead to a set of requirements to be covered by the meshing algorithm developed in this thesis. The main ones are: robust and fast mesh generation, and ability to mesh from non-watertight geometries. The explanation of all the requirements covered by the new mesher is detailed below in Sections 1.2.1, 1.2.2, and 1.2.3.

1.2.1 Mesher requirements

In this section, the requirements of the mesher developed in the thesis regarding its behavior are detailed:

- *Robustness.* This is one of the key objectives and it can also be seen as a requirement for the input data, as this robustness refers to the capability of generating the mesh independently from the input data quality. Some of the existing unstructured volume meshers accomplish all (or almost all) the requirements presented in this work, but they require a very specific tuning of their parameters for successfully generating the mesh. Independently of the sizes assigned by the user, the quality of the input geometrical data or the general parameters chosen, the mesher should be able to generate always a mesh suitable for the simulation. The idea of the new algorithm developed in this thesis is that it should generate always a mesh accomplishing the requirements defined in this section without a special training in the use of the mesher.

- *Fast mesh generation.* One of the key motivations of this work is to reduce the time consumed in the pre-processing operations for a numerical simulation. For this purpose the mesher has to be fast. Moroves, it has to be *naturally* fast, in the sense that it should imply fast and simple operations. Furthermore, its implementation should take profit from parallel computing strategies.

- *Prepared to generate really large meshes.* As the computer technology is evolving very fast, each time the solvers are capable to solve larger and larger algebraic systems. Thinking about the requirements from different simulations fields for the next years, the mesher should be able to generate billions of tetrahedral elements without problems. This characteristic is really important when deciding some implementation aspects of the algorithm.

1.2.2 Input data requirements

This section focuses on the requirements to be covered by the mesher concerning the input data:

- *Accept non-cleaned input geometry.* This is another key point of the mesher. The idea is that it should be able to generate the mesh of the model although its geometrical contour comes from a non-cleaned input geometry. In this context, a non-cleaned input geometry is defined as the one containing:
  - Non-watertight volumes. This means volumes with contours presenting gaps or overlapping entities.
  - Non-coherently oriented contour entities. This means that the contour entities of a volume are not needed to be oriented (all of them) towards the same part of the volume (inner or outer).

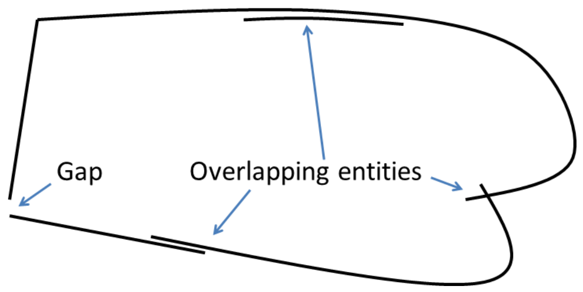


Figure 1.2: Example of a 2D non watertight input boundary with gaps and overlapping entities.

  - Low quality definition. The geometrical contour can be defined by geometrical entities (like NURBS) or by a surface mesh. In this context, a geometrical contour with low quality definition means a highly distorted parametrization of the surfaces (in case of geometrical representation), or a mesh containing low quality elements (in case of surface mesh representation). Examples of low quality input geometries and no water-tight domains are shown in Figures 1.1 and 1.2. It has to be noted that the algorithm proposed in this document is for 3D volume meshing. However, for sake of simplicity, 2D examples are shown in some figures.

- *Accept input data in geometry and mesh format.* The natural input for a mesher is the geometrical definition of the contours of the model. This geometrical definition can be represented in several ways, but the most common ones are CAD or mesh entities. In this document, when talking about CAD entities, the NURBS surfaces and curves [Far97] (trimmed or not) will be considered, as they are the most general mathematical representation able to represent all the geometrical shapes.

  Although the mesher should generate a mesh from a non-cleaned input geometry, some minimum criteria concerning topology or shape definition may be needed.

  While mesh entities are simpler to be defined, CAD entities are a more precise way of defining a geometry. Meshes often loose continuity in the geometrical definition depending on the smoothness or curvature of the shape to be represented. It also has to be considered that, almost always, the mesh presents a *chordal error* compared to the smooth original geometry (the chordal error is the distance between a point on the mesh and the original smooth shape the mesh is trying to represent). In the Figure 1.3 a graphical representation of this chordal error is shown.

  The simpler definition and treatment of mesh entities has made them the most common

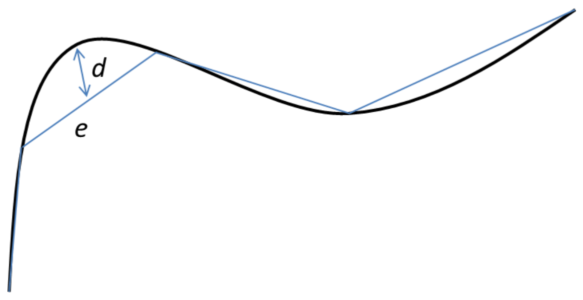


Figure 1.3: Graphical representation of the chordal error in a 2D case. The black thick line represents the smooth geometry, and the gray thin line represents its mesh. The distance d is the chordal error of the element e

used as input for the existing volume meshers. There are cases where a surface (or a line) cannot be meshed because of its bad parametrization. In these cases, a volume mesher requiring a mesh as input cannot be used. This is the reason why the proposed volume meshing algorithm should be prepared to get as input data mesh entities, as well as NURBS surfaces and curves. Actually, it should be prepared not only to get these kinds of entities as an input, but also to work with them in the geometrical operations needed during the whole meshing process. The methodology presented in this thesis is prepared to work either with CAD and mesh entities. Hence, the general form of *geometrical entities* will be used to refer the input entities defining the contours of the domain. Only if some specific operation is needed for just one of the representations it will be specified if the geometrical entity is a CAD or a mesh one. Following this duality, in this document a single term will be used to refer the different natures of geometrical entities:

- *Surface entities* for surface mesh elements (especially triangles) and geometrical surfaces (especially NURBS surfaces and patches of connected NURBS surfaces).
- *Line entities* for line mesh elements and geometrical curves (especially NURBS lines and groups of connected NURBS lines).
- *Point entities* for nodes of a mesh and geometrical points.

1.2.3 Final mesh requirements

The requirements to be covered by the mesher regarding the final mesh generated are:

- *Maintain volume topology.* The topology of the initial domain to be meshed must be preserved in the final mesh. For most meshing algorithms this requirement is automatically satisfied, but it is not automatically guaranteed in all meshing techniques. Considering the final mesh, it has to be possible to identify the volume each tetrahedron belongs to. Then, if getting all the tetrahedra belonging to a given volume, the concept of the mesh of that volume can be defined. The maintenance of the volume topology of the initial domain means that there must be a mesh for each of the initial volumes, and each volume mesh must represent the same relationship between the others. For instance, if two volumes are connected by a line, the meshes of these two volumes must be neighbors sharing line elements. Furthermore, the mesh of a volume itself must represent the topology of the volume (for instance, if the volume has a hole, the mesh must have a hole). In Figure 1.4 two examples are shown with meshes preserving or not the topology of the original model.

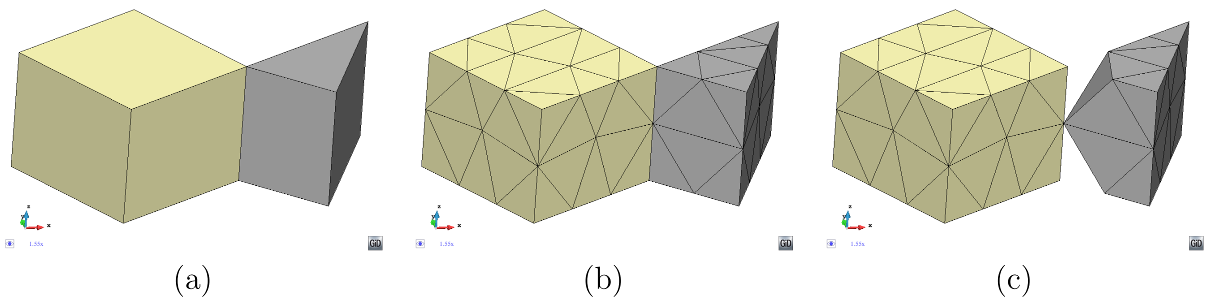
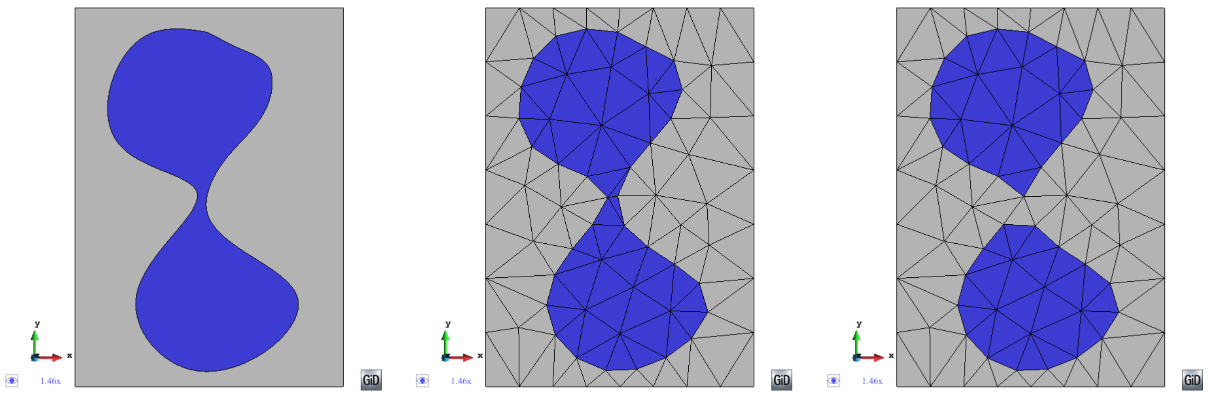


Figure 1.4: Topology preservation between two volumes and their meshes. **(a)** Two volumes in contact sharing a line. **(b)** Meshes of the volumes shown in (a) sharing the line elements corresponding to the mesh of the line shared by both volumes: the topology is preserved. **(c)** Meshes of the volumes shown in (a) not sharing line elements: the topology of the original model is not preserved.

A special case which evidences the importance of maintaining the initial topology is the situation where the domain has very thin parts representing relevant details of it. A 2D case of this kind is shown in Figure 1.5, where a thin channel-like part can be identified in a surface (Figure 1.5(a)). Independently on the mesh desired size required by the simulation, the mesher must generate elements small enough not to close this channel-like zone, as the mesh shown in Figure 1.5(b). The mesh depicted in Figure 1.5(c) is not acceptable, as it does not preserve the topology of the domain.

- *Maintain a representative lines and surfaces topology.* Often it is not required to preserve the topology of all the surface and line entities of the domain to be meshed. As the



(a) (b) (c)

Figure 1.5: Example of surface mesh preserving or not the topology of the initial domain. **(a)** 2D geometrical domain formed by two surfaces (colored as blue and gray). **(b)** Mesh of the surfaces preserving the topology of the initial domain. **(c)** Mesh of the surfaces not preserving the topology of the initial domain.

proposed algorithm is a volume mesher, the surface or line elements in the final mesh are of interest just in order to assign properties to them for the simulation. These properties can be of different nature: material properties, boundary conditions, etc. At the end, the mesher should maintain the topology of some patches of lines and surfaces on user demand, or following some automatic criteria.

Note that this requirement is not as hard as the typical constrained condition in the boundaries of the domain. Especially when the input data for a mesher is a surface mesh, it is common to require the mesher to be constrained at the boundary. This means that the contour mesh of the final tetrahedral mesh (the triangles representing the skin of the generated tetrahedra) must be topologically identical to the input surface mesh defining the contours of the domain.

In the Figure 1.6(a) an example of the contour of a volume is shown, where a set of surface entities (in this case triangles)are colored in red. Let us call A the region represented by those triangles region. If the simulation requires a set of triangles representing the region A, the mesh generator should provide with a tetrahedral mesh whose skin should have a set of triangles representing that region, but it is not needed for those triangles to be identical as the ones in the input geometry. A tetrahedral mesh accomplishing this criterion is shown in Figure 1.6(b). A totally unconstrained mesh is depicted in Figure 1.6(c), where it cannot be identified a set of triangles corresponding to the region A. It is obvious that the mesh in Figure 1.6(b) is not constrained with the

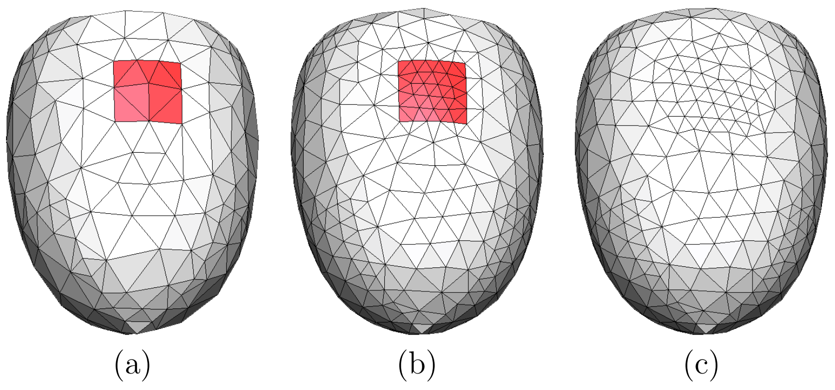


Figure 1.6: **(a)** Contour mesh of a volume with a set of triangles highlighted in red representing the region A. **(b)** A partially constrained tetrahedra mesh generated from the contour mesh shown in (a) where the highlighted set of tetrahedra faces corresponds to the region A. **(c)** A not constrained tetrahedra mesh of the contour mesh shown in (a) (it cannot be identified a set of tetrahedra faces representing the region A).

input boundary shown in Figure 1.6(a), as the triangles representing the region A are different. However, it can be seen that the requirement of maintaining a representative topology of the surface and line entities of the input data allows a bijective relationship between groups of surface and line entities in the input data and surface and line mesh elements in the final mesh. This requirement is enough to allow an automatic assignment of data from the input boundary geometry to the final mesh entities.

- *Generate automatically the contour mesh of the volumes.* Although the mesher is thought to generate volume elements (tetrahedra), it should give as an output also the triangular elements corresponding to the contours of the volumes, and the line elements corresponding to the boundaries of certain patches of triangles. This surface and line elements correspond to faces and edges of the final tetrahedral mesh.

- *Preserve geometrical features.* This is automatically achieved by the boundary constrained meshers, but it is not guaranteed at all by the types of meshers. This requirement is crucial for the final mesh to represent the shape of the domain precisely. For several kinds of numerical simulations, the presence of sharp edges and corners in the domain affects drastically the results, as they often govern the physical behavior of the process to be simulated. The importance of preserving sharp edges and corners to represent some domains can be appreciated in Figure 1.7. Part of the contours of a volume are shown in Figure 1.7(a). The tetrahedral mesh of that volume generated without preserving the sharp edges of the input geometry is shown in Figure 1.7(b).

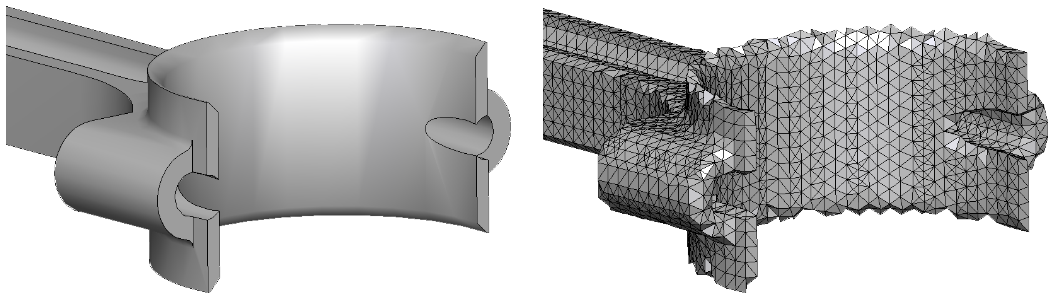
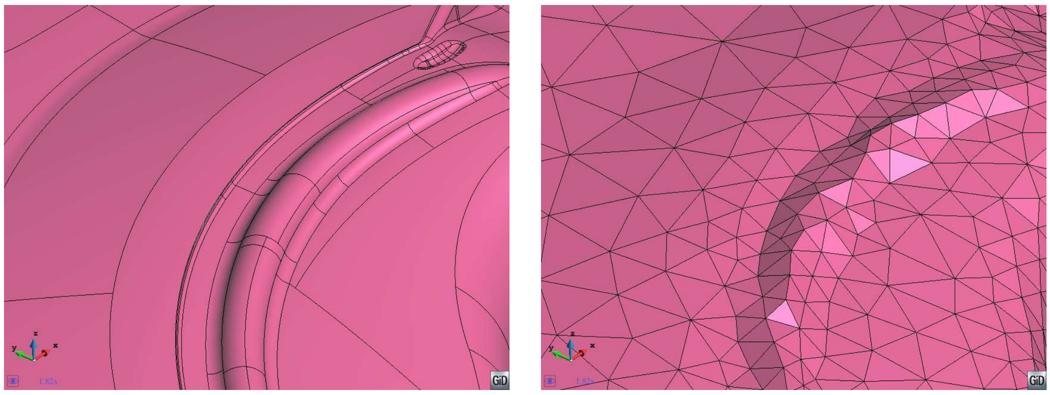


Figure 1.7: **(a)** View of a part of the contours of a mechanical piece. **(b)** Tetrahedral mesh of the mechanical part generated without preserving the sharp edges of the input geometry.

In terms of the mesher, this requirement means that it should be partially constrained to some line entities from the input geometrical data, and must include fixed nodes in the final mesh (corresponding to specific point entities in the input data). In this context, partially constrained means that if some specific line elements in the input geometrical data must be preserved, a collection of edges from the final mesh should follow the path of those line elements. This is not as restrictive as totally constrained condition.

As it will be explained later on (Section 3.1), the strategy to cover this requirement will be also useful to reach the requirement of maintaining a representative lines and surfaces topology.

- *Allow to skip given details of the domain.* This requirement is somehow complementary to the previous one. What it means is that the mesher must be not constrained in some regions. The definition of the geometrical domains includes often very thin or small entities in given parts. The reason for the size of these entities may be the size of the part of the domain represented itself, but the presence of these small entities often responds to the application of a given tangency criteria for the geometrical definition, or they are just the result of some geometrical operation previously done (intersections, Boolean operations, etc.). The basic idea of this requirement is that the sizes of the geometrical entities used to define the contours of the domain are not necessary related to the mesh size needed for the simulation. In the Figure 1.8, an example is shown where the triangles generated from a patch of surfaces skipping the inner lines between them are much larger than the size of some of the surfaces of the patch. Clearly the simulation should not need such thin triangles in the regions where the surfaces are so thin.



(a) (b)

Figure 1.8: **(a)** Zoom of a patch of surfaces representing a mechanical part. **(b)** Triangle mesh of the patch of surfaces shown in (a).

Even when the sizes of the input entities are related to the representative size of a specific shape to be represented, the user may desire a bigger mesh size to skip the representation of given details because they are not of interest for the simulation. Instead of forcing the user to modify the input geometry which defines the domain, this requirement makes the mesher skip the detail within the mesh generation process.

The accomplishment of this requirement will let the user to define a desired mesh size distribution in the final mesh just because of the simulation requirements, and not because of the initial way of defining the geometry. In terms of the mesher, this requirement means that it should be able to skip some of the line and point entities from the input geometrical definition of the domain. The entities to be skipped should be defined automatically following some given criterion, or by the user.

- *Precise representation of the contour geometry.* This requirement focuses on the need of the nodes of the contours of the final mesh to lay exactly on the geometrical entities defining the contours of the domain. Some mesh generators only approximate the shape of the domain to be meshed, or represent its contours in a staircase manner (specially the meshers based on octree or bin structures). Several numerical methods (specially the FEM) need a precise representation of the contour, as its shape and smoothness may affect drastically the result of the simulation. This is the main reason for this requirement.

- *Final elements quality ensured.* The acceptable quality of the mesh elements is a relative parameter, as the different methods used for the numerical simulation are applicable for different ranges of element qualities [She12]. Usually, most numerical methods perform better as the shape of the element is closer to the corresponding regular polyhedron, however this cannot be extended to all numerical simulations. This requirement will try to reach a minimum dihedral angle in the final tetrahedra.

- *Allow a 3D spatial size assignment.* The use of a non-uniform mesh is crucial for some simulations. In some specific regions of the domain the size of the elements may have an upper bound, but if that size is used in the whole domain the final mesh should have too many elements. Typically, in the regions where there are higher gradients of the variable under study, the simulation needs a finer mesh than in other regions. This requirement is also important in the accurate representation of the shape of the domain, as the curvature of the domain geometry is strongly related with the mesh size needed to accomplish a given chordal error criteria. Just for geometrical purposes, a refined mesh in some regions would be needed.

- *Control of sizes transitions.* As the final mesh can be non-uniform size distributed, the size of the elements should vary from a region of the domain with a desired size to another. Some numerical simulations do not only require a specific mesh size in some specific zones, but also a given growing law between neighbor elements. The mesher should let the user to control how the element sizes grow (or decrease) from a region of the domain, depending on the distance from the elements to that region.

- *Applicable for immersed methods.* A family of methods for numerical simulations are the so-called *immersed methods* and *embedded methods* [LP00, LCC$^+$08]. The main characteristic of these methods is that they do not require a body-fitted mesh to represent the domain. The shape is implicitly introduced by distance from the nodes to the boundaries and a casting process (determine whether a node is inside or outside the domain). In some sense, generating a mesh for a simulation using an immersed method is less restrictive as the geometrical features should not be preserved, but in contrast, the final mesh must have extra information attached. This information is the distance from each node of the mesh to the boundary of the domain.

1.2.4 Surface meshing

Although the main objectives of the thesis are related to volume meshing, there is a secondary objective for the new mesher, which is to be able to mesh 3D surfaces and lines which are

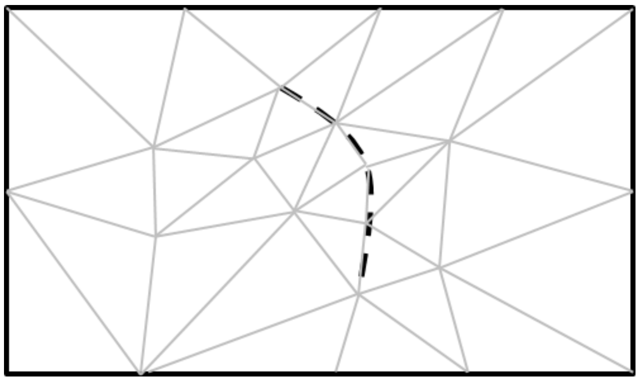


Figure 1.9: 2D example of a surface mesh (gray lines) conformal with an inner line of the surface (dotted line).

not boundary of any volume accomplishing the requirements defined above. This capability would give several advantages to the method:

- Generate conformal meshes of volumes containing inner surfaces or lines. These are entities totally inside the volume, but not belonging topologically to the boundaries of the volume. An example of this case could be the representation of a sail (3D surface) into a control volume in order to perform a CFD simulation. The sail is totally inner to the control volume, and the mesh of it must be conformal with the tetrahedra representing the control volume. A 2D example of this case is shown in Figure 1.9. In this example a surface (bounded by solid black lines) has an inner line (dotted black curve), and the mesh of the surface (in gray) is conformal with this line.

- Mesh in a conformal way a model with volumes and surfaces or lines connected to them. A case of two volumes connected by a surface is depicted in Figure 1.10. The mesh of the volumes and the surface must be conformal in the contact lines between them. This case can be also meshed by generating separately the volume meshes with the octree mesher, and afterwards mesh the surface using other meshing algorithms. This would not take profit on the advantages of meshing the whole model at once.

- Mesh patches of surface entities together preserving features, but skipping the inner line entities between them if needed. The entities to be skipped could be detected automatically following a given smoothness criteria, or under user demand. Figure 1.8 shows an example of triangle mesh representing a collection of surface entities, but skipping the inner line entities.

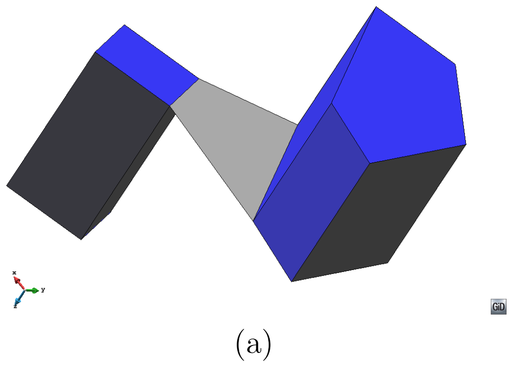


Figure 1.10: **(a)** Example of a model containing two volumes (in blue) and a surface (in grey) connecting them. **(b)** A conformal mesh of the model considering the volumes and the surface mesh.

- If the patches of surfaces are not topologically connected, the proposed method could act as a CAD cleaning tool, as it could extract a continuous manifold surface mesh from a collection of surface entities with gaps or overlaps. The extraction of surface skin meshes from a volume discretization has been used in [CCL02] and [NT03], and it is interesting for CAD cleaning, as well as for boolean operations involving non-watertight geometries. The preservation of geometrical features is essential for these purposes.

1.3 Structure of the thesis

The thesis is structured in the following chapters:

- *Chapter 1: Introduction.* This chapter focuses on introducing the context of the presented work. Its motivation is explained (Section 1.1), and its objectives are detailed (Section 1.2).
- *Chapter 2: State of the art on mesh generation.* In this chapter the different mesh types are presented (Section 2.1) as well as the state of the art of different meshing techniques (Sections 2.2, 2.3, 2.4 and 2.5). The advantages and drawbacks of each meshing method is studied, and the proposed solution for the meshing algorithm is justified (Section 2.6).
- *Chapter 3: Basic concepts of the new mesher.* Several concepts and some auxiliary algorithms are defined in order to allow a better understanding of the meshing algorithm presented. Its main parts are:

  - Definition of the input data for the presented meshing algorithm and a proposal of how the mesher can interact with CAD data, integrated within a pre-processing system (Section 3.1).
  - Octree structure characteristics (Sections 3.2 and 3.3). The definition and main characteristics of the octree structure (which plays a key role in the presented work) are presented. Special interest is given to its specific properties for the meshing process. The notation is defined which will be referred to in the following chapters.
  - Geometrical intersections (Section 3.4). As the geometrical intersections are crucial for some of the algorithms presented, this section analyzes them deeply and studies the pathological configurations that may occur.

- *Chapter 4: Coloring algorithm.* The coloring process determines where a point in space is topologically: it determines if it is outside the domain, inside a given volume, or onto an interface between volumes. This is one of the main operations involved in the presented mesher and can be understood as an auxiliary algorithm for the meshing itself, so a whole chapter is devoted to it. In this chapter different approaches to solve the coloring problem are studied (Section 4.1), and a new algorithm based in the ray casting technique is exposed (Section 4.2). The implementation of this algorithm is presented in Section 4.3.

- *Chapter 5: Octree based mesher.* This chapter focuses on the new meshing algorithm itself. After an introduction, where its main idea is highlighted (Section 5.1), the algorithm for embedded (Section 5.2) and body-fitted (Section 5.3) meshes is detailed.

- *Chapter 6: Implementation aspects.* In this chapter all the implementation details for the meshing algorithm developed are explained. After some general aspects of the implementation (Section 6.1), the implementation of the octree structure is analyzed (Section 6.2). Sections 6.3, 6.4, and 6.5 focus on the implementation of specific parts of the meshing algorithm itself. Some considerations on the parallel implementation of the mesher are pointed out in Section 6.6. A list of the values of the parameters relevant for the mesher used in the presented implementation of the algorithm is detailed in Section 6.7.

- *Chapter 7: Examples.* In this chapter the results of some validation examples highlighting specific characteristics of the presented mesher are shown (Section 7.1). Sections 7.2

and 7.3 analyze the results of the meshing algorithm applied to real complex geometries under different configurations.

- *Chapter 8: Conclusions and future research lines.* This chapter lists the conclusions of the presented work (Section 8.1) and proposes some futures lines of research (Section 8.2).

- *Appendix A: Profiling tables and complete data of examples.* In this appendix, tables with all the profiling data concerning times and memory for the whole configurations used in the examples run to validate the meshing algorithm are provided.

It is highlighted that, although the meshing algorithms presented are directly applied to volume meshing, most of the geometrical operations are applicable (with slight modifications) to surface meshing in 2D cases. Taking into account that some concepts are much more clear to understand from a 2D scheme (especially when dealing with geometry), in this thesis 2D examples are used sometimes to illustrate some of the concepts explained.

Chapter 2

State of the art on mesh generation

In this thesis isotropic volume meshers are considered. Isotropic meshers can be defined as the ones trying to generate elements as much regular as possible, understanding a regular element as the one whose edges have the same length.

2.1 Mesh types

There are two main families of meshers depending on the kind of mesh they generate: structured and unstructured [Geo91]. Actually, a third kind of meshers can be classified as semi-structured ones. A structured mesh is defined as a mesh which all inner nodes have the same degree (the degree of a node is the number of elements owning it), while the nodes of an unstructured mesh have different degrees. Semi-structured meshes can only be applied to topologically prismatic geometries, and they basically repeat the structure of an unstructured mesh (in the tops of the prismatic shape) in different layers following the structured direction. An example of this kinds of mesh is shown in Figure 2.1.

Unstructured meshers [Löh08, She12, FG00] can be divided in three main families: *advancing front*, *Delaunay* and *space decomposition* methods.

In the following sections, the main characteristics of these methods as well as their main advantages and drawbacks are detailed, focusing on the requirements defined in section 1.2. The aim of this chapter is to highlight which of those requirements are covered by each meshing method, so the algorithms are not deeply detailed and only their main characteristics are pointed out.

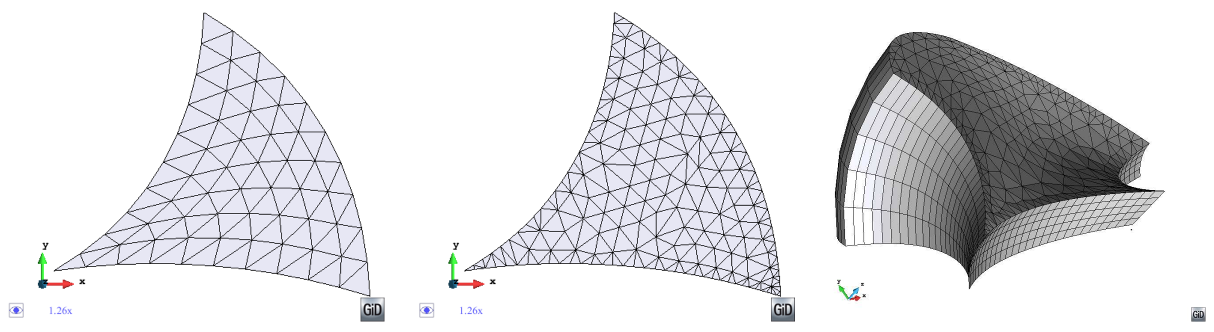


Figure 2.1: Examples of different types of mesh**(a)** Structured triangle mesh. **(b)** Unstructured triangle mesh. **(b)** Semi-structured prism mesh.

2.2 Structured meshers

Structured and semi-structured meshers often get as input data the position of the nodes in the contours of the domain, and generate the inner nodes positions from a given interpolation [Geo91, FG00, Far97]. The quality of the final mesh obtained is directly related to the kind of interpolation used and the degree of distortion of the contours of the domain, so a minimum level of element quality cannot be guaranteed for arbitrary domains. However, for good shaped volumes and uniform sizes distributions these methods provide with very good quality meshes.

The main advantages of structured meshers are:

- *Fast and robust.* As they are based on a given predefined interpolation, their are really fast and robust.
- *Parallelizable.* Apart from being fast, these kind of meshers are very parallelizable.

On the other hand, these meshers have some important drawbacks:

- *Need of specific topology for the input data.* The main problem of this kind of meshers remains in a requirement for the input data: they need a specific topology for the geometrical definition of the contours of the domain. As an example, to generate a structured mesh of hexahedra, the input geometry must be topologically an hexahedra. This means it has to be a volume with 6 contour surfaces, and each one of their must have 4 contour lines. This requirement makes impossible to use this kind of meshers for arbitrary geometries with complex topology. A family of methods have been proposed

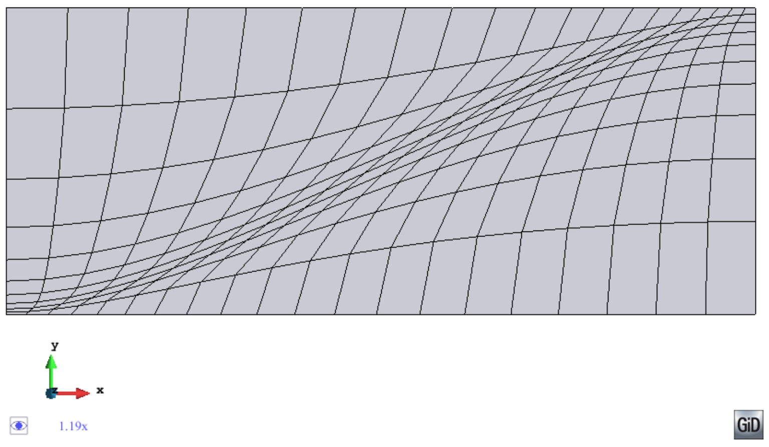


Figure 2.2: Example of a 2D structured quadrilateral mesh with a non-uniform size distribution. A hight level of element distortion can be appreciated.

that are able to skip this problem [ZP71]. They are basically based in decomposing the original domain to be meshed into different parts which do accomplish with the topology demanded by the mesher. Then, each of these parts can be meshed in a structured manner, so they are often referred as *structured by blocks* or *multi-block* meshers. Several implementations of these methods have been carried out [Löh08, Geo91], but in practice they are meaningful for very specific domains only. Depending on the topology required by the mesher, some geometries are impossible to be decomposed in such this way, and even when this decomposition is possible, from complex original geometries is not obvious to generate this decomposition automatically. This often implies an extra preprocessing operation before using the mesher, which is the splitting (manually) of the domain in different parts.

- *High distortion in non-uniform meshes.* As the nodes of structured meshes have the same degree, a predefined topology of the mesh is forced. It generally tends to generate uniform meshes. Structured meshers can relatively distort the elements in some region in order to provide with a non-uniform mesh, but the fixed topology of the mesh does not allow a sharp sizes transition without an excessive distortion in the elements. A 2D example illustrating this problem is depicted in Figure 2.2.

2.3 Advancing front method

The advancing front method [Lo85, LP88, PVMZ87, NVP82] is a common technique for generating unstructured meshes. It gets a closed and oriented mesh of the boundary of the domain as input and mesh its inner part. The surface elements of this mesh are the ones in the *active front*, and the algorithm can be summarized with the following points:

- From each element in the front (face) a new volume element is generated. An optimal position of the node needed to build the element is obtained and the nodes and faces of the active front close to it are considered. The new element is build using a new node in the optimal position or an existing close node, depending on the resulting configuration: the new element should not intersect any one of the close already existing faces.

- The active front is updated adding the new faces created, and removing the ones already shared by two elements.

- These two steps are repeated until there are no faces in the active front. Then all the domain has been filled with volume elements.

Although the advancing front is mainly used for generation of tetrahedral (or triangle in 2D) elements, some adaptations of the method have been done to generate other types of elements [OS00], and even for generating particles for DEM simulations [LO98]. The present work is focused in tetrahedra mesh generation. There are several possible implementations of the advancing front method [Löh08] depending on the way of creating the new elements from a face of the front, the way of considering the mesh desired size in the inner part of the domain, or the order in which the faces of the front are processed, among others. In special, much work have been done in order to improve the efficiency in evaluating the desired mesh size in a specific region of the domain, and control a smooth size transition in the final mesh. This is one of the strong points of advancing front techniques, as the creation of each element can fit very well the desired mesh requirements. Different approaches use a background grid to set the mesh desired size [PVMZ87, LP88, JT93, Fry94], or sources from which the desired mesh size vary following a given function [Löh93], which provides with a very smooth transition between element sizes. A combination of different methods can be used in order to improve accuracy and reach an efficient implementation of the method [Löh96]. However, independent of the implementation, one can identify some general advantages and disadvantages of advancing front based techniques. The main advantages are:

- *Good quality meshes.* In the advancing front method, each new element is created connecting a face of the active front with a new node or an existing one. This choice is done based on an optimal position for this new node in order to get the best element (in terms of quality) taking into account the desired size of the mesh in that region. This methodology for creating the elements gives high quality elements in the final mesh, even before any smoothing process.
- *Automatic preservation of topology.* The advancing front method is naturally constrained, as each element of the surface mesh of the contours of the domain to be meshed will become a face of the final tetrahedra.
- *Good control on the size transitions.* As explained before, each new element is created from an existing face and an optimal position of the node to create the element from the face. Considering the desired size for the final mesh in the region where the face is, and the size of the face itself, the size transitions from one part of the mesh to another is totally controlled.
- *Parallelizable.* The creation of each new element using the advancing front method is clearly local. The geometrical and topological checks needed to create the new element only take into account the nodes and faces inside a specific radius of influence of the face the new element is created from. This aspect makes totally independent the creation of an element from the creation of another element which is far enough. Although it is not obvious to implement an efficient parallel version of the advancing front method, at least the element generation method is local enough to make the implementation affordable. [LC99] proposed a first octree subdivision of the domain, and then apply the advancing front technique in each of the cells of the octree, seaming afterwards the interface parts between cells.

The main drawbacks of the advancing front method are:

- *Strong dependency on the contour mesh quality.* Considering that the surface elements of the initial contour mesh will be faces of the final volume elements, the quality of these contour elements is totally related with the quality of the tetrahedra in contact with them: if a triangle of the contour of the domain has a very small angle in some of its nodes, the resulting tetrahedron from it will have a very small dihedral angle. This implies a great effort in the mesh generation of the contours of the volumes to be meshed. Often this effort takes most of the part of time in the pre-processing operations of the whole simulation when using the advancing front technique.

- *Requires watertight input geometry.* The reason for this is that the initial surface mesh representing the boundary of the domain will be the skin of the tetrahedra in the final mesh.

- *Not very robust.* Theoretically, the advancing front is a robust method. However, the locality of the element creation process and the checks needed to evaluate if the new element created intersects or not the active front require some tolerances. These tolerances must be tuned precisely in order not only to avoid the crossing of the active front, but also to avoid different parts of the front to get very close. This situation can lead to geometrical configurations of the active front which make almost impossible the creation of a new element. The tuning of these tolerances and other parameters can be done as general as possible, but as the active front evolves during the meshing process, these geometrical configurations are not under control, and cannot be predicted. This aspect, together with the hard dependency on the contour mesh quality makes the advancing front method not so robust in practice.

- *Not very fast.* Every candidate new element to be created needs to be checked in order to verify if the active front is intersected by some of its faces and edges. This involves the use of several geometrical intersections operations in the whole meshing process. This aspect, together with other characteristics of the advancing front method, makes it not very fast in comparison with other meshing algorithms.

2.4 Delaunay method

A Delaunay mesh is defined as a mesh which elements accomplish the Delaunay condition: the circumcircle (in 2D case) or the circumscribed sphere (in 3D case) of any element has no node from the mesh inside [Geo91]. Given a cloud of points, a Delaunay triangulation can always be created from their Dirichlet tesselation connecting them with a set of triangles (in 2D) or tetrahedra (in 3D) without adding any extra node [Geo91].

The Delaunay meshing methods [FG00, CDS12, She12, Löh08] depart from the contour of the domain and generate the Delaunay triangulation of its nodes. This mesh is the convex hull of the domain to be meshed. Although it is already a mesh, its elements may have a low quality, or may not fit with the desired mesh size in that region of the domain: these are the *bad elements*. The following strategy is applied recursively to all the bad elements:

- A node is created in the centroid of the element. Some strategies allow the creation of the new node at the edges or faces of the elements [GB98] in order to fit improve the quality of the elements respecting the contours of the domain.

- All the elements of the mesh which circumscribed sphere (or circumcircle in 2D) includes the new node are deleted. Note that a void region is created containing the new node inside.
- A Delaunay triangulation is created with the new node and the contour nodes of this void region.

This procedure leads to a Delaunay mesh (accomplishing the Delaunay condition), but this does not guarantee a given level of quality by itself. Often, some elements can present very low quality (specially in 3D cases). These elements may have null volume and are called *slivers*. Even if its volume is equal to zero, they can accomplish the Delaunay condition. For this reason, it is common to relax the Delaunay condition in some regions in order to avoid quality problems [GHS90].
The main advantages of this method are:

- *Robust.* Mathematically, we can always obtain a Delaunay mesh from a cloud of points. However, sometimes the modifications of the method to cover the requirements of the simulation makes it not so robust.
- *Fast.* Although it depends strongly on the implementation, it can be said that Delaunay methods are naturally faster than the advancing front ones.

The main drawbacks of the Delaunay method are:

- *Naturally not constrained.* Delaunay methods use the representation of the contour of the domain as an initial configuration for the insertion of nodes procedure, but the final mesh boundaries are not guaranteed to be constrained with that contour. Some strategies can be applied [Bak87, Bak89, Wea92] in order to guarantee that the nodes in the initial contours will lay on the contour of the final mesh, or even a constrained condition in the faces of the initial contour, but these modifications reduce the performance and robustness of the method. Furthermore, if there are huge different mesh desired sizes in the domain, these strategies are not very robust.
- *Requires watertight input geometry.* As in the advancing front method, the contours of the domain are required to be watertight. Actually, a strategy could be followed to

treat non-watertight geometries. It is based on generating the first mesh (the convex hull in the traditional Delaunay methods) using only the nodes of the contours, with an automatic recognition of the boundaries: the so called *alpha-shape method* [AEF$^+$95]. This would indeed generate a volume mesh from a non-watertight geometry, but there is no enough control in the boundaries recognition to ensure that mesh correctly represents the topology of the domain.

- *Not naturally parallelizable.* The check of the Delaunay condition of an element requires to take into account all the mesh entities participating in the circumscribed sphere of the element. As its radius depends on the position of the nodes of the element (cannot be bounded a priori), the treatment of one element can involve the whole mesh in some configurations. This implies a hard dependency ranging from an element to the whole mesh, so a parallel implementation of the method is not obvious. However, some parallel implementations (mainly for shared memory paradigm) of the Delaunay method have been carried out, [KKŽ05] or [BBK06].

2.5 Space decomposition methods

Space decomposition-based methods follow a different philosophy than the methods explained before. To generate the mesh, they basically subdivide the space into cells providing with a spacial decomposition covering the space where the domain is (overlapping the domain). These cells can be thought in a general way, but it is common to use one of the following main structures which govern their configuration:

- *Bin*: a bin structure provides with an homogeneous grid as the space decomposition formed by regular cells (squares in 2D or cubes in 3D). A graphical view of a 2D bin is shown in Figure 2.3(a).
- *Octree*: an octree (quadtree for the 2D case) is basically a hierarchical spacial structure that partitions the 3D space into regular cells [Sam06]. These cells can be refined in given zones of the domain. A graphical view of a quadtree is shown in Figure 2.3(b). A more detailed definition of the octree structure is given in Section 3.2.

The bin structure is suitable for homogeneous discretizations. It is common to use an octree structure as the regular grid (which gives the name of the family of methods), because it is more flexible for mesh generation purposes, as domains to be meshed and desired mesh

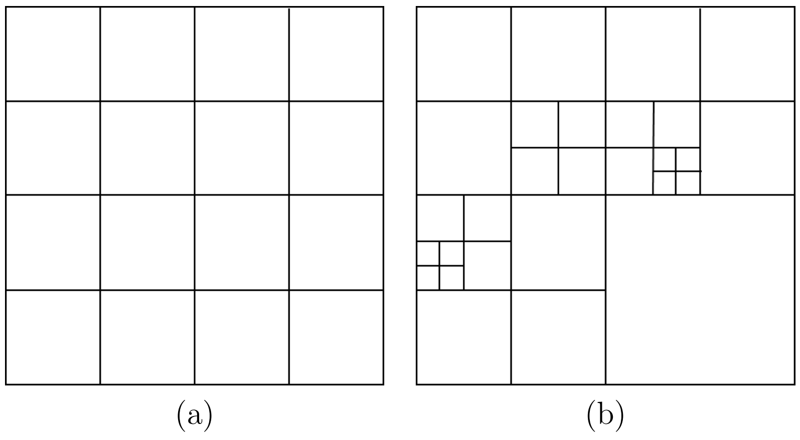


Figure 2.3: 2D examples of typical structures used in space decomposition based meshing algorithms. **(a)** Bin. **(b)** Quadtree.

sizes are commonly non-homogeneous. Octree-based meshers were pioneered by Yerry and Shephard in [YS84] and, since then, several approaches have been proposed [Sch96, LS07, MV92, BEG94, Mar09, SG91, QZ10].

The octree structure was thought for the first time for space searching purposes [Sam06], and the specific topology of the spacial decomposition it represents (detailed in Section 3.2) gives several advantages for mesh generation. Somehow, an octree can be considered a mesh itself (it can be thought as a non-conformal hexahedra mesh), so it is really natural to build a mesh from it.

Although several algorithms have been proposed parting from the octree-based family of methods, almost all of them follow three main steps:

- Generate the regular grid for the space decomposition.
- Generate the elements of the final mesh directly using given patterns from the cells of the regular grid.
- Fit somehow the boundaries of the domain.

Concerning the first step, as it has been pointed out before, the octree is the most common structure used for the space decomposition. From the theoretical point of view, all octrees are similar, but depending on the way the octree will be used, different implementations have been proposed by several authors [SF02, Sam06] in order to improve the efficiency of the octree, the performance for searching processes, the optimization considering the memory needed to store it, etc... More details on the implementation of the octree are presented in Section 6.2.

The generation of the elements of the final mesh from the octree is a simple process. It is based in creating the mesh elements directly from the *octree cells* (the definition of octree cell, as well as other octree related basic concepts are explained in detail in Section 3.2). Some of the existing methods apply different splitting patterns from the cells to get tetrahedra [YS84, YS83, LS07]. Other methods can get directly the cells of the octree as hexahedra elements of the final mesh (in cases where the final mesh is not needed to be conformal), or create transition elements when two neighbors present hanging nodes [QZ10]. [Mar09] proposes a different approach: create special cells where the octree is refined (where the neighbor elements are not conformal) and build the dual of the octree. The cells of it are directly the elements of the final mesh. With this approach a final mesh of conformal hexahedra is obtained automatically.

The key difference of each method remains in the third step, which is the most complex one. As the octree is a regular space decomposition, their cells do not fit exactly the contours of the domain to be meshed. Getting only the elements coming from the cells which are in the inner part of the domain (or even the ones intersecting its boundaries), the contours of the final mesh are staircase-like, so they are not able to represent smooth shapes with a given curvature.

Considering the inner cells (the ones totally inside a volume) and the interface ones (the ones colliding with the contours of a volume), different strategies have been proposed to fit the contours:

- Get only the inner cells and fill somehow the empty space between them and the real contour of the domain. This family of methods profits from the main advantages of octree-based methods for the inner part of the domains, but presents the same limitations near the contours as the methods used to fill those empty spaces.
- Project onto the contour of the domain the boundary nodes of the elements created from the inner cells using different techniques [Sch96, BEG94]. This strategy warps the nodes coming from the octree, so the shape of the octree changes, but its topology remains the same. [QZ10] proposes a pillowing technique near the contours to avoid bad shaped elements (hexahedra) after mapping the nodes.
- Move some nodes and split the elements intersecting the contours of the domain in order to represent it precisely [LS07].

Although these methods achieve the smooth representation of the contours, they have to follow specific strategies to preserve the geometrical features (corners or sharp edges). [SG91] proposes a re-tetrahedralization of the octants of the octree containing sharp edges using advancing front or Delaunay technique, taking into account the intersection points between the sharp edges and the octree cells. [Mar09] follows a strategy based on detecting which triangle from the input boundary the final nodes lay onto, and assuming that if two nodes lie onto two triangles connected by a sharp edge, there should be a sharp edge between that nodes of the final mesh. This strategy is not so robust, as it assumes that the sizes of the final elements is quite similar to the sizes of the triangles of the contour, so the triangles where two neighbor nodes of the final mesh lie onto are supposed to be neighbors connected by an edge. This is not a general situation.

As it has been explained, several approaches have been proposed departing from the octree-based family of methods. Although each approach has its own characteristics, some common advantages can be detected:

- *Robust.* The operations involved in most common octree-based methods are designed to be robust independently on the tuning of their control parameters. This is applicable to the operations dealing with the octree itself, but some processes dealing with the mesh generated (specially for body-fitted meshes) may not be so robust.
- *Fast.* As mentioned above, the octree is a structure used to make faster the searching algorithms in space. Indeed, almost all the meshing algorithms uses an octree as a tool for searching purposes. In this case, this structure is also the base of the mesh generation itself, so the algorithm takes profit from it to improve its performance. Furthermore, the inner elements of the mesh are created directly based on a division pattern of the octree cells, which is really fast in comparison with other methods which have to take care on geometrical aspects when generating each element.
- *Naturally parallelizable.* The octree itself is a partition of the space, so it is really easy to identify the parts of the domain affected when some part of the mesh is modified or generated. Furthermore, typical operations involved in an octree-based mesher are performed at octree cell level, and most of them are independent from one cell to another, so the parallelization of some parts of the algorithm is almost automatic. Several parallel implementations of the octree structure have been carried out taking advantage on these features [TOG05, CC12].
- *Good quality meshes.* As the mesh elements in these methods come from a predefined

pattern of decomposition of an octree cell, all the inner elements are in a range of qualities known a priori. However, the elements in the contour of the domain are typically more distorted (depending on the method used).

- *Allow not cleaned geometry as input.* Most octree-based methods only take care on the position in space of the boundaries of the domain rather than a topological relationship between the contour entities. This is a key point to allow some of the octree-based meshers to work with non-watertight input geometries. This property also makes octree-based meshers independent from the quality of the contour meshes used as input.

The main drawbacks of octree-based methods are:

- *Naturally not constrained.* The space decomposition of the octree does not respect the topology of the model. Octree-based methods are often not constrained, and the topology preservation is usually linked to the sizes of the octree cells used. This ends with a need for the user of the mesher to assign properly the desired sizes of the mesh, taking into account some kind of characteristic size of the model in specific regions of the domain.

- *Hard to preserve geometric features.* This drawback is strongly linked with the previous one: as the mesher is not constrained, it is hard to preserve sharp edges or corner points from the input data in the final mesh. Furthermore, cases where the sharp edges involve a small dihedral angle are very unfavorable for these methods, as the octree cells are very regular by its nature: it is hard to represent a small angle parting from a regular shape.

- *Predefined size transitions.* As most elements come from a direct pattern from the octree cells, the size of an element in comparison with the size of its neighbor is strongly related with the sizes ratio between neighbor cells in the octree. This link makes quite stiff the possibility of applying a smooth size transition in the final mesh. For some simulation methods, the meshes obtained by octree-based meshers present (locally) too sharp variations in the sizes of neighbor elements.

- *Alignment of edges in a preferred direction.* Because of the regular partition of space the elements come from, these meshes tend to generate meshes aligned with it. This alignment may influence the solution of the numerical simulation depending on the method used.

- *Creation of too refined meshes.* Octree-based meshers tend to generate a large number of elements, specially when the model has sharp features. This is because these methods often try to guarantee a minimum element quality by refining the octree near them.

2.6 Proposed solution

The strategy chosen in this work to cover all the requirements described in Section 1.2 is to develop an octree-based mesher. The election of an octree-based mesher in this work has been made taking into account the main advantages and disadvantages of the different methods:

- Structured and semi-structured meshers have been disregarded because the requirement for the input geometry to present a given topology is so restrictive that it makes them non applicable for many complex cases.
- Advancing front like methods have not been taken into account because the hard dependency they have on the quality of the input mesh for being successful in the mesh generation. Furthermore, they are directly not applicable if the input geometry is not watertight, unless some geometry repairing process is applied prior to the meshing itself.
- Delaunay methods have not been taken into account because they are not naturally constrained, and the proposals for them to be constrained blur some of its advantages. Furthermore, Delaunay methods are not naturally parallelizable, and they often generate localized bad quality elements.
- Octree-based methods are fast, robust and lead to very good quality meshes. Their main disadvantage are the inability to preserve the input topology, and their not constrained nature. However, the advantages of this family of methods, together with the ideas presented in this thesis to overcome these drawbacks have tipped the balance to these methods.

As explained in Section 1.2.4, although the main objective of this work is to develop a new volume mesher, the methodology proposed is applicable to generate meshes of 3D surfaces and lines not belonging to any volume. The case of lines is automatically solved with the special treatment of line elements in the volume mesher (Section 5.3.1), and some adaptations are maid to the volume mesher in order to mesh surfaces as it is explained on Section 5.3.10.

Chapter 3

Basic concepts of the new mesher

This chapter focuses on defining the different concepts involved in the proposed meshing algorithm, as well as some auxiliary algorithms needed to understand it. The following concepts will be described:

- Input data (Section 3.1). The input data needed for the mesher is defined, as well as the way of interaction of the mesher with a CAD system.
- Octree structure (Section 3.2). As the octree is the base structure for the mesher, an introduction to it is carried out in this section, highlighting its main characteristics.
- Octree properties for mesh generation (Section 3.3). Some specific properties of the octree are used in the meshing algorithm. A detailed explanation of them, as well as the key notation is introduced.
- Geometrical intersections (Section 3.4). The way the new algorithm deals with the geometrical intersections is presented. Also, the notation of pathological intersection types is introduced. This issues will be referred to in other parts of the work.

3.1 Definition of input data

The essential input for the mesher is the geometrical definition of the boundaries of each volume of the domain. As indicated in Section 1.2.2, this definition can be carried out using CAD or mesh entities. In this document the general concepts of *surface*, *line* and *point entities* will be used for both representations.

At this point, it has to be commented that the mesher considers the outer part of the domain as another volume to be meshed. It takes the name of *outer volume*, or volume number zero. Of course, the outer volume extends until infinite and it has no sense to consider it as a closed volume, so it is treated in a special manner. Later on it will be explained in more detail how the mesher deals with it.

Together with the surface entities an extra information is needed: the identification of the volumes each surface entity is interfacing. Note that considering the outer part of the domain as a virtual volume, all the surface entities defining the domain are interfacing two volumes. It may have sense for a surface entity to interface more than two volumes, but this would imply overlapping definitions of the 3D space (parts of space belonging to more than one volume). These kinds of topology are not considered in the present work.

Apart from the geometrical definition of the boundaries of each volume of the domain, extra information can be given to the mesher in order to specify some characteristics of the final mesh. It is important to note that this extra information is optional, as the mesher should generate the mesh of the domain with or without it. This information is given by the *mesh size entities*, the *forced point entities*, the *forced line entities* and the *general parameters*. Hereafter the characteristics of these data are detailed:

- *Mesh size entities.* The mesh size entities are geometrical entities used to provide the mesher with the desired size of the mesh in different regions of the 3D space. These geometrical entities can be point, line, surface or volume entities (both in mesh or CAD form), and they have a desired mesh size associated to them. These entities can be part of the input boundaries or not. If a desired mesh size is needed to be assigned to a volume, it can be provided just with the identification of the volume, without the need of creating the corresponding geometrical entities (volume entities in this case).

- *Forced point entities.* These are positions in space where the final mesh is forced to have nodes. They can be part of the input boundaries or not.

- *Forced line entities.* These are line entities to be preserved in the final mesh. This means that the final tetrahedra mesh will have a connected path of edges identifiable as a linear mesh of each forced line entity. As an example, the line entities in Figure 3.1(a) can be considered as forced line entities, as a path of connected edges representing it can be identified in the mesh shown in Figure 3.1(b). Forced line entities can be part of the input boundaries or not.

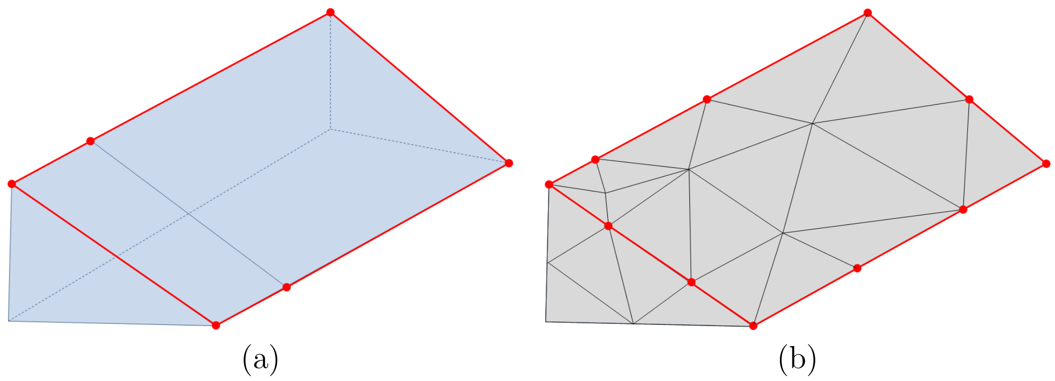


Figure 3.1: **(a)** Contours of a volume highlighting some of its forced line entities. **(b)** View of the tetrahedra mesh of the volume highlighting the sharp edges corresponding to the forced line entities in (a).

- *General parameters.* These are some global parameters which govern the behavior of the mesher and are not attached to geometrical entities. These parameters are optional, as the mesher should work without them, but they can help to fit the final mesh into the requirements of the simulation. Hereafter there is a list of these parameters:
  - *General mesh size.* This is the mesh size desired in the regions of the domain where there are no mesh size entities. If this parameter is not entered, the general mesh size is taken as the size of the bounding box of the domain.
  - *Size transition factor.* It controls the behavior of the size transition function. The definition and use of this function is explained in Section 5.2.2, and it basically determines if the size of the elements grows faster or slower from the areas where the mesh size is smaller to the ones where it is larger.
  - *Maximum angle for sharp edges.* This is the dihedral angle below which an edge of the input geometry is considered to be preserved in the final mesh.
  - *Maximum chordal error.* Two values may be needed: a relative and an absolute limit for the chordal error allowed for the mesh to be generated.
  - *Maximum distance between overlapping boundaries.* This parameter is named tol_{nw}. It is only applicable in case of non watertight geometries. It represents an estimation of the maximum distance between overlapping surfaces interfacing the same volumes. The definition and use of this parameter is explained in Section 3.4.

Apart from all these data, extra information can be attached to the entities defining the input boundaries: the *external data*. The external data can be of any nature and it has no relevance for the meshing process, but the mesher will transfer it to the corresponding entities of the final mesh. As an example, if a surface entity defining part of the boundary of a volume has some external information, the nodes of the final mesh placed on that surface entity, or the faces of the tetrahedra with all its nodes onto it will have the same external data attached to them.

3.1.1 Integration with CAD data

To prepare all the data needed for a numerical simulation it is common to use a software tool: the pre-processor. As explained in Chapter 1, part of these data is the mesh representing the geometry of the domain. As the geometry of the domain is often provided in a CAD format, pre-processors typically are forced to work with CAD data and generate the meshes for the simulation. This is the reason why pre-processors are often CAD systems, and they include several meshers inside.

From the point of view of a general pre-processor or a CAD system willing to use the presented mesher, there are some interesting aspects to be considered. Typically the geometrical entities inside these systems have several information attached related with the simulation data (boundary conditions, material properties, etc...) or to the CAD system structures (layers grouping the entities, topological information, etc...). This information must be transferred to the generated mesh, which implies the following requirement for the mesher: it should return specific meshes representing given geometrical entities (not only volumes, but also curves and surfaces).

The mesher should provide not only with the tetrahedra generated, but also with triangular meshes (made of triangles which are faces of the tetrahedra) and linear meshes (made of linear elements which are edges of the tetrahedra). Actually point entities can also be identified with a final node in the mesh, just by setting the corresponding point entity as a forced point entity.

The way to make the mesher returns (apart from the tetrahedra meshes of the volumes of the domain) the mesh of some line or surface entities from the input data is explained hereafter.

How to get the mesh of a line entity:

- Set the line entity as a *forced line entity* in the input data.
- Assign a specific *external data* to the line entity.
- The mesher will return a set of linear elements with the same external data attached.

How to get the mesh of a surface entity:

- Considering the line entities which are contour of the surface entity, set them as *forced line entities* in the input data. Note that the contour line entities of a surface entity form a closed path of line entities (actually, more than one set of closed paths can be present when the surface entity has holes).
- Assign a specific *external data* to the surface entity.
- The mesher will return a set of triangular elements enclosed by the forced line entities with the same external data attached. This situation can be seen in Figure 1.6: the patch of triangles in Figure 1.6(c) is the surface mesh of the surface entity represented by the highlighted triangles in Figure 1.6(a) (region A). To achieve this situation, the line entities surrounding the region A should be set as forced line entities.

Following this mechanism, the pre-processor can obtain not only the mesh of the volumes of the domain, but also the mesh of any line or surface entity. Once a mesh of an entity can be identified, all the information of the input entity can be transferred to the corresponding part of the final mesh.

3.2 Octree structure

As the octree is a key structure in the proposed mesher, a brief introduction is given in this section to highlight its characteristics.

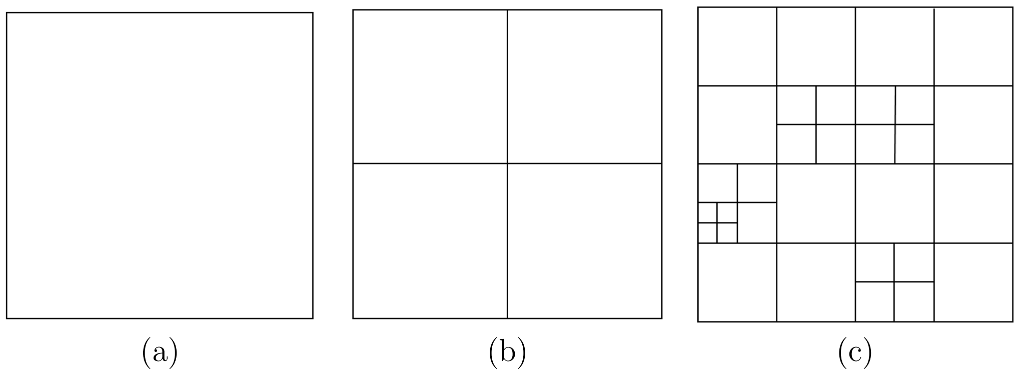


Figure 3.2: Example of a quadtree structure. **(a)** The root cell of a quadtree. **(b)** Root cell subdivided in 4 cells. **(c)** Example of a quadtree refined 4 levels.

An octree (quadtree in the 2D case) is basically a hierarchical spacial structure that partitions the space [Sam06]. The basic structure of an octree is the *cell*, which is a cubic portion of space (square in the 2D case). Actually, the cell can be a parallelepiped (or parallelogram in 2D). In this work the octree used is an homogeneous one, which implies that the cells are regular parallelepipeds (cubes). From a first cell which is the bounding box of the space to be partitioned (the so called *root* of the octree), a successive subdivision can be performed, where each cell is subdivided in eight cells (four in 2D case). These eight cells are the *sons* of the cell they come from, which is their *father*. Cells with no sons are called *leaves*.

In the Figure 3.2 a graphical view of the root of a quadtree and different levels of refinement are shown.

Considering the root of the octree, it can be equilateral or not. This, together with the way the cells are subdivided, leads to different configurations of the octree (Figure 3.3). In particular, for notation purposes, an octree accomplishing the following two properties receives the name of *isotropic octree*:

- The octree root is equilateral.
- The division of a cell results in eight sons, and the division criterion is equidistant (the eight sons of a cell are identical). In other words, the division of a cell is done parallel to its faces passing by its center.

To fix the notation, some interesting concepts related with the octree are detailed hereafter:

- In a cell there can be identified vertices, edges and faces (analogously as the nodes, edges and faces of an hexahedral mesh element).
- The cell size is the length of the edge of the cell.
- Two cells are *neighbors by face* if they are in contact by a face.
- Two cells are *neighbors by edge* if they are in contact only by an edge.
- Two cells are *neighbors by vertex* if they are in contact only by a vertex.
- An octree is refined if some of its leaves is subdivided.
- The *degree of neighborhood* between two cells is defined as the minimum number of cells needed to go from a cell to the other traveling by neighbor cells. As an example, the degree of neighborhood between two neighbor cells is one.

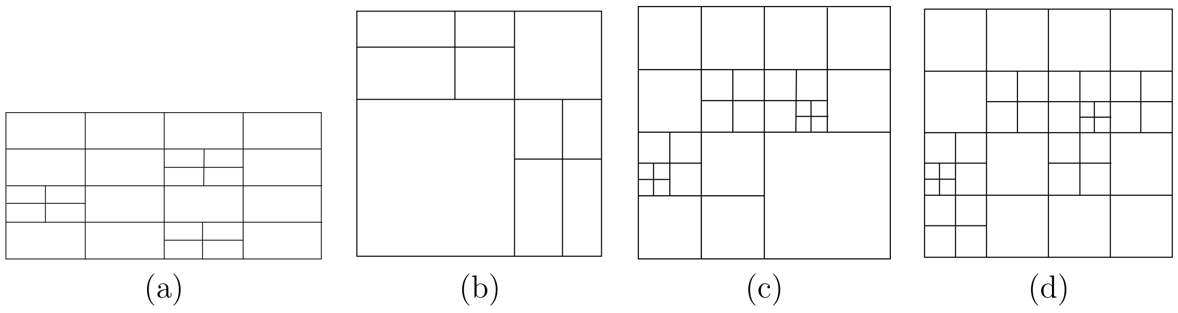


Figure 3.3: Different kinds of quadtree. **(a)** Quadtree with a non-equilateral root, with a equidistant cell division criterion. **(b)** Quadtree with an equilateral root, with a non-equidistant cell division criterion. **(c)** Isotropic non-balanced quadtree. **(d)** Isotropic balanced quadtree.

- A maximum depth of cells subdivision can be defined. Lets call it N_L. A *level* can be assigned to each cell, so as: the root has level equal to $N_L - 1$, and the sons of a cell has one level less than their father. Note that the smallest cell of the octree has a level equal to 0.

3.3 Specific octree properties for mesh generation

As explained in previous sections, the octree structure was thought for the first time for space searching purposes [Sam06]. In this section, the adaptations to the octree structure done in order to use it for mesh generation and its main properties to understand the algorithm are explained.

The decision of using the octree for isotropic mesh generation leads to use an isotropic octree. This decision has an important relevance at the time of implementing the algorithm (as it will be seen in Section 6.2).

The proposed meshing algorithm should be valid using other kinds of octree (like the analogous quadtree examples shown in Figures 3.3(a) and (b)), but it would lead to non isotropic meshes.

Another important characteristic of the octree chosen for the method is the so called *constrained two to one* condition. This is a widely used condition in octree based meshers, and limits the number of neighbors of a cell. The *two to one* name comes from the two dimensional case (quadtree), and limits the maximum number of neighbors of a cell to two. In the octree case (3D), this condition implies that a cell cannot have more than four neighbor cells by face, or two by edge. In the present document an octree accomplishing the constrained two to one condition is referred as a *balanced* octree. Two configurations of an isotropic quadtree (non-balanced and balanced) are shown in Figures 3.3(c) and (d).

The main reason to use a balanced octree for the proposed meshing algorithm is to simplify the patterns to build the tetrahedra from the octree cells, ensure a better quality in the final tetrahedra and avoid a very strong sizes transitions in the final mesh. The tetrahedra generation process from the octree has the following characteristic: the more difference between sizes of neighbor cells, the worse aspect ratio will have the tetrahedra generated from them.

3.3.1 Octree cell types

As it has been pointed out in previous sections, octree cells are the result of the space partitioning by the octree structure. Besides this, they have some information attached as the input boundary entities colliding with them.

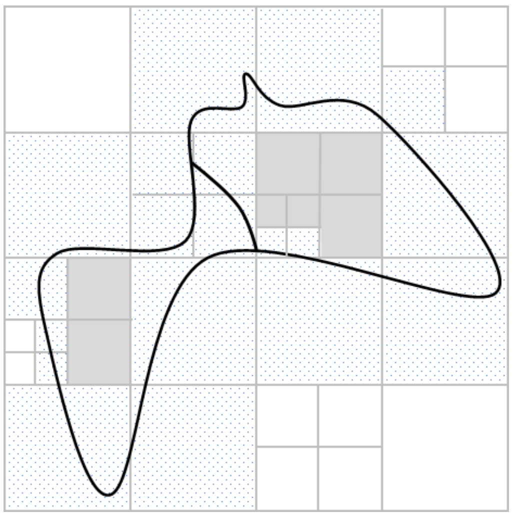


Figure 3.4: 2D example where the three kinds of cells can be identified. The black curved line defines the contours of a domain formed by two surfaces (which are in contact), and the light gray lines represent the octree. Outer cells are the white ones, interface cells are marked with dots, and inner cells are colored in gray.

In the frame of this work, the octree cells are classified in three categories:

- *Interface cells.* These are the cells colliding some input boundary entity. This means that some input boundary entity is inside them, or intersects them. Also the cells containing a *forced node* (Section 5.3.2) are considered as interface cells.

- *Outer cells.* These are the cells which are completely out of the domain to be meshed and they do not contain any forced node.
- *Inner cells.* These are the cells which are not interface or outer ones. These cells can be identified with some volume of the domain, as they are totally inside one volume.

Figure 3.4 shows a 2D example where the three kinds of cells can be identified.

3.3.2 Octree positions and nodes

As it is explained in Section 3.3.3, tetrahedral elements will be generated following a pattern from the octree. The nodes of these tetrahedra are called *octree nodes* and they are assigned to some predefined positions in space: the *octree positions*. Each cell has 27 octree positions corresponding to the vertices of the cell (8), the center of the cell (1), the center of its edges (12) and the center of its faces (6). A graphical view of the octree positions of a cell is shown in Figure 3.5.

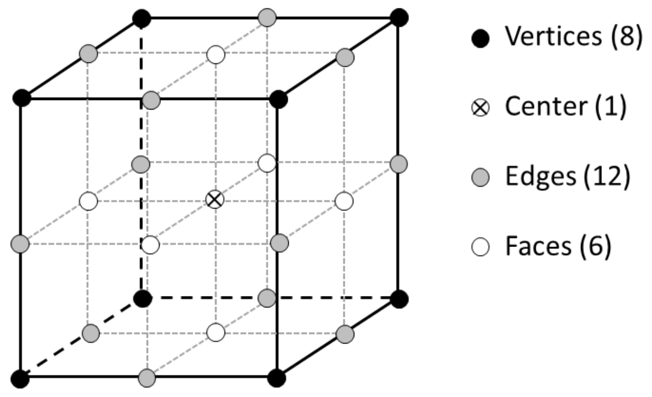


Figure 3.5: Octree positions of an octree cell. The cell is represented by the black lines.

It has to be noted that an octree position can be shared by more than one cell, as it can be seen in Figure 3.6.

The *linear positions* of an octree cell are defined as the ones corresponding to the vertices and the center of the cell. The other positions are called *quadratic positions*. As an extension, the term of *linear* or *quadratic* octree node can be used to refer an octree node associated to a linear or quadratic cell position.

When referring to the whole octree, a linear position (or octree node) is the one which is linear in some cell. It has to be noted that an octree node can be linear regarding one cell,

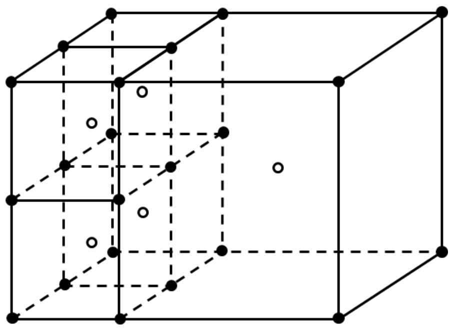


Figure 3.6: Linear octree positions of a part of an octree. White dots are center of cells, and black dots correspond to vertices of cells.

but quadratic from the point of view of another cell containing it. The linear octree positions of a part of an octree are shown in Figure 3.6 as an example.

It is important to note that not all the octree positions are forced to have an octree node associated, but all the octree nodes are linked to an octree position.

In Section 5.3.2 the concept of forced node is introduced. It basically corresponds to a node linked to an octree position, but occupying a different position in space.

3.3.3 Tetrahedra patterns

Parting from a given octree configuration, there are several possible tetrahedra patterns to be applied in order to split it. One important consideration is that the pattern chosen must fill the space with tetrahedra in a conformal way: it must not leave hanging nodes. A hanging node is a node lying on an element not being a vertex of it (it is on an edge or a face).

The option chosen in this work is based on the body centered cubic (BCC) lattice, which lead to a space-filling tetrahedra [Som23]. The use of BCC patterns linked to octree structures is quite natural considering the spacial distribution of the octree, and was proposed by [Fuc98, Nay99]. This option fills the space in a conformal way with a set of identical high quality tetrahedra: dihedral angles are 60 or 90 degrees, and edge lengths are 1 and $\frac{\sqrt{3}}{2}$ times the cell size. This provides the tetrahedra with an edge ratio of 1.155 (the ratio of the longest and shortest edges of the element).

A BCC lattice based pattern is local in the sense that each tetrahedra generated only depends on one cell and one of its neighbor. A pattern only depending on each cell (independent from the neighbor ones) would be more efficient for parallelization. However, this kind of patterns often provides with a lower quality tetrahedra (although their quality is acceptable) and a larger number of them [Nay99].

The BCC lattice is defined in a regular grid (an octree with all its leaves equal-sided). As the octree used in the proposed method is not regular (has leaf cells in different levels), other tetrahedra patterns must be defined.

To reach a tetrahedra mesh with no hanging nodes, all the linear octree nodes are used, and (in case they exist) the forced nodes linked to quadratic positions (forced nodes are defined in Section 5.3.2).

Basically, the tetrahedra pattern proposed focuses the tetrahedra generation cell by cell, and focusing on one cell, using its six faces independently. The only information required to generate the tetrahedra from a face F of a cell C is: the octree nodes of the face F, the center octree node of the cell C and the neighbor cell of C from the face F, denoted by C_{neig}.

As explained previously, the octree used is balanced (it accomplishes the constrained two to one condition). This ensures that the level of C_{neig} is one less, equal, or one more than the level of C. These three configurations lead to the different tetrahedra patterns:

- C and C_{neig} have the same level (this means they have the same size). In this configuration the center node of C_{neig} is used to build the tetrahedra. Taking into account that the face F (the face in contact between C and C_{neig}) has four sides (edges), from each edge e from the face F one, two or four tetrahedra are generated, depending on the possible presence of octree nodes in quadratic positions. The tetrahedra generated under each situation are shown in Figure 3.7.

  The minimum dihedral angle of the tetrahedra generated in this configuration is 45 degrees, and it occurs when some node in a quadratic position is used.

  This case involves the creation of tetrahedra in two cells. Hence, only one of them should create them to avoid repetitions. The criteria used is that cell C creates the tetrahedra only if it is *lower* than C_{neig}, otherwise the tetrahedra will be created by C_{neig}. Considering two leaves (A and B), A is *lower* than B if the x coordinate of the center of A is lower than the x coordinate of the center of B. If the x coordinates are equal, the y coordinate is checked, and if it is also equal, the z coordinate is used to compare the cells. It has to be noted that, because of the characteristics of the octree structure, there cannot be two leaves with their centers in the same position.

- C has one level more than C_{neig} (this means C is bigger than C_{neig}). In this configuration eight tetrahedra are created from the face F. These tetrahedra are shown in Figure 3.8(a). It has to be considered that if C is bigger than C_{neig}, the constrained

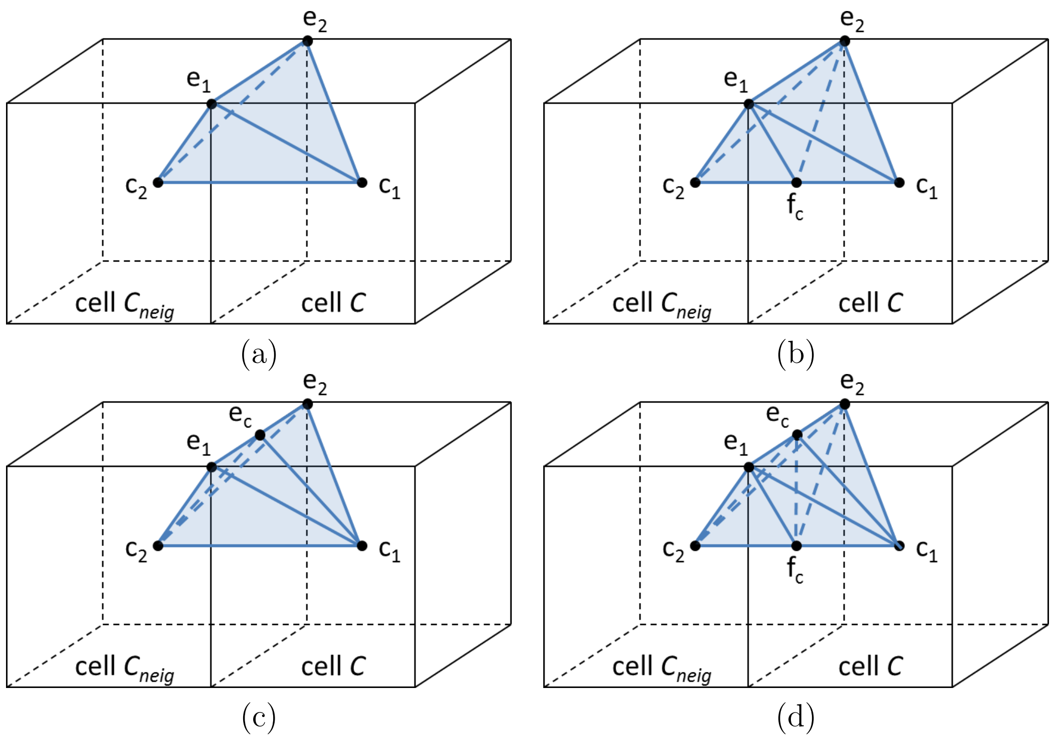


Figure 3.7: Tetrahedra pattern in case where cell C has the same level as C_{neig}. Tetrahedra generated from edge e ($\overline{e_1e_2}$) of common face between C and C_{neig} in different situations. **(a)** No quadratic octree nodes involved: one tetrahedron is generated ($c_1e_2c_2e_1$). **(b)** Octree node in the center of face F: two tetrahedra are generated($c_1e_2f_ce_1$ and $f_ce_2c_2e_1$). **(c)** Octree node in the center of edge e: two tetrahedra are generated($c_1e_2c_2e_c$ and $c_1e_cc_2e_1$). **(d)** Octree nodes in the center of face F and edge e: four tetrahedra are generated($c_1e_2f_ce_c$, $c_1e_cf_ce_1$, $f_ce_2c_2e_c$ and $f_ce_cc_2e_1$).

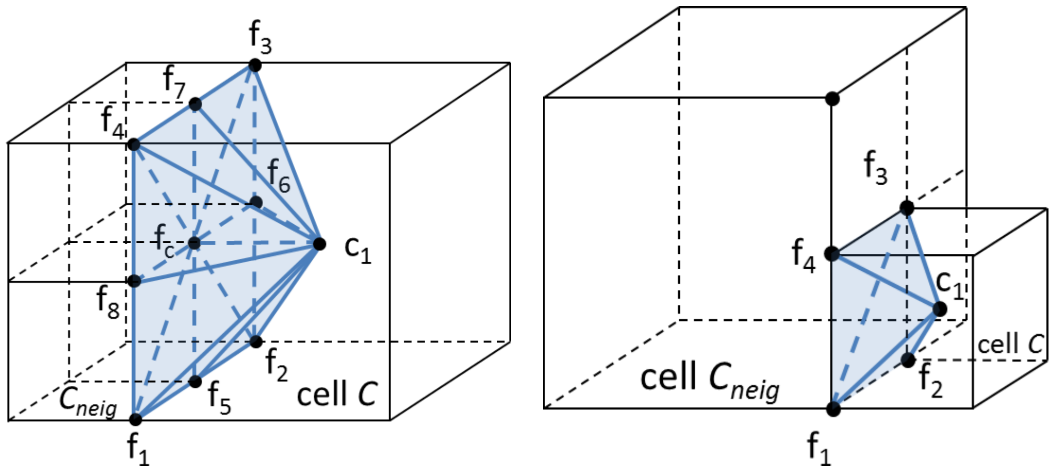
two to one condition ensures that there will be four cells neighbors of C in the other side of F.

The minimum dihedral angle of the tetrahedra in this configuration is 45 degrees.

- C has one level less than C_{neig} (this means C is smaller than C_{neig}). In this case two tetrahedra are created from the face F, shown in Figure 3.8(b). Note that this configuration could build the two tetrahedra crossing the face F by diagonals f_1f_3 or f_2f_4. To ensure the final mesh to be conformal, the diagonal chosen is the one passing by the octree node which is the center of face in C_{neig}.

  The minimum dihedral angle of the tetrahedra in this configuration is 45 degrees.

It can be seen that the quality of the tetrahedra generated using these patterns depends



(a) (b)

Figure 3.8: Tetrahedra pattern in case where cell C has different level than C_{neig}. **(a)** C is bigger than C_{neig}: 8 tetrahedra are generated ($f_1 f_5 f_c c_1$, $f_5 f_2 f_c c_1$, $f_2 f_6 f_c c_1$, $f_6 f_3 f_c c_1$, $f_3 f_7 f_c c_1$, $f_7 f_4 f_c c_1$, $f_4 f_8 f_c c_1$ and $f_8 f_1 f_c c_1$). **(b)** C is smaller than C_{neig}: two tetrahedra are generated($f_1 f_2 f_3 c_1$ and $f_3 f_4 f_1 c_1$).

on the configuration chosen, but it is always very good: the minimum dihedral angle of any tetrahedra coming from this predefined patterns is 45 degrees.

The case where a cell has no neighbor by one of its faces occurs in the cells in contact with the boundaries of the octree root. This case is not considered because the building of the octree root (Section 5.2.3) ensures these kind of cells are totally out of the domain to be meshed, so they are not used to generate any tetrahedron.

3.4 Geometrical intersections

Geometrical intersections play a key role in the new meshing algorithm and affect several parts of it. This section focuses in defining how this intersections are considered. Their treatment has special interest when the input boundaries are non-watertight.

The intersection operation involved in the algorithm is typically the one between a segment and the surface entities defining the contours of a volume. This segment is understood as a portion of a straight line between two points. Depending on the part of the algorithm, it could be referred with different names such as edge (when talking about tetrahedra or triangle edges) or ray (when talking about ray casting operations). In this document some figures are based in 2D examples for clarity purposes. The equivalent intersection operation in these

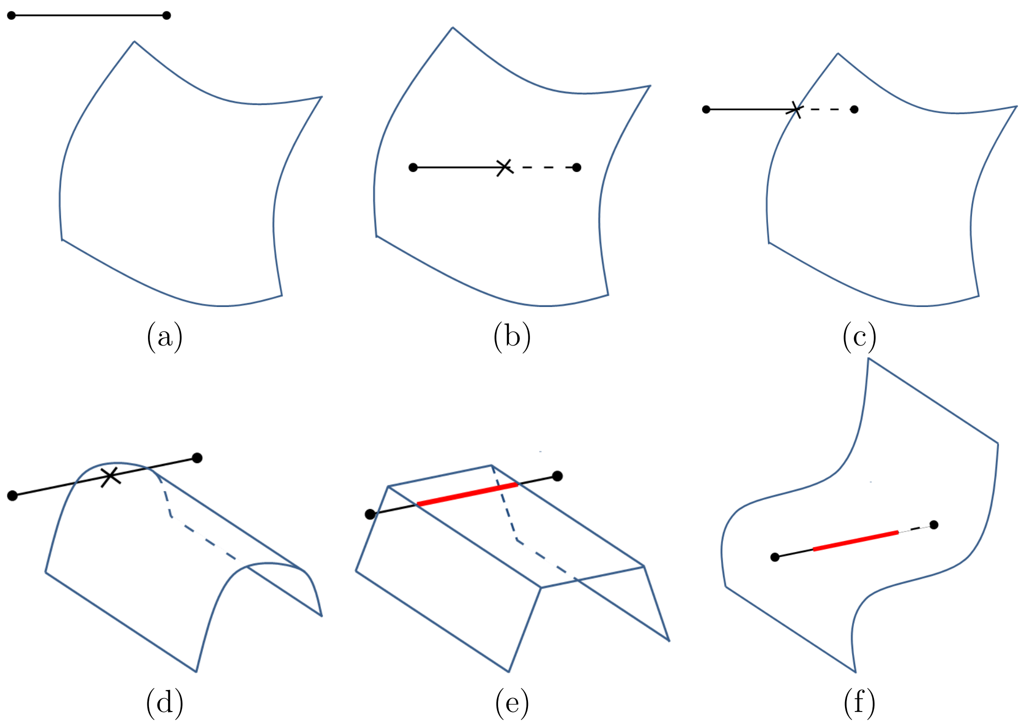


Figure 3.9: Types of intersections between a segment and a surface entity. Crosses are the intersection points. **(a)** No intersection. **(b)** S type intersection. **(c)** C type intersection. **(d)** T type intersection. **(e)** PT and **(f)** PI type intersection; the red thick part of the segment is co-planar with the surface entity.

cases is between segment and line entities defining the input boundaries.

Five situations are distinguished when evaluating the intersection between a segment and a surface entity (a graphical interpretation of these cases using simple examples is depicted in Figure 3.9):

- No intersection: the segment and the surface entity do not intersect (Figure 3.9(a)).
- Single intersection (S type): they intersect in one point within the surface entity (Figure 3.9(b)).
- Contour intersection (C type): they intersect in one point laying on the contours of the surface entity (Figure 3.9(c)).
- Tangent intersection (T type): they intersect in one point tangentially to the surface entity (Figure 3.9(d)) and the segment remains in the same side of the surface entity at both sides of the intersection point.

- Co-planar intersection (P type): the segment and some part of the surface entity are parallel and the distance between them is zero. This case can be split in two:
  - Tangent (PT type): before and after the intersection, the segment remains at the same side of the surface entity (Figure 3.9(e)).
  - Intersected (PI type): if the PT case is not accomplished (Figure 3.9(f)).

Analytically, the P type of intersection has an infinite number of intersection points. As it will be seen in further sections, these intersection types only take relevance in the node coloring process (Section 4), that determines inside which volume a node is. For this reason the treatment of P type intersections is detailed in Section 4.2.

Types S, C and T have only one intersection point. However, as the contours of a volume may be formed by more than one surface entity, each intersection may involve a different number of intersection points (one for each intersected surface entity). This situation is illustrated in the 2D examples of Figure 3.10 (as a 2D example, line entities play the role of surface entities in 3D). In Figure 3.10(b) a C intersection type has two intersection points (one for each intersected line entity), as well as in Figure 3.10(c) with the T type intersection.

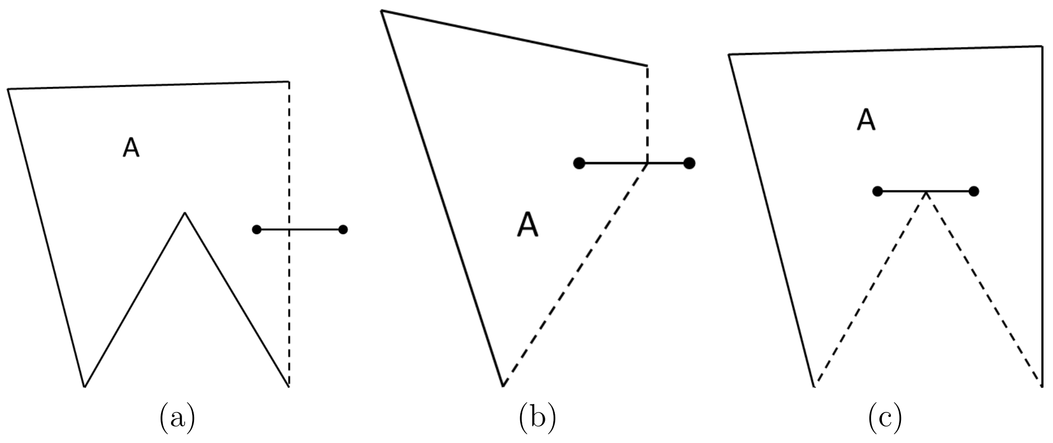


Figure 3.10: Line entities enclosing surface A intersected by a segment (bounded by two dots) presenting different intersection types. Intersected line entities are drawn in dotted line. **(a)** S intersection type. **(b)** C intersection type. **(c)** T intersection type.

These cases are characterized by the fact that all the involved intersection points are really close one from each other. Theoretically, all the intersection points should be in the exact same position, but because of the tolerances involved in the numerical computation of intersections this cannot be guaranteed. The tolerance tol_c is defined in order to determine if a collection of close intersection points corresponds to the same intersection: if all of them are within a distance lower than tol_c, they are collapsed into a *multiple intersection point (MIP)*. The value of tol_c is a portion of the model bounding box size:

$$tol_c = \alpha_{bb} \cdot s_{box} \tag{3.1}$$

being s_{box} the length of the minimum side of the model bounding box, and α_{bb} a real value between zero and one. The value used in the present work for α_{bb} is detailed in Section 6.7.

The position of the MIP corresponding to a collection of intersection points is the mean position between all of them. In the present work, when evaluating the intersection point between a segment and the contours of a volume, in cases where there are intersection points close enough, their MIP is considered instead of them. This matches with the theoretical number of intersection points corresponding to each intersection type: C and T intersection types have only one intersection point (a MIP).

3.4.1 Non-watertight geometries

For non-watertight geometries, special cases may be treated when evaluating the intersection between a segment and the contours of a volume. There are basically two specific intersection types (a 2D example of this cases is depicted in Figure 3.11):

- Overlap intersection (W type). This situation happens when the part of the boundaries of the domain where the segment intersects is defined by more than one overlapped surface entity (Figure 3.11(a)).
- Gap intersection (G type). This situation happens when the part of the boundaries of the domain where the segment intersects has a gap (Figure 3.11(b)).

For the W intersections type, the corresponding MIP of the intersection points involved is considered. For this purpose, in cases where non-watertight geometries define the domain, tol_c takes the value of tol_{nw}. This parameter must be provided in the input data and is a characteristic length of the gaps and the overlappings of the model. tol_{nw} must be an upper limit of the distance between overlapping entities. The MIP corresponding to the 2D example of Figure 3.11(a) is shown in Figure 3.12(a).

G intersections do not provide with any intersection point. However, as it will be seen in further sections, there are cases where the extremes of a segment are known to be inside different volumes. Situations where both extremes belong to different volumes indicate that

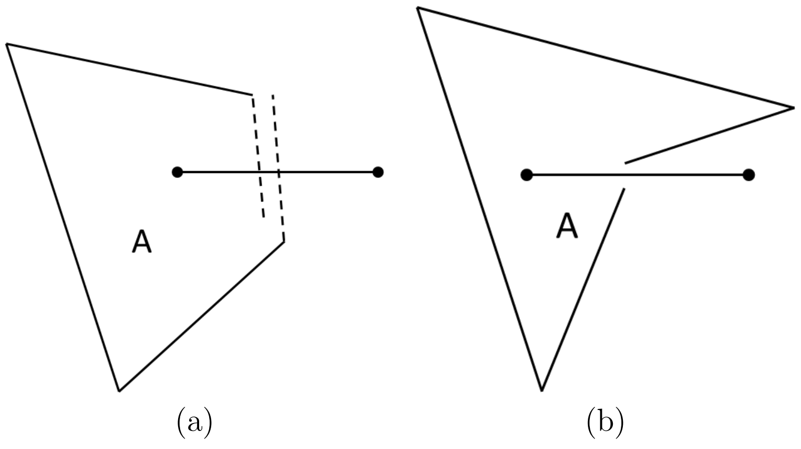


Figure 3.11: Intersection between a segment and the line entities enclosing surface A in non-watertight situations: **(a)** overlap (W intersection type) and **(b)** gap (G intersection type).

there should be an intersection point although it is not detected. In this cases the *gap intersection point (GIP)* is created. Considering the surface entities surrounding the segment, the closest point from each surface entity to the segment can be computed. The GIP takes the position of the closest one. The GIP corresponding to the 2D example of Figure 3.11(b) is shown in Figure 3.12(b).

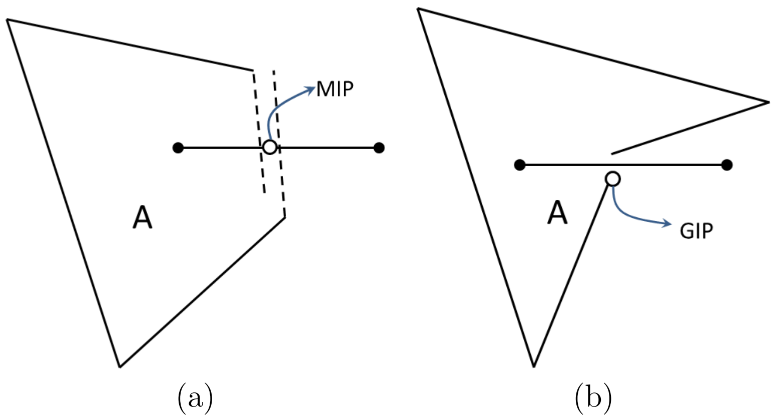


Figure 3.12: Treatment of intersections of non-watertight geometries depicted in Figure 3.11. **(a)** MIP corresponding to the W intersection depicted in Figure 3.11(a). **(b)** GIP corresponding to the G intersection shown in Figure 3.11(b).

In the present work, the intersection point between a segment and the contours of a volume could be a single intersection point, a MIP or a GIP.

Chapter 4

Coloring algorithm

As explained in Section 2.5, space decomposition meshing methods use a regular grid (in our case, an octree) over the domain to be meshed. The regular grid is larger than the domain, so at some point of the algorithm, this family of methods are forced to use a strategy to know if a given position of the grid is inside or outside the domain.

The coloring operation consists in determining where the entities are in the topological sense. This is, to determine whether each entity is inside a volume, out of the domain or laying on an interface entity between volumes. Applied to points, this operation is known in the literature as the *point-in-polygon (PIP)* problem [NRNTfI67, HS97, Sch08].

In the presented meshing method, the coloring operation is applied to nodes (the case explained in this section) and to tetrahedra (as explained in Section 5.3.7).

This is one of the key points of the meshing algorithm, as it is not obvious to determine if a point in the space is inside a volume or not when the contours of the volume are non-watertight. Actually, if the contour of the volume has gaps, the concept of interior or exterior of the volume is not even defined from the topological point of view.

It has to be noted that several existing octree-based methods are focus on meshing a domain formed by a single volume. In this work arbitrary domains with several volumes are meshed at once, so coloring a node is not reduced to identify if it is inside or outside the domain: it has to be determined inside which volume (or interface entity) is.

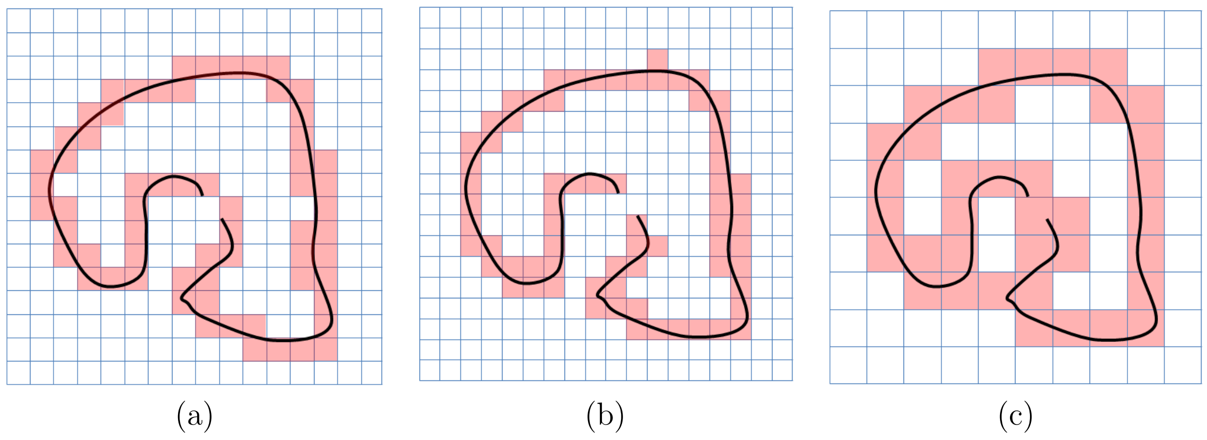


Figure 4.1: Example of three voxelizations of a 2D model. The model has a gap in its contour and it is represented by the black curved line. Contour voxels are drawn in red. **(a)** The size of the voxels is large enough to close the gap of the domain: the topology of the voxelized model is watertight. **(b)** The size of the voxels is too small, so the gap of the domain is not closed. **(c)** The size of the voxels is too large: the gap is closed, but the final topology does not represent correctly the domain.

4.1 Coloring strategies

There are several ways of coloring points considering a watertight definition of the volumes. These cases have some clear advantages as the contours of each volume can be oriented coherently (towards the interior or the exterior of the volume). This orientation provides with valuable information at the time of determining if a point near the contour entity is inside or outside the volume.

However, this work is focused on non-watertight geometries. This implies that a coherent orientation of the contours of a volume cannot be guaranteed. To deal with such geometries it is common to work with a *voxelization* of the model [ITN08, HYFK98, WK93]. These strategies aim to converting the non-watertight geometries into water-tight ones in order to be able to apply the coloring strategies of points. A voxel representation of a model is a regular grid (typically axis aligned and isotropic) where each voxel contains a topological information. In the case of study, it contains the color of the voxel. If a voxel is intersected by a surface entity it has the color of that surface (lets call it a *contour voxel*), otherwise it has the color of the volume it is into (*inner voxel*), or the color of the outer part of the domain if it is not inside any volume (*outer voxel*). All the points inside an inner voxel can be considered as interior to the corresponding volume. An example of voxelized 2D model is shown in Figure 4.1.

The advantage of working with voxelized models is that, depending on the size of the voxel, the topology of non-watertight geometries can be improved. If the gaps of the input boundary or the distance between overlapped contour entities is lower than the voxel size, the voxeled model may close the gaps or join the overlapped entities. This situation is shown in Figure 4.1(a), where the voxelized model presents a closed set of inner voxels (with no gap in its contour). As a drawback, the voxelized model can neglect some important topological information of the domain: the topology of the voxelized model shown in Figure 4.1(b) represents exactly the topology of the model (with its gap). Despite the size of voxels used in Figure 4.1(c) is large enough to close the gap (which is desirable), it represents an undesired topology, as it includes different sets of unconnected inner voxels.

The problem is that the voxel size needed to represent correctly the topology of the model cannot be estimated a priori and it may not be valid for the whole domain. Figure 4.1 shows that three different voxel sizes lead to three different topologies of the same domain.

If the voxelized model is watertight, one strategy for the coloring of the voxels is by propagation [ITN08]. This strategy consists in the following steps:

- Determine all the contour voxels (the ones colliding a contour entity).
- Determine one (and only one) voxel interior to each volume solving the corresponding PIP problem.
- From a known inner voxel, assign its color to its neighbour voxels if they are not contour ones.
- Repeat the last step until all the voxels have color.

This propagation method is totally robust in watertight representations and has the advantage of solving very few times (one for each volume) the PIP problem, which can be computationally expensive. However, it has the drawback that is not naturally parallelizable. Only the voxels which are neighbour of a determined voxel (a voxel with a known color) can be colored at a time.

Voxelizing the model can help the coloring strategy as it can make the model watertight, enabling the use of the propagation method (which requires the solving of PIP problem only in a few points). However, it has been seen that even the voxelized model may be non-watertight. In this case the color propagation cannot be done, so the PIP problem has to be solved for each one of the points to be colored (in our case, the octree nodes).

4.1.1 Ray casting

The solution chosen in this work for coloring the octree nodes is based on the *ray casting* technique. The ray casting algorithm was first developed by Arthur Appel for rendering purposes in 1968 [App68]. It proposes a solution for determining the visibility of a 3D object from a given point of view, and uses this information to paint a representation of the 3D object in a 2D image (made of pixels). The idea behind ray casting is to shoot rays (straight lines) from the point of view (one per pixel) and find the closest part of the object intersecting the ray. The algorithm needs to compute first all the intersections between a ray and the contours of the object (surface entities), and then get the closest to the point of view.

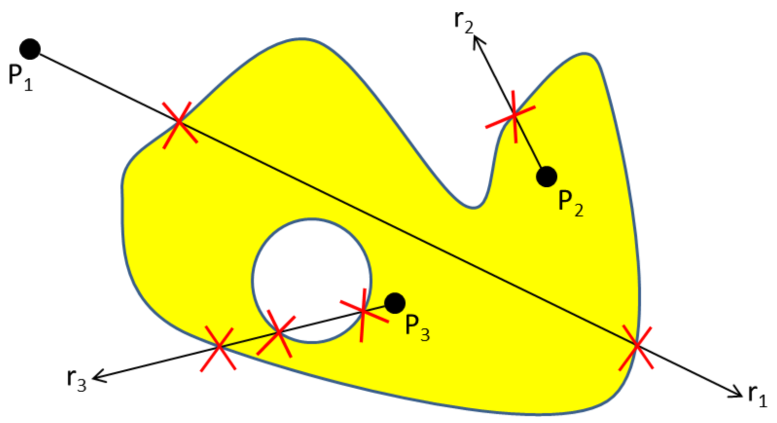


Figure 4.2: 2D example of the ray casting technique to solve the PIP problem. Point P_1 is considered outside of the polygon because ray r_1 has an even number of intersection points (2). Points P_2 and P_3 are considered inside of the polygon because rays r_2 and r_3 have an odd number of intersection points (1 and 3 respectively).

Despite the first applications of ray casting were focused on rendering of 3D objects, its use has been generalized for several purposes following the philosophy of analyzing the intersection points between the ray and given 3D objects.

A common use of the technique is to solve the PIP problem. Among the several existing methods to solve this problem [NRNTfI67, HS97, Sch08], a classical approach is the *ray intersection method* [NRNTfI67]. It consists in tracing a random ray from the point of analysis and compute the number of intersections between it and the contours surface entities. If the number is even, the point is outside the domain, and if it is odd, it is inside.

A 2D example illustrating the application of ray casting technique to solve the PIP problem is depicted in Figure 4.2. In this example, P_1 is considered outside of the polygon because the ray r_1 has two intersection points (an even number). Points P_2 and P_3 are considered inside of the polygon because their rays (r_2 and r_3) have an odd number of intersection points (1 and 3 respectively).

4.2 Ray casting-based proposed technique

As pointed out previously, in the present work the coloring process of a node must identify not only if it is inside or outside the domain, but also the specific volume of the domain it is into. To fit this requirement, the ray casting method in this work focuses more on the topological information of each intersection rather than in the number of them. Following this philosophy, the proposed method consists in tracing rays in space and take care about the intersection of these rays with the contours of the volumes: if a ray intersects the interface between volumes A and B, it can be ensured that, near the intersection point, at one side of the intersection point the ray will be inside A, whereas at the other side it will be inside B. Following this principle, if we move along a ray from a point which color is known, we can color all the points of the ray just looking at the intersections of it with the contours of the volumes. Figure 4.3 shows a 2D example where a ray beginning in the outer part of the domain is colored in different parts corresponding to the contours it is intersecting.

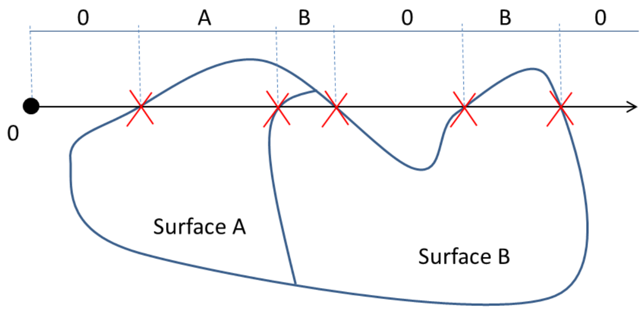
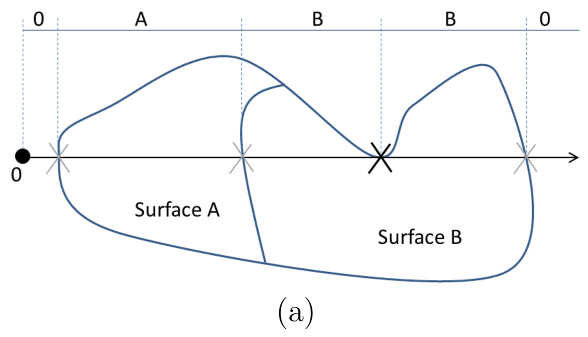


Figure 4.3: A 2D example of coloring parts of a ray (represented by the black arrow). The black dot is the beginning of the ray which color is known: 0. The crosses are the intersection points between the ray and the contours of the surfaces A and B (which are the ones forming the domain). In the upper line the color of the different parts of the ray is shown.

It is important to note that the presented technique only need the topological information of the surface entities: which volumes they are intersecting. Other techniques require the use of normal vectors in some points, or a given connectivity among the surface entities defining the contours of the domain. This is crucial considering the proposed algorithm must work with non-watertight definitions of the volumes. In these cases it is not obvious to define the oriented normal of a surface entity pointing towards the inner or the outer part of a volume.



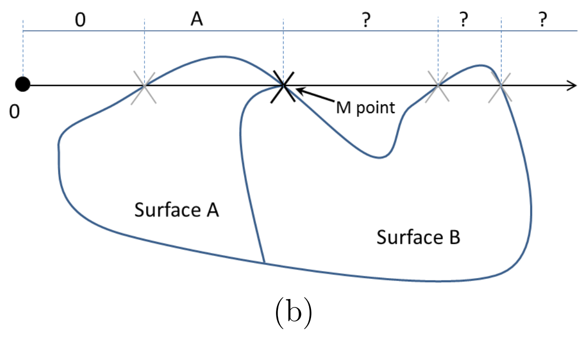


Figure 4.4: 2D examples of pathological configurations for the ray casting technique. **(a)** Both sides of the ray from the dark cross intersection are colored equally (surface B) although there is an intersection point because the ray is tangent to the contours in it. **(b)** M point is interface between surface A, B and 0 (outer part of the domain), so the color of the right part of the ray cannot be set.

4.2.1 Pathological configurations

The ray casting technique presents some pathological configurations [Sch08] for the PIP problem. One of these is the case when the intersection between the ray and the volume boundaries is done tangentially (T type intersection defined in Section 3.4). In this situation the ray intersects the boundaries of the geometry, but both sides of the intersection are in the same volume. To solve this problem, this kind of intersections (tangentially to the boundaries of the volumes) are not taken into account for coloring the regions of the ray. Figure 4.4(a) shows an example of this pathological case: both sides of the ray from the dark intersection point are inside surface B, although there is an intersection point.

Another pathological configuration occurs when the intersection point is a point of the boundary interfacing more than two volumes (in 2D case, more than two surfaces). This is the case shown in Figure 4.4(b). For notation purposes, this kind of intersections are referred as a M intersection type.

If a M intersection is detected in a ray, a color has to be chosen for the following part of the ray (from the M intersection point on). The strategy followed to decide this color is explained in the Implementation chapter (Section 4.3.1). It is based on a try and error approach considering all the possible colors.

The case of co-planar intersections (PT and PI intersection types defined in Section 3.4) presents also a pathological configuration for the ray casting technique. In these cases, the part of the ray co-planar with the boundaries must be colored with the color of the interface itself. In this case, the color does not correspond to a volume, but to an interface between

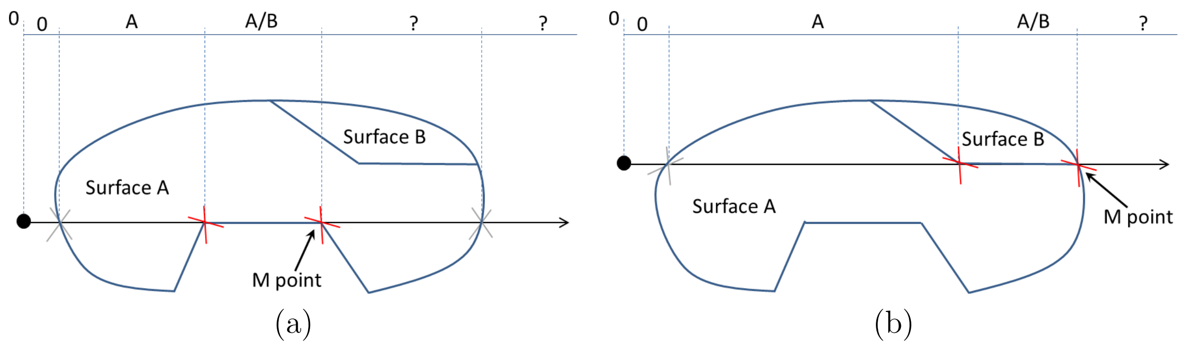


Figure 4.5: 2D examples of co-planar intersections of the ray. **(a)** PT intersection type. The color at the right part of the M point could be A or zero. **(b)** PI intersection type. The color at the right part of the M point could be A, B or zero.

volumes. 2D examples for PT and PI intersection types between the ray and the boundaries of a model is depicted in Figure 4.5. The end point of the co-planar part of the ray is considered as an M point from the coloring point of view.

For non-watertight contours of the domain volumes, two more pathological configurations affect the ray casting technique. They are related to the possible gaps and overlapping entities in the contours (G and W intersection types defined in Section 3.4):

- If a ray passes through a gap of the contours of a volume (G type intersection), it does not detect that is entering into the volume, so it would assign the following regions an invalid color. This is the case shown in Figure 4.6(a), where the ray casting technique only identifies two zones in the ray separated by the intersection point: the left region as outer, and the right one as inside surface A.

- When a part of the contour of a volume is overlapped (W type intersection), if a ray intersects the contour by that region, it intersects more than one time the contour. The case where the distance d between the overlapping entities is smaller than tol_c presents no problem, as only one intersection point is considered: the MIP. This can be understood as a local voxelization of the model, as the topological information of different entities is collapsed into one data (voxel). The case where d is greater than tol_c disturbs the coloring operation as the color of the ray changes two times instead of one. This is the case shown in Figure 4.6(b), where the part of the ray inner to surface A is colored as outer because of the two close intersections highlighted in the figure.

Both pathological cases (gaps and overlappings greater than tol_c) can lead to set the ray as

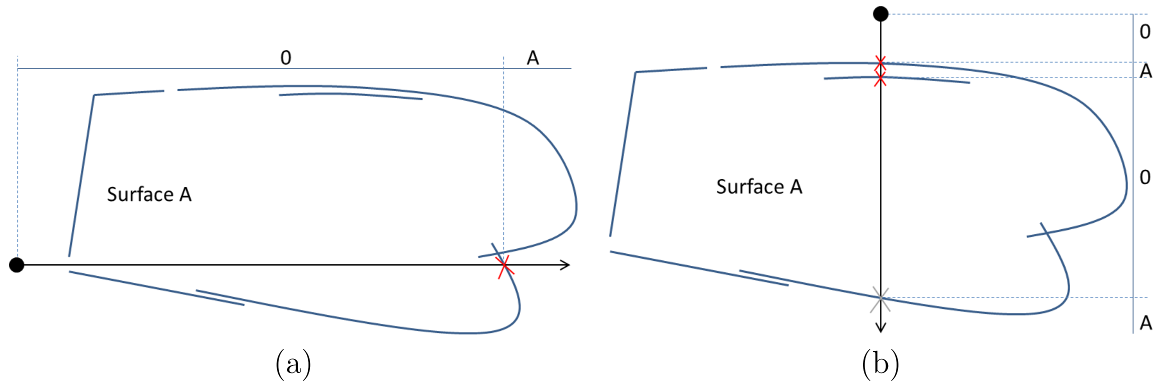


Figure 4.6: 2D examples of pathological configurations for the ray casting technique with non-watertight domains. **(a)** G type intersection of the ray. The part of the ray inside the surface A is not colored as A because the ray has not intersected its boundary entities. **(b)** W type intersection of the ray. The part of the ray inside surface A is colored as zero because the ray intersects two times the boundary in the same region (crosses).

invalid. An invalid ray is a ray with some coloring contradictions. There are two kinds of contradictions, depicted in Figure 4.7(a) and (b):

- The last part of the ray (from the last intersection point to the outer part of the bounding box of the model) has a color different than zero (see Figure 4.7(a)). Considering the principle that the parts of the ray outside the bounding box of the domain must be outer, the beginning and end of the ray must be outer (color zero).

- A part of the ray colored as i ends in an intersection point with a surface entity not interfacing the ith volume (Figure 4.7(b)).

Situations of invalid rays can also happen even if the contours of the domain are watertight, taking into account that the intersection operations in 3D are done numerically, so they depend on tolerances. Some specific configurations of the ray and the contours may lead to an invalid ray.

4.2.2 Adaptations of the method

A special kind of rays are the *Cartesian rays*, which are straight and parallel to the x, y and z directions. The ray casting technique can be applied to any kind of ray, but in the proposed algorithm Cartesian rays are used because they simplify the intersection operations, and they take profit on the spacial distribution of the nodes provided by the octree structure (this aspect will be seen in more detail in Section 4.3).

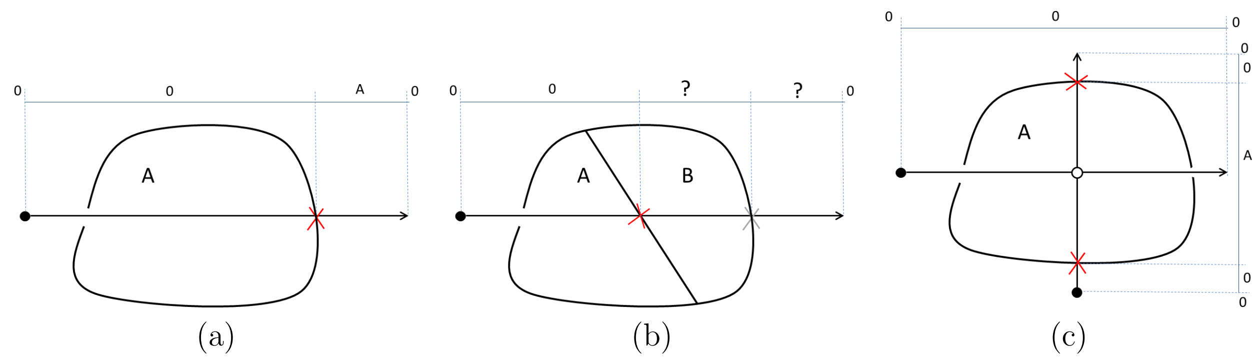


Figure 4.7: Types of invalid rays (crosses are intersection points). **(a)** The last part of the ray is colored as A, but it should be zero.**(b)** At the dark cross intersection point the ray should arrive with color A or B, as these are its interfacing surfaces, but its color is zero. **(c)** Rays to be canceled because they provide with different colors to a given position.

Taking into account that the domain to be meshed is totally inside its bounding box, the octree nodes which are outside it can be directly colored as zero (they are out of the domain) without the need of any coloring process. All the other octree nodes are colored using rays which begin out of the model bounding box, so as the color of the initial point of the ray is known: zero. The color of a node is directly the color of the part of the ray the node is into. As there are situations in which a ray is considered as invalid to color a point because of a pathological configuration, the following strategy is applied in order to color all the octree nodes:

- For each octree node, three Cartesian rays (one for each direction x, y and z, from the lower to the higher coordinate) passing by the position of the node are shooted from the outer part of the bounding box of the domain.
- Each one of these three rays can be invalid or valid. If it is valid it provides with a candidate color for the octree node. This opens the following cases:
  - There is only one valid ray. The node is colored with the color given by the ray.
  - There is more than one valid ray and they give the same color. The node is colored with that color.
  - There is more than one valid ray and they give different colors. A 2D example of this case is depicted in Figure 4.7(c). In this case the three rays are canceled. To cancel a ray means not to use its color information. It has to be taken into account that a ray can be used to color more than one node, as some nodes may be aligned

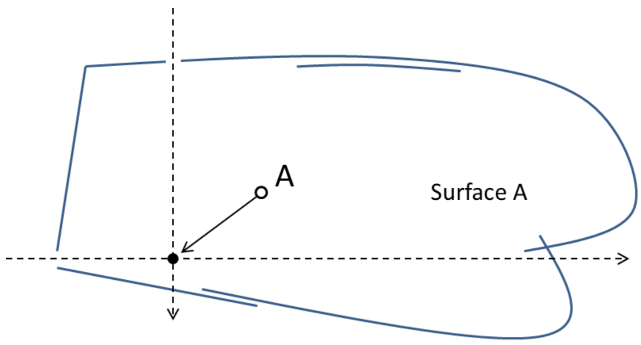


Figure 4.8: A 2D example of local ray casting. A non watertight representation of surface A is shown. The two Cartesian rays passing by the black dot (drawn with dotted arrows) are not valid. The color of the white dot is known (inside surface A). As the ray from the white dot to the black one has no intersection with the boundaries of the domain, the color of the black dot is also set to A.

with the Cartesian directions. In this case, canceling a ray also affects the other nodes it passes throw.

  – The three rays are invalid. In this case the node is not colored.

- At this point of the process there may be nodes with no color. These ones are colored using a *local ray casting*. This operation consists in launch a ray (not a Cartesian one) from a neighbor node with a known color to the node itself. The color is set evaluating the intersections of this local ray with the input boundaries. A 2D example of this local ray casting operation is shown in Figure 4.8. It has to be noted that the situation where the node cannot be colored with the Cartesian rays is rather improbable, as it should imply the coincidence of pathological situations in the three directions of the space aligned with the node.

This strategy have been proved in several examples, and it solves successfully the coloring process for the octree nodes with pathological configurations for the ray casting technique. The use of three rays for each node provides with a redundancy minimizing the possible effect of an invalid undetected ray.

4.3 Implementation of nodes coloring algorithm

This section is focused in the implementation of the algorithm explained in Section 4.2.2 for nodes coloring. It requires three Cartesian rays for each one of the nodes in order to color them (determine where they are topologically). Applying this algorithm to a general cloud of n nodes would imply to compute the intersections of $3n$ rays. However, as the nodes of the tetrahedra mesh treated in this work are octree nodes, most of them are placed on regular positions in space, aligned with the Cartesian directions. To take advantage on this configuration, one ray can be used in the coloring of several nodes: all the ones aligned with a Cartesian direction. Following this strategy an important time saving is achieved in the nodes coloring operation.

It is important to note that only the nodes with unknown color must be colored following the presented algorithm. The octree nodes with known color before the coloring operation are:

- The forced nodes (section 5.3.2); as they are linked to an entity coming from the input data, they directly have the color of the corresponding input geometrical entity.
- The octree nodes outside the bounding box of the domain. These ones are outer nodes, so their color is zero.

A 2D example of the rays used to color the nodes of a given configuration of a quadtree is shown in Figure 4.9. In the example, there are 28 nodes to be colored and 22 rays are used (12 for x direction and 10 for y direction).

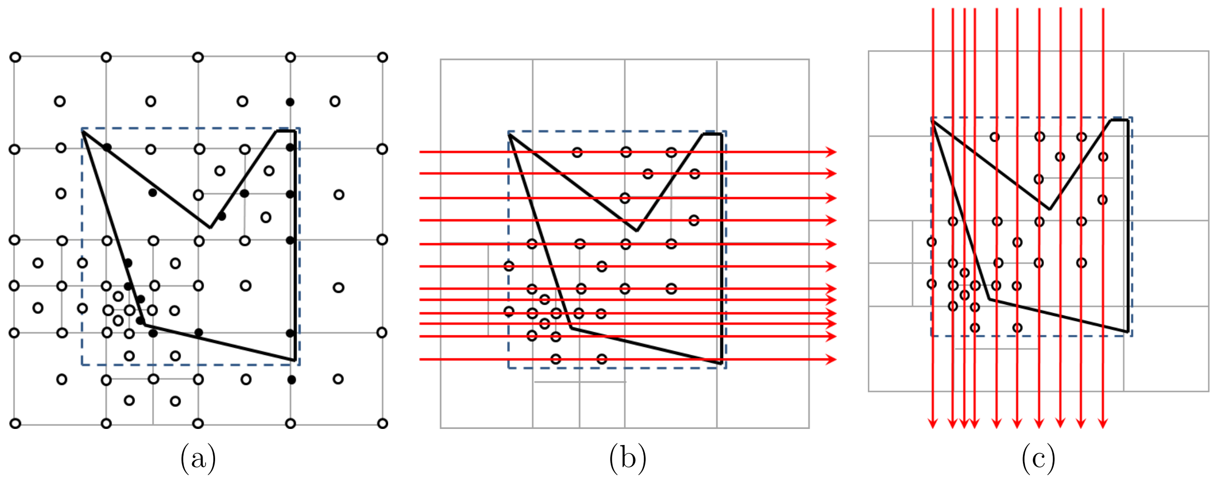


Figure 4.9: Rays used to color the quadtree nodes of a 2D example. The contour of the domain is the solid black line and the dotted line represents its bounding box. Rays are represented with red arrows. **(a)** Quadtree configuration with all the octree nodes represented. The black nodes are forced nodes. **(b)** The rays used in x direction to color the nodes. Only the nodes to be colored are plotted in order to clarify the figure. **(c)** The rays used in y direction.

The implementation of the coloring operation is based in the following steps:

- *Creation of all the rays.* For each one of the three directions of the space (x, y and z), the coordinate of each one of the nodes to be colored is considered. Then, one ray is created for each one of the coordinates. This process generates a collection of rays. Each one of the rays has the following information:
  - The alignment direction.
  - The two coordinates corresponding to the plane perpendicular to its direction.
  - The nodes it passes through, sorted from lower to upper coordinate in the direction of the ray.
- *Color the rays and set the contribution to each node.* In this step each one of the rays is colored. It has to be noted that this process is fully parallel, as each ray is totally independent from the others. In this process all the intersections between the ray and the input boundaries are computed. They are sorted from lower to upper coordinate, dividing the ray in different parts corresponding to the regions between intersection points. Looking at the interfaces intersecting at each intersection point the color of the parts of the ray can be set. Pathological situations in the intersection operations are detailed in Section 4.3.1. As explained in Section 4.2.2, each node has three possible colors, corresponding to the three Cartesian rays passing through it. At this point, the color contribution of the ray is set to all the nodes it passes through.
- *Determine the invalid rays.* As explained in Section 4.3.1, the invalid rays are identified. Then, the rays with contradictions in one node are canceled.
- *Set the color of the nodes.* The process described in Section 4.2.2 is used to determine the color of the node considering its three color contributions: if there is some valid ray the color is set automatically, otherwise, the local ray casting is performed. In case the local ray casting provides with different candidate colors (from the local rays from different neighbors), a voting process is performed: the color set is the one given by more neighbors.

4.3.1 Pathological situations

In this section the strategy followed to solve the pathological situations for the ray casting technique defined in Section 4.2 is detailed. These situations are related with the T, C, P, W and G intersection types and the M intersection points.

In all these situations the intersection point is a MIP (Section 3.4). Actually, the G intersection type (situation where the ray passes through a gap in the boundaries) involves a GIP, but this kind of intersections is not solved in the present approach because trying to find the corresponding GIP to all the parts of the ray not intersecting the input boundaries would be too computationally expensive. Leaving this situation as unsolved could lead to rays declared as valid, but giving wrong information in terms of coloring. It has to be noted that if this situation occurs, it does not mean necessary that an invalid color will be assigned to the nodes. As the three Cartesian rays passing by a node contribute to the decision of its color, only in cases where the three rays are considered valid, provides with wrong information and this information is compatible (within the three rays), the color of the node would be wrong. This is a rather improbable situation, which have never happened in the examples run in this work.

A MIP in a ray indicates the presence of a pathological configuration. Each intersection point has the information of the two volumes interfaced by the surface entity where it is (the *interfacing volumes* of the intersection point). The interfacing volumes of a MIP are the union of the interfacing volumes of each intersection point involved, so a MIP can have two or more interfacing volumes.

In Section 4.2 the situation of P intersection types (co-planar ones) is explained. They correspond to a M intersection point at the end part of the co-planar part of the ray. Considering that the presented algorithm uses linear triangle elements as surface entities for defining the boundaries of the domain, the situation where the ray is co-planar can be detected easily. If geometrical surface representations (line NURBS) are used for defining the contours of the volumes, solve this situations becomes much more difficult. For coloring purposes, the co-planar part of the ray takes no relevance, as it is ensured there will be no node to be colored there. If a node is so close to a surface entity it is a forced node in interface (Section 5.3.2), so its color is already set without the need of ray casting technique. The problematic situation comes at the time to color the following part of the ray (next to the co-planar one), where a color have to be assigned among the corresponding interfacing volumes.

When a MIP (a pathological configuration) is detected, its type has to be determined (T, C, W or M) in order to apply the right strategy to color the following parts of the ray from that point on. For this purpose, a sort of auxiliary local rays are built: the *surrounding segments*. These are segments parallel to the ray at a distance tol_c of it. In the present implementation, a number of 4 surrounding segments has been chosen following the Cartesian directions. Longitudinally, the surrounding segments are centered in the MIP position and have a length

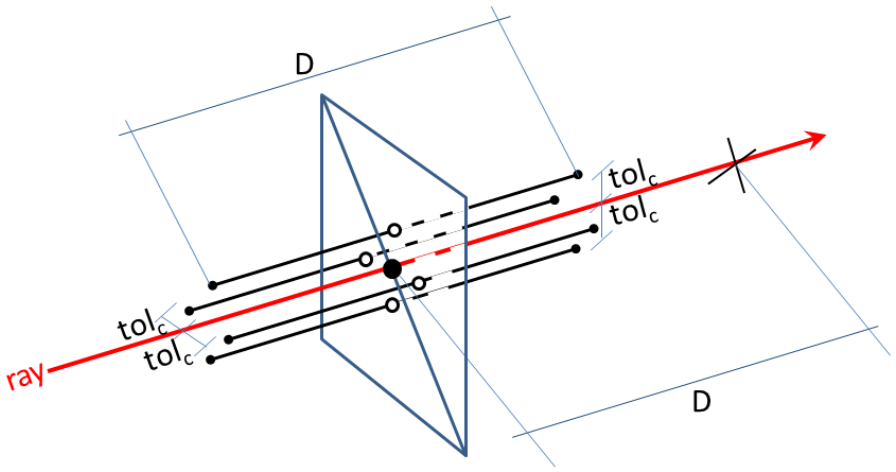


Figure 4.10: Surrounding segments of a ray around a MIP. The two triangles represent a part of the input boundaries. The black dot is the MIP and the white ones are the intersection points between the surrounding segments and the input boundaries. The black cross is the nearest intersection point of the ray to the MIP.

of D, being D the distance to the closest intersection point of MIP (in the ray). An example of the surrounding segments of a ray is shown in Figure 4.10. In this example the ray intersects two triangles by its contour edge, so it is a C intersection type.

Then, the intersection points between the surrounding segments and the input boundaries are computed. Considering them and the MIP interfacing volumes, the following cases should be accounted for distinguishing the type of pathological configuration:

- *Case I*: there are more than two interfacing volumes among the ones of the MIP and the ones of the intersection points of the surrounding segments. This case corresponds to the M intersection type.

- *Case II*: case I is not accomplished (so only two interfacing volumes are involved) and some surrounding segment has no intersection point. This case corresponds to the T intersection type.

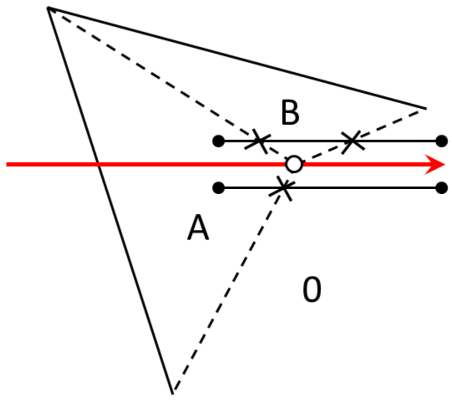
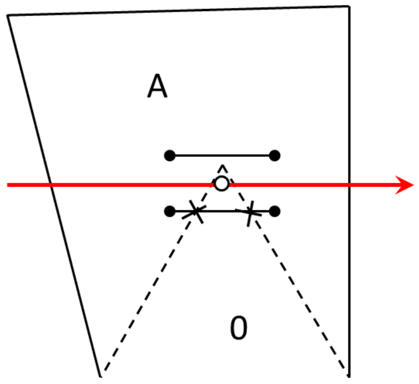
- *Case III*: when cases I and II are not accomplished. This means that all the surrounding segments have intersection points, and there are only two interfacing volumes involved. The following sub-cases are considered:

  – *Case IIIa*: each one of the surrounding segments have only one intersection point (MIP or single one). This case corresponds to the C intersection type.

- *Case IIIb*: all the surrounding segments have only one intersection point and some of them is a MIP (this means all the intersection points of the surrounding segment are enclosed in a distance tol_c). This case corresponds to the W intersection type.
- *Case IIIc*: there are surrounding segments with more than one intersection point. This case corresponds to the T intersection type.

Actually, cases IIIa and IIIb could be merged, as case IIIb is included in IIIa. The differentiation has been made only to highlight that a W intersection type always implies a MIP, and the C one can present a MIP or not in the surrounding segments. This is not relevant because the treatment of C and W intersection types is analogous for coloring purposes.

A graphical interpretation of these cases in a 2D example is illustrated in Table 4.1. As a 2D cases, only two surrounding segments take sense for a MIP.

Case I: there are three interfacing surfaces (A, B and zero) involved in the intersection points of the surrounding segments. M intersection type.	
Case II: all the intersected entities have the same interfacing surfaces (A and zero), and there is one surrounding segment with no intersection. T intersection type.	

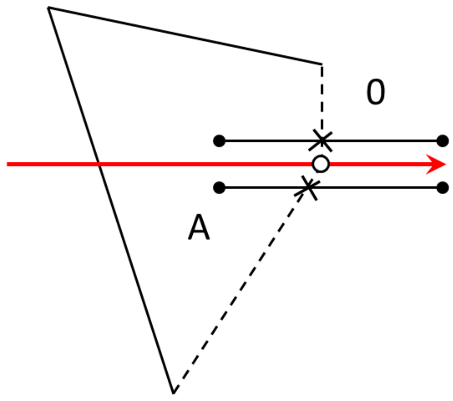
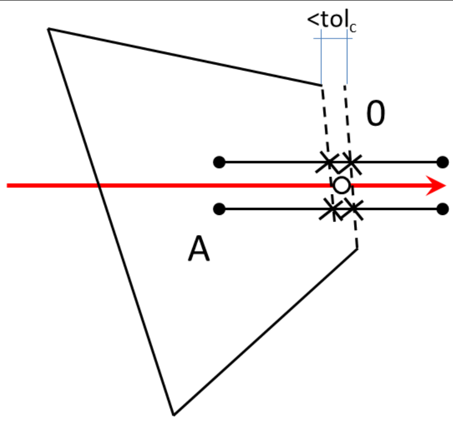
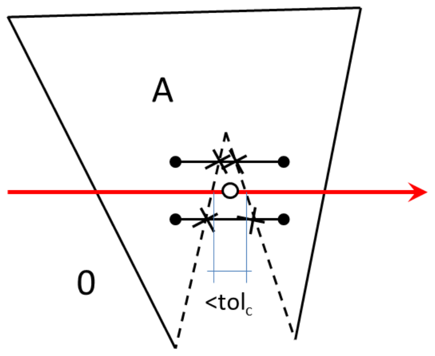
Case IIIa: each of the surrounding segments intersects the boundaries in one point and all the interfacing surfaces are the same (A and zero). C intersection type.	
Case IIIb: all the surrounding segments can create a MIP (its intersection points are closer than tol_c) and all the interfacing surfaces are the same (A and zero). W intersection type.	
Case IIIc: Some surrounding segment has more than one intersection point. T intersection type.	

Table 4.1: Different cases for surrounding segments for identification of intersection types corresponding to a MIP (white dot) in a 2D example.

Once the type of pathological intersection has been determined, the following strategy is carried out to color the following part of the ray (the one next to the MIP):

- T type intersection. In this case the color of the ray is not changed in the following ray part.

- C and W type intersections. In this case the color of the ray is changed accordingly with the interfacing volumes: as there are two interfacing volumes (A and B), if the previous part of the ray was colored as A, the next one is B and viceversa.

- M type intersection. This case is the more complicated one. Considering the n interfacing volumes of the corresponding MIP and the color of the previous part of the ray, the following part of it can have any color among the rest $n-1$ colors. The strategy chosen is based on a try and error approach: $n-1$ rays are shooted from the MIP, one for each possible color, and if one of them is valid, it is taken as the good one. If no one of them is considered valid, then the whole ray is set as invalid.

There is one situation not solved by the presented strategy. It is a mix between the T and the M intersection types. It occurs when the ray passes tangentially to two parts of the boundaries of a volume at the same point. A 2D example of this situation is shown in Figure 4.11. If the corresponding surrounding segments have one MIP each one, it corresponds to the case IIIb, so the intersection would be considered as W and the color of the ray would change when it should remain with the same color A (as if it was a T intersection).

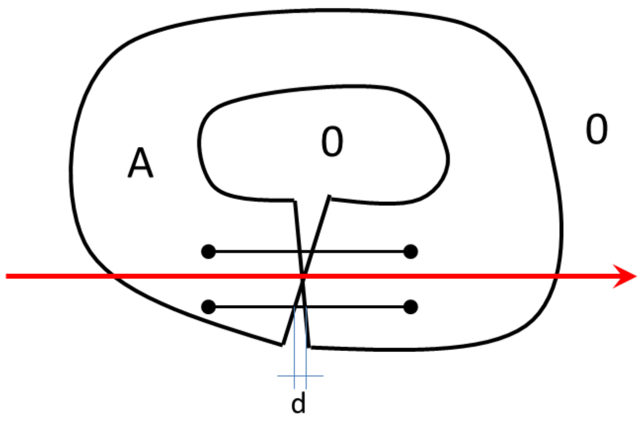


Figure 4.11: Pathological situation where the analysis of the surrounding segments can lead to erroneous classification of intersection type if $d < tol_c$.

4.3.2 Computation of distances

As explained in previous sections, this work proposes a meshing algorithm for body fitted meshes and for embedded ones. The later ones are used in embedded and immersed methods, and require for the nodes of the mesh their distance to the input boundaries in addition to their color (Section 5.2). This section focuses in the computation of these distances for the embedded meshing algorithm.

As explained in Section 5.2.1, the computation of distances from the nodes of the tetrahedra mesh to the input boundaries for embedded methods takes advantage on the ray casting technique used in the coloring algorithm. This is mainly because the Manhattan distance [Kra86] has been chosen for approximating the distance for far nodes (nodes further than a given distance of the boundaries of the domain). The Manhattan distance between two points is the sum of the distances of the three Cartesian components of the vector defined by the points. As the rays used in the coloring algorithm are Cartesian, its use fits perfectly for this purpose.

To improve the efficiency in a parallel implementation of the method, each ray contributes with a distance to the nodes it passes through. This implies each node has three contributions of distances, as each node has three rays passing trough it (one for each direction: x, y and z).

The process of computing the distances from the nodes of the tetrahedra mesh to the input boundaries follows the steps detailed hereafter:

1. Compute the exact distance to the input boundaries of the octree nodes belonging to an interface leaf of the octree. The exact distance is the one from the octree node to its closest coordinate onto the input boundaries. The strategy chosen to compute this distance profits from the octree itself. The surface entity from the input boundaries the closest coordinate is onto must collide an octree leaf containing the octree node. So only the surface entities colliding these octree leaves are considered at the time of finding the closest coordinate to a node.

2. Set to the maximum possible distance the three distance contributions of the rest of octree nodes.

3. Proceed with the ray casting technique corresponding to the coloring process. Apart from the color assignment to the nodes, an extra operation is performed in order to set the distances. Each ray launched in the coloring process has the information of its intersection points with the input boundaries, and the octree nodes it passes through (the ray nodes). Being d_0 the existing distance contribution of a ray node and d_i the distance from the node to its closest intersection point in the ray, d_0 is equalized to d_i if $d_i < d_0$.

4. Propagate the distance of each node within each ray. This step only affects the rays with more than one node. Let us consider the nodes of a ray sorted out by the coordinate corresponding to the direction of the ray. Being n one of the nodes and n_{-1} and n_{+1} its previous and next nodes, d_n, d_{n-1} and d_{n+1} are defined as their corresponding distance contributions of the ray. (For the first node only the n_{+1} is considered, and for the last one only the n_{-1}). Two propagated distances are defined for each node:

$$pd^{-}(n) = d_{n-1} + d(n_{-1}, n) \tag{4.1}$$

$$pd^{+}(n) = d_{n+1} + d(n, n_{+1}) \tag{4.2}$$

   where $d(a,b)$ is the distance between nodes a and b. Note that, as the nodes are aligned in the ray, the distance between them is directly the difference between their coordinates corresponding to the ray direction.

   Considering each node n of the ray, its distance contribution is set to the minimum between d_n, $pd^{-}(n)$ and $pd^{+}(n)$.

5. Update the distance contributions of each node. This operation consists in equalizing the three distance contributions of each node to the minimum one of them.

6. Repeat Steps 4 and 5 (propagate and update) two more times. This two steps must be done a number of times equal to the dimension of the problem (in our case, three times). This is to allow a complete propagation of the distances from the nodes. As this propagation is done using Cartesian rays, the information provided by one direction is not passed to the other ones until the distance contributions are updated. A 2D example is depicted in Figure 4.12 where the iso-lines for distance are plot before and after the second propagate and update operation (as a 2D case, propagate and update operation must be done only two times). It can be seen (in Figure 4.12(b)), that the distances in the lower left and right part of the surface are not well computed. As the hole of the surface is not aligned with them, the second propagation and update operation is needed to compute correctly the distances to the boundaries of all the nodes (Figure 4.12(c)).

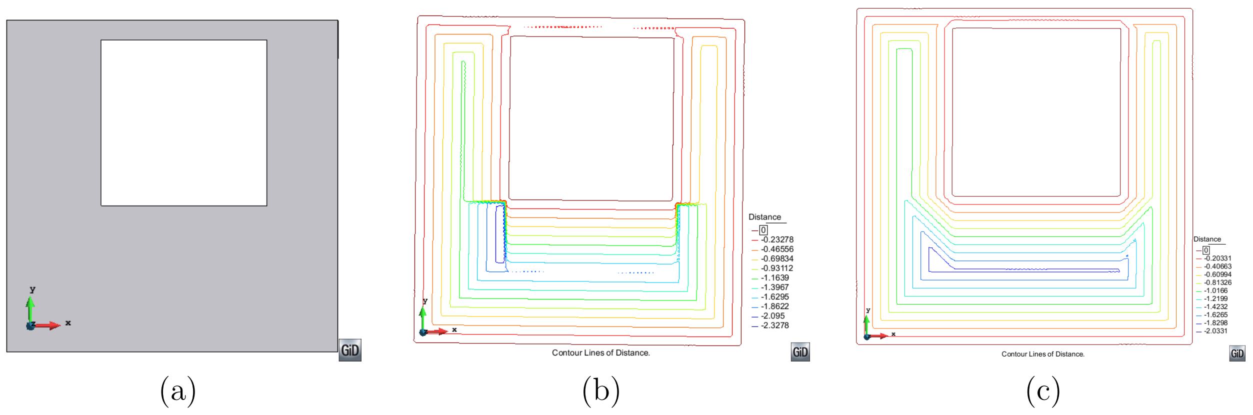


Figure 4.12: 2D example for computation of distances to the boundaries. **(a)** Surface representing the domain. The surface is a square with a squared hole inside. **(b)** Iso-lines of distance to the boundaries of the surface when only one propagation and update operation have been done. **(c)** Iso-lines of distance performing the propagation and update operation two times.

Chapter 5

Octree based mesher

5.1 Introduction

This chapter present the octree-based mesher developed in this thesis.

One of the requirements defined in Section 1.2.3 refers to the immersed [LP00] and embedded [LCC$^+$08] methods. This family of methods uses the so called *embedded meshes*. An embedded mesh is a mesh of a part of the 3D space containing in its interior the domain to be simulated, but the mesh is not limited to the inner part of the domain and, furthermore, does not fit the contours of it. The effect of the contours of the domain in the results of the numerical simulation to be run is captured by the assignment of special boundary conditions in the mesh elements and nodes. Because of its nature, it takes no sense to talk about preserving features or topology preservation for embedded mesh generation.

On the other hand, there are the *body-fitted meshes*, which must capture precisely all the contours of the domain. They need a faithful representation of the boundaries preserving the geometrical features, the topology of the volumes of the domain, and forcing the nodes and elements to follow the shape of their contours.

The algorithm proposed in this thesis considers the embedded meshing as a particular case of mesh generation where there is no need to apply any strategy for preserving geometrical features or volume topology. In this sense, the generation of embedded meshes will be easier than the generation of body-fitted ones.

The detailed explanation of the meshing algorithm requires the definitions of specific concepts and auxiliary algorithms explained in Chapter 3. In Sections 5.2 and 5.3 the meshing algorithm itself is presented for embedded and body-fitted meshes.

As a general view, the main steps of the mesher are pointed out hereafter. Let us consider that the input data for the mesher is the definition of the contours of the volumes of the domain to be meshed (from now on, *input boundaries*) and a given list of parameters (detailed in Section 3.1). The new embedded meshing algorithm can be summarized in five steps:

1. Create an octree enclosing the domain.
2. Refine the octree following some given criteria.
3. Generate a tetrahedra mesh from predefined patterns from the octree.
4. Find the volume containing each node of the mesh.
5. Compute the distance of the nodes of the resulting mesh to the input boundaries.

The first three steps are directly related to the octree structure. The two other ones involve mesh operations that can be applied to any unstructured tetrahedra mesh independently on its generation method, although the octree structure can be used as an auxiliary tool in order to improve the efficiency of the algorithms. They are detailed in Chapter 4 and Section 5.2.1.

Concerning the body-fitted mesher, the four first steps are the same ones as for the embedded mesher (adding some extra refinement criteria to refine the octree in the second step), and the fifth step is replaced by the following three ones:

- Apply given mesh edition operations to fit into the tetrahedra mesh the sharp edges and corners to be preserved and the surface entities of the domain. These mesh edition operations are detailed in Sections 5.3.5 and 5.3.6, and they basically involve nodes movement and element splitting.
- Identify the volume each tetrahedra of the mesh is into. This part of the algorithm is detailed in Section 5.3.7.
- Improve the mesh quality if needed using make-up and smoothing operations. This step is optional, as it is only needed if the mesh quality is poor. It is detailed in Section 5.3.8.

All these specific steps related to the body-fitted mesher are also applicable to any unstructured mesh, independently on the generation method chosen.

An important aspect of the new mesher is that it generates the mesh of the whole domain at once, including all the volumes which are part of it. Other existing meshers are designed to generate the mesh of just one volume, so they generate the mesh of the whole domain on a volume by volume manner.

5.2 Embedded mesher

One of the objectives of the presented meshing algorithm is to generate meshes for embedded methods. This section is focused in the specific aspects of the algorithm for this kind of methods.

The main characteristic of embedded methods is that the mesh used is not body-fitted, and its nodes have the information of:

- where they are topologically (inside which volume or onto which interface between volumes);
- the distance they have to the contours of the domain.

Knowing if a node is inside or outside a volume is solved by the proposed coloring algorithm (Chapter 4), and the computation of distances also takes profit of that algorithm, as explained in the following section. Furthermore, it is common to apply these methods to domains where only one volume is involved, so only two colors take part: inside and outside.

Embedded meshes are mainly used in Computational Fluid Dynamics (CFD) simulations, where the behaviour of a fluid around solid bodies is studied. If the solid bodies are in movement, rather than remeshing the whole domain at each time step, this family of methods maintains the volume mesh static and updates the nodes information (color and distance).

5.2.1 Computation of distances

The distance function plays a key role in embedded methodology in order to apply the boundary conditions. It is common to combine the coloring and distance function into one signed distance function where negative distance indicates inside and positive distance indicates outside nodes. A direct result of this definition is the fact that the iso-surface representing the zero distance defines the approximated embedded boundary condition, as shown in Figure 5.1.

To ensure the accuracy of the method the distance near the boundary and especially in cut elements must be calculated exactly. For the nodes far from the boundary the exactness of the distance becomes less important so, in order to reduce the computational cost, it is convenient to calculate the exact distance only for the points inside a given distance range from the boundary, and leave the rest with a maximum value (an upper distance limit). However, in cases with moving boundaries one may convect the distance function given the interface motion. In this procedure, having sharp gradients in distance functions can lead to

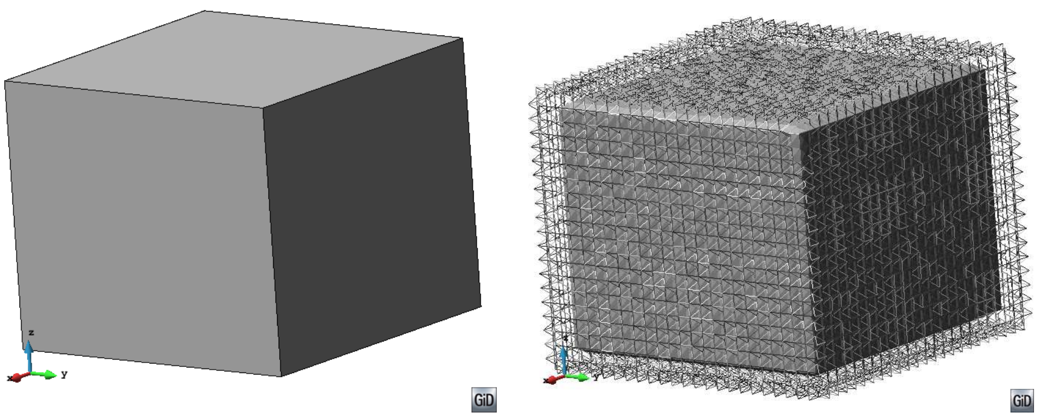

(a) (b)

Figure 5.1: **(a)** Surface entities defining a cube. **(b)** The cube representation via a zero iso-surface of the distance function in the tetrahedral embedded mesh generated.

numerical error and having a constant maximum distance near the boundary may affect the convergence and results. In order to deal with this problem an approximation of the distance function from a given distance of the boundary would be interesting.

In this work the exact Euclidean distance is calculated for the octree nodes belonging to the interface leaves. For the rest of them the Manhattan distance [Kra86] is computed. It is an estimation of the exact distance given by Equation 5.1:

$$d_M(\mathbf{p},\mathbf{q}) = \sum_{i=1}^{n} \parallel \mathbf{p}(i) - \mathbf{q}(i) \parallel \tag{5.1}$$

where $d_M(\mathbf{p},\mathbf{q})$ is the Manhattan distance between positions $\mathbf{p}$ and $\mathbf{q}$, and i represents the i-coordinate of the vector. n is the dimension of the space considered, in this case: three.

The Manhattan distance provides with the desired continuity and smoothness of the distance function in order to deal with moving objects with minimum distance calculation overhead. The reason to use this distance measure is to take profit on the coloring algorithm, as explained in Section 4.3. Cartesian rays are used for the coloring of the nodes, so using the distances onto the rays provides directly with the Manhattan distance.

5.2.2 Octree refinement criteria for embedded meshes

As explained in Section 5.1, one of the main steps of the meshing process is refine the octree following given criteria. The basic idea is that the octree must be refined in such a way that the application of the further steps of the meshing process ensures the accomplishment of the requirements to be covered.

In this section all the refinement criteria (RC) needed for embedded meshes are defined. It is important to highlight that all the octree refinement criteria defined in this section (the ones needed for embedded meshes) only depend on the desired mesh sizes and the topology of the octree itself (the balance criterion). This implies that the application of these refinement criteria does not need to consider the input boundaries. For notation purposes, the collection of all the refinement criteria defined in this section is called *size refinement criteria*.

Uniform size refinement criterion

Let us consider the bounding box of the model ($Bbox_m$). It is a parallelepiped with its faces parallel to the Cartesian axes (note that $Bbox_m$ is not needed to be coincident with the octree root). s_{box} is defined as the length of the smaller side of $Bbox_m$.

For given desired mesh sizes (via *mesh size points* or *general mesh size* parameter), lets define s_{dmax} as the maximum of the desired mesh sizes entered in the input data. Then, S_{max} is defined by Equation 5.2:

$$S_{max} = min(s_{dmax}, s_{box}) \tag{5.2}$$

If no size has been introduced in the input data, then S_{max} is directly s_{box}.

It has to be noted that because of the tetrahedra pattern definition (Section 3.3.3), tetrahedra sizes are directly related to the octree cells sizes. Actually, the size of the edges of the tetrahedra generated from a cell is always equal or smaller than the cell size. Refining the octree implies dividing by two some of the cells, reducing accordingly the corresponding tetrahedra size. This aspect makes impossible to reach exactly a given size for a tetrahedron. The algorithm tends to fit it by subdividing cells until their size is close to the desired one approximating it above or below. To incorporate in the algorithm the difference between the desired mesh size and the cell size and avoid an excessive level of refinement, the parameter $\alpha_{ms/cs}$ is defined. This is a real value ranging between 1 and 2. The value used for $\alpha_{ms/cs}$ is specified in Section 6.7.

Considering the isotropic octree structure, the longest edge of the final mesh of the model must be always smaller than S_{max}. This leads to the first refinement criterion:

RC 1 (Maximum size)**.** *If an octree cell collides with $Bbox_m$ and its size is greater than $(\alpha_{ms/cs} \cdot S_{max})$, the cell must be subdivided.*

Mesh size entities refinement criterion

In order to account with the possible desired mesh sizes required by the simulation defined with the *mesh size entities*, the following refinement criterion is defined:

RC 2 (Mesh size entities)**.** *If an octree cell collides with a mesh size entity with a desired size s and its size is greater than $(\alpha_{ms/cs} \cdot s)$, the cell must be subdivided.*

At this point the concept of *generalized* mesh size points needs to be introduced. To make easier the implementation of the method, and to automatize the methodology, the use of points instead of generic entities (lines, surface and volume ones) helps. The idea is to replace (only for mesh size purposes) each mesh size entity by a collection of mesh size points with the same desired mesh size: its *generalized mesh size points.* Actually only mesh size lines, surfaces and volumes are involved in this process, as the mesh size points are already points.

If we consider a mesh size entity with a desired mesh size s, its generalized mesh size points are located onto the mesh size entity in such a way that there is no point onto the entity further than s from a generalized mesh size point. An example of the generalized mesh size points of a surface mesh size entity (a triangle) is shown in Figure 5.2.

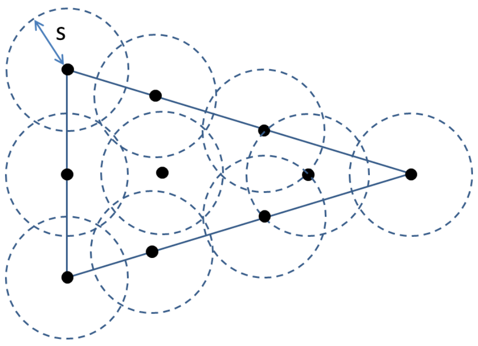


Figure 5.2: Representation of a surface mesh size entity (a triangle) with a set of its generalized mesh size points (black dots). It can be seen that all the coordinates inside the triangle have at least one mesh size point closer than s.

It has to be noted that, following this definition, there can be infinite sets of generalized mesh size points. The process used to obtain them is detailed in Section 6.3.

The implementation of the mesh size entities criterion is simplified if the generalized mesh size points are considered instead of the mesh size entities.

Volume mesh sizes refinement criterion

For a given mesh size desired for each volume (considering $s_{vol,i}$ as the desired size for the ith volume) the following criterion is defined:

RC 3 (Volume size)**.** *If an octree cell is an inner cell of the ith volume and its size is greater than* $(\alpha_{ms/cs} \cdot s_{vol,i})$*, the cell must be subdivided.*

Balance refinement criterion

The following refinement criterion is widely used in octree based meshers. It limits the number of neighbors of one cell, and it is often referred as *constrained two to one* condition (Section 3.3). It is an essential criterion for the pattern used for the creation of tetrahedra from the octree leaves:

RC 4 (Balance)**.** *If an octree cell has more than four neighbor cells by face or two by edge, the cell must be subdivided.*

Size transition refinement criterion

The balance criterion gives an upper limit for the size transition: as the size of two neighbor cells differs (at maximum) by a factor of 2, the difference between the size of the tetrahedra generated from those cells is bounded as well. However, it may be required for the final mesh to present a more smooth size transition between regions with small and large elements.

Considering the desired mesh size for given regions of the domain (from the input data), and a given size transition function, an envelope function can be defined to set an upper limit for the element size allowed in each position of the space $S_u(\vec{x})$. The definition if this function is detailed in Section 6.4, and leads to the following refinement criterion:

RC 5 (Sizes transition)**.** *Being* $\vec{c_i}$ *the center of an octree cell, if the size of the cell is greater than* $(\alpha_{ms/cs} \cdot S_u(\vec{c_i}))$*, the cell must be subdivided.*

5.2.3 Octree root

As explained in Section 3.2, the octree root is the bounding box of the octree. Clearly it should contain totally the domain to be meshed in its interior, but considering the meshing process, it has to accomplish some other requirements.

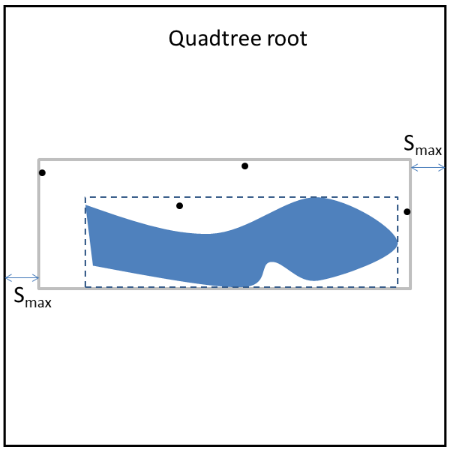


Figure 5.3: 2D example where the domain is the solid surface. Black dots are the generalized mesh size points. The bounding box of the model is represented by the dotted line. The $Bbox_m^+$ is the gray line, and the quadtree root is the black square.

For the creation of the octree root, the *extended bounding box* of the model is needed (from now on $Bbox_m^+$). $Bbox_m^+$ is the minimum bounding box containing the bonding box of the model and all the generalized mesh size points. It has to be noted that the mesh size entities can be outside the domain. The octree root is built centered in the center of $Bbox_m^+$, and with a size equal to the maximum size of $Bbox_m^+$ plus S_{max} (Equation 5.2). This offset of the octree root with respect to $Bbox_m^+$ is needed to ensure the tetrahedra creation by the tetrahedra pattern, considering that in some cases this creation involves one cell and its neighbor. A graphical 2D example (quadtree instead of octree) of the $Bbox_m^+$ and the octree root is shown in Figure 5.3.

5.2.4 Meshing algorithm

Hereafter, the steps of the meshing algorithm for embedded meshes are presented.

1. Process input data and create the octree root. This includes:
    - Creation of *generalized mesh size points* (in case they exist).
    - Compute S_{max}.
    - Creation of the octree root (Section 5.2.3).
2. Refine the octree accomplishing the size refinement criteria (defined in Section 5.2.2):
    - RC 1: Maximum size.

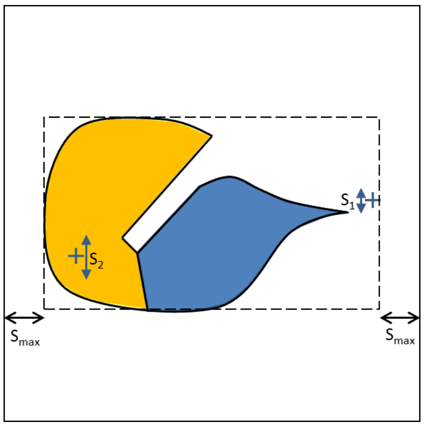
- RC 2: Mesh size entities.
- RC 3: Volume size.
- RC 4: Balance.
- RC 5: Sizes transition.

3. Classify the input boundary entities into the octree. This step implies the identification of the octree leaves colliding with the input boundary. Until now the octree has been refined, but each one of the cells had no information about its type: interface, inner or outer. Prior to this step the input boundaries only have been taken into account for building the bounding box of the model. At this point is where the input boundaries (surface entities from the input data) are analyzed in order to determine the interface cells. Furthermore (as it will be explained in Section 6.2) in the implemented octree, each interface cell stores the input boundary entities colliding with it.

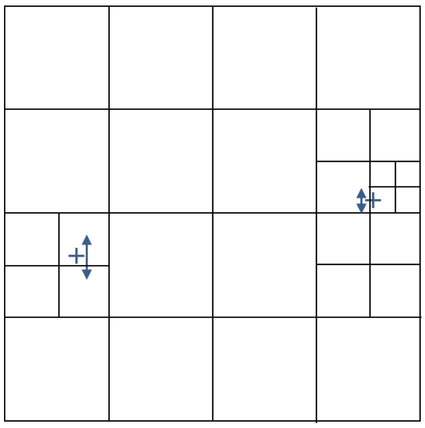
4. Create and color the linear octree nodes (Section 3.3.2) and compute the distances to the input boundaries (Section 5.2.1).

5. Apply the tetrahedra pattern.(Section 3.3.3).

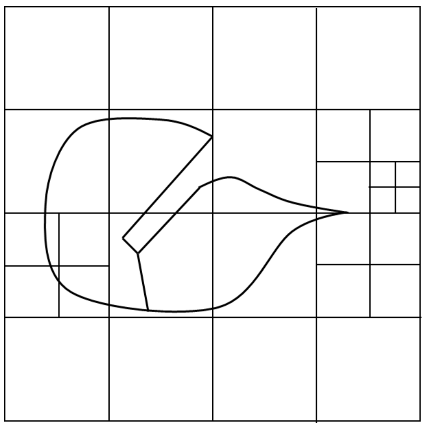
A 2D example of the algorithm is depicted in Table 5.1, where each step is illustrated by a figure. The model is formed by two surfaces in contact (drawn in orange and blue), and it contains only two mesh size points (represented by a cross in the figures) in the input data.

1- *Process input data and create the octree root.* The crosses are the mesh size points. The arrow beside each mesh size point represents its associated size. The octree root is the black square, and the $Bbox_m^+$ of the model is drawn with dotted lines. S_{max} in this example coincides with S_2, as it is the largest size coming from the generalized mesh size points, and it is smaller than s_{box}.	

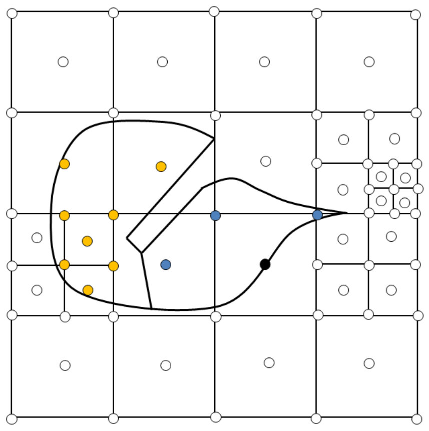
2- *Refine the octree accomplishing the size refinement criteria.* In this example the size transition factor is equal to one (the transition corresponds to the constrained two to one condition) to make the figure clearer.



3- *Classify the input boundary entities into the octree.* It is important to note that, until now, the input boundaries only have been considered to build the bounding box of the model, but they have not been implied in the octree refinement process.



4- *Create and color the linear octree nodes and compute the distances to the input boundaries.* The nodes of the figure are painted with the corresponding color: orange or blue if they are inside the corresponding surface, black if they are onto the input boundaries, and white if the are outside the domain.



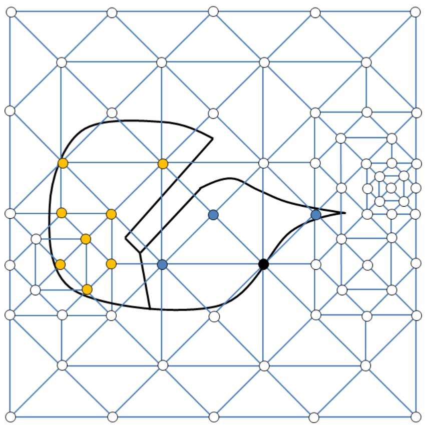
5- *Apply the tetrahedra pattern.* It can be appreciated in this figure that the final tetrahedra (triangles in this 2D case) are not body-fitted, as they do not preserve the original shape of the domain.	

Table 5.1: Steps followed by the embedded meshing algorithm applied to a 2D example with two surfaces and two mesh size points.

5.3 Body-fitted mesher

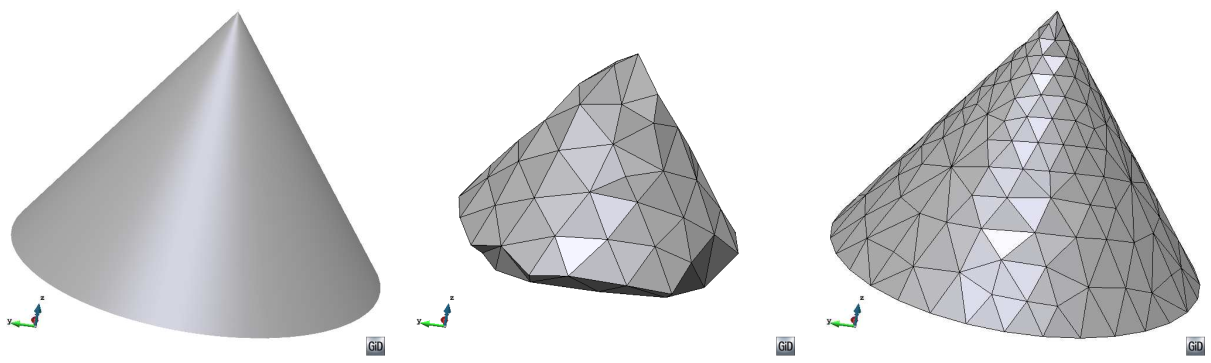
The concept of body-fitted mesh is applied to a mesh which contours represent precisely the shape of the contours of the domain. Considering the presented algorithm is a volume mesher generating a body-fitted mesh implies, from the geometrical point of view, that the final tetrahedra mesh must take into account lower level entities: point, line and surface ones.

The adaptation of the final mesh to the surface entities defining the different volumes of the domain plays a key role to ensure the preservation of the topology of the model and the accuracy of the mesh near the contours. This process is detailed in Section 5.3.6.

In the literature, the concept of *geometrical features* is used to refer the point and line entities relevant for the shape definition of the model, which must be preserved in the final tetrahedra mesh. Point entities to be preserved are the ones where the surface normal has multiple discontinuities and are referred as *corners*. Line entities to be preserved are the ones shared by two surfaces forming a sharp angle in it and are referred as *ridges*.

An example of two meshes of the same model (one preserving and the other one not preserving the geometrical features) is shown in Figure 5.4.

The preservation of the geometrical features is one of the weak points of the octree-based meshers. As these family of meshers are based in a regular grid (the octree) rather than in the shape of the domain, when the shape of the domain has specific distorted regions in comparison with the octree cells, some strategy has to be followed. One can think that



(a) Geometrical model. (b) Non body-fitted mesh. (c) Body-fitted mesh.

Figure 5.4: Example of body-fitted and non body-fitted mesh of a geometrical model of a cone. The non body-fitted mesh does not preserve the apex of the cone (corner), nor the line entities (ridges) defining its base.

refining the octree in those regions should solve the problem, but this is not true, as some configurations can lead to an infinite refinement process. This is because the geometrical problems governed by angles are reproduced exactly in all the refinement levels.

In this work, not only the geometrical features are taken into account to be preserved in the final mesh, but also forced point and forced line entities coming from the input data, needed to preserve the topology of the model or to provide with specific attached data to the final mesh entities. The generalized concept of *forced edges* and *forced nodes* (detailed in Sections 5.3.1 and 5.3.2) is used to include all the line and point entities to be preserved, independently on its purpose: preservation of topology or preservation of geometrical features.

5.3.1 Forced edges

In case the input data for the mesher has forced line entities (from the input data), or a minimum angle for sharp edges is defined, the so called *forced edges* must be created. These are edges the final tetrahedra mesh will preserve. Forced edges plays a key role for preserving sharp edges and representative surface and line topology.

At this point it has to be noted that the mesher is *partially constrained*. In this context, the term partially constrained means that if there are some forced line entities to be preserved, a collection of edges from the final mesh should follow the path of those line entities. This is not as restrictive as totally constrained condition, which would force to have as much edges in the final mesh as the number of forced line entities, and bounded (each one of the edge)

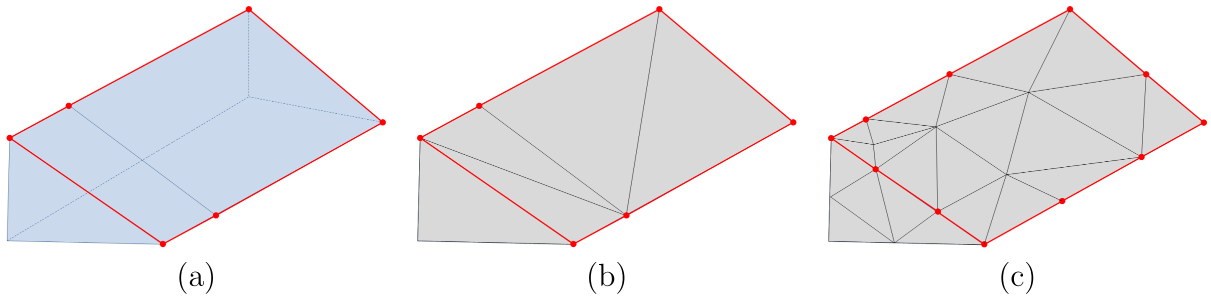


Figure 5.5: **(a)** Contours of a volume with some of its sharp edges highlighted. **(b)** Constrained tetrahedra mesh of the volume highlighting the sharp edges corresponding to the ones in figure (a).**(c)** Partially constrained tetrahedra mesh of the volume highlighting the sharp edges corresponding to the ones in figure (a).

by the same nodes bounding the forced line entities.

This difference can be appreciated in the Figure 5.5, where the definition of a domain is shown with some of its sharp edges highlighted (Figure 5.5(a)), and two different final meshes of the domain being partially or totally constrained are shown. In this example forced edges come only from the sharp edges of the domain. As it can be seen in Figure 5.5(b), generating a totally constrained mesh yields a mesh with the same sharp edges present in the input data. In Figure 5.5(c) it can be appreciated that a different number of edges are generated to represent the initial forced edges: generating a partially constrained mesh, a collection of sharp edges follows the path of the sharp edges present in the input data, so as the shape of the domain to be meshed is well captured.

The creation of the forced edges is based on three steps: identification of the *base line entities*, creation of the *polyline entities* and linear mesh generation from the polyline entities:

1. Identification of the *base line entities*. The base line entities are line entities coming from three different sources:

    - Forced line entities: each forced line entity from the input data is directly a base line entity.
    - Line entities shared by more than two surface entities from the input boundaries. A T-junction is an example of this case, where a line entity is shared by three surface entities.
    - Line entities shared by two surface entities from input boundaries which form an angle (along the line entity) smaller than the maximum angle for sharp edges (in case it is defined in the input data). The angle between two surface entities is

not well defined, as it can only be evaluated point by point. In the present work, the angle between two surface entities along their common line entity is defined as the mean of the angles formed by the normal vectors of each surface entity in some sampled points of the line entity. The number of points used depends on the curvature of the surface and line entities.

Note that the base line entities is a collection of line entities in the space, and they can be related or not to the input boundaries of the domain.

In case of non watertight input boundaries, a previous collapse of nodes and edges may be done to the input boundaries in order to be able to capture the sharp edges, as two surface entities forming a sharp edge may not be in contact topologically. This aspect is treated later on.

2. Creation of the *polyline entities*. In this step, the collection of base line entities is clustered in different groups called *polyline entities* in such a way that two base line entities are in the same polyline if:

   - they share a point entity,
   - and the shared point entity does not belong to any other base line entity,
   - and the angle formed by the tangent vectors of each line entity in the shared point is smaller than the maximum angle for sharp edges.

   The result of this clustering operation leads to a collection of polyline entities having each one of them one or more base line entities. Two point entities can be identified as extremes of each polyline entity.

   The polyline entities corresponding to the base line entities shown in Figure 5.6(a) are depicted in Figure 5.6(b). In this example base line entities l_2 and l_3 are not part of the same polyline entity because they form a small angle in the common point (smaller than an hypothetical maximum angle for sharp edges).

3. Generate a mesh from the polyline entities. In this step a linear mesh is generated from each polyline entity taking into account three criteria for the linear element creation:

   - If a polyline entity is closed (both extreme nodes are the same one), it must have at least three elements. This condition avoid the creation of zero-volume elements in the final mesh.

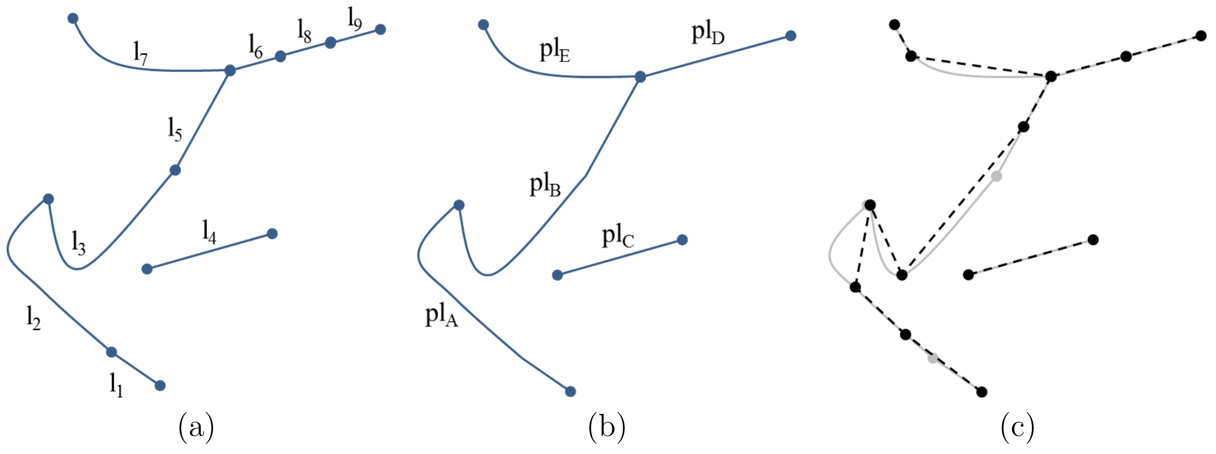


Figure 5.6: Example illustrating the entities involved in the creation of the forced edges. **(a)** A collection of base line entities l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8 and l_9. **(b)** The polyline entities pl_A, pl_B, pl_C, pl_D and pl_E created from the base line entities shown in Figure **(a)** considering the angle between l_2 and l_3 smaller than the maximum angle for sharp edges (from in input data). **(c)** A possible distribution of forced edges (shown in dotted line) created from the polylines present in Figure **(b)**.

- If the base line entities forming the polyline entity has some desired size assigned (from the input data), a desired size s_m can be computed for each linear element. This s_m is the mean of the desired sizes of the base line entities in the positions of the nodes of the element. Each element of the mesh must have a length equal or lower than its s_m.
- The chordal error of the elements of each polyline meshes is limited by the most restrictive value between the chordal error required by the simulation (in case it is defined in the input data) and a given relative chordal error Ec_l. The relative chordal error is defined as the chordal error of the element divided by its length. The parameter Ec_l tries to ensure that the forced edges describe correctly the shape of the forced line entities and the input boundaries. The value of this parameter is discussed in Section 6.7.

The discretization of the polyline entities can be done using any mesh generation method. This linear mesh will be used only auxiliary (it will not be part of the final mesh) and its quality is not relevant more than providing a sort of sizes distribution in the octree. In this work a simple recursive splitting method is used to generate the mesh of each polyline entity. The method creates one linear element using the two extreme nodes of the polyline entity, and subdivide it recursively until all the elements accomplish with the three criteria defined above. It has to be considered that each new edge node created when an element is subdivided is mapped onto the polyline entity in order to capture well the shape of it.

The *forced edges* are directly the elements of the meshes generated from the polyline entities.

Note that the two first steps refer to line entities in general, without specifying if they are CAD or mesh entities. The forced edges created in the third step are always mesh entities.

For notation purposes, the extreme nodes of a forced edge are called *edge nodes*. Each forced edge has two (and only two) edge nodes, and each edge node can belong to more than one forced edge.

It will be seen later on that the octree cells near a forced edge should have a similar size to it not to produce too distorted tetrahedra. To reach this goal, the *forced edge condition* has been defined:

Condition 1 (Forced Edge Condition)**.** Being cell A the octree leaf containing one edge node of a forced edge, and cell B the octree leaf containing the other edge node of the forced edge, the degree of neighborhood between A and B must be lower or equal than two.

Considering a given configuration of the octree, all the forced edges must accomplish the Forced Edge Condition. If a forced edge violates it, it must be subdivided in other forced edges until all of them accomplish it. The split of the edge is done by its middle point. It has to be taken into account that the subdivision of a forced edge implies the creation of new edge nodes. These ones are mapped onto the polyline entity where the forced edge comes from to yield a better approximation of the shape.

Pathological configurations of forced edges

There are some pathological situations for forced edges that can occur due to given configurations of input boundaries:

- *Very thin surface entities.* The presence of very thin surface entities in the boundary may be a problem due to the distortion in the computation of their normal vector. In case of triangles, the normal vector is computed using the vector product of two of their adjacent sides. If one of the them is very small (almost zero), the normal vector calculated becomes erroneous: it can point to any direction. For this reason, when comparing normal vectors of adjacent triangles in order to set sharp line entities, some fake ones may be detected.

- *Very close forced line entities.* The presence of very near forced line entities without intersection may lead to very distorted elements in the final mesh. Furthermore, some of the octree refinement criteria (Section 5.3.3) may lead to infinite level of refinement in this situations.

A possible strategy to solve the first situation is to collapse the line entities with a length lower than a given tolerance, or directly exclude these small ones as forced line entities. Collapsing the small line entities may not solve the problem, because they may belong to a longer polyline entity (also fake) which won't be collapsed. The option of not considering the small line entities also may fail, because there may be some cases where a relevant polyline entity is formed by very small line entities.

For the second situation, a possible solution may be to collapse the line entities which are close enough one from each other. However, these geometrical operations are not trivial in some 3D configurations.

In order to detect the sharp line entities, only the ones belonging to two, and only two, surface entities are considered. Cases where more than two surface entities share a line may be important for the topological definition of the domain. In these cases, the corresponding line entity should be set as forced line entity in the input data, not in the sharp edges detection process. The edges surrounding a gap (in case of non-watertight boundaries) are not considered to be preserved.

These automatic strategies may not solve all the pathological situations that can occur in the input boundaries. Hence, it may be needed to preprocess them in order to specify the desired forced line entities following given criteria to accomplish the simulation requirements. This is the case of the example shown in Figure 5.7. The input boundaries of this example are depicted in Figure 5.7(a). As it can be seen, there are very thin triangles. The normal vector of these triangles is not well computed numerically. When comparing the normal vector of two adjacent triangles, some edges which are not sharp are detected as so. It is the case of the small edges depicted in Figure 5.7(b).

In this example, some of the fake sharp edges (forced line entities) are connected creating the corresponding polyline entity. These polyline entities are not so small, so collapsing the nodes of the small forced line entities will not eliminate these fake ridges: the polyline entity would remain, but with less forced line entities. A possible solution here is to consider only

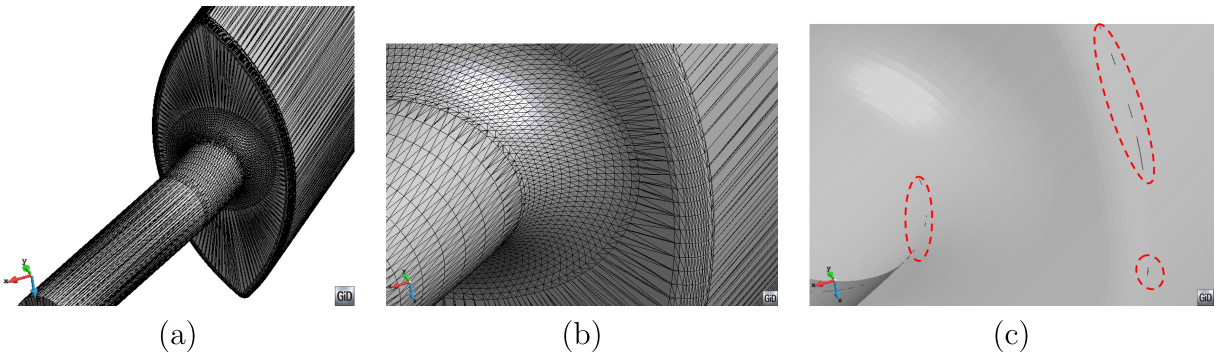


Figure 5.7: **(a)** Input boundaries (triangles) of a part of an example model of a mechanical piece. **(b)** Zoom view of Figure (a), where the shape is smooth enough not to present sharp edges. **(c)** Same view of (b), showing only the sharp line entities (black lines) automatically detected considering the normal vectors of the triangles of the boundaries (dotted lines surround them in order to highlight their position). As there are very thin triangles, some normal is not well computed, so some of the detected sharp line entities are fake.

the polyline entities longer than a given tolerance (as the polyline entities useful to define the shape of the domain are much more longer). However, this strategy cannot be applied in general. Some models may not have a so clear separation between the lengths of relevant and not relevant sharp entities.

In cases where the 3D model is not very complex, another option could be to select manually a priori the line entities to be forced ones.

In conclusion: Depending on the quality of the input boundaries of the domain, more information may be needed in order to identify clearly the forced line entities to be preserved by the mesher. In these cases, the geometrical definition of the domain is not enough to set the appropriate forced edges.

5.3.2 Forced nodes

The *forced nodes* are octree nodes (linked to an octree position) with a prescribed position in space: the *forced position*. The forced position of a forced node is not coincident with the octree position it is linked to. There are three kinds of forced nodes:

- *Forced isolated nodes*. These ones are created directly from the forced point entities given by the input data. Every forced point entity becomes a forced isolated node whose forced position is the one of the forced point entity and the related octree position is the closest one to it.

- *Forced nodes in edge.* These forced nodes correspond to the edge nodes (Section 5.3.1). Every edge node become a forced node in edge which forced position is the one of the edge node and the related octree position is the closest one to it.

- *Forced interface nodes.* These correspond to octree nodes which octree position is very near to the input boundaries. Taking into account an octree node, lets define **cp** as its closest coordinate onto the input boundaries. Considering D the distance from an existing octree node to its **cp**, and s_c the size of the octree leaf the octree node is into, an octree node is a forced interface node if

  $$D < \alpha_{ip} \cdot s_c \tag{5.3}$$

  where α_{ip} is a real positive value. The value of α_{ip} must be lower than 0.25, otherwise it could lead to inverted tetrahedra (with negative jacobian) when applying the tetrahedra pattern (Section 3.3.3). The tuning of this value is explained in Section 6.7.

  The forced position of a forced interface node is the **cp** of the octree node it is generated from. It is important to note that the definition of the forced interface nodes depends on the size of the octree cells, so an octree node can be considered as a forced interface node or not depending on the size of the cells surrounding it. It can be seen that refining an octree can lead to the creation or deletion of some forced interface node.

It will be seen in Section 5.3.3 that the forced nodes are involved in an octree refinement criterion (RC 6). To anticipate the general idea, each forced node is linked to an octree position, and each octree position cannot have more than one forced node associated. So if two forced nodes are very close one from each other, the octree cell containing them should have a size similar to the distance between them. This could lead to an excessive level of refinement of the octree, specially in cases where there are non-watertight definitions of the boundaries.

In extreme cases where the forced nodes can be almost exactly in the same position, the refinement criteria could lead to an infinite level of refinement. To solve this problem, the forced nodes which are closer than a given tolerance (a portion of the mesh desired size) are collapsed. It has to be considered that if two forced nodes are collapsed and they belong to a forced edge, it must be collapsed as well.

5.3.3 Octree refinement criteria for body-fitted meshes

All the size refinement criteria (the ones applied for embedded meshes, detailed in Section 5.2.2) are also applicable to body-fitted meshes, as they are based on the desired mesh size.

In this section the specific refinement criteria (RC) needed for body-fitted meshes are defined. Some of them may not be applied in the meshing process depending on the input parameters.

These refinement criteria are a key point for the body-fitted mesher, as they ensure the geometrical features of the domain and its topology will be preserved by the tetrahedra generated from the octree.

Forced nodes refinement criterion

The following criterion refers to the forced nodes. As it has been explained in Section 5.3.2, each forced node is associated to an octree cell position.

RC 6 (Forced nodes)**.** *If an octree cell has a forced node inside, and the octree position associated to that forced node is occupied by another forced node, the cell must be subdivided.*

In order to ensure that this refinement criterion does not lead to an infinite level of refinement, the distance between forced nodes must be greater than a minimum value, corresponding to the minimum cell size allowed in the octree. The fulfillment of this condition involves a treatment of the forced nodes before the refinement criterion: if two forced nodes are closer than the minimum cell size allowed in the octree they are collapsed into one. If they are part of a forced edge, the forced edge is also collapsed.

Tetrahedra distortion refinement criterion

For the refinement criteria defined hereafter, the tetrahedra created from the octree following the tetrahedra pattern are considered. For this reason, these criteria only take sense if the RC 4(balance) and RC 6(forced nodes) are accomplished. From now on the concept of the tetrahedra created from an octree cell is used to refer the tetrahedra result from applying to the cell the patterns defined in Section 3.3.3.

The quality of the tetrahedra generated from cells without forced nodes is very high (and it can be evaluated a priori), but the presence of forced nodes implies that their positions in space are not the octree positions, so the tetrahedra generated may be distorted.

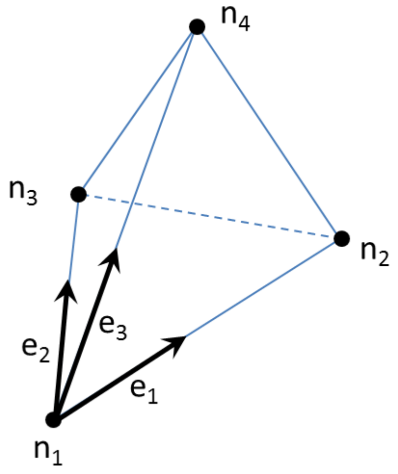


Figure 5.8: Tetrahedron with the local node numeration following the right hand rule. Vectors involved in the definition of an inverted tetrahedron are depicted.

The concept of *inverted* tetrahedron must be defined at this point in order to introduce the following refinement criterion. Considering a tetrahedron with its nodes n_1, n_2, n_3 and n_4 sorted in a given way (as the tetrahedron depicted in Figure 5.8), it is inverted if the scalar product of ($\mathbf{e_1}$x$\mathbf{e_2}$) by $\mathbf{e_3}$ is negative. $\mathbf{e_1}$, $\mathbf{e_2}$, and $\mathbf{e_3}$ are vectors aligned with the directions n_1n_2, n_1n_3 and n_1n_4 respectively, oriented from n_1 towards the other nodes. In the case where the scalar product is null, the tetrahedron has zero volume and receives the name of *sliver*.

RC 7 (Tetrahedra distortion)**.** *If a tetrahedron created from an octree cell is inverted or a sliver, the cell must be subdivided.*

Topological refinement criterion

The following refinement criteria are the basis to ensure the preservation of the topology for the mesher. Some auxiliary definitions are needed here. If we consider an edge of a tetrahedra generated from a cell, a limit distance of the edge d_{edge} is defined as

$$d_{edge} = \alpha_{edge} \cdot l_{edge} \tag{5.4}$$

where l_{edge} is the length of the edge and α_{edge} is a real value between zero and one (the value taken for this parameter is detailed in Section 6.7).

The so called *intersection points* of an edge are the intersection points between the edge and the input boundaries. As an intersection operation between an edge and surface entities, the following situations must be considered (see also Section 3.4):

- If the intersection between an edge and the the surface entity defining the input boundaries is co-planar (P intersection type), any intersection point is considered.
- For the evaluation of the following refinement criterion, a MIP is created only if all the intersected surface entities involved interface the same volumes.
- As the extremes of an edge are already colored, if both belong to different volumes and no intersection point has been detected, the corresponding GIP must be considered as the intersection point.

If we define the portion of a given volume enclosed in an octree cell delimited by the input boundaries and the cell faces, the interface cells can have several portions of volumes. Let us name *face limit surfaces* to the boundaries of these portions of volumes laying onto the the cell faces. An example of the portions of volumes of a cell is shown in Figure 5.9.

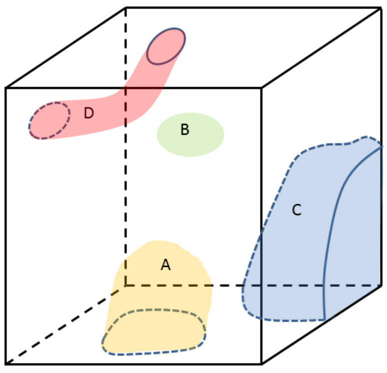


Figure 5.9: Example of portions of volumes enclosed in an octree cell. Volume A lies onto one face of the cell creating one *face limit surface*, volume B does not lay onto any face of the cell, volume C lies onto three faces of the cell creating three connected *face limit surfaces*, and volume D lies onto two faces of the cell creating two unconnected *face limit surfaces*.

RC 8 (Topology). *This criterion is split in three levels:*

- *8(a) If an edge of a tetrahedra created from an octree cell fulfill some of these conditions:*
  - *both extreme nodes of the edge are forced nodes and there are more than one intersection point,*
  - *one extreme node of the edge is a forced node and the distance between it and an intersection point is lower than* d_{edge}*,*

- *there are two intersection points and the distance between them is lower than d_{edge}, or*
- *there are more than two intersection points,*

then the cell must be subdivided.

- *8(b) If a portion of volume enclosed in an octree cell does not intersect any edge from the tetrahedra created from the cell, and it lies onto more than one face of a cell creating unconnected face limit surfaces, then the cell must be subdivided.*
- *8(c) If a portion of volume enclosed in an octree cell does not lay onto any face of the cell and does not intersect any edge from the tetrahedra created from the cell, then the cell must be subdivided.*

The implementation of these refinement criteria is detailed in Section 6.5.

Examples of different configurations of an edge fulfilling the refinement criterion 8(a) are shown in Figure 5.10.

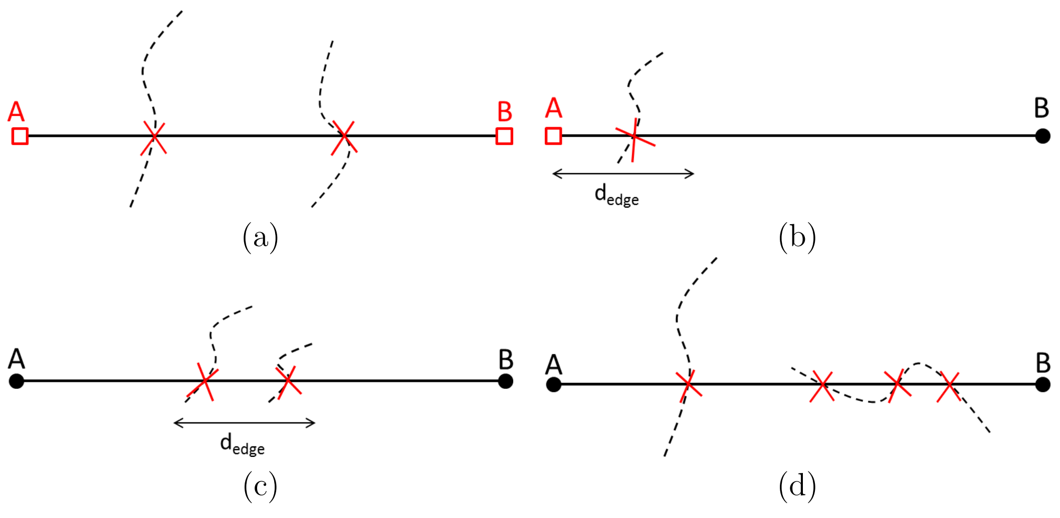


Figure 5.10: Different configurations of the edge $\overline{AB}$ that force the refinement of the octree in order to accomplish the refinement criterion 8(a). Dotted line represents the input boundaries and crosses are the intersection points between the edge and the input boundaries. **(a)** Both A and B nodes are forced nodes and there are more than one intersection point. **(b)** A is a forced node (B could be forced node or not), and there is an intersection point closer than d_{edge} to it. **(c)** there are two intersection points and the distance between them is lower than d_{edge}. **(d)** There are more than two intersection points.

An example of a pathological configuration where the refinement criterion 8(b) is needed is the portion of volume D of Figure 5.9. On the other hand, the portion of volume B of the same figure evidences the need for the fulfillment of the criterion 8(c) in order to preserve the topology of the input data.

5.3.4 Meshing algorithm

Hereafter, the steps of the meshing algorithm for body-fitted meshes are presented:

1. Process input data and create the octree root.
    - Creation of *forced edges* and *forced nodes* (in case they exist).
    - Creation of *generalized mesh size points* (in case they exist).
    - Compute S_{max}.
    - Creation of the octree root. The creation of the octree root is done in the same way as for embedded mesher (Section 5.2.3).
2. Refine the octree according to the size refinement criteria (the ones used for the embedded mesher, defined in Section 5.2.2).
3. Classify the input boundary entities into the octree. Here is where the input boundaries (surface entities from the input data) are analyzed in order to determine the interface cells. Before this step the input boundaries only have been taken into account for building the bounding box of the model.
4. Color the octree nodes (Section 4) and detect the forced interface nodes.
5. Refine the octree to fulfill the specific criteria for body-fitted meshes (defined in Section 5.3.3):
    - RC 6: Forced nodes. This implies enter the forced nodes into the octree, which is assign an octree position to each one of them.
    - RC 7: Tetrahedra distortion. (See page 93).
    - RC 8: Topology. (See page 94).

This step implies the coloring of the appearing octree nodes. RC 4 (balance) has to be also taken into account during this refinement process, as it is mandatory for the tetrahedra generation following the given patterns.

From now on, the octree structure is frozen in the sense that its cells will not be refined any more.

6. Apply tetrahedra pattern (Section 3.3.3). In this step, for all the interface and inner cells (not the outer ones) the tetrahedra are created following the tetrahedra patterns.

7. Preserve geometrical features (Section 5.3.5). After this step all the forced edges are edges of the tetrahedra mesh, and the octree nodes linked to a forced node have been placed in the corresponding position.

8. Apply surface fitting algorithm (Section 5.3.6). Some nodes are placed onto the surface entities defining the contours of the volumes, and some tetrahedra may be split.

9. Color tetrahedra and create skin triangles of volumes (Section 5.3.7). After this step all the tetrahedra can be assigned to a specific volume or the outer part of the domain. Now the tetrahedra and the nodes of the outer part of the domain can be deleted. The skin triangles of the volumes is formed by the faces (triangles) between tetrahedra of different color, or interfacing a tetrahedron with the outer part of the domain.

10. Make-up and smoothing. This step can be considered out of the mesh generation algorithm itself, as it consists in the improvement of the quality of the elements generated, and it can be applied to any tetrahedra mesh. However, it is highly necessary considering the quality of the tetrahedra in the contours of the domain (Section 5.3.8).

Note that the refinement process involved in the fifth step is an iterative process. Every time an octree cell is subdivided, several aspects have to be considered in the new configuration of the octree:

- New octree nodes are created, so they have to be colored.
- Some of the new octree nodes may be forced interface nodes.
- Some previously existing forced interface nodes may become regular nodes (not forced ones).

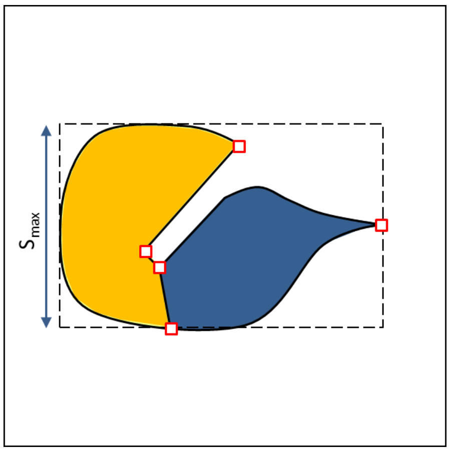
- Some forced edge may be subdivided in order to accomplish the maximum Forced Edge Condition(Section 5.3.1).
- New forced nodes (interface or edge ones) may be created.

All this aspects may force the subdivision of other cells due to the other refinement criteria, so an iterative process is required in order to achieve an octree configuration where all the refinement criteria are accomplished. However, in our experience, few iterations are enough for satisfying all of them. The implementation of the algorithm is detailed in Section 6.5.

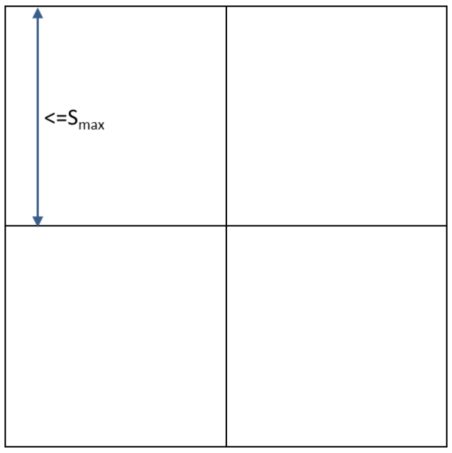
Another important characteristic of the algorithm is that the first six steps are based on the octree, while from the seventh step onwards, the operations are applied to the unstructured tetrahedra mesh.

A graphical example of the steps of the meshing process is shown in Table 5.2 using a 2D model to make the figures more understandable. In the first figure, the offset between $Bbox_m^+$ and the octree root should be equal to S_{max}, but it has been put smaller to make the following figures clearer.

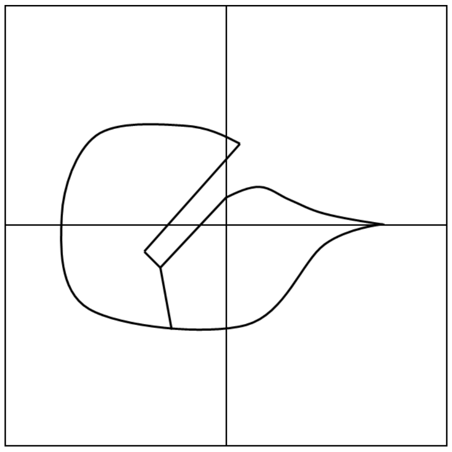
It also has to be considered that some of the parts of the algorithm are intrinsically 3D (such as the process to preserve forced edges), so they cannot be illustrated with a 2D model. This example is formed by two surfaces and it has no mesh size information in the input data. It is important to note the presented algorithm is able to generate a body-fitted mesh with the only information of the input boundaries.

1- *Process input data and create the octree root.* The red small squares represent forced points. The octree root is the black square, and the $Bbox_m^+$ of the model is represented with dotted lines. In this example S_{max} coincides with the minimum side of the model bounding box (s_{box}), as there are no mesh size points.	

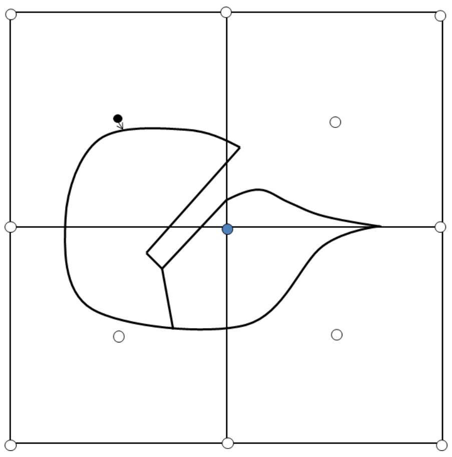
2- *Refine the octree accomplishing the size refinement criteria.* As there are no mesh size point in the model, the octree is refined uniformly with S_{max}.



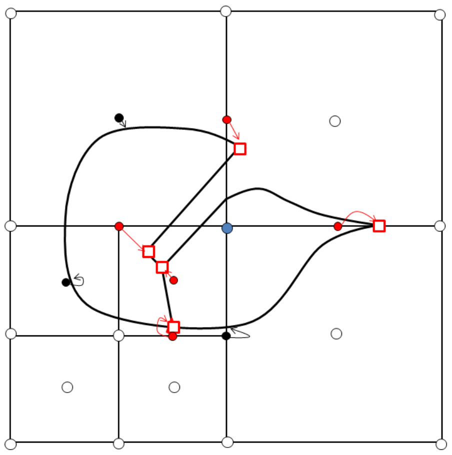
3- *Classify the input boundary entities into the octree.* It is important to note that until now, the input boundaries only have been considered to build the bounding box of the model, but they have not been implied in the octree refinement process.



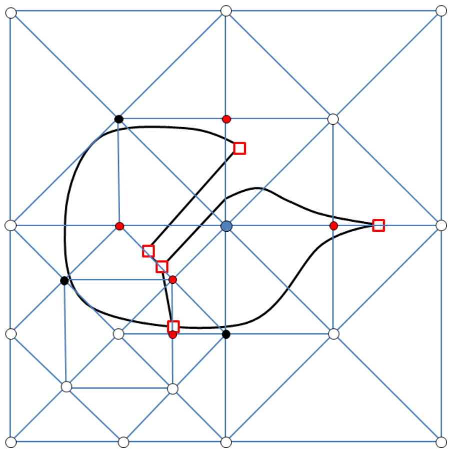
4- *Color the octree nodes and set forced interface points.* The nodes of the figure are painted with the corresponding color: orange or blue if they are inside the corresponding surface, black if they are close enough to the contour entities to be forced interface nodes. Nodes outside the domain are white.



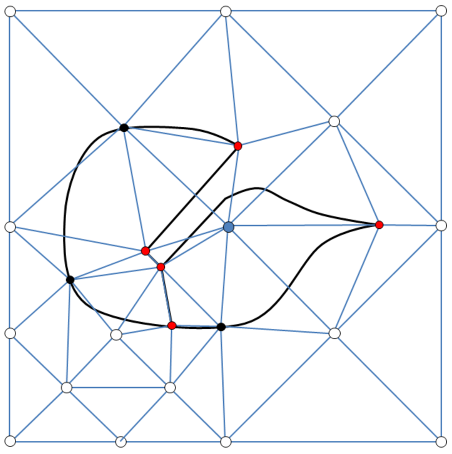
5- *Refine the octree according with the body-fitted refinement criteria.* It can be appreciated that the octree refinement process leads to the creation of new octree nodes. Some of them become forced isolated nodes (red ones) or forced interface nodes (black ones). The arrows indicates the forced position of each forced node.



6- *Apply the tetrahedra pattern.* The tetrahedra are generated from the octree nodes. From this step on, the operations are performed to the corresponding tetrahedra mesh rather than the octree one.



7- *Preserve geometric features.* At this point forced nodes are moved into their forced positions. As a 2D example, only the moving of forced nodes can be appreciated in the figure (the preservation of edges has no sense in 2D).



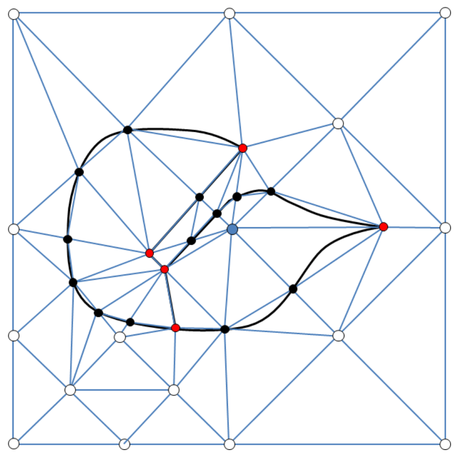
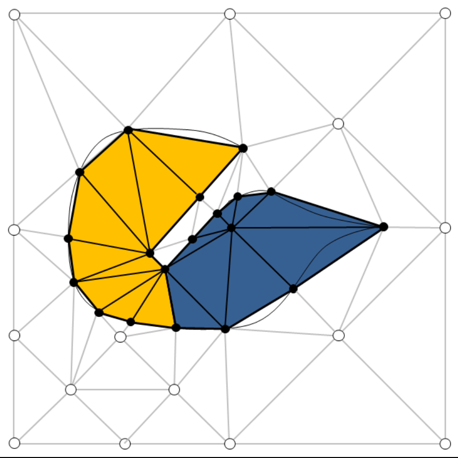
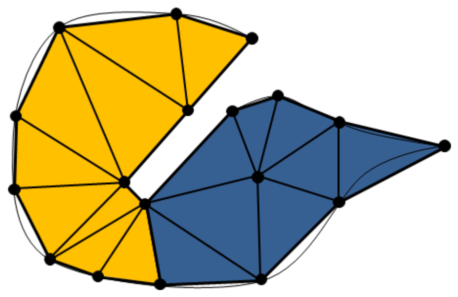
8- *Surface fitting process.* This is the last step of the process which ensures the final mesh could represent the original topology of the input boundaries.	
9- *Tetrahedra coloring and identification of skin mesh.* Elements owning to the orange or blue surface are painted accordingly in the figure. After this process the outer elements can be deleted.	
10- *Make-up and smoothing.* In this step some mesh editing operations are done in order to improve the mesh quality.	

Table 5.2: Steps followed by the body-fitted meshing algorithm applied to a 2D example with two surfaces with no mesh size assigned.

5.3.5 Preserve geometric features

The preservation of geometrical features (corners and ridges) lies on the preservation of forced nodes (Section 5.3.2) and forced edges (Section 5.3.1).

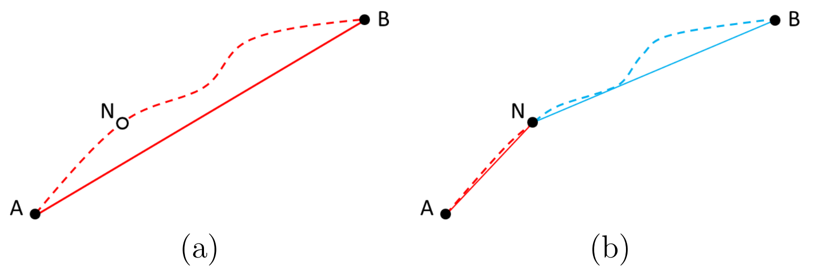


Figure 5.11: Process of splitting a forced edge by inserting a node onto its base line. **(a)** Forced edge $\overline{AB}$ to be split by the node N; the dotted line is the base line of the forced edge. **(b)** Forced edges $\overline{AN}$ and $\overline{NB}$ result from splitting the forced edge $\overline{AB}$ by the node N; dotted lines are the corresponding base lines of the forced edges.

The operations described in this section are performed after the creation of tetrahedra from the octree leaves following the tetrahedra pattern (Section 3.3.3), so at this point there are forced nodes, forced edges and a tetrahedra mesh got directly from the octree cells. This tetrahedra mesh covers the domain to be meshed, but does not preserve the geometrical features and does not fit the surface entities representing the interfaces between volumes (there are nodes outside the domain and tetrahedra with edges crossing volume interfaces). The aim of this process is that the tetrahedra mesh has nodes in the corresponding positions of the forced nodes, and edges corresponding to the forced edges.

Each forced node has an octree node associated to it. The process of preserving the forced nodes is reduced to move the corresponding octree nodes to the position of its forced node.

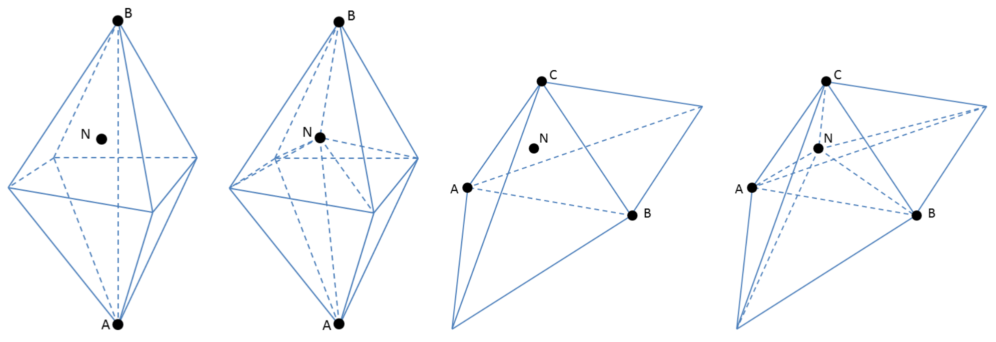
Concerning forced edges, the goal is to force the tetrahedra mesh to have edges coincident with them. As explained in Section 5.3.1, the forced edges are obtained as a linear mesh from the polyline entities. The portion of polyline entity enclosed between the extreme nodes of a forced edge can be defined as the *base line* of the forced edge. Note that this is a pure notation, as the polyline is made of generic line entities (in mesh or geometrical format). For notation purposes, a forced edge will be defined by its extreme nodes: the forced edge $\overline{AB}$ is the one which extreme nodes are node A and node B. The same notation is used for a generic edge of a tetrahedron.

From now on the concept of fitting the tetrahedra mesh to the forced edges is used to define the process of having an edge of the tetrahedra mesh for each forced edge. To achieve this goal a splitting process of forced edges and tetrahedra is proposed. This process involves three basic operations:

- Split a forced edge by inserting a node onto its base line. This process is shown in

Figure 5.11 and it creates two forced edges from an original one connecting its extreme nodes with the splitting node. The base line of the forced edge is split by the node as well.

- Split an edge of a tetrahedron. This process involves all the tetrahedra surrounding the edge. Each of them is split into two tetrahedra as shown in Figure 5.12(b).
- Split a face of a tetrahedron. This process involves a tetrahedron and the neighbor to the face. Each of these two tetrahedra generate three tetrahedra as shown in Figure 5.12(d).



(a) (b) (c) (d)

Figure 5.12: Process of splitting tetrahedra by a node. **(a)** Edge $\overline{AB}$ to be split by the node N, with its surrounding tetrahedra around (4 in this example). **(b)** 8 tetrahedra result from splitting the edge $\overline{AB}$ by the node N. **(c)** Face ABC to be split by the node N, with the two tetrahedra sharing the face. **(d)** 6 tetrahedra result from splitting the face ABC by the node N.

Hereafter the process needed for fitting the tetrahedra mesh to the forced edges is detailed. For each forced edge $\overline{AB}$ we check whether nodes A and B are nodes of a same tetrahedra or not. In case they are, the tetrahedra mesh has an edge coincident with the forced edge, so the objective is already achieved. In case there is no tetrahedra containing nodes A and B, the following procedure must be considered (a graphical example of the steps followed to fit the tetrahedra mesh with a forced edge is shown in Table 5.3):

- Find the tetrahedron surrounding node A which opposite face with respect to A (from now on $face_A$) is intersected by the base line of the forced edge. Note that the base line can be represented by geometrical or mesh entities (NURBS curve or linear mesh). In case of a linear mesh, the intersection operation between the base line and the faces of tetrahedra is much more robust and computationally cheaper.

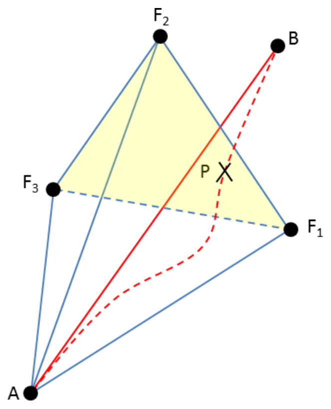


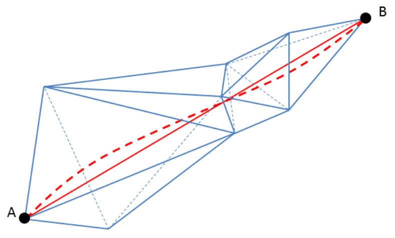
Figure 5.13: Entities involved in the process of making the tetrahedra mesh to have an edge coincident with a forced edge. Red line is the forced edge $\overline{AB}$ and the dotted red line is its base line. The tetrahedron $AF_1F_3F_2$ is the tetrahedron surrounding node A which opposite face with respect to A ($face_A$) is intersected by the base line. Face $F_1F_2F_3$ is the $face_A$ (drawn in yellow). P is the intersection point between the base line and $face_A$. The closest node of $face_A$ to P is F_1, and the closest edge from $face_A$ to P is the edge F_1F_2.

- Find the intersection point P between the base line and $face_A$. Defining min_{side} as the length of the shortest edge of $face_A$, a node (N) is created taking care on the following casuistic (a graphical view of the entities involved is shown in Figure 5.13):
  - If the distance between P and its closest node of $face_A$ is lower than ($\alpha_{vertex} \cdot min_{side}$), then this node is moved to the position of P and it is considered as the node N. The parameter α_{vertex} is a real value between zero and one; the value used for it in the presented implementation of the algorithm is discussed in Section 6.7. This is the case shown in figures d) and e) of Table 5.3.
  - If the previous case is not accomplished and the distance between P and its closest edge of $face_A$ is lower than ($\alpha_{side} \cdot min_{side}$), node N is created in the position of P, and the closest edge is split by the node N. The parameter α_{side} is a real value between zero and one; the value used for it in the presented algorithm is discussed in section 6.7. This is the case shown in figures f) and g) of Table 5.3.
  - If the previous cases are not accomplished, node N is created in the position of P and $face_A$ is split by the node N. This is the case shown in figures b) and c) of Table 5.3.
- Split the forced edge $\overline{AB}$ by the node N. Now the forced edge $\overline{AN}$ is an edge of the tetrahedra mesh.

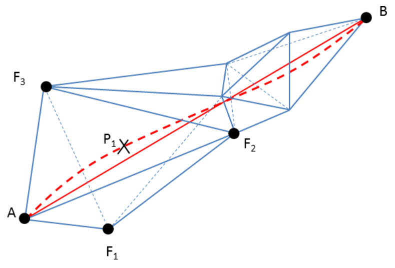
- If the forced edge $\overline{NB}$ is not an edge of the tetrahedra mesh, repeat the process with it.

Note that the nodes created in this process are not octree nodes, as they are not related with any octree position. Furthermore, these nodes do not need any coloring algorithm, as they lay on a interface entity.

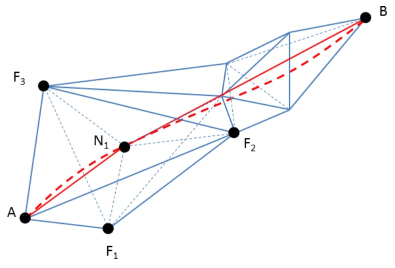
(a) Initial configuration of the forced edge $\overline{AB}$ (red line) with its surrounding tetrahedra (in blue), and its base line (dotted red curved line).



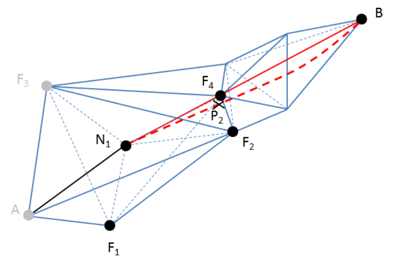
(b) Find the tetrahedra $(AF_1F_3F_2)$ owning node A which opposite face to it $(F_1F_2F_3)$ intersects the base line of forced edge $\overline{AB}$. P_1 is the intersection point.



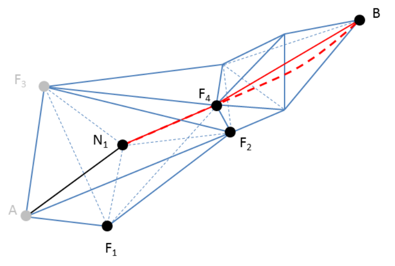
(c) As P_1 is not close enough to F_1, F_2 or F_3, and neither to the edges of face $F_1F_2F_3$, create the node N_1 in the position of P_1 and split the face $F_1F_2F_3$ and the forced edge $\overline{AB}$ by the node N_1. Now the forced edge $\overline{AN_1}$ is already an edge of the tetrahedra mesh.



(d) Proceed the treatment of forced edge $\overline{N_1B}$. Find the tetrahedra $(N_1F_2F_1F_4)$ owning node N_1 which opposite face to it $(F_1F_2F_4)$ intersects the base line of forced edge. P_2 is the intersection point.



(e) As P_2 is close enough to F_4 (closer than α_{vertex} times the minimum edge of $F_1F_2F_4$), move F_4 to the position of P_2 and split forced edge $\overline{N_1B}$ by node F_4. Now the forced edge $\overline{N_1F_4}$ is already an edge of the mesh.



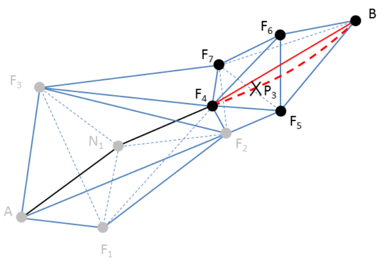
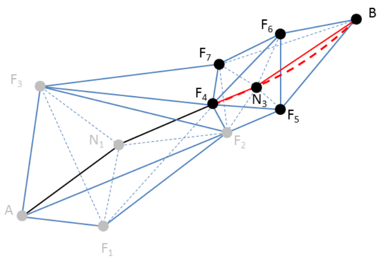
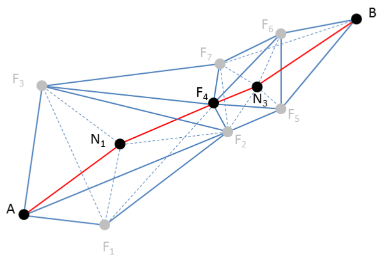
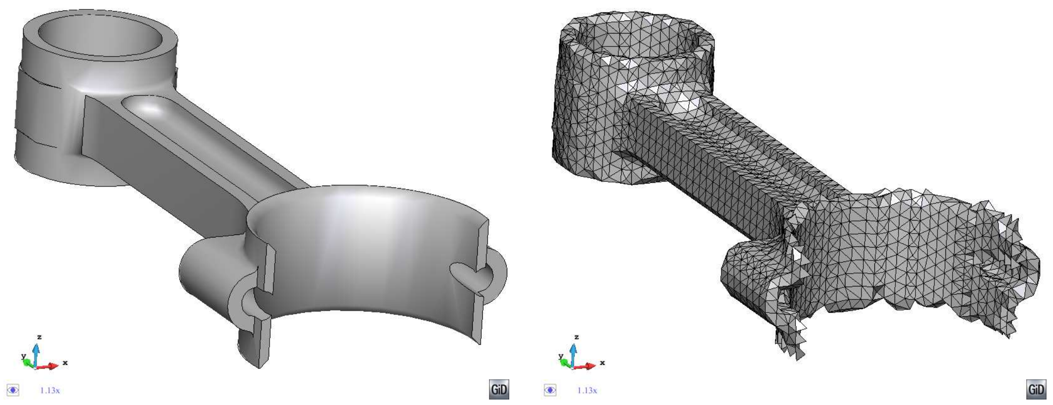
(f) Proceed the treatment of forced edge $\overline{F_4B}$. Find the tetrahedra $(F_4F_5F_7F_6)$ owning node F_4 which opposite face to it $(F_5F_6F_7)$ intersects the base line of forced edge. P_3 is the intersection point.	
(g) As P_3 is close enough to edge F_5F_7 (closer than α_{side} times the minimum edge of $F_5F_6F_7$), creation of node N_3 in the position of P_3 and split of edge F_5F_7 and the forced edge $\overline{F_4B}$ by the node N_3. Now the forced edges $\overline{F_4N_3}$ and $\overline{N_3B}$ are edges of the mesh, so the process is finished.	
(h) Final configuration of the tetrahedra with the new forced edges created: $\overline{AN_1}$, $\overline{N_1F_4}$, $\overline{F_4n_3}$ and $\overline{N_3B}$.	

Table 5.3: Example of the process to fit a tetrahedral mesh with the forced edge $\overline{AB}$.

5.3.6 Surface fitting

This section describes the process of fitting the tetrahedra mesh into the volumes of the domain to be meshed, representing their interface surfaces accurately (from now on *surface fitting* process). The methodology presented is applied to the tetrahedra mesh (come from the tetrahedra pattern defined in Section 3.3.3), in which the process of preserving geometrical features (defined in Section 5.3.5) has been carried out. Some of the nodes of the mesh are forced nodes (fixed in a position in space) and some of its edges are coincident with the forced edges. The process defined from now on preserves the forced nodes as well as the edges corresponding to the forced edges.

The surface fitting process is based on the isosurface stuffing method published in [LS07] (from now on *isostuffing method*). However, it has some differences, as its objectives and restrictions are different:



(a) (b)

Figure 5.14: Example of a 3D model of a mechanical piece **(a)** and a tetrahedra mesh of it generated using the isostuffing method **(b)**.

- The isostuffing method is applied to an isolated volume. In this work all the volumes of the domain are meshed together, and they can be in contact one to each other.
- The isostuffing method do not preserve sharp edges nor corners. The present algorithm must respect the forced nodes and forced edges in order to preserve the topology of the model and its geometrical features.
- The isostuffing method does not preserve the topology of the domain, as the size of the cells used is chosen by the user a priori; if this size is much more greater than a thin part of the domain, this part will not be represented by the final mesh. The surface fitting process presented in this work ensures the topology of the mesh will be the same as the one of the input geometry. This is due to the topological refinement criteria applied to the octree the mesh is generated from.

An example of tetrahedra mesh generated using the isostuffing method is shown in Figure 5.14. It can be appreciated that the sharp edges relevant for the definition of the shape of the model are not preserved by the mesher.

The surface fitting process is based on the edges of the tetrahedra mesh which are not forced edges: the *iso-edges*. It consists in the following steps:

1. *Compute iso-edges intersections.* Compute the intersection of the iso-edges with the input boundaries (intersection points). As described in Section 3.4, the following situations must be considered:

- If the intersection between an edge and the the surface entity is co-planar (P intersection type), no intersection point is considered.
- Intersection points closer than tol_c are collapsed into a MIP.
- As the extremes of an edge are already colored, if both belong to different volumes and no intersection point has been detected, the corresponding GIP must be considered as the intersection point.

Some of the edges can intersect more than one time the input boundaries, but not more than two due to the topology criterion (a). In these cases with two intersections only one of them is taken into account (no matter which one). From now on we only take care of the edges intersecting the input boundaries; the ones that do not intersect them are taken out from the iso-edges.

2. *Move nodes.* For each node of the iso-edges, we consider its closest intersection point (between the intersection points of the edges containing the node). It can be assigned to each of these nodes (nod) the closest intersection point (ip) and the edge the ip is into (e). Then, if nod is not a forced node, move it to the position of ip if

$$d < \alpha_{iso} \cdot l_{edge}$$

where d the distance between nod and ip, and l_{edge} the length of the edge e. The parameter α_{iso} is a real value between zero and one. The value used in the algorithm is explained in Section 6.7. This process of moving the node is only performed if the resulting configuration does not generate poor quality tetrahedra.

Moving a node implies to recompute the possible intersection points of all its connected edges, as one of their extremes is moved. This leads to an iterative process where new edges can be added to the iso-edges, and some existing one can be taken out (the ones with no intersection point). A 2D example of a situation where the movement of a node creates a new intersection point is shown in Figure 5.15.

3. *Split edges.* Split the remaining iso-edges (the ones with intersection point after the node moving process) by their intersection point reconnecting the surrounding tetrahedra, as shown in Figures 5.12(a) and (b).

4. Repeat steps 1, 2 and 3 with a new set of iso-edges: the ones of the mesh having at least one if its nodes onto the input boundaries.

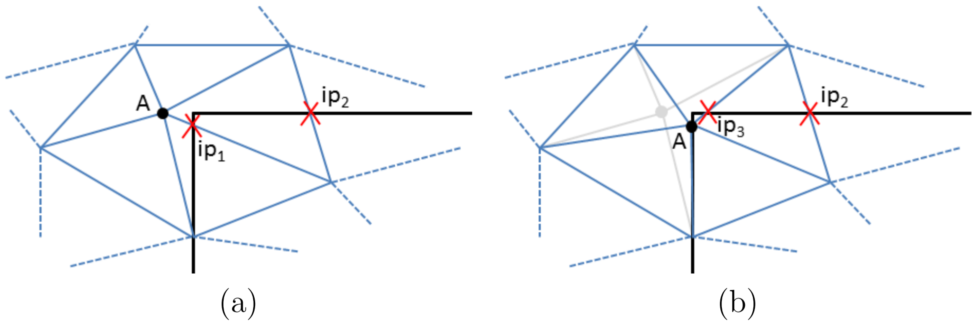


Figure 5.15: 2D example of creation of a new intersection point when moving a node. Black line represents the input boundaries, and part of the triangle mesh is shown in blue. **(a)** Initial configuration: node A is close enough to intersection point ip_1, so it is moved. **(b)** Final configuration after moving A: the new intersection point ip_3 has appeared.

5. Repeat steps 1, 2 and 3 with a new set of iso-edges: the ones of the mesh fulfilling the following conditions (both of them):
   - At least one of the nodes of the edge is onto the input boundaries.
   - At least one of the nodes of the edge is connected (by an edge) to a forced node in edge.

The accomplishment of the topology criterion (a) allows the presence of edges intersecting twice the domain in the mesh coming from the tetrahedra pattern. For this reason the moving and splitting process has to be applied the second time to the iso-edges (step 4). The involved iso-edges this second time have at least one of their nodes onto the input boundaries (as they have already been processed the first time).

A 2D example applying two times the process of moving nodes and splitting edges is shown in Table 5.4.

For pathological situations, applying a third time the process of moving nodes and splitting edges is needed (step 5). These pathological situations happen when some of the new edges created in the previous steps has an intersection point relevant for the topology of the mesh. These cases occur near sharp edges, so the iso-edges involved are only the ones connected to a forced node in edge. An example is shown in Figure 5.16, where the difference between applying or not the moving and splitting process the third time can be appreciated. If there were not forced edges it will not be needed to repeat a third time the moving and splitting operations, because a set of tetrahedra would already represent topologically well each volume of the domain.

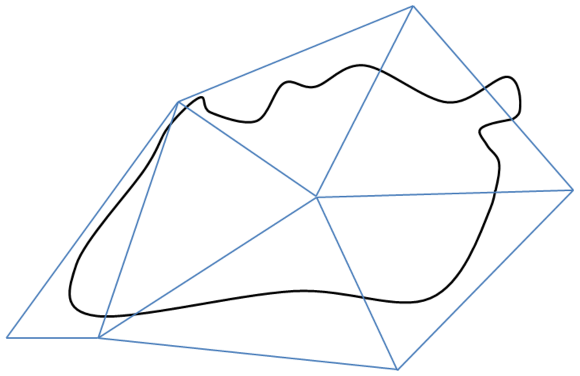
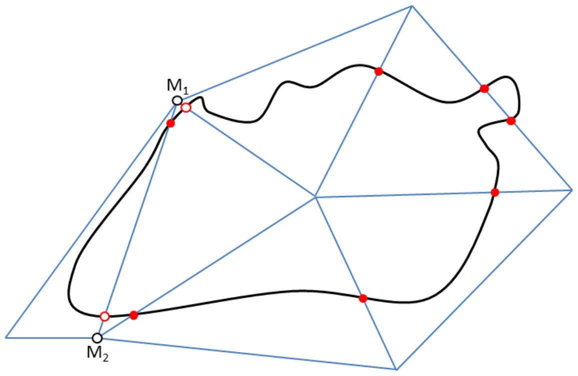
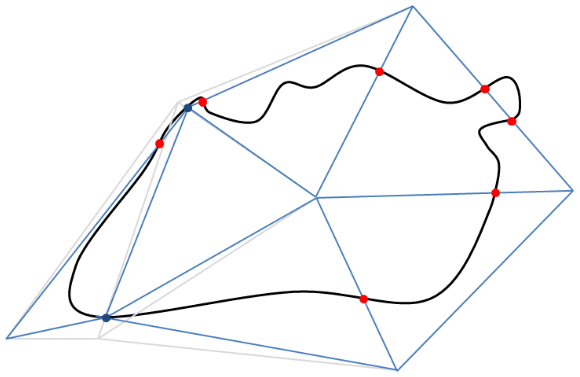
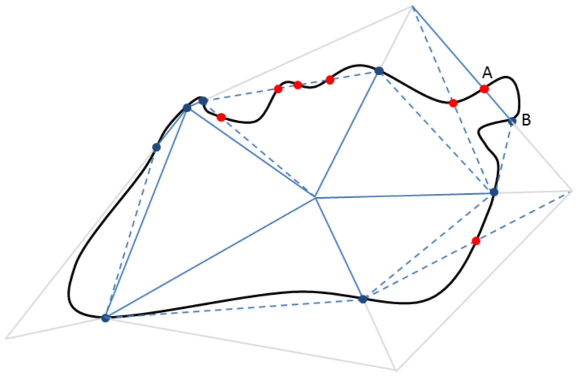
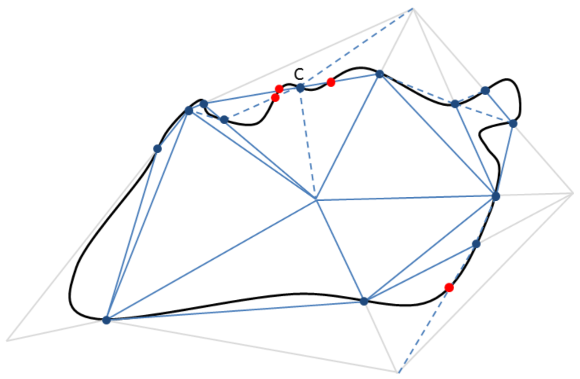
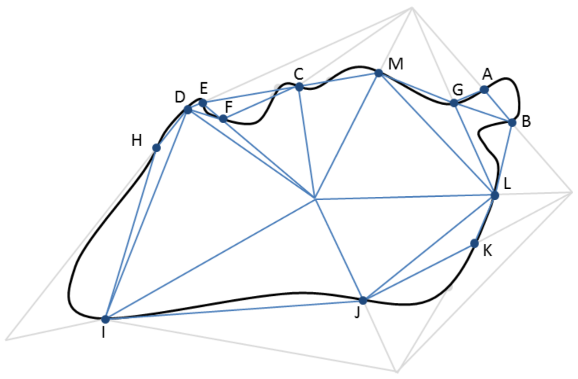
 **(a)** Initial configuration of the mesh before surface fitting process.	 **(b)** Detection of the intersection points (red dots). Nodes M_1 and M_2 are close enough to its closest intersection points (red circumferences) to be moved.
 **(c)** Mesh configuration after moving points M_1 and M_2. Some intersection point have disappeared and a new one have appeared. Light gray lines represent the mesh in the previous step.	 **(d)** Splitting process by the intersection points. Dotted lines are the new edges created. There is an edge with two intersection points (A and B). Node B has been used for the splitting (A could be used as well). Edges in gray represent outer edges with no intersections.
 **(e)** Second iteration. As there are no nodes to be moved, edges with intersections are split. There is an edge with three intersection points. Point C is chosen (arbitrarily) for the splitting.	 M A C D E G F B H L K J I **(f)** Final configuration of the mesh.

Table 5.4: 2D example of surface fitting process. The black line represents the input boundaries, and the triangle mesh is represented in blue.

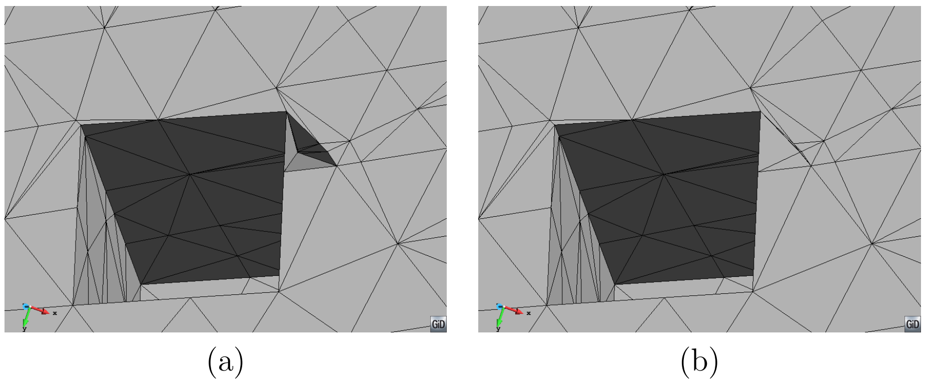


Figure 5.16: Example of mesh generated applying **(a)** or not applying **(b)** the process of moving nodes and splitting edges the third time. It can be appreciated in Figure (a) that the topology of the mesh near a forced edge does not represent well the topology of the domain.

One can think about repeating the process of moving nodes and splitting edges as many times as needed until the mesh does not present any intersection point. This is not feasible as it could lead to almost infinite loops (in some configurations with curved input boundaries). Also, each time it is applied, the new elements will have worse quality.

As explained before, the process of moving nodes may create new intersection points. Also the splitting process can create more of them, as it implies the creation of new tetrahedra with the corresponding new edges. The new edges created may have more than two intersections with the input boundaries, but these intersection points are not relevant to preserve the topology of the volumes, so it is not needed to take them into account.

A 2D example of the described surface fitting process is depicted in Table 5.4. In this example, the process of moving and splitting could be repeated more times, as there are more intersection points (in edge $\overline{FC}$ of the mesh shown in (f)). However, the topology of the input boundaries is already well represented, so no more iterations are needed.

5.3.7 Tetrahedra coloring

Once the surface fitting and preserving features operations have been performed, the mesh is in the following situation: it is ensured that the tetrahedra represent the topology of the input boundaries, and all the nodes of the mesh are colored (it is known if they are inside a volume or onto interfaces between volumes). Now the operation of tetrahedra coloring must be done: this is, assign a volume to each one of the tetrahedra. In this work the concept of assigning a color to an entity is used to determine inside which volume the entity is, so *color* and *volume* are used indistinctly.

The tetrahedra to be colored can present two basic situations:

- At least one node of the tetrahedra is inside a volume (inner node).
- All the nodes of the tetrahedra are onto interfaces between volumes (surface or line entities). From now on this kind of tetrahedra will be referred as *interface tetrahedra*.

Tetrahedra presenting the first situation are obvious to be colored. Because of the topological refinement criteria and the surface fitting properties, it is ensured that there is no edge of the the tetrahedra with nodes inside different volumes. This implies the tetrahedra belongs to the same volume of the inner node.

The second situation (interface tetrahedra) is much more complicated to solve. It has to be noted that each node of an interface tetrahedra is interfacing a number of volumes; they are the so called possible volumes of a node. Considering the forced edges lying on an interface between volumes, the concept of possible volumes of a forced edge can be defined analogously as for the nodes.

The possible volumes of each tetrahedron are clearly identified: they are the volumes which are interfaced by all the nodes of the tetrahedron. There are some configurations under which the color of an interface tetrahedron can be determined. This is the case where the tetrahedron has only one possible volume. The rest of interface tetrahedra are considered as undetermined ones; these are the cases that must be solved.

A 2D example of these different types of elements (in terms of coloring process) is shown in Figure 5.17. Here the mesh for two surfaces (A in blue and B in orange) is shown. All elements containing node M belong to surface A, as M is an inner node to A. The other elements are interface ones. The possible colors of each one of the interface nodes are listed in Table 5.5 (note that the outer part of the domain is considered as color zero):

Considering the interface elements, their possible volumes can be easily gotten analyzing the common possible volumes of each one of their nodes. They are depicted in Table 5.6.

It can be seen that the elements HGJ and JPK are undetermined in this example, as they can belong (topologically) to surface B or the outer part of the domain indistinctly.

As each undetermined tetrahedron has more than one possible volume, there are several possible configurations of colors for them. A configuration is understood as the assignment of a color to each one of the undetermined tetrahedra. It has to be noted that the triangles of the skin meshes of each volume are the faces of the tetrahedra shared by tetrahedra of different

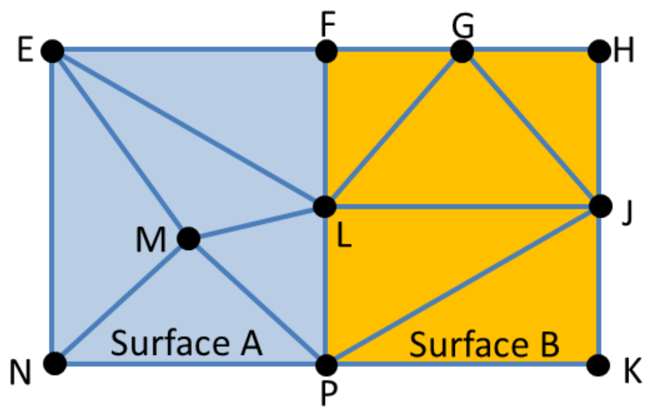


Figure 5.17: 2D example of a triangle mesh of two surfaces: A (blue) and B (orange). Elements containing node M are directly colored as A, as M is a inner node to A. The other triangles are interface elements. Elements HGJ and JPK are undetermined, as they have two possible colors: B and 0 (exterior).

Node	Possible colors
E	A , 0
F	A , B , 0
G	B , 0
H	B , 0
J	B , 0
K	B , 0
L	A , B
N	A , 0
P	A , B , 0

Table 5.5: Possible colors of nodes of the mesh of Figure 5.17.

Element	Possible colors
ELF	A
FLG	B
HGJ	B , 0
JPK	B , 0
JLP	B

Table 5.6: Possible colors of tetrahedra of the mesh of Figure 5.17.

color, so each configuration of colors lead to a different tetrahedra skin mesh. Taking care about the topology of the input boundaries, the final configuration of colors has to accomplish the following conditions to be considered as valid:

1. Between the tetrahedra surrounding one node, it must be at least one tetrahedron for each possible volume of the node. This implies, for example, that the tetrahedra surrounding a node in interface cannot be all of them of the same color.

2. Between the tetrahedra surrounding one forced edge, it must be at least one tetrahedron for each possible volumes of the forced edge.

3. Considering the input boundaries in the near region from a given node in interface, if the input boundaries are manifold in this near region, the skin of the tetrahedra surrounding the node must be manifold. This condition can only be applied for watertight volumes and implies, for example, that if a node is in a manifold region of the input boundaries, there must be a closed loop of triangles around the node in the skin of volume's tetrahedra.

4. The mesh of a volume must present the same topology as the volume. This means, for example, that if the volume has n holes, its mesh must have also n holes.

The accomplishment of these conditions can reduce to one the possible colors of some tetrahedra. In this cases, these ones would not be undetermined. However, other tetrahedra may remain undetermined.

A good 2D example of undetermined elements considering only two different colors (interior and exterior) is shown in the (f) figure of Table 5.4 (see page 110). The triangles AGB, LBG, LGM, CEF, FED, DHI and KLJ have all their nodes in the boundaries. AGB, DHI and KLJ are not undetermined elements, as they can only be colored as inside the surface taking into account the first condition defined above: if they were colored as outside, nodes A, H and K respectively won't have any element inside the surface, and they are nodes interfacing the surface and the outer part. Then, in this example, the elements LBG, LGM, CEF and FED are the undetermined ones. These elements could be colored as interior or exterior each of them, and both situations may lead to topologically correct meshes. This example shows that the problem of coloring the undetermined tetrahedra has more than one solution, and it is not obvious (actually, in some cases it is impossible) to determine whether a solution is better or worse than another.

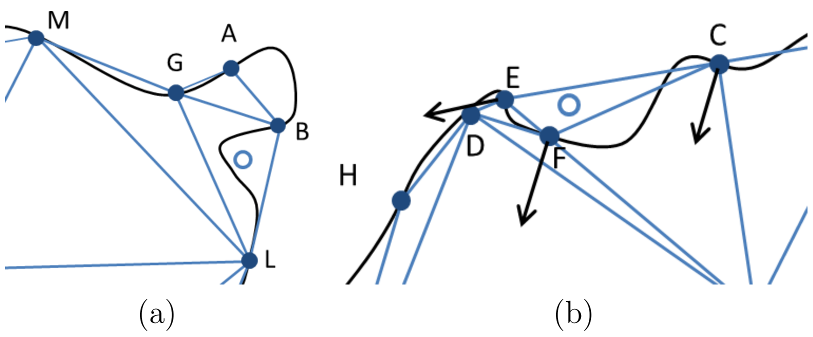


Figure 5.18: 2D examples of pathological configurations for element coloring. Both cases are taken from the mesh shown in (f) figure of Table 5.4. **(a)** The element LBG is a triangle of the surface, but its center (white dot) is outside it. **(b)** The element CEF could be colored as inside or outside of the surface; the oriented normal vectors in nodes C, E and F does not point to the center of the element. Black arrows are the oriented normal vectors towards the surface, and the white dot is the center of the triangle.

A possible strategy for the tetrahedra coloring should be to assign the tetrahedra the color of their center using the node coloring technique explained in Chapter 4, but this can lead to wrong decisions as there are several pathological configurations where the center of an element of a volume is not inside it. This is the case of triangle LBG in (f) figure of Table 5.4. A zoom of this triangle is shown in Figure 5.18(a): LBG is topologically a triangle of the surface A, but its center is outside it.

Another possible strategy for undetermined tetrahedra should be take care about the normal vectors of the interface surface entities in the nodes of the tetrahedra. This normal vector can be oriented in the sense that it can be determined which volume it is pointing to (between the volumes interfaced by the surface entity). The notation *oriented normal to a volume* V is used from now on to indicate the normal vector of a surface entity pointing towards the volume V.

The computation of the normal of a surface of the volume is not obvious taking into account that the contours of the domain can be non-watertight. We are leaving this aspect unsolved by now, and let us suppose the oriented normal vectors can be created. One could think about situations which the oriented normal vectors can determine whether the color of a tetrahedron is one or another depending on where they are pointing to. For example: the cases where the four normal vectors of an interface tetrahedron (one for each node) oriented to a given volume are pointing at the center of the tetrahedron could be considered as determined: the color of the tetrahedron should be the volume one. (In this context, point to the center of a tetrahedron means that the center is in the semi-space defined by the oriented normal vector).

Unfortunately, not all the tetrahedra accomplish with this condition, so there may still be some undetermined tetrahedron. This is the case presented in Figure 5.18(b): it is a 2D case where it can be seen that the oriented normal vectors to the surface in the nodes of the element CEF are not leaving the center of the element in the same semi-space.

Furthermore, the same topological situations can be represented by different volume boundaries which present a drastically different configuration of normal vectors in the nodes of the elements. This is the case shown in Figure 5.19, where different interfaces between surface A and B present the same element with the normal vectors in two of its nodes identical, and the third normal different for each shape of the boundaries. This 2D example demonstrates that the process of tetrahedra coloring cannot be based on the normal vectors of the interface entities in the nodes of the elements. It has to be taken into account that in 3D the possible configurations are much more complicated than in 2D.

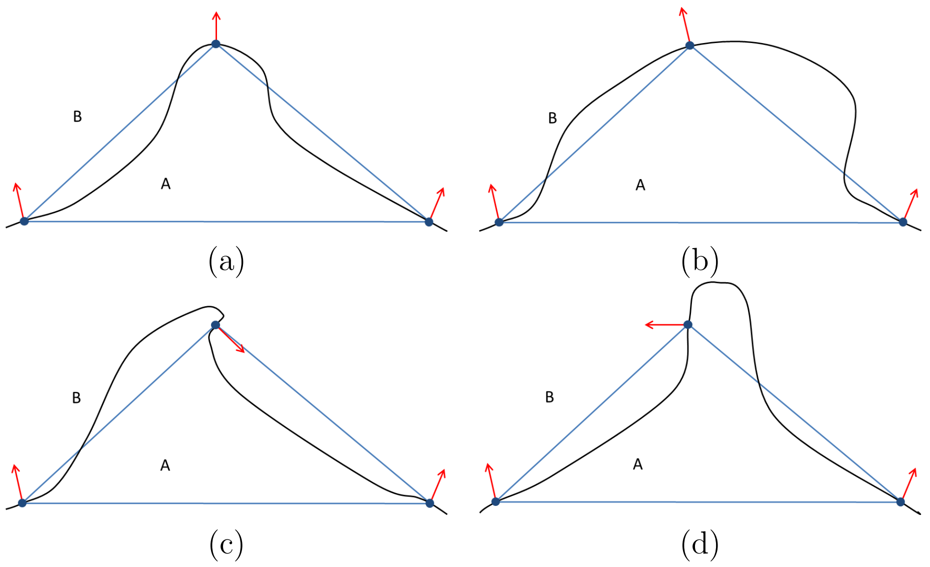


Figure 5.19: 2D example. The black curved line represent the interface between surfaces A and B. **(a)**, **(b)**, **(c)** and **(d)** represent different shapes of the interface. In all the cases an undetermined triangle is depicted and the vectors are the normal vectors pointing to surface B. As it can be seen, in all the cases the element should be colored as inside surface A. The normal vectors in two of the nodes are identical for all the cases, and the third (the upper one) is different in all of them.

The reason why the two strategies defined above fail is they are based on geometrical conditions, and the problem to solve is more topological than geometrical. The color of an undetermined tetrahedron is independent on the percentage of the element inside or outside a volume: it is determined for topological conditions (the ones defined previously). To take

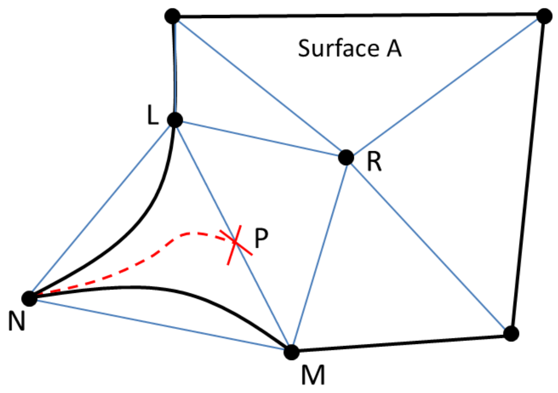


Figure 5.20: Graphical interpretation of Proposition 1 in a 2D example. The black line represents the contour of surface A, and its triangle mesh is represented by blue lines.

care about the topology, a strategy is proposed to color the undetermined tetrahedra starting from the tetrahedra with known color. It is based in the following proposition:

Proposition 1. Considering an undetermined element T and an element R of color A which is neighbor of T by face F. Being N the node of T opposite to face F, if there is a point P inside volume A laying on face F, and a continuous curve in space from P to N completely inside element T with no intersection with the boundaries of A, then the color of T is A.

A graphical view of this proposition in a 2D case is shown in Figure 5.20. In this example the element NML is undetermined, as all its nodes are on the interface between surface A and the outer part of the domain. All the other elements are fully determined (they belong to surface A), as they contain node R, which is inner to A. Point P is inner to A, and it lays on the face shared by elements NML and MRL. As it exists a continuous curve (shown in dotted red line) from N to P, totally inner to element NML and with no intersections with the boundaries of A, element NML can be considered inner to surface A. It has to be noted that this process imply the coloring process of point P, which is done following the algorithm described in Chapter 4.

Proposition 1 is used to color the undetermined tetrahedra (the implementation of the algorithm is detailed in Section 6.5). However, it cannot be applied in some cases as the ones where an undetermined tetrahedron has no neighbor by face with a determined color, or the cases where the points in the common face are onto a surface entity (they are not inner to a volume), or the coloring process of these points fails. Another case where the proposition cannot be applied is when the volumes involved are non-watertight. For this reason, it has to be planned that after this coloring process some undetermined tetrahedra can still remain.

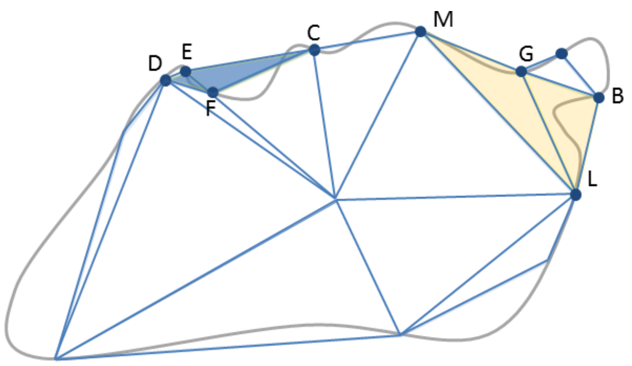


Figure 5.21: Two clusters of undetermined elements (blue and orange) of the mesh shown in (f) figure of Table 5.4.

From now on the methodology to color the remained undetermined tetrahedra is explained.

As it can be seen, the coloring of one tetrahedron can affect the coloring of its neighbors, as the skin of the tetrahedra changes and can affect the manifold condition of each node. However, each tetrahedron affects only to its neighbors, so there can be defined different clusters of undetermined tetrahedra which are independent one from each other; these clusters are made by the undetermined tetrahedra connected at least by one node. As an example, the clusters of undetermined tetrahedra of the mesh in figure (f) of Table 5.4 are identified in Figure 5.21. As these clusters are independent one from each other, the methodology presented can be applied on a cluster by cluster manner.

Considering all the undetermined tetrahedra of a cluster, each of them has its possible colors. All the possible color configurations can be obtained, and each one of these configurations implies a different skin mesh of each volume's tetrahedra. Checking the topological conditions that must be accomplished (detailed previously in this section) for the final mesh, some of the configurations are not valid, and some of them are valid. For the tetrahedra coloring purposes, any one of the valid configurations is set as the solution.

As an example, lets see the case of the orange cluster of undetermined elements shown in Figure 5.21. It is made of two elements: LBG and LGM. Each of them has two possible colors: inside or outside the surface. All the different configurations are shown in Figure 5.22. The configuration shown in Figure 5.22(a) is not valid because the topology of the final mesh is not the same as the one of the surface it is representing (the mesh has two unconnected sets of elements). The configurations of Figure 5.22(b) and (c) are not valid because they have non-manifold nodes in regions where the contour of the surface is manifold (node G in case (b) and node L in case (c)). The only valid configuration is the one shown in Figure 5.22(d).

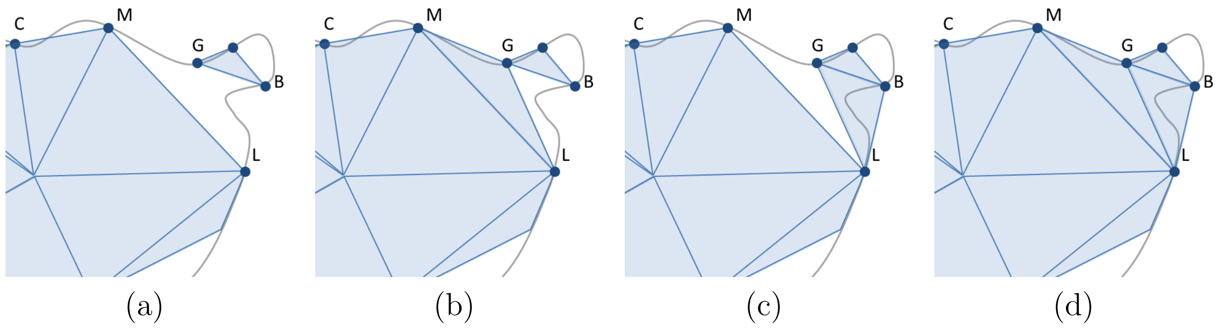


Figure 5.22: Zoom of the orange cluster of elements in Figure 5.21. All the possible configurations taking into account the different colors (inside or outside the surface) of elements LBG and LGM.**(a)** Both elements are outside the surface. **(b)** LBG is outside and LGM inside. **(c)** LBG is inside and LGM outside. **(d)** Both elements are inside.

5.3.8 Make-up and smoothing

The make-up and smoothing operations are used to improve the quality of the mesh once it has been generated. Make-up operations are the ones changing the topology of the mesh, and smoothing process only implies movement of the nodes, maintaining the original connectivity of the elements.

The acceptable quality of the mesh elements is a relative parameter, as the mesher aims to be applied to different kinds of numerical simulations, and they are applicable in different ranges of qualities for the elements [She12]. A clear example of this is the boundary layer meshes, which elements should have an aspect ratio higher than 10000 in some cases; such distorted elements are not valid for standard FEM analysis. However, almost all the methods require non-inverted elements. This means that the Jacobian of the transformation of the element (from the parametric to the 3D space) should be positive. This lead to require, at least, a strictly positive Jacobian in the elements of the final mesh. A graphical interpretation of an inverted element is provided in Section 5.3.3.

In this section there are references to tetrahedra, triangles and edges. Edges are edges of the tetrahedra elements, and triangles are considered as the faces of the tetrahedra interfacing tetrahedra of different color. Thus, each volume tetrahedra mesh is surrounded by a triangle mesh.

Considering the mesh generated may have forced nodes and forced edges, there are some restrictions to be applied to the make-up and smoothing operations:

- Forced isolated nodes (Section 5.3.2) cannot be moved nor deleted.

- Forced nodes in edge which are extreme of a *polyline entity* (Section 5.3.1) cannot be moved nor deleted.
- Forced nodes in edge can only be moved along the polyline entity they belong to.
- Forced interface nodes can only be moved lying onto the surface entity they belong to.
- If both extreme nodes of a tetrahedron edge are forced nodes in interface and the edge is not a triangle edge, the edge cannot be collapsed.
- If both extreme nodes of a tetrahedron edge are forced nodes in edge and the edge is not a forced edge, the edge cannot be collapsed.
- If a tetrahedron edge has one forced node in edge and one forced node in interface, it cannot be collapsed.

From now on, the operations involving the collapse of an edge, or the movement of a node are only applied to the entities not violating these restrictions. It has to be taken into account that the collapse operation involves the deletion of one of the two nodes involved.

The elements obtained from the tetrahedra pattern defined in Section 3.3.3 have very good quality if the nodes are in the octree positions (see Section 3.3.2), so the make-up and smoothing operations are only needed for the tetrahedra containing a forced node, the ones coming from interface cells and the ones resulting from surface fitting or preserving features operations.

The parameters α_{vertex} and α_{side} (Section 5.3.5) try to guarantee a minimum quality in the tetrahedra resulting from the preserving features process, and α_{iso} tries to do the same for surface fitting operations. However, some configurations of the mesh and the input boundaries may lead to low quality elements. This situation often causes the presence of small edges in the mesh. An example of these small edges is shown in Figure 5.23(a). To improve the quality of the elements surrounding these small edges, an edge collapsing step is performed (make-up operation). Taking into account that the octree cell containing each node gives an idea of the mesh size required for the mesh in that region (because of the user desired sizes or as a result of a topological refinement process), the edges smaller than a given portion of the size of the cell they are inside are collapsed. An edge is collapsed if it does not violate the restrictions defined at the beginning of this section and

$$l_{edge} < \alpha_c \cdot s_c \tag{5.5}$$

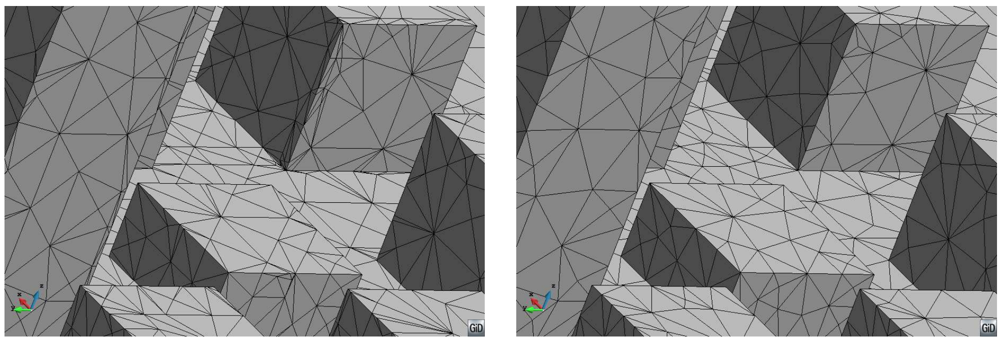

(a) (b)

Figure 5.23: Example of a mesh (a) before and (b) after the make-up and smoothing process.

where s_c is the size of the smaller octree cell between the ones where the extreme nodes of the edge are inside, l_{edge} the length of the edge and α_c is a real value greater than zero. Its value is discussed in Section 6.7.

The smoothing operation applied consists in a Laplacian-like smoothing that displaces a node into a position such that its surrounding elements improve their quality. The smoothing process is applied on a node by node basis, and follows these steps:

1. Compute the quality of all the tetrahedra surrounding the node, and identify the worst of them (q_0).
2. Move the node to a candidate position.
3. Compute the quality of the tetrahedra surrounding the node in the new configuration and identify the worst of them (q_1).
4. If q_1 is better than q_0, the node is left in the candidate position, otherwise, the node is returned to its original one.

This process implies the definition of a quality measure of the elements, as well as a procedure to obtain the candidate position. As a quality measure, the minimum dihedral angle of the tetrahedra has been chosen, considering that a tetrahedra is worse than another one if it has a lower minimum dihedral angle. The election of the candidate position is explained later on.

It has to be noted that the mesh can have three kinds of elements: tetrahedra, triangles and linear elements (corresponding to the forced edges). All the nodes have tetrahedra around them, but only some of them have triangle or linear (1D) elements. Taking into account the restrictions defined in the beginning of this section, the degrees of freedom for moving a node are restricted by its nature: if it is a forced node in edge, it can only move along the polyline entity corresponding to the forced edge, and if it is a forced interface node it must lay on the surface entity it belongs to. This aspect governs the candidate position for each node:

- If the node is a forced node in edge, the mean position m of the opposite nodes of the forced edges it belongs to is calculated. Considering the polyline entity those forced edges come from, the candidate position is the closest point to m laying on the polyline entity.
- If the node is a forced interface node (it has triangles around it), all the nodes of the triangles around it (except the node itself) are considered. Their mean position m is calculated, and the candidate position is the closest point to m laying on the surface entity the triangles belong to.
- If the node has no triangles nor forced edges, the candidate position is the mean position of the nodes of the surrounding tetrahedra of the node (except the node itself).

As the movement of a node affects the quality of all its surrounding elements, the smoothing operation is thought as an iterative process where several loops over all the nodes are performed in order to improve the quality of the mesh.

Apart from edge collapsing and nodes smoothing, the edge flipping operation (a make-up operation) is performed in the mesh [GB03]. The cases *2 to 3*, *3 to 2*, *4 to 4* and *5 to 6* are implemented. These operations imply the removal of a face (the first case) creating an edge, or the removal of an edge (the other cases) creating some extra faces. Apart from the pathological configurations described in [GB03] which determine the situations where the edge flipping cannot be made, some topological configurations must be considered in order not to invalidate the topology of the mesh generated. In particular, the following restrictions are considered:

- If an edge is a forced edge or the edge of a triangle, it cannot be deleted.
- If a tetrahedra face is a triangle of the mesh, it cannot be deleted.

The collapse, edge flipping and smoothing operations are applied in a sequential iterative manner to account for the updated configurations of the mesh each time. A maximum of four iterations is set in the present work. The result of applying them to the mesh shown in Figure 5.23(a) is depicted in Figure 5.23(b). In Chapter 7 several examples are shown indicating the mesh quality before and after the make-up and smoothing operations.

5.3.9 Mesh quality

This section focuses in the analysis of the quality of the final mesh generated by the new mesher.

For the embedded mesher, the quality of the mesh is totally guaranteed, as all the tetrahedra come directly from the predefined patterns. However, for the body-fitted mesher, it has to be considered that a body-fitted mesh is forced to respect the boundaries of the domain. If a part of the domain is bounded forming a very small dihedral angle, the tetrahedra representing it will have the same dihedral angle. This limits the scope of minimum quality guarantee to the inner parts of the domain, or the boundaries which are relatively smooth.

The tetrahedra of the inner parts of the domain come directly from the predefined patterns, so they have a very good quality. Their minimum dihedral angle is 45 degrees (Section 3.3.3). For the tetrahedra near the boundaries there are several aspects that make impossible to ensure a minimum quality for the elements in the final mesh:

- Forced nodes. As the forced nodes are not placed in the octree positions, the tetrahedra pattern generated may yield tetrahedra that are distorted with respect to the ones coming from regular octree nodes. The tetrahedra distortion refinement criterion (RC 7) avoids inverted tetrahedra, which ensures a minimum dihedral angle equal to zero. A solution could be to set a higher minimum dihedral angle, thus ensuring a better quality for the elements. However, considering that the boundaries of the domain may have small dihedral angles, this could lead to infinite refinement in certain regions, because it may be impossible to represent a given shape from a tetrahedron coming from an octree cell.

- Preserving forced edges operations. These operations involve displacement of nodes and split of tetrahedra edges and faces. Parameters α_{vertex} and α_{side} govern the behavior of these operations. Depending on the position of the intersection point of the forced edge and the given tetrahedron face, these parameters establish whether a node must be displaced or a splitting operation must be performed. Theoretically, the tuning of these parameters could bound the quality of the final configuration of tetrahedra after these operations, if the minimum element quality of the initial configuration is known. This is not the case, because of the presence of forced nodes (as explained in the previous paragraph), so a minimum element quality cannot be ensured after the process of preserving the forced edges.

- Surface fitting process. The isostuffing method [LS07] ensures a minimum element quality by tuning two parameters that govern the decision of moving a node or splitting an edge in the surface fitting process. This is possible because the method does not preserve the geometrical features. The mesh where these operations are applied comes directly from a predefined tetrahedra pattern, which quality is known a priori. In our case, the quality of the mesh before the surface fitting process is not bounded, as explained in the previous two points. Hence, the tuning of the parameter α_{iso} tries to get well shaped tetrahedra, but cannot guarantee any minimum quality for the final mesh.

For all these aspects, although the make-up and smoothing operations described in Section 5.3.8 reach an acceptable quality for the meshes generated, a given minimum element quality cannot be guaranteed theoretically.

5.3.10 Extension for surface meshing

As explained in Section 1.2.4, a secondary objective of the present work is to be able to apply the octree mesher to the meshing of surfaces not belonging to any volume. In this section the concept of *inner surface* is used to refer a surface entity not belonging topologically to the boundaries of a volume. It can be inside or outside a volume. From the mesher point of view this difference takes no sense as the outer part of the model is considered as volume zero.

Taking into account the new mesher can provide with the surface mesh corresponding to the boundaries of a volume, it could seem that it can be applied directly to an inner surface. Actually, after the surface fitting process there are tetrahedra at both sides of the inner surfaces that conform to them. However, the faces of the tetrahedra which form the triangle mesh of the inner surface cannot be detected in the same way as the surface entities boundary of a volume (*regural* surface entities).

Extracting the surface mesh of a regular surface entity is automatic after the process of tetrahedra coloring (described in Section 5.3.7): the triangles of the surface entities are the faces of the tetrahedra interfacing two tetrahedra of different color. In the case of inner surfaces, tetrahedra at both sides of the triangles have the same color, so the same strategy cannot be applied. Some modifications have to be performed in order to detect the triangle mesh of an inner surface.

At the time of extracting the surface meshes of the inner surfaces all the tetrahedra have been generated, the processes for preserving geometrical features and surface fitting have been carried out, and the tetrahedra have been colored. At this point, there are already tetrahedra

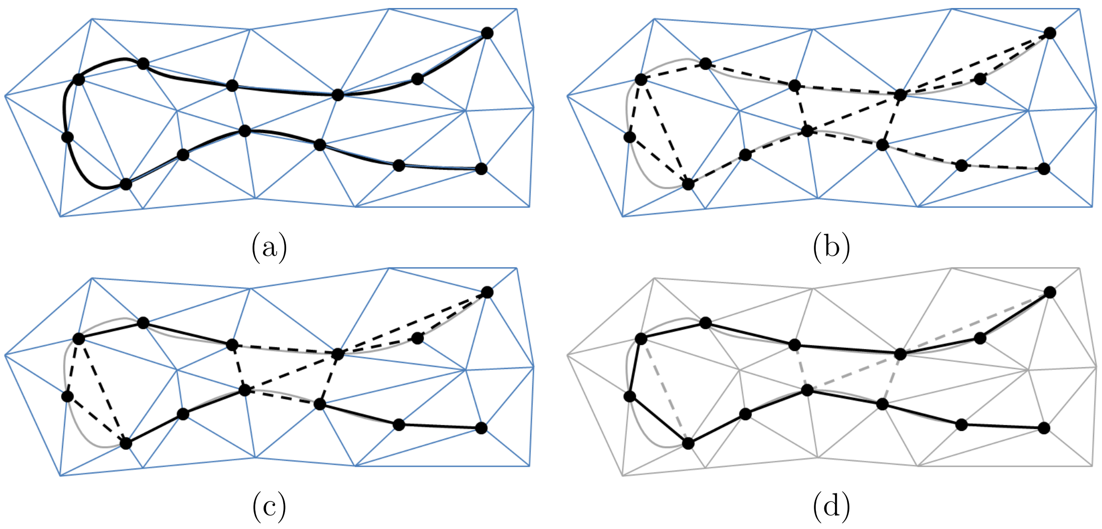


Figure 5.24: 2D example of the steps for detecting the line elements of an inner line. **(a)** Initial configuration: elements at both sizes of the inner line. Thick black line is the inner line and the mesh is in blue. Black dots are forced nodes in interface or in edge. **(b)** All the candidate 1D linear elements in dotted black lines. **(c)** Candidate 1D linear elements (in black) accomplishing the topological properties set to definitive. **(d)** Final mesh (in black) for the inner line once the remaining invalid candidate linear elements have been discarded by Proposition 2.

faces corresponding to the triangles of the inner surface mesh, but they have to be detected. This situation is illustrated in Figure 5.24(a) using a 2D example (an inner line and the surrounding triangles is used instead of an inner surface and the surrounding tetrahedra).

A first consideration must be done: all the triangles in the mesh of a surface entity have their nodes laying on it. This means that the nodes of those triangles are forced nodes in interface or in edge. The tetrahedra faces with this characteristics (all their nodes being forced nodes in interface or in edge) are called *candidate triangles*. In Figure 5.24(b) the candidate linear elements (in 2D there are candidate 1D linear elements instead of candidate triangles) of the configuration shown in Figure 5.24(a) are depicted in dotted lines.

Considering all the candidate triangles of an inner surface, some topological properties analogous to the ones made in the tetrahedra coloring strategy can be done. If a node is inside a surface entity, the surface mesh around it must be manifold (let us call it a *manifold node*). This implies that there should be a closed loop of triangles surrounding it in the mesh of the surface entity. If there are a closed loop of candidate triangles around a manifold node, and they are the only candidate triangles containing the node, they can be set as valid (triangles belonging to the inner surface final mesh). Also the requirement for every forced node in interface or in edge to belong at least to one triangle of the final mesh can be used: if one of these nodes belongs to one candidate triangle only, this one must be valid. For the case shown in Figure 5.24(a), the candidate elements set as valid considering these topological properties are depicted in Figure 5.24(c).

However, this topological properties may not solve the problem of detecting all the triangles of an inner surface mesh, as there are nodes which are not needed to be manifold (the ones in edges), and there may be triangle configurations around manifold nodes which are not manifold. This leads to the need for another strategy to detect the right triangles among all the candidate ones. The strategy proposed is based on disregarding the wrong candidate triangles, rather than detecting the right ones, using Proposition 2. It is based on Proposition 1 (defined in section 5.3.7 for tetrahedra coloring), but with slight modifications:

Proposition 2. Consider a candidate triangle T of an inner surface S and the two tetrahedra around it R_1 and R_2. Being P_1 and P_2 the nodes of R_1 and R_2 not belonging to T, if there is a point P inside triangle T and a continuous curve in space from P_1 to P_2 passing through P completely inside the union of R_1 and R_2 with no intersection with S, the candidate triangle is invalid.

A graphical view of this proposition(using a 2D case) is shown in Figure 5.25.

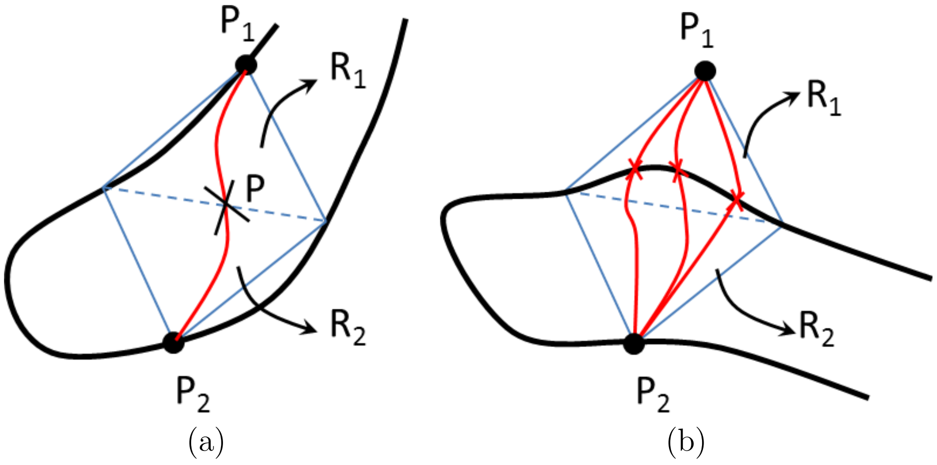


Figure 5.25: Graphical interpretation of Proposition 2 in a 2D example. Black line represents the inner line (S). Triangle mesh is represented by blue lines and the candidate linear element to be checked (T) is the dotted line. **(a)** The candidate linear element is invalid, because there is a curve (in red) not intersecting the inner line. **(b)** The candidate 1D linear element cannot be set as invalid because all the possible curves intersect the inner line.

In Figure 5.25(a) the candidate 1D linear element is set as invalid because there is a curve (drawn in red) accomplishing Proposition 2. Figure 5.25(b) shows that there could not be any curve accomplishing the proposition, so the candidate linear element cannot be set as invalid.

Once all the invalid candidate triangles are disregarded using Proposition 2, the rest of the candidate triangles are the ones corresponding to the final mesh of the inner surface (Figure 1.10(d)).

Note that this proposition can set as invalid some correct triangle in some cases, specially when the the surface entities are curved. Some improvements should be applied to the algorithm to be more robust in some pathological configurations.

The implementation of the algorithm defined for extracting the mesh of an inner surface is detailed in Section 6.5.4.

Chapter 6

Implementation aspects

This chapter details all the implementation aspects relevant for the meshing algorithm described in the previous chapters. The specific implementation of the ray casting technique used for the nodes coloring is explained in Section 4.3, as it can be considered as an independent process.

It is important to note that implementation matters do not affect the result of an algorithm, but they affect its performance and efficiency. In this sense, the implementation of a meshing algorithm is crucial if it has to be applied to industrial problems with complex geometries, or to generate really large meshes. A bad implementation can lead to unaffordable problems in terms of memory and computational time.

The algorithm has been implemented as a static library. Although the implementation of the mesher is done taking into account the GiD pre and post-processing system [CRP$^+$10a, CRP$^+$10b, CRP$^+$10c] for getting the input data and the visualization of the meshes, its connection to GiD has been done as an external library, by a general interface specially designed to make it accessible for other programs. It is really important to provide access to the mesher as a library, as some numerical simulations require interaction with the mesher during the simulation itself (i.e. for optimization loops or in remeshing processes).

6.1 General aspects

The implementation of the new algorithm has been carried out paying special attention to saving as much memory as possible, and trying to improve its performance taking into account shared memory parallel processing (Section 6.6).

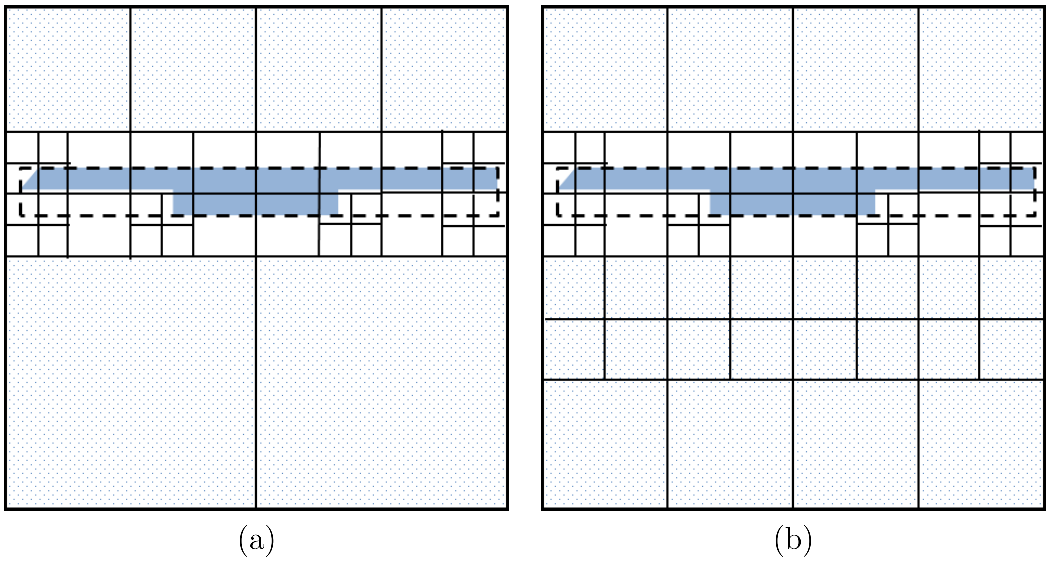


Figure 6.1: 2D example of a thin model (the solid surface) with its bounding box (dotted line) and the quadtree (black lines). Cells out of the bounding box are marked with points. **(a)** Quadtree refined considering only the cells colliding the model bounding box. It can be appreciated that the quadtree is not balanced out of it. **(b)** Quadree refined considering all its cells.

One important characteristic common to many meshing algorithms is that they need an extra memory to generate the mesh compared to the memory needed to store the mesh once it has been generated. This leads to the need of memory saving strategies in the implementation of the mesher:

- The main part of operations involving octree cells are applied only to inner and interface cells (not to outer ones), so they are the ones from which the tetrahedra are generated. These operations include cells refinement and creation of octree nodes. It has to be considered that in the first stages of the meshing algorithm it is not obvious to know if a cell is outer or not, as the coloring operations are not done yet. For this reason, in these stages a simple check is made in order to detect some of the outer cells in an easy way: if the cell is outside the bounding box of the model, it is taken as an outer cell. Checking whether a cell is inside or outside a bounding box is a really inexpensive process.

  Depending on the aspect ratio of the bounding box of the model, this strategy can save a lot of memory. This is the case of very thin models in some direction (like the 2D example depicted in Figure 6.1) where, actually, the main part of the octree root is out of the bounding box of the model. Not considering the refinement criteria of the octree out of the model bounding box (like, for instance, the balance criterion) can reduce drastically the number of cells of it, as it can be seen in Figure 6.1. It can be appreciated that the quadtree refined considering all the cells (b) has much more cells than considering only the cells colliding the model bounding box (a).

- Tetrahedra from the inner part of the volumes are generated at the very last step of the algorithm. Tetrahedra from the inner leaves have really good quality, they are not affected by the operations involved in the body-fitting process (preserve geometric features and surface fitting) and their color is totally determined (all its nodes have the same color). For this reason, these tetrahedra can be generated in the last step of the algorithm without affecting the resulting mesh. This part is implemented as follows:
  - Once the octree is refined following all the refinement criteria, the interface cells and their neighbors (one level of neighborhood) are considered. Then, their tetrahedra are generated.
  - The following operations are performed on these tetrahedra: preserve geometrical features, surface fitting process, tetrahedra coloring and make-up and smoothing operations. These operations require typically an extra topological information of the mesh (like the neighbor elements of a node). Once these processes are finished, this extra information is deleted in order to save memory.
  - Then, as a last step, the tetrahedra generated from the inner cells are created. In this step no additional information than the mesh itself must be stored.

  The flowchart of the body-fitted algorithm illustrating this process can be seen in Figure 6.7.

Considering the input data, the meshing algorithm is designed in a way that the input boundaries can be mesh or geometrical entities. The only operations required for them to be used in the algorithm are:

- Get the bounding box of the entity.
- Compute the intersection between an edge and the entity.
- Get the closest point of the entity to a given position of the space.

The algorithm has been implemented considering only mesh entities for the definition of the input boundaries. This simplify much the geometrical operations defined above.

6.2 Octree implementation

The octree is the key structure of the presented algorithm. As mentioned in previous sections, it is not only used for generating the tetrahedra, but also for searching purposes. Actually, octrees were created originally for this topic, so it is natural to take advantage from on its characteristics in this field.

The main operation the octree is designed for is to find the leaf containing a coordinate in space. For this purpose, beginning from the octree root, the given coordinate is analyzed to evaluate which of its children contains it (considering the uniform subdivision of a cell, determining in which octant of a cell the coordinate is represents an inexpensive process). If the child is not an octree leaf, the process is repeated (with itself instead of the octree root) until an octree leaf is reached. This process can be seen as *going downstream* by the octree structure, and often implies the use of recursive functions (the same function applied to a cell is applied to its child). The use of recursive functions is not desirable because it decreases the performance of the algorithm due to the function call overhead.

In order to achieve a good performance, a very efficient implementation of the octree has been carried out based on [SF02] work. This octree has the peculiarity that works in the normalized unitary space $[0, 1]$x$[0, 1]$x$[0, 1]$. This is required because it uses the binary representation of all the coordinates involved in the process casted to integer: the so called *key* of the coordinate.

From a given coordinate in the normalized space, the following steps are performed in order to find the octree leaf containing it:

- Get the key of the coordinate: it is the representation of the coordinates in binary form (i.e. for coordinate $x = 0.623$ the binary representation will be 1010000).
- For each level i of the octree (beginning from the octree root), the ith bit of the key representation is used for selecting the left or right child for each coordinate (0 for left and 1 for right child). In each level, having the combination of three coordinates bits provides the correct decision of the child.

The identification of a cell is done by its lower coordinate using the corresponding key; it receives the name of the *location code* of the cell. A graphical view of a 1D binary tree is depicted in Figure 6.2. Each one of the nodes represents a cell, and its key is depicted. For the leaves (the cells in the last level) the coordinate in the unitary space is also shown (framed). Checking the keys of the different cells it can be appreciated that traveling downstream the

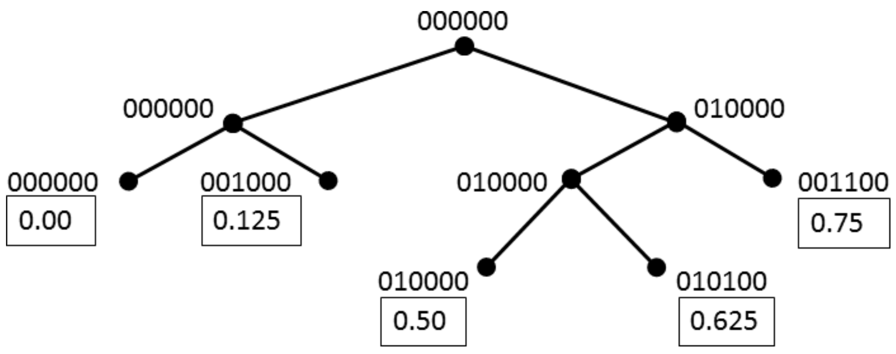


Figure 6.2: Binary tree representation of a 1D spatial partitioning over [0, 1] indicating the cell's location code (its key) and its corresponding coordinate (inside frames) of the leaves cells.

tree is really easy looking at the bit corresponding to the level of the cell: if the bit is 0 take the left branch, and if it is 1, take the right one.

A detailed description of the algorithm can be found in [SF02] including its 2D implementation, as well as how other basic operations in octree are done (like traversing from a cell to another).

This implementation of the octree forces to work in a normalized space (between 0 and 1 in each dimension). As it will be seen in Section 6.5, some geometrical transformations are performed on the input boundaries and the sizes information from the input data in order to fulfill this requirement.

One important advantage of using the keys instead of the coordinates, is that they are integer values while the coordinates are real ones. Some geometrical operations profit from this, like the detection of nodes aligned with the Cartesian directions. If two nodes have the same coordinate (in some direction), its real value can be slightly different because of the storing procedures of the computer. However, their key, as an integer, is the exact same number. This aspect is used in the ray casting implementation and other parts of the algorithm.

The main characteristics of the implemented octree are:

- The octree is represented in a hierarchical tree structure composed of a root cell, intermediate cells and leaf cells.
- Each cell stores pointers to each child cells, but not to its father cell. This reduces the memory used by the octree.
- The operation of traveling downstream of the octree is not recursive. As it have been

seen, knowing the key of the coordinate to be analyzed is enough to take the corresponding branching decisions in a global way.

- The maximum number of octree levels is 30. This is the direct result of storing the key as a 32 bit integer. Due to some implementation issues the minimum level must be 2, so in practice the octree can have 30 levels. This implies that the smallest octree cell can be $\frac{1}{2^{30}} \approx 10^{-9}$ of the bounding box.

The memory usage can be improved by keeping only the leaves and making a hash function to search leaves as the work of [SSA+07]. Also the performance of the octree can be improved by precomputing the bit shifting in each level and the traversing algorithm can be improved by creating a parents trace from root to leaves in time of traversing as proposed in [KWPH06]. However, the current implementation is good enough for the meshing algorithm developed in this work, as it is not a bottle-neck in terms of speed nor memory.

In the new meshing algorithm, the main role of the octree is to create the tetrahedra from given patterns. For this purpose each octree leaf has its octree nodes stored. Furthermore, the octree is also used for searching purposes.

Specifically, the octree is used to find in an efficient way the input surface entities near a given node, or near an edge of the tetrahedra mesh. For this purpose, the input surface entities must be stored somehow in the octree. A surface entity can collide one or more leaves. The strategy followed in this work is to store a pointer of each surface entity inside each leaf colliding with it. This option gives a very good performance in the octree operations. The memory consumed for this purpose depends on the ratio between the size of the surface entities defining the domain and the octree leaves sizes. In the examples studied this memory is not relevant, considering the whole process.

6.3 Generalized mesh size points

The generalized mesh size points are a collection of points placed on each mesh size entity in such a way that, being s its desired mesh size, there is no point inside the entity further than s from a generalized mesh size point (see page 78). The idea is to replace the mesh size entities by their generalized mesh size points in order to simplify some of the algorithms involved in the mesher, such as the size refinement criteria.

There are infinite configurations of generalized mesh size points of an entity accomplishing the condition described above. For improving the performance of the global algorithm it is necessary to find a simple way (computationally inexpensive) of obtaining a configuration of them trying to minimize its number.

The distribution of sizes of the leaves of the octree is not continuous (a leaf can have the same, the double or half of the size of its neighbor). However, the desired mesh sizes imposed by the mesh size entities can take any real positive value. To take into account this, the parameter α_{msp} is used to scale the desired size of a mesh size entity when creating its generalized mesh size points. The scaled size s_α of a mesh size entity is defined as:

$$s_\alpha = s \cdot \alpha_{msp} \tag{6.1}$$

where s is the desired mesh size of the mesh size entity and α_{msp} is a real value greater than one. The value used in the presented implementation is detailed in Section 6.7.

There are four kinds of mesh size entities: point, line, surface and volume entities. The presented implementation has been done considering the following mesh element types as mesh size entities: node, linear, triangle and tetrahedron. To simplify the notation, the concepts of *mesh size node*, *mesh size edge*, *mesh size triangle* and *mesh size tetrahedron* is used to refer to the corresponding entities.

All the generalized mesh size points created from a mesh size entity E have a desired mesh size s. As E can be shared by other higher level mesh size entities (for example, an edge can be shared by some triangles and tetrahedra), s takes the lower value between the desired mesh size of the mesh size entities E belongs to and the one from E itself.

Hereafter, the methodology used for the creation of the generalized mesh size points from each type of entity is presented:

- *Creation from a mesh size node.* This is the simplest case. From a mesh size node a generalized mesh size point is created in its position with a desired mesh size s.

- *Creation from a mesh size edge.* The extreme nodes of a mesh size edge are considered as mesh size nodes (with a desired mesh size s).

  If the length L of the mesh size edge is higher than s_α, the edge is subdivided uniformly in n divisions, with n defined as:

  $$n = (int)\frac{L}{s_\alpha} >= 1 \tag{6.2}$$

  where (int) represents the truncated value to the closest integer.

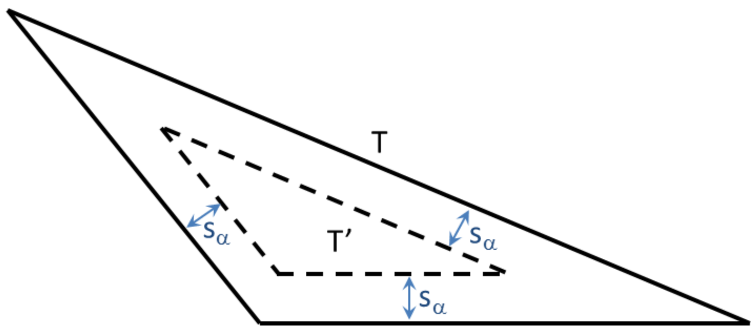


Figure 6.3: Auxiliar triangle $T^{'}$ (dotted line) of mesh size triangle T (solid line) used for the generation of its inner generalized mesh size points.

From each position inner to the edge resulting from the edge subdivision a generalized mesh size point is created with a desired mesh size s.

- *Creation from a mesh size triangle.* The three edges of the triangle are considered as mesh size edges (with a desired mesh size s).

  Generalized mesh size points from the inner part of the triangle are created only if the three heights of the triangle are higher than $2s$. In this case, an auxiliary triangle $T^{'}$ is created from the mesh size triangle T. $T^{'}$ is similar to T (their sides are parallel), it is smaller than T and the distance between their corresponding sides is equal to s_α (see Figure 6.3).

  The edges of $T^{'}$ are then considered. If they all have a length lower than s_α, a generalized mesh size point is created in the center of $T^{'}$. Otherwise this strategy is followed:

  – If the length of an edge is higher than s_α, it is considered as a mesh size edge (with a desired mesh size s).

  – If the length of an edge is lower than s_α, a generalized mesh size point is created in the center of the edge.

  – Then the process of creating the auxiliary triangle is repeated from $T^{'}$ recursively.

- *Creation from a mesh size tetrahedron.* The four faces of the tetrahedron are considered as mesh size triangles (with a desired mesh size s).

  If the four heights of the tetrahedron are higher than $2s$, an analogous procedure as for the triangle is carried out (in this case creating an auxiliary tetrahedron) in order to create the inner generalized mesh size points. This process can create extra mesh size triangles, edges and nodes.

6.4 Sizes transition function

This section focuses on the definition of the $S_u(\vec{x})$ function, used in the sizes transition criterion. This function provides the upper limit for the mesh size in each position $\vec{x}$ of the domain.

The definition of S_u leans on two main inputs, both from the input data (see section 3.1):

- The mesh size entities. These entities indicate the desired mesh size for the simulation in given regions of the domain.
- The *size transition factor* (from now on TF). TF is a positive value between zero and one, and controls if the size variation between regions of the domain of different mesh size is smoother or sharper.

From a given position in space $\vec{p}$, and a desired mesh size s_p defined on it, two functions can be defined ($S_{up}^{\vec{p},s_p}$ and $S_{low}^{\vec{p},s_p}$) in order to bound between an upper and lower limit the mesh size in a specific position of space $\vec{x}$: the *size transition functions*. Different functions can be used for this purpose in order to fulfill the simulation requirements (in terms of element size distribution). In the present work the following linear functions have been chosen:

$$S_{up}^{\vec{p},s_p}(\vec{x}) = s_p + r \cdot d(\vec{p},\vec{x}) <= S_{max} \tag{6.3}$$

$$S_{low}^{\vec{p},s_p}(\vec{x}) = s_p - r \cdot d(\vec{p},\vec{x}) >= S_{min} \tag{6.4}$$

where $d(\vec{p},\vec{x})$ is the distance between $\vec{p}$ and $\vec{x}$ and r takes the following expression:

$$r = \alpha + (1-\alpha) \cdot TF. \tag{6.5}$$

S_{max} is defined in Section 5.2.2. S_{min} is the maximum value between the minimum desired size coming from the mesh size entities and the size of the smallest leaf in the octree (so the evaluation of this function depends on the configuration of the octree). α is a weight parameter (Equation 6.9) and its interpretation is given hereafter.

A graph of the size transition functions with different values for the parameter TF is depicted in Figure 6.4. As it can be seen, for TF equal to one, $S_{up}^{\vec{p},s_p}$ is a straight line with a slope equal to one. This corresponds to the maximum variation of mesh size allowed in the octree, which is given by the constrained two to one condition.

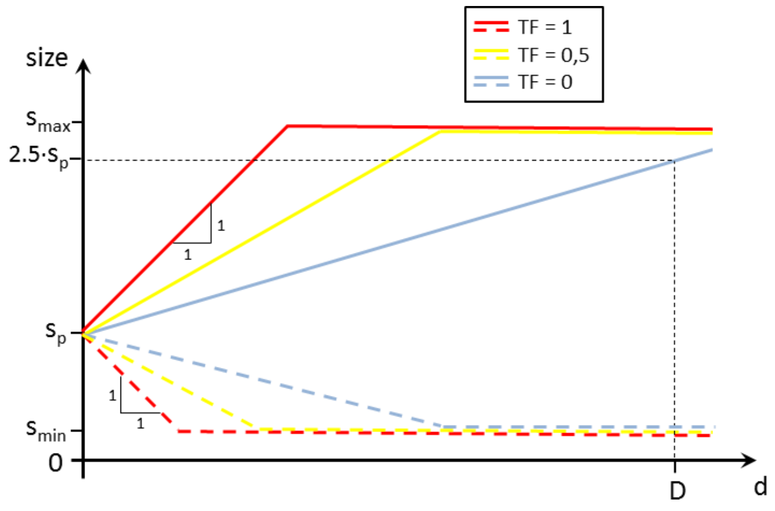


Figure 6.4: Graph of the size transition functions 6.3 (solid line) and 6.4 (dotted line) with different parameters of TF. The functions are plotted as a function of d: the distance between $\vec{p}$ and $\vec{x}$.

The α parameter should be tuned in order to set the lower slope of the function (when TF is equal to 0). In order to do so, the following aspects must be considered:

- The distance between two points of the domain is always lower than the diagonal of its bounding box D.
- The minimum size transition should allow, at least, to double the mesh size between the furthest points of the domain. This means the sizes of these points differs by a factor f equal or higher than two. A factor f lower than this could lead to uniform size meshes (without size transition).

These parameters are applied to Equation 6.3 in order to get the expression of α. Taking into account above conditions this expression can be written now as:

$$f \cdot s_p = s_p + r \cdot D \tag{6.6}$$

Replacing r using Equation 6.5, and considering we are in the case of $TF = 0$ we get:

$$f \cdot s_p = s_p + \alpha \cdot D \tag{6.7}$$

The expression of α can be found from Equation 6.7 as:

$$\alpha = \frac{(f-1)\cdot s_p}{D} \tag{6.8}$$

A factor $f = 2.5$ has been chosen in this work. It can be appreciated in Figure 6.4 (the function corresponding to $TF = 0$) that the mesh size at a distance D is 2.5 times the one at the origin. This leads to the final expression of α as:

$$\alpha = \frac{1.5\cdot s_p}{D} \tag{6.9}$$

The selection of these functions has been done mainly for its simplicity. One could think about more sophisticated ones, but it has to be considered that these functions are used for octree refinement purposes. Once an octree cell is refined, its child cells have half of its size. This limits the control of the local mesh size variation and governs the sizes transitions at a more global scale of the domain.

Considering the generalized mesh size points from all the mesh size entities, functions $S_{up}^{msp_i}$ and $S_{low}^{msp_i}$ can be defined from Equations 6.3 and 6.4 when $\vec{p}$ is the position of the generalized mesh size point i, and s_p its desired mesh size. Being n_{msp} the number of mesh size points, function S_{up}^{msp} is defined as:

$$S_{up}^{msp}(\vec{x}) = \min S_{up}^{msp_i}(\vec{x}) \qquad , i = 0, n_{msp} \tag{6.10}$$

Analogously, function S_{min}^{msp} gives a minimum value for the mesh size in all the points of the domain:

$$S_{low}^{msp}(\vec{x}) = \max S_{low}^{msp_i}(\vec{x}) \qquad , i = 0, n_{msp} \tag{6.11}$$

The concept of generalized mesh size points has been introduced in Section 5.2.2. As explained before, the presented algorithm works with them instead of the mesh size entities. For this reason, only for notation purposes, from now on *mesh size points* will be used to refer all the *generalized mesh size points*.

One option for providing an envelope for the upper limit in the element size covering all the domain could be to use directly the equation 6.10 as $S_u(\vec{x})$. This lead to a mesh size distribution governed by the lower mesh sizes. A graphical 1D example of the $S_u(\vec{x})$ function following this approach is shown in Figure 6.5. Dotted lines are the $S_{up}^{msp_i}$ of each mesh size point, and $S_u(\vec{x})$ is the red solid polygonal line (the envelope of $S_{up}^{msp_i}$). It can be seen that the mesh size in the extremes of the line (s_{2real}) is lower than the desired one (s_2) because

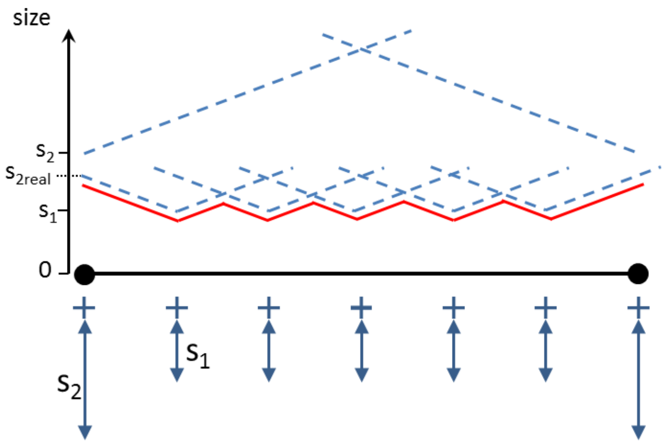


Figure 6.5: Graphical view of the $S_u(\vec{x})$ function for a 1D example (red line). Black horizontal line represents the domain (a line entity) to be meshed with a size equal to s_1. The crosses are the generalized mesh size points. Extreme points of the line have a mesh desired size equal to s_2, higher than s_1. Dotted lines represent the $S_{up}^{msp_i}$ function of each generalized mesh size point.

there is some mesh size point with lower size (s_1) assigned near them.

In general, it is common to let the small mesh size regions dominate the mesh size distribution all over the domain. However, for many numerical simulations, the mesh size on the contours of a geometrical entity has a special interest. A strategy has been followed in order to try to preserve the mesh sizes on this areas (the contours of the entities). It consists in applying a correction to the desired size assigned to the mesh size points before evaluating Equation 6.10. This correction is done in the following way:

- Classify the mesh size points following the nature of the mesh size entity they come from: point, line, surface or volume entity. This yields four collections of mesh size points (one for each entity type).
- For each group of mesh size points from the lower to the higher level (from points to volumes), execute the following actions:
  - Sort the mesh size points of the group according to its desired size (from smaller to larger size).
  - For each mesh size point ms_i (from small to large size):
    - Evaluate functions $S_{up}^{msp_i}$ and $S_{low}^{msp_i}$ of ms_i in the positions of the all the other mesh size points. This gives an upper (s_u) and lower (s_l) size limit for each mesh size point M of the model.

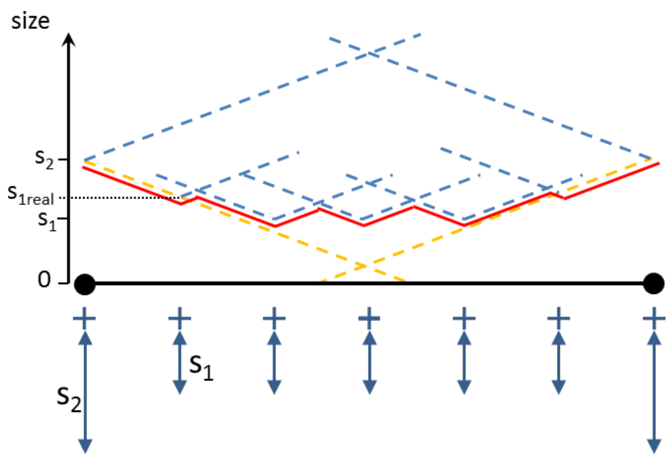


Figure 6.6: Graphical view of $S_u(\vec{x})$ function for the 1D example of in Figure 6.5, after applying the size correction in the mesh size points. Dotted yellow lines represent the $S_{low}^{msp_i}$ function of the mesh size points corresponding to the extremes of the lines. $S_{low}^{msp_i}$ functions for inner mesh size points are not plotted to make clear the visualization (they are not relevant for the example).

∗ For each mesh size point M, check whether its size (s) is inside the range defined by the limits s_l and s_u.

· If s is higher than s_u, then the size of M is set to s_u.

· If s is lower than s_l, then the size of M is set to s_l.

The graphical view of $S_u(\vec{x})$ for the 1D example shown in Figure 6.5 after the correction of sizes for the mesh size points is shown in Figure 6.6. It can be appreciated that the desired mesh size in the contours of the line entity to be meshed (its extreme points) is not modified (s_2), but some of the inner mesh size points of the line (the closer to the extremes) have a maximum size higher than the one desired (s_{1real}).

It has to be considered that the evaluation of $S_{up}^{msp_i}$ and $S_{low}^{msp_i}$ at a given mesh size point only has to be effectively done in the mesh size points around it, as both functions are truncated by S_{max} and S_{min}. These values define a radius of influence for each function.

The way this algorithm is implemented takes advantage from the octree structure itself: instead of evaluating the functions from a given mesh size point in all the others, the upper and lower limits provided by the functions are stored in the octree cells surrounding of it (near the radius of influence). Then, to check the size limits of a mesh size point, the values of the octree cell containing it are considered.

6.5 Body-fitted mesher

The main steps of the meshing algorithm for body-fitted meshes are explained in Section 5.3. This section is focused in its implementation aspects. The implementation of the embedded mesher is not detailed because it basically involve the algorithm detailed in Section 5.2 with no more particularities.

A general flowchart of the meshing algorithm implementation is depicted in Figure 6.7. Its processes involved are listed hereafter (the ones requiring a more detailed explanation are detailed in further sections):

- *Process input data.* This step basically consists in getting all the information from the input data: input boundaries, forced entities (points and lines) and general parameters. In this step some new forced entities can be generated in case the input data define some tolerance to detect automatically ridges or corners.

- *Geometrical transformations to the input data and creation of octree root.* Before the creation of the octree root, some geometrical transformations are applied to the input data (the input boundaries and some of the input parameters) in order to fit it to the implemented octree characteristics, and to improve efficiency in the algorithm. These geometrical transformations are detailed in Section 6.5.1.

- *Octree refinement and nodes coloring.* This process receives the octree root with all the input data already processed, and ends up with the octree in its final configuration: its leaves will not be further subdivided. The implementation of the octree refinement operations is complex, as it require an iterative process. It is detailed in Section 6.5.2. The coloring of the nodes is done inside this process, so after the octree refinement all the nodes are colored.

- *Generate tetrahedra from the interface cells.* At this point only the tetrahedra from the interface leaf cells and their first level neighbor leaves are generated following the given patterns (defined in Section 3.3.3). The tetrahedra from the inner part of the volumes have all their nodes in the octree positions (they have no forced nodes), so they have good quality (no further smoothing is needed). Furthermore they are not involved in the surface fitting or the preserving features processes, so there is no need to edit them. The possibility not to generate the inner tetrahedra allows an important memory saving, taking into account that in most models the number of tetrahedra of the inner parts

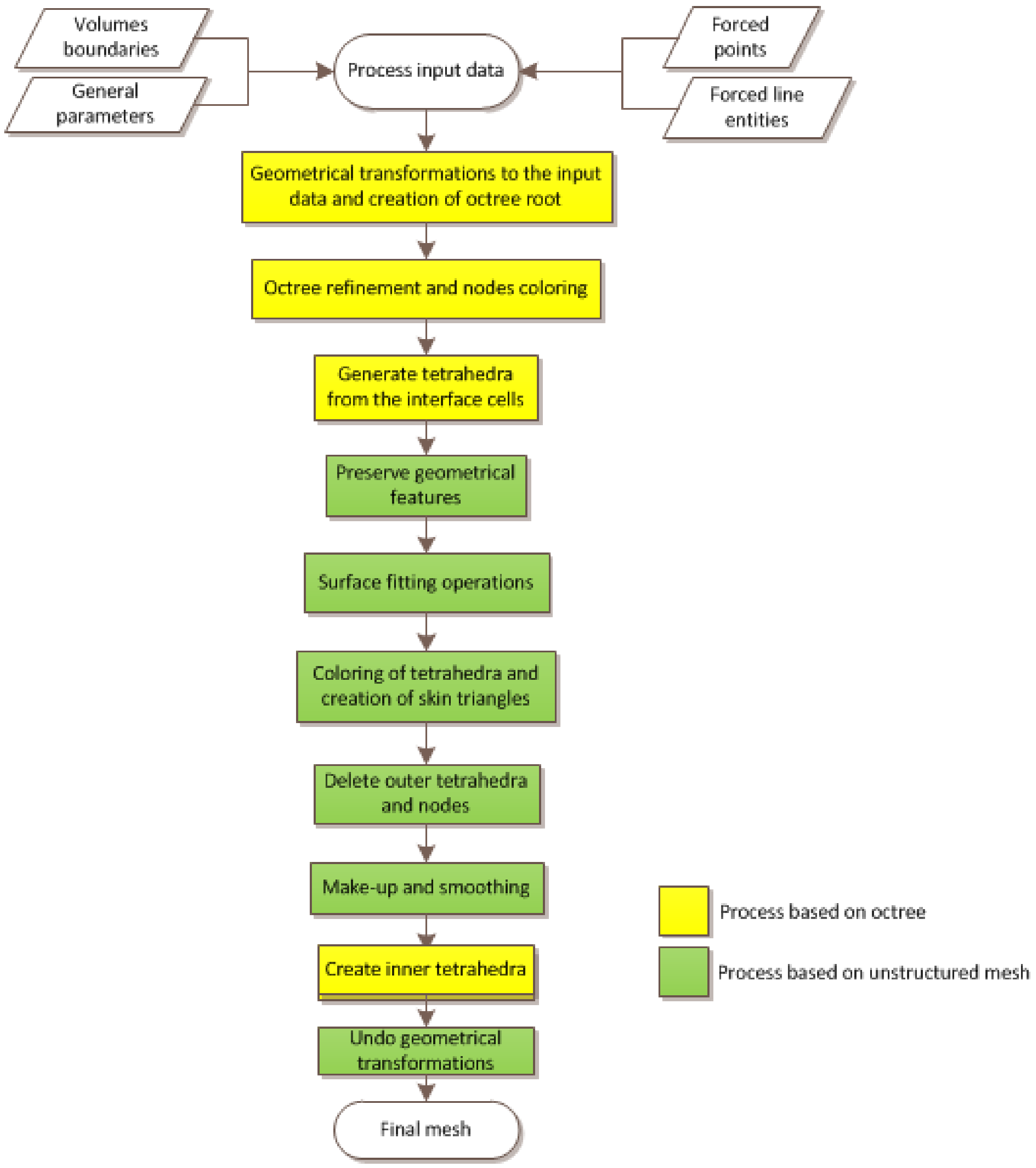

Volumes boundaries
General parameters
Process input data
Forced points
Forced line entities
Geometrical transformations to the input data and creation of octree root
Octree refinement and nodes coloring
Generate tetrahedra from the interface cells
Preserve geometrical features
Surface fitting operations
Coloring of tetrahedra and creation of skin triangles
Delete outer tetrahedra and nodes
Make-up and smoothing
Create inner tetrahedra
Undo geometrical transformations
Final mesh
Process based on octree
Process based on unstructured mesh

Figure 6.7: Flowchart of the body-fitted meshing algorithm implementation.

of the volume is much larger than the number of tetrahedra intersecting the volume contours.

This memory saving allows to store extra information in the nodes and tetrahedra (only the ones coming from interface cells) in order to improve the performance of the algorithms. This extra information is mainly related to the neighborhood of nodes and elements.

- *Preserve geometrical features.* This step involves the operations described in Section 5.3.5, and ends up with all the forced edges and forced nodes belonging to the some tetrahedron of the mesh.

- *Apply surface fitting operations.* This step involves the operations described in Section 5.3.6.

- *Coloring of tetrahedra and create skin of triangles.* Tetrahedra are colored (the volume they belong to is determined) following the implementation described in Section 6.5.3. Then, the skin of triangles of each volume of the domain is built with the tetrahedra faces interfacing tetrahedra of different colors.

- *Delete outer tetrahedra and nodes.* At this point the nodes and tetrahedra owning to the outer part of the domain (color equal to zero) can be deleted.

- *Make-up and smoothing.* As part of the tetrahedra coming from the inner leaf cells are not generated yet, in the make-up process the nodes belonging to one of these leaf cells must be preserved: they cannot be deleted in the edge collapsing process. As the more distorted elements come from the interface leaf cells, this limitation in the make-up process does not affect the final mesh quality improvement.

- *Create inner tetrahedra.* In this step the tetrahedra from the inner cells are generated following the given patterns (defined in Section 3.3.3). It has to be considered that these tetrahedra are always determined in terms of color, as all their nodes belong to the same volume.

- *Undo geometrical transformations.* The inverse of the geometrical transformations applied to the geometrical input data must be applied to the final mesh generated, in order to fit it in the original position of the domain.

6.5.1 Geometrical transformations to the input data

As explained in section 6.2, the octree used is a unitary octree: the octree root goes from the values 0.0 to 1.0 for the coordinate in each dimension. This enforces all the geometrical input data (input boundaries, forced entities and mesh size entities) to be transformed in order to avoid any coordinate outside the range $[0, 1]$. Because of the octree implementation chosen and efficiency matters, before the octree root creation three main geometrical transformations are applied to the geometrical input data:

- *Rotation.* As explained in Section 3.3, the octree structure chosen for the mesher is isotropic. Considering a domain and its possible solid rigid motion, the way it fits inside the octree is different in the sense that different configurations of cells are intersected by the model. In general, when working with octrees, the more the model shape is aligned to the Cartesian axes the more efficient is the octree in terms of memory and performance. For this reason, the geometrical input data is rotated aligning the main inertia axes of the external skin of the model with the Cartesian axes. It is important to note that the meshing algorithm is valid without rotating the model (this is done for efficiency purposes only).
- *Translation.* Once the geometrical input data is rotated, a translation is applied to it in order to place the center of $Bbox_m^+$ in the position $(0.5, 0.5, 0.5)$ which is the center of the octree root.
- *Scale.* As explained in Section 5.2.3, the octree root is created from $Bbox_m^+$ and S_{max}. Considering the octree root must have a unit length, the gometrical input data is scaled to accomplish this condition. All the mesh sizes information in the input data (the general mesh size, or the size assigned to mesh size entities) are also scaled with the same factor.

A graphical 2D example of these three transformations (with a quadtree instead of an octree) is shown in Figure 6.8.

A secondary advantage of applying these transformations to the model before the meshing process is that the final mesh is more invariant with solid rigid movements of the input data. This is not a crucial aspect, but it is always desirable that the mesh for a given domain remains topologically identical if the domain is transformed following a rigid solid movement.

In cases where the desired mesh size is uniform in all the domain, an extra transformation can be applied to the octree root after the previous ones. It consists in increasing the size

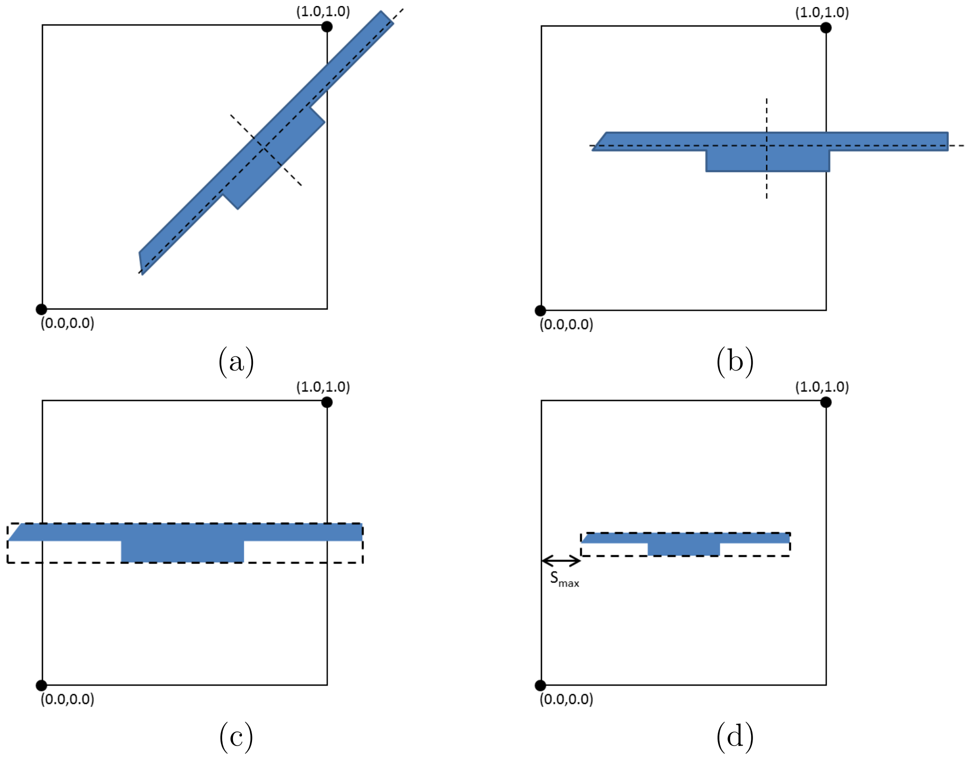


Figure 6.8: Geometrical transformations applied to a 2D model before quadtree root creation. Solid surface is the model and the black line the quadtree root. **(a)** Original configuration of the model and the quadtree root (inertia axes drawn with dotted lines). **(b)** Model rotated aligning its main inertia axes with the Cartesian ones. **(c)** Model translated to the center of the quadtree root (bounding box of the model with dotted line). **(d)** Model scaled leaving an offset of S_{max} between its bounding box and the quadtree root.

of the octree root until it fits with a power of two of the desired uniform size. Doing so, it is guaranteed that the final mesh will have elements of a size very close to the desired one. Furthermore, there is more control on the number of elements in the final mesh. If this transformation is not done, assigning slightly different sizes to the same model would yield the same mesh.

As an example, generating the mesh of the same model with different uniform sizes (from large to small size), the generated mesh would be identical until the desired size is small enough to force an octree refinement. In this case, the number of elements in the final mesh will be approximately eight times the one of the previous mesh, as refining an octree cell creates eight cells more.

6.5.2 Octree refinement

As pointed out in Section 5.3, the refinement of the octree is an iterative process, because the subdivision of an octree cell may lead to the creation of new forced nodes, which may affect the configuration of the octree itself. The flowchart of the octree refinement process is shown in Figure 6.9.

In this flowchart it can be appreciated that, in the first steps of the refinement process (specially the ones related with the mesh sizes criteria), the octree cells are not classified yet as interface or inner, so these steps do not require to consider the input boundaries.

The last step in the octree refinement process is the so called *Delete data on outer octree cells*. From this point of the meshing algorithm on, the cells of the octree will not be subdivided anymore and only the inner and interface cells are involved in the meshing process, so all the information of the outer cells is deleted in order to save memory. More than this, in the cases where the eight sons of a cell are outer cells, they are deleted, and their father becomes a leaf. This derefinement process may make the octree violate some of the refinement criteria (in particular, the octree may be unbalanced). As the only leaf cells involved in the meshing process are the interface and the inner ones, only these ones must fulfill the refinement criteria.

Concerning the topological refinement criteria (RC8), only the criterion a) has been implemented, which is the one involving tetrahedra edges. The other two criteria (focused on surface and volumetric entities) involve more complex geometric operations which may lead to a more computational expensive algorithm. In order to take them somehow into account, a cheaper refinement criterion has been implemented: it consists in refining a cell if it is an interface cell with no forced node and no edge from the tetrahedra generated from the cell intersects the input boundaries. This criterion is not as restrictive as the whole RC8 one, but it is considerably less computationally expensive. The examples run show that it is fulfilled for most of the cases, specially if a reasonable desired mesh size is provided in the input data. Only some special configurations of boundaries would need a full implementation of the complete RC8 criteria.

6.5.3 Tetrahedra coloring

The tetrahedra coloring algorithm is described in Section 5.3.7, where the Proposition 1 gives a theoretical solution for some configurations of tetrahedra to be colored. However, it only determines the color of the tetrahedron if the continuous curve (with no intersections with the boundaries) between P and N exists. There are infinite continuous curves in space going

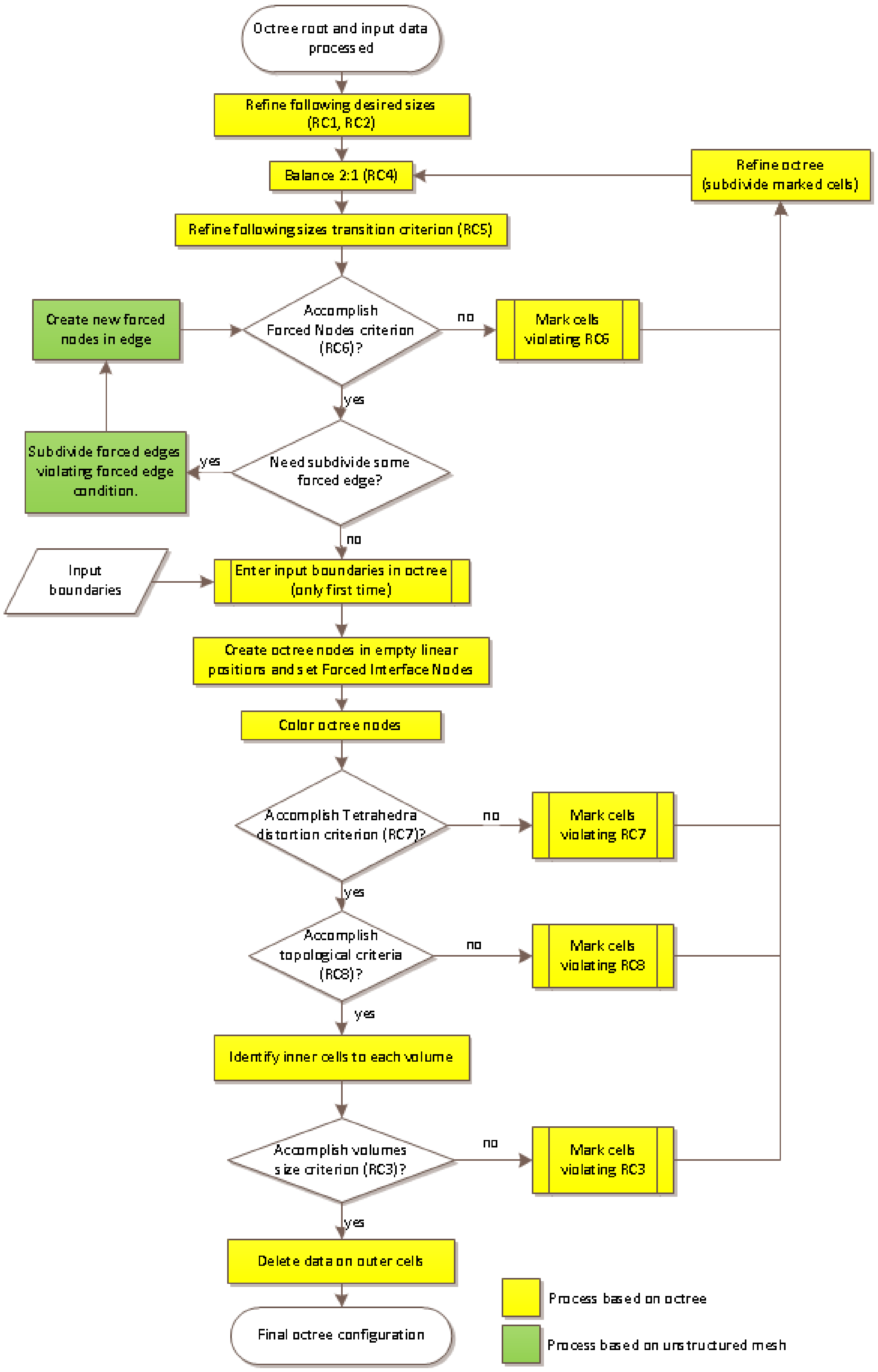

Octree root and input data processed
Refine following desired sizes (RC1, RC2)
Balance 2:1 (RC4)
Refine octree (subdivide marked cells)
Refine following sizes transition criterion (RC5)
Create new forced nodes in edge
Accomplish Forced Nodes criterion (RC6)?
no
Mark cells violating RC6
yes
Subdivide forced edges violating forced edge condition.
yes
Need subdivide some forced edge?
no
Input boundaries
Enter input boundaries in octree (only first time)
Create octree nodes in empty linear positions and set Forced Interface Nodes
Color octree nodes
Accomplish Tetrahedra distortion criterion (RC7)?
no
Mark cells violating RC7
yes
Accomplish topological criteria (RC8)?
no
Mark cells violating RC8
yes
Identify inner cells to each volume
Accomplish volumes size criterion (RC3)?
no
Mark cells violating RC3
yes
Delete data on outer cells
Final octree configuration
Process based on octree
Process based on unstructured mesh

Figure 6.9: Flowchart of octree refinement process.

from P to N contained inside the tetrahedron, and infinite positions inside face F for point P, so a strategy should be used to find the one (if it exists) fulfilling the proposition.

The strategy chosen in the present work is to check between a limited number of curves and points P, based on make the algorithm as simpler as possible. Specifically, only straight lines are taken into account, and only a few sample of points P are chosen inside face F. If some of these chosen curves satisfy the proposition, the tetrahedron can be colored. However, if they do not fulfill the proposition no decision can be taken, because nothing guarantees that there should not be other point P with a related curve fulfilling it.

In non-watertight geometries the Proposition 1 should not be applied because there is no guarantee to be valid if some of the curves passes through a gap. In the present work it has been applied also in these cases with the risk of assigning a bad color to a tetrahedron.

We finally mention that in some pathological configurations of tetrahedra, some elements may remain uncolored because no point P chosen fulfills the proposition. In these cases, as a last chance, the color of the tetrahedron is set to the color of its center, which is colored using the standard node coloring process.

6.5.4 Inner surface meshing

As explained in Section 5.3.10, when meshing inner surfaces the triangles corresponding to the mesh of the inner surface must be detected among the faces of the tetrahedra containing nodes on the surface. This process is done just after the tetrahedra coloring process, and involves the following steps:

- *Detect all the candidate triangles.* The candidate triangles are the ones which all its nodes are forced nodes in interface or in edge, placed on an inner surface.
- *Set as valid the topologically right ones.* These are the ones accomplishing the topologically properties defined in Section 5.3.10.
- *Set as invalid the ones accomplishing the Proposition 2.* Not considering the already set as valid, the Proposition 2 is used to set as invalid the corresponding candidate triangles.
- *Check the final configuration of triangles.* After the previous steps, there can be some configuration not accomplishing the topology of the inner surface. This is, some node which should be manifold belongs to a set of triangles which are not manifold. In these

cases a last procedure is performed in order to try to get a final valid mesh. It consists in trying to set as valid some invalid triangle (or set as invalid a valid one), and then check if the topology is accomplished. This is done for the triangles in contact with problematic nodes. Usually after applying the Proposition 2 almost all the triangles are correct and only some local region present topological inconsistencies, so this last strategy often gives good results and solves the local irregularities.

An extra consideration must be taking into account concerning the inner surfaces. As they are surrounded by the same volume (considering the outside part of the domain as volume zero), they do not affect to the coloring of the nodes. For this reason, when computing the intersections of the rays used in that process, the surface entities coming from an inner surface are not considered.

6.6 Parallel processing

The core of the meshing algorithm developed in this work is serial in the sense that the it implies a number of tasks to be done one after another. However, some of these tasks can be parallelized. In the present work shared memory paradigm following OpenMP strategy has been used to parallelize the more costly parts of the algorithm in terms of CPU time. These are the key parts of the algorithm parallelized:

- Nodes coloring. Each one of the rays used to color the nodes it passes through is independent from the others, so the rays are shot in parallel.
- Surface fitting process. Using topological information (the color of the nodes and the cells they belong to), the edges candidates to be affected by the surface fitting process are detected. Then, it has to be checked whether they intersect the input boundaries or not. This process is done in parallel, as each one of the edges are independent from the other ones.
- The checks needed to verify if a cell must be subdivided or not following the refinement criteria RC1, RC2, RC3, RC7 and RC8 only depend on the cell itself and the input boundaries. This allows to treat in parallel the cells to be checked for these refinement criteria.
- Tetrahedra generation through patterns. This process is totally parallel, as the creation of the tetrahedra are independent from each cell and, inside a cell, from each of its

faces. Actually, in some configurations the creation of the tetrahedra also depends on the neighbor cell, but in this case it is easy to prioritize which cell generate the tetrahedra (Section 3.3.3). It has to be noted that this process is very fast and only slight differences can be appreciated between the serial and parallel versions of the algorithm.

- Edge collapsing. The edge collapsing process is performed as follows. First of all the detection of edges suitable to be collapsed is performed, ensuring no node of an edge belongs to another edge to be treated. This step is done in serial, but it could be improved including locks in a parallel implementation. Then, in parallel, each edge is checked in order to evaluate if it can be collapsed, and if so, it is collapsed. This process is repeated a number of times in order to ensure that all the edges of the mesh have been taken into account.
- Smoothing process. The smoothing process is performed as follows. First of all, different non overlapping clusters of elements (containing bad quality elements) are detected, and then each one of these clusters is smoothed in parallel only moving its inner nodes. The clusters detection is done in parallel, but is could be improved in a parallel implementation including some locks in the code. This process is repeated a number of times in order to ensure all the nodes have been treated.

The performance of the code could be improved by parallelizing other parts of algorithm. In the present work, only the parts representing a bottle-neck and the ones naturally parallel have been considered for parallelization.

Considering the structure of the octree, a parallelization based on distributed memory can be thought for the algorithm. Briefly explained, the octree cells could be subdivided in different domains, and the whole meshing process could be applied to each part, taking into account to pass the information from one part to the other through the cells interfacing different parts of the domain. This process should be studied in detail and is out of the scope of this work.

6.7 Parameters used

In this section the value of the parameters governing the mesher are detailed.

- α_{ip} (defined in Section 5.3.2, page 91). It corresponds to the portion of the octree leaf size to be used as a limit distance. If a node is closer than it to the input boundaries, it is considered forced interface point. α_{ip} is a positive value, and cannot be higher than

0.25 in order to avoid the creation of inverted tetrahedra due to an excessive distance of the octree nodes with respect to their octree position. In the current implementation α_{ip} takes the value of 0.08.

- Ec_l (defined in Section 5.3.1, page 87). It represents the maximum chordal error allowed at the time of creating the forced edges from the forced line entities. The examples run show that the presence of chordal error (even if it is very small) in the forced edges may cause problems at the time to perform the surface fitting operations. For this reason, in the present implementation its value is really low: 0.000,1.

- $\alpha_{ms/cs}$ (defined in Section 5.2.2, page 77). This value is used for checking whether a cell must be subdivided or not, considering a desired size in its region. The subdividing process divides by two the size of the resulting children cells. A value of 1.33 ensures the cell will be subdivided when the size of the resulting children is closer to the desired size than the size of the cell itself. However, in the present implementation a value of 1.5 have been chosen in order to avoid an excessive level of refinement in some examples.

- α_{edge} (defined in Section 5.3.3, page 93). This is a real value between 0 and 1 indicating the percentage of the length of an edge to be use as a limit distance to consider two intersection points in the edge close enough to refine the cell following the topological refinement criterion. The value used in the present implementation is 0.1.

- α_{vertex} (defined in Section 5.3.5, page 104). In the preserving features operations, this parameter controls if the intersection point between a base line entity and the tetra face is close enough to one of the tetrahedra nodes to move the node to its position. It is a percentage of the size of the shortest edge of the tetrahedra (so it must be a value between 0 and 1). The value used in this implementation is 0.3.

- α_{side} (defined in Section 5.3.5, page 104). This parameter controls if the intersection point between a base line entity and the tetrahedra face is close enough to one of the tetrahedra edges to split it in the preserving features operations. It is a portion of the size of the shortest edge of the tetrahedra (a value between 0 and 1). The value used in this implementation is 0.3.

- α_{iso} (defined in Section 5.3.6, page 108). This parameter is the portion of an edge length to set the limit distance in the surface fitting process used to decide if the edge must be split, or one of the nodes of the edge must be moved to the intersection point. Its value must be compressed between 0 and 0.5. A value close to 0 will split all the edges, and a value close to 0.5 will move the edge nodes in the surface fitting process. The value used in this implementation is 0.3.

- α_c (defined in Section 5.3.8, page 120). It is the portion of the cell size to set the limit length under which an edge will be collapsed in the make-up operations. A value equal to 0 will not collapse any edge. The value used in the present implementation is 0.2.

- α_{msp} (defined in Section 6.3, page 135). This parameter controls the distribution of generalized mesh size points onto the mesh size entities. It must take a value greater than one. A value equal to one would create too much generalized mesh size points, and a value greater than two could leave areas in the mesh size entity not covered by any generalized mesh size point. The value used in this implementation is 2.5, considering the region of the octree cells affected by a mesh size entity is always larger than the mesh size entity itself.

- α_{bb} (defined in Section 3.4, page 50). It is the portion of the model bounding box size used to create the MIPs when more than one intersection point are very close one from each other. It must be a very low value not to collapse relevant details of the model when dealing with the pathological intersections situations. The value used in the present implementation is 1×10^{-5}.

Chapter 7

Examples

In this chapter some examples of meshes generated using the proposed algorithm are shown. In the first part, some simple validation examples are shown in Section 7.1. These examples try to demonstrate the effectiveness of the mesher in dealing with some of the typical drawbacks of octree based meshers (mainly geometrical features and topological preservation), so they do not involve large meshes.

Examples in Sections 7.2 and 7.3 involve complex real geometries trying to evaluate the performance of the new meshing algorithm for complicated geometries and the generation of a large amount of elements.

The analysis of the results considering all the representative examples is carried out in Section 7.4. Some representative cases are used in Section 7.4.1 to compare the prefomance of the new mesher with other two meshers:

- An advancing front implementation (the one included in the GiD v11 version [CRP$^+$10a, CRP$^+$10b, CRP$^+$10c]).
- A Delaunay implementation (the one provided in Version 1.4 of the Tetgen library [ST10]).

In order to present a realistic view of the results and allow their correct evaluation, the following data of the model to be meshed is provided for each example:

- Sphericity of the model (S). The specific surface of a solid (volume) is computed as the ratio between the area of its surfaces and its volume. Its units are of $length^{-1}$. This ratio is useful to asses the speed of tetrahedra generation. In the presented method the tetrahedra within the volumes are extremely fast to generate, and the most time consuming operations are related to the tetrahedra near the boundary. The more specific surface has the model, the more expensive (in computational terms) the mesh generation will be. However, the specific surface measure is not dimensionless. For instance, scaling homogeneously a volume changes its specific surface, so this measure is not suitable for comparing models with different sizes.

To be able to compare different models (with different sizes), the sphericity is provided instead of the specific surface. This measure is dimensionless, and indicates how far is the volume measured from a sphere (which is the shape with less specific surface enclosing a given volume). The sphericity S of a model is computed using the following equation:

$$S = \frac{\pi^{\frac{1}{3}}(6V)^{\frac{2}{3}}}{A} \tag{7.1}$$

where V is the volume of the model and A is the area of the surface entities defining the domain. The sphericity takes values from 0 (shapes with high specific surface) to 1 (a perfect sphere). Theoretically, the presented algorithm should be faster for models with higher sphericity.

- Input mesh size. Some of the examples are meshed with a uniform mesh size, and some of them have different desired mesh sizes assigned to different parts of the domain. The desired mesh sizes of each example are detailed.
- Number of volumes present in the model.
- Number of surface entities in the input boundary. In all the examples shown these are triangle mesh elements. This can give an idea of a ratio between the input boundary entities size in comparison to the octree leaves (which are more less of the same size of the final mesh elements).
- Watertight condition. If the model is watertight or not.
- List of the relevant general parameters used for the mesher (Section 3.1).

For each case, some figures of the input boundaries and the final mesh are shown as well as the following data:

- Number of nodes generated.

- Number of tetrahedra generated.
- Number of triangles in the final mesh (faces of tetrahedra interfacing different volumes).
- Number of line elements (coming from forced edges) in the final mesh.
- Quality of the elements generated. In order to distinguish between the mesher itself and the make-up and smoothing operations, the quality of the elements will be shown before and after these operations. The minimum dihedral angle of the element is taken as a quality measure, and its accumulated distribution among all the elements is provided. The value corresponding to the worst quality element of the mesh is also provided, as well as the number of elements with a minimum dihedral angle lower than 5 degrees (in case there are some).
- Memory used by the algorithm. The memory needed to store a given mesh can be determined a priori, as it consists of the coordinates of the nodes and the connectivity of the elements. However, during the mesh generation process, extra memory is needed to store auxiliary data. These data are deleted once the mesh has been generated. The ratio between the peak of memory needed to generate the mesh and the memory required to store that mesh is given in order to evaluate the efficiency (in terms of memory) of the implementation of the algorithm. A value of this ratio higher than one is expected, as the algorithm should consume more memory for generating the mesh than the needed to store it (once it has been generated).
- Time needed to generate the mesh. The time of the whole mesh generation process is given, as well as the time consumed by the different parts of the algorithm, to allow the evaluation of possible bottle necks. As the implementation of the algorithm has been done following parallel processing techniques, in the relevant examples the time will be provided taking into account different scenarios: using only one thread (as a reference time), and using two, three and four threads. This allows to evaluate the scalability of the algorithm. The time is measured using the *omp_get_wtime*() function [Ope13], and the value provided is the mean among five measures taken.
- Speed: number of tetrahedra generated per minute. This speed is provided in order to compare with other methods, but it has to be taken into account that it can lead to misleading evaluations because of the sphericity considerations mentioned above. Another important factor to be considered when assessing the tetrahedra generation

speed is the mesh desired size in comparison with the final sizes of the mesh. Specially in the operations needed to preserve the volume topology, the algorithm is more time-consuming if it has to refine automatically than if the sizes introduced by the user fulfill directly the topological criteria.

- Speed-up: the speed-up is defined as the ratio between the time needed to run a process in serial and the time needed to run it in parallel (using more than one thread). This value is given in the examples run in parallel, in order to evaluate the efficiency in the parallel implementation of the algorithm.

In order to evaluate the efficiency and performance of some parts of the algorithm as well as their implementations, the following data is also provided:

- Number of local ray castings performed: local ray casting operations are needed in the coloring algorithm when the three Cartesian rays incident to a node are invalid. The number of local ray casting operations performed provides an idea of the robustness of the coloring algorithm presented in this work.
- Number of undetermined tetrahedra: these are the ones colored following the tetrahedra coloring algorithm explained in Section 5.3.7.
- Number of tetrahedra in interface cells: as explained in Section 6.5, almost all the operations of the mesher are applied to the tetrahedra generated from interface cells, and the tetrahedra from the inner cells of the volumes are not generated until the end of the meshing process. The number of tetrahedra in interface cells gives an idea of the memory saved during the meshing process thanks to the fact that the tetrahedra from the inner octree cells are not generated yet. Two values are provided:
  - The number of tetrahedra generated directly from the interface cells applying the tetrahedra pattern.
  - The number of tetrahedra just after the make-up and smoothing operations. This number results from applying to the previous one the operations of preserving geometrical features, surface fitting operations, deleting the outer tetrahedra and performing the make-up and smoothing operations.
- Number of octree cells. The total number of cells of the octree is provided (as well as the number of interface and outer cells) considering the refinement criteria in the

whole octree root. Also the total number of cells not applying them to the outer cells is provided in order to give an idea of the memory saved thanks to the presented implementation.

The characteristic sizes, as well as the mesh sizes of the examples are expressed in general units of length (*uol*).

All these data are not provided in all the examples, as each one of them has specific results to be highlighted, so only its relevant data will be provided. Complete tables of the data for all the examples studied in this chapter are available in Appendix A.

The meshing algorithm has been implemented as a library, and it has been integrated in the pre and post-processor GiD [CRP+10a, CRP+10b, CRP+10c]. The examples shown in this document are meshes using the version of the algorithm present in the 11.1.9d version of GiD, which can be downloaded from the website [MCP+13]. All the meshes have been generated in a computer with the characteristics shown in Table 7.1.

Processor	Intel Core i5 3570K @3.40GHz
RAM memory	32.0 GB DDR3 800 MHz
OS	Windows 8 64-bits

Table 7.1: Characteristics of the computer used to run the examples

The code is written in C++ and it has been compiled using Microsoft Visual Studio 2012 Version 10.0.402191.1 SP1Rel with OpenMP enabled [Ope13].

7.1 Validation examples

In this section, some validation examples are shown. Some of them have been chosen in order to demonstrate that the presented mesher can generate meshes from input data which is not treatable in raw by other volume meshing techniques (like advancing front or Delaunay). This is the case for non-watertight input boundaries, or input triangle meshes with very low quality elements. Other examples may be more suitable to be meshed with these techniques, but they have been used to demonstrate that the presented algorithm solves some common problems in octree based methods (like the preservation of topology or geometric features).

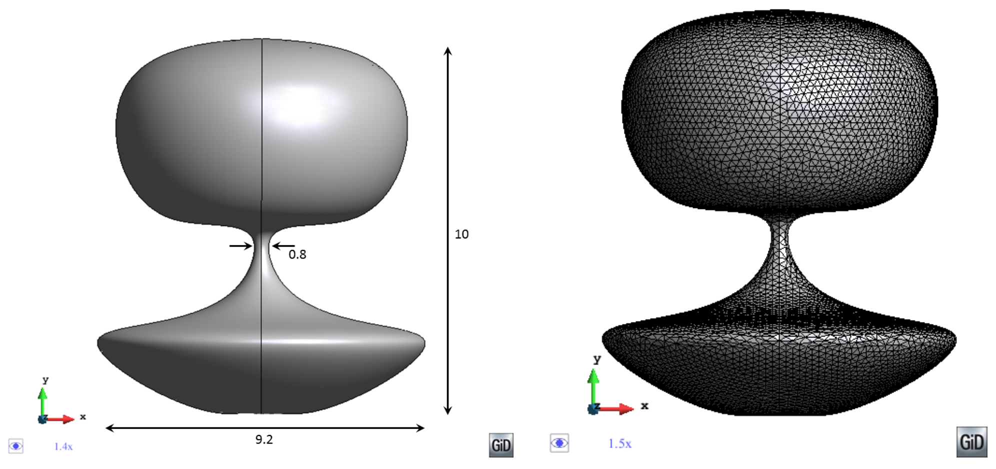
7.1.1 Preserving topology

As mentioned in Section 2.5, one of the disadvantages of the octree-based meshers is the difficulty to preserve the topology of the model. Some examples are presented in this section showing that the proposed algorithm solves this problem. The combination of the topology refinement criterion, the surface fitting process and the tetrahedra coloring are the main parts of the algorithm in charge of preserving the topology.

Some of these examples may be obvious to be meshed using other techniques like advancing front or Delaunay based meshers, but this is not the case for classical octree-based meshers.

Validation example $VE-T1$

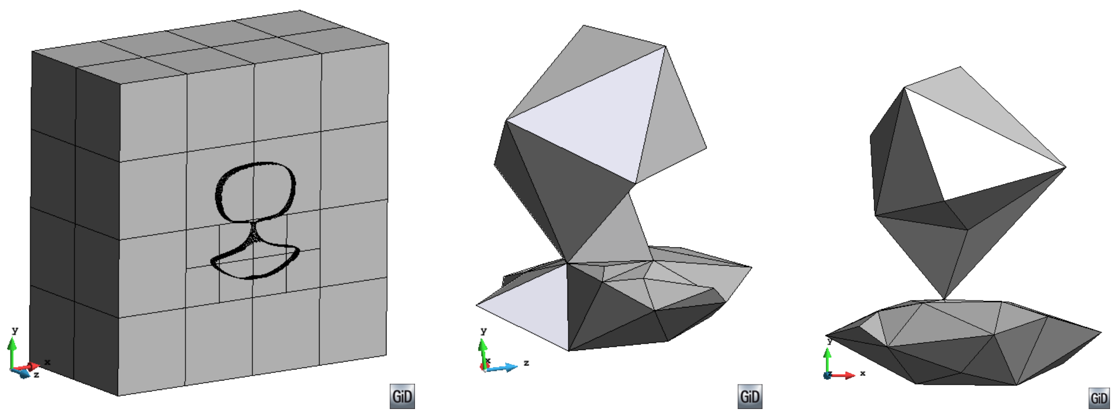
The first example is the one shown in Figure 7.1. This model is formed by an axisymmetric watertight volume. It has no geometrical features to be preserved and all its surfaces are curved. The particularity of the model is that its central part is very thin in comparison with its characteristic size. A view of the geometrical model (represented with NURBS surfaces) is shown in Figure 7.1(a), where its characteristic sizes are depicted. The contour mesh of the volume used as the input boundaries for the volume mesher is shown in Figure 7.1(b).



(a) (b)

Figure 7.1: Model used in the validation example $VE-T1$. **(a)** View of the geometry of the volume and its characteristic sizes. **(b)** View of the triangle mesh used as the input boundaries for the volume mesher.

In this example no desired mesh size has been set in the input data, and a size transition factor (TF) equal to 1 has been used. This is the most unfavorable situation for the mesher,



(a) (b) (c)

Figure 7.2: Results for the validation example $VE - T1$. **(a)** View of a cut of the octree cells after the refinement process and part of the input boundaries. **(b)** Final tetrahedra mesh generated. **(c)** Mesh generated if the tetrahedra coloring is based on the color of the center of each element.

as it implies that all the octree refinement process is based on the topology criteria and the constrained two to one condition given by the balance criterion.

A view of the octree cells in a cut plane passing trough the rotation axis of the model is shown in Figure 7.2(a). This configuration of cells is the result of applying the octree refinement process to the octree root. It can be seen that, even when no desired mesh size has been assigned, the octree is refined because of the topology criteria.

The final mesh is depicted in Figure 7.2(b). It is a very coarse mesh, as no mesh size was entered in the input data. However, it represents perfectly the topology of the model. In this case, the tetrahedra coloring strategy plays a key role for preserving the topology. The use of the simple tetrahedra coloring technique based on assigning to each element the color of its center would yield the mesh shown in Figure 7.2(c). It can be seen that the model topology would not be preserved using such simple tetrahedra coloring technique, as the upper and lower parts of the model would be connected by just one node in a non-manifold way.

Although the mesh generated in example $VE - T1$ accomplish with the requirements of the mesher, the elements are too large to capture reasonably well the shape described by the input boundaries. Hence, it may not be useful for a given numerical simulation. A mesh of the same model with a desired mesh size of 1 uol is depicted in Figure 7.3(a) in order to empathize that the mesher is flexible in terms of mesh size assignment: if no mesh size is entered in the input data, a *legal* mesh is generated, but if some specific mesh size is needed,

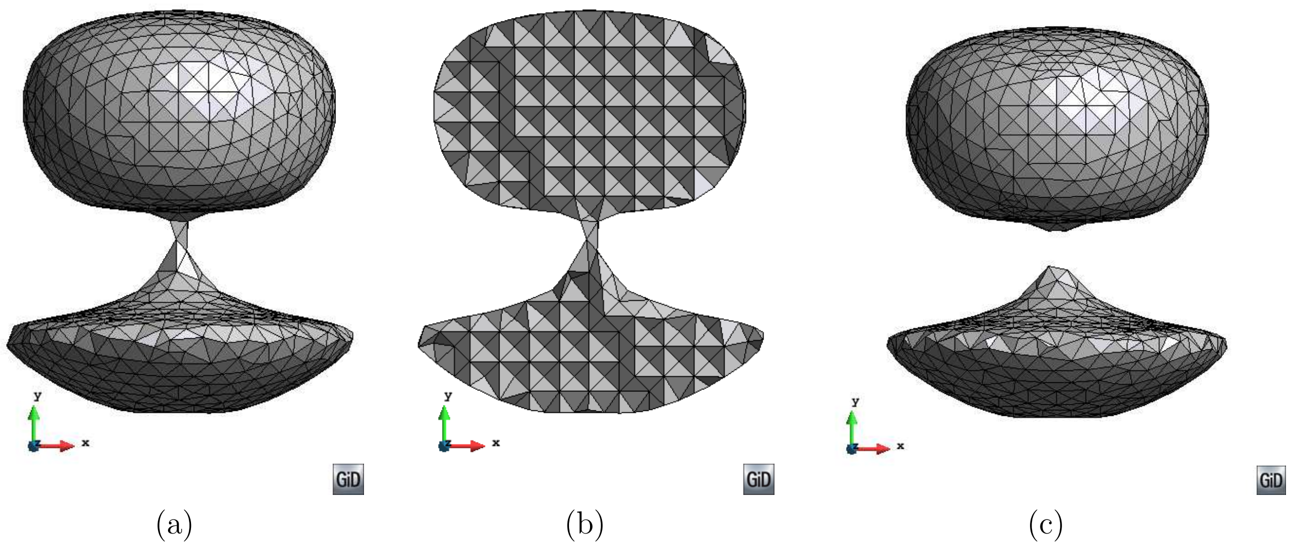


Figure 7.3: Results for the validation example $VE - T1$ with mesh size equal to 1.0 uol. **(a)** Final tetrahedra mesh generated. **(b)** View of the inner elements of the mesh. **(c)** Mesh generated if no topological mesh refinement criterion is applied. It can be seen that the topology of the model is not preserved in this case, as there are two unconnected parts of the final mesh

the mesher takes care of it. A view of the inner elements is depicted in Figure 7.3(b). The regular distribution of elements coming directly from the tetrahedra patterns from the octree cells can be appreciated in the inner parts of the volume.

The data of the generated mesh is detailed in Table 7.2.

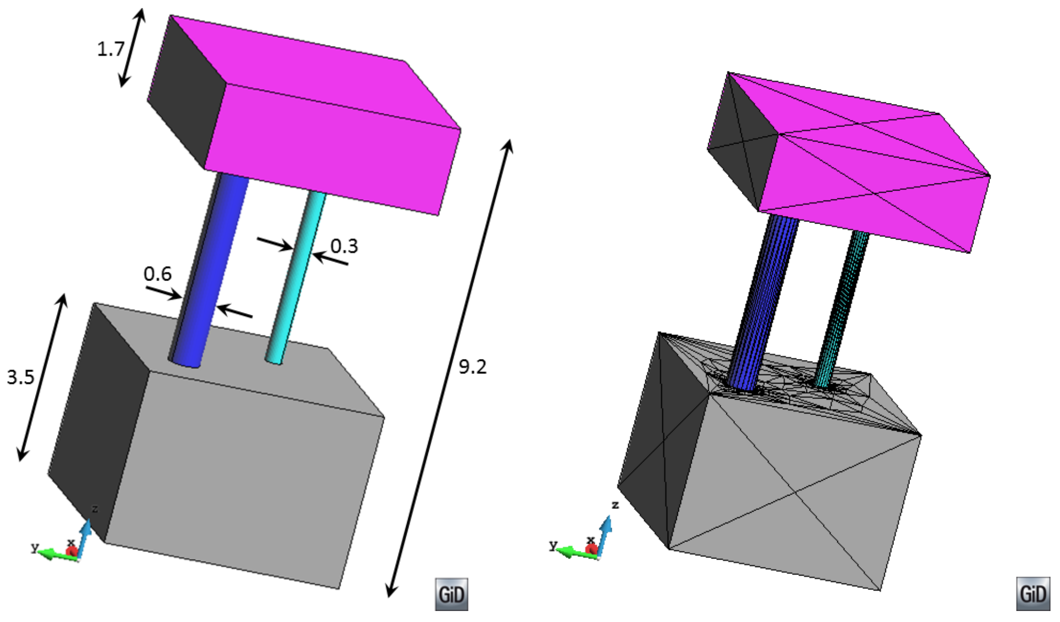
Number of threads	1
Mesh size (uol)	1.0
Number of tetrahedra	9,474
Number of nodes	2,211
Minimum dihedral angle (degrees)	5.7
Time to generate the mesh (seconds)	0.75
Speed (Mtetrahedra per minute)	0.8

Table 7.2: Data for the validation example $VE - T1$.

It has to be noted that, decreasing the desired mesh size, the resulting mesh fits much more with the original shape of the domain. However, even with a mesh size of 1 uol, the classical approach for octree based meshers would generate an invalid mesh like the depicted in Figure 7.3 (c). This is because the size of the cells is larger than the thin part of the model. In this situations it is crucial to use a strategy for the octree refinement and a surface fitting process able to preserve the topology of the model. In the case of the presented mesher the refinement criteria takes into account the posterior surface fitting process.

Validation example $VE - T2$

The validation example $VE - T2$ (Figure 7.4) also presents some thin parts. In this case the model is formed by 4 watertight volumes: the upper part, the lower part and the two cylindrical parts connecting both. The extreme surfaces of the cylindrical volumes are shared by two volumes. As it can be seen, this model has sharp edges to be preserved by the mesher.

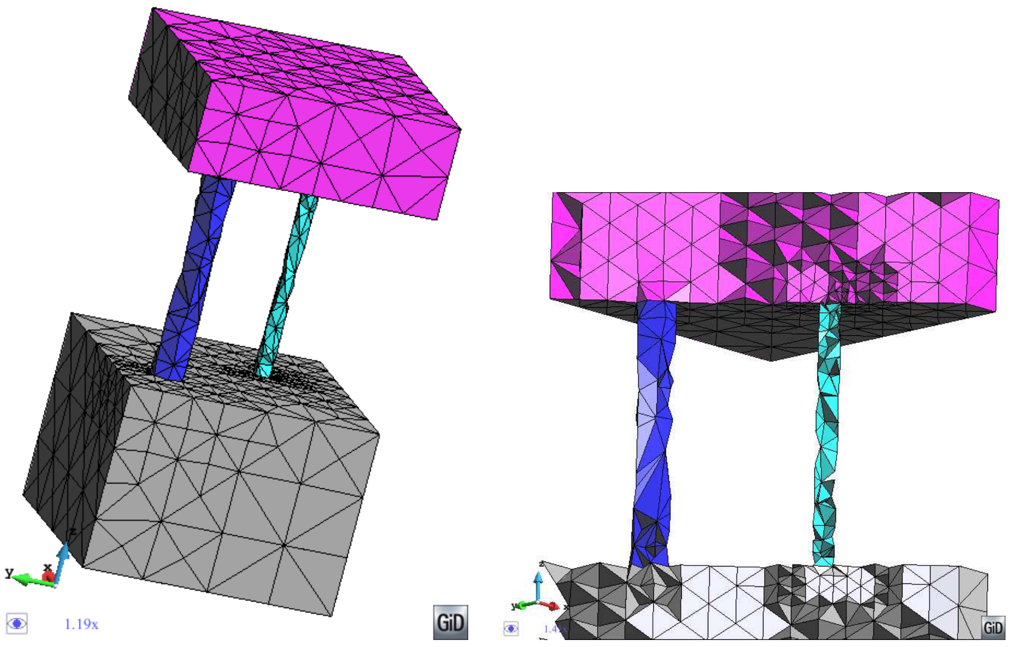


(a) (b)

Figure 7.4: Model used in the validation example $VE-T2$. **(a)** View of the geometry of the model and its characteristic sizes (in uol). Different volumes are drawn in different colors. **(b)** View of the triangle mesh used as the input boundaries for the volume mesher.

Analogously as the validation example $VE - T1$, no desired size has been provided to the mesher and TF is equal to 1, so the refinements needed to preserve the topology of the original model are performed automatically by the mesher.

The tetrahedral mesh generated is shown in Figure 7.5(a). As it can be seen, the topology of the model has been preserved, as well as its sharp edges. Each volume of the model has



(a) (b)

Figure 7.5: Results for the validation example $VE-T2$. **(a)** Final tetrahedra mesh generated. **(b)** Zoom view of some internal elements of the final mesh.

its corresponding tetrahedra, and they fit the contour surfaces. A zoom view of the inner elements is shown in Figure 7.5(b).

The data of the generated mesh is detailed in Table 7.3.

Number of threads	1
Mesh size (uol)	$None$
Number of tetrahedra	16,304
Number of nodes	3,830
Minimum dihedral angle (degrees)	6.2
Time to generate the mesh (seconds)	0.37
Speed (Mtetrahedra per minute)	2.6

Table 7.3: Data for the validation example $VE - T2$.

No mesh size has been provided to the mesher in order to check the preservation of the topology of the model by the mesher. While the topology has been perfectly captured, the element size in the thin parts of the model is too large to define properly the curved shape. These elements have a high chordal error, which is a typical weak point of the octree based

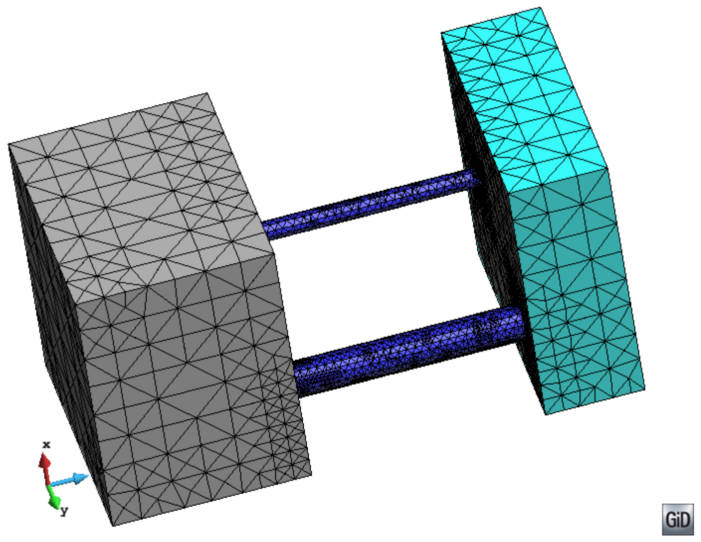


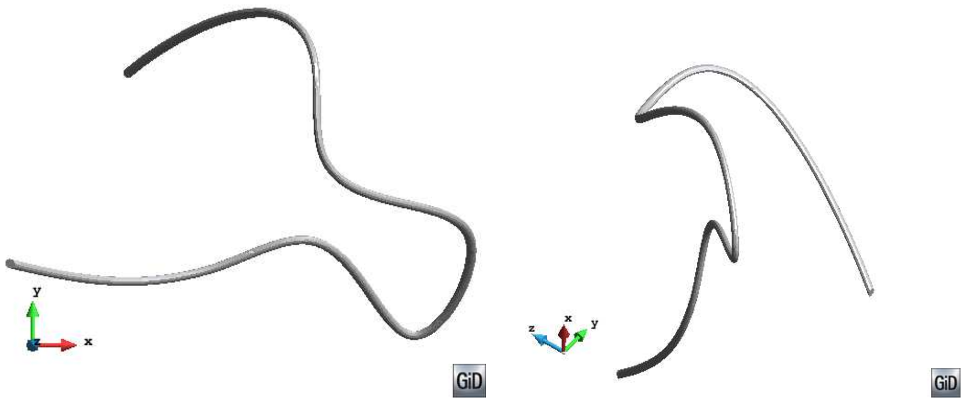
Figure 7.6: View of a mesh of the validation example $VE-T2$ with a smaller mesh size in the curved parts of the domain.

meshers. Nevertheless, assigning a smaller size in the curved regions a better quality mesh is obtained as shown in Figure 7.6. Some specific make-up and smoothing operations may also reduce the chordal error of the mesh.

In this example the independence of the mesh generator to the input mesh quality is demonstrated. In Figure 7.4(b) it can be appreciated that the boundaries of the volumes are well defined in terms of chordal error, but the quality of the triangles is very low. However, this does not affect the meshing algorithm. In methods based on the advancing front technique, the input boundaries of this example could not be used directly as the active front for the volume meshing. A previous mesh improvement of this input surface mesh should be required, or even the generation of a new one. In the other hand, this provides the advancing front methods with a better mesh quality in terms of chordal error.

Validation example $VE-T3$

As the topology refinement criteria concerning surfaces and volumes ((b) and (c)) are not implemented, the more unfavorable case for the presented algorithm is the one where the model has very thin parts, understanding thin as a distance much smaller than the bounding box of the domain. In these situations, the octree only will be refined properly to capture the topology of the model if some of the edges of the tetrahedra generated from the cells intersects the input boundaries. If not, the octree should be too coarse and the surface fitting process should not be able to capture well the boundaries.



(a) (b)

Figure 7.7: Model used in the validation example $VE-T3$. Two views (**(a)** and **(b)**) of the triangle mesh used as the input boundaries for the volume mesher are shown in order to get a proper 3D perception of the geometry. The diameter of the cylindrical volume is 0.6 uol, and the side of its bounding box is around 8 uol.

This validation example corresponds to this unfavorable situation. A view of the model and the input boundaries used to feed the mesher is shown in Figure 7.7. The model is a unique watertight volume representing a cylindrical rod curved in space. The diameter of the cylinder is 0.6 uol, and the bounding box of the model is around 8 uol.

A mesh with a mesh size of 0.1 has been generated using the presented algorithm. The final mesh is shown in Figure 7.8. It can be appreciated that the topology of the model is preserved. The data of the generated mesh is detailed in Table 7.4.

Number of threads	1
Mesh size (uol)	$None$
Number of tetrahedra	9,986
Number of nodes	3,733
Minimum dihedral angle (degrees)	6.5
Time to generate the mesh (seconds)	2.52
Speed (Mtetrahedra per minute)	0.2

Table 7.4: Data for the validation example $VE-T3$.

A mesh generated with no mesh size assigned (which is the most unfavorable situation for the mesher) is shown in Figure 7.9(a). As commented above, the no implementation of the

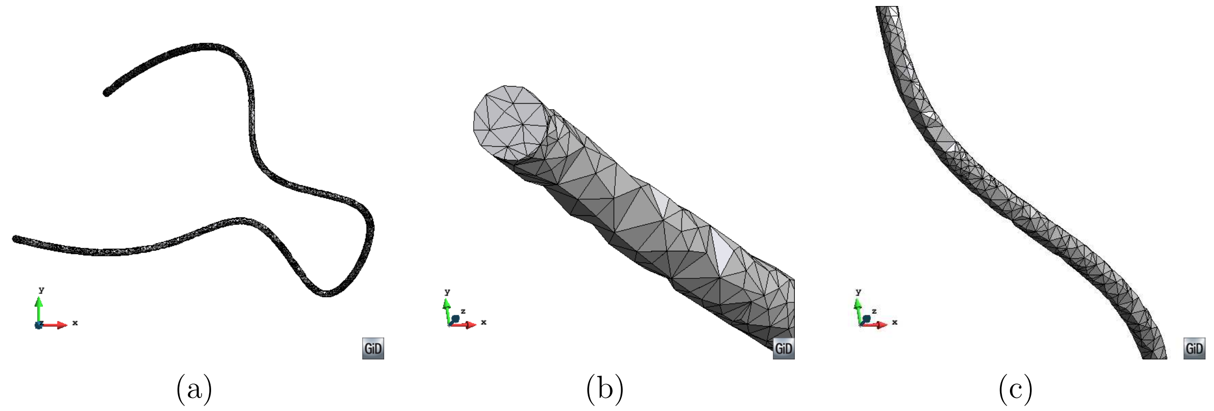


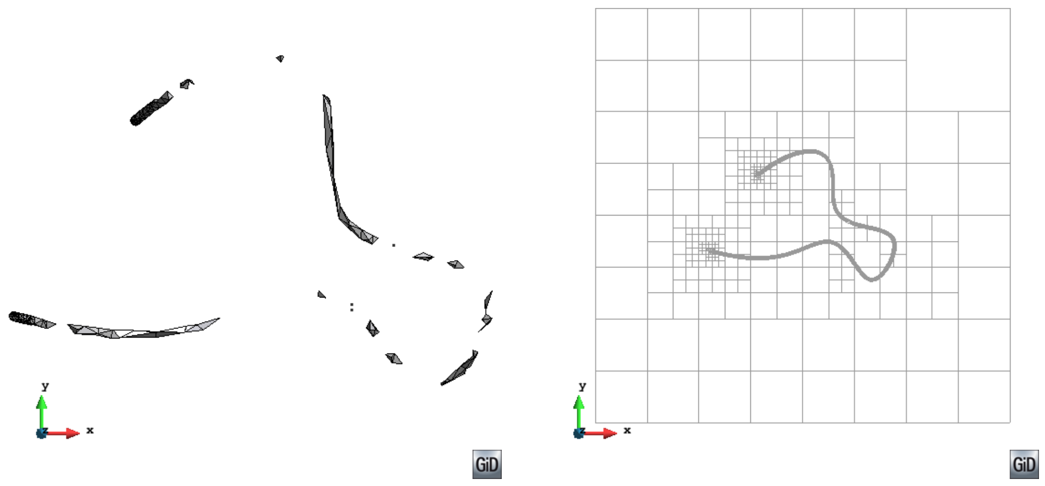
Figure 7.8: Result mesh of the validation example $VE - T3$. Different views of the mesh generated with a desired mesh size of 0.1 uol.

surface and volumetric parts of the topological refinement criterion can lead to situations where a cell does not recognize the need to be subdivided. This occurs in this case. It can be appreciated that the topology of the model has not been preserved, as there are parts of unconnected meshes representing the unique volume of the model. This example shows the importance of the topology refinement criteria.

A view of the octree refined merged with the input boundaries is depicted in Figure 7.9(b). Although the octree is relatively refined in the regions where the volume is, the level of refinement is not enough to let the surface fitting process capture well the topology of the model. It can be seen that the octree is more refined in the end parts of the volume. This is due to the presence of sharp edges there, so the corresponding forced edges are created. They involve forced points which activate the forced nodes refinement criterion.

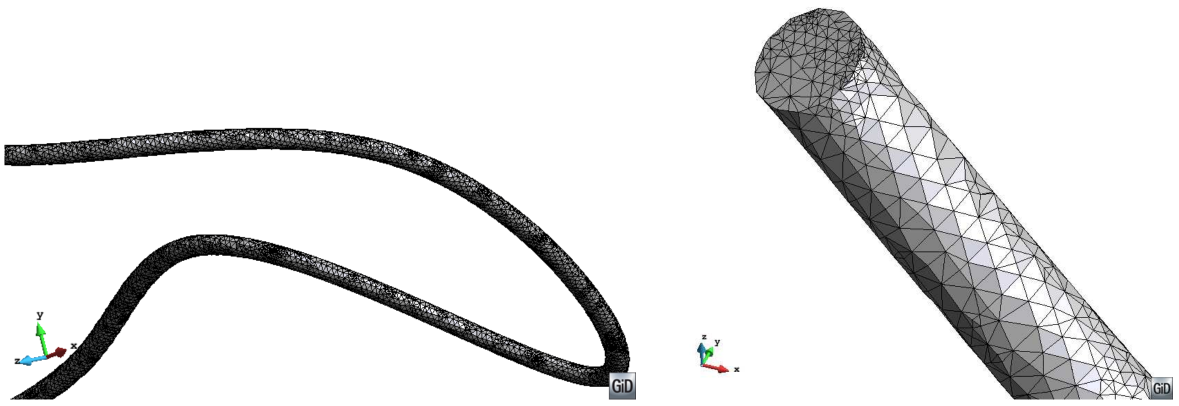
It is uncommon not to provide with any characteristic size of the model at the time of generating a mesh for a numerical simulation. Due to the solver needs or for the need of accuracy in the shape representation, often a size is provided to the mesher locally or globally in order to reach a successful mesh for the simulation. Actually, in the presented case it can be appreciated that the mesh obtained with a mesh size of 0.1 uol (although it preserves the topology) may be too coarse to represent the curvature of the rod with an acceptable chordal error. A zoom view of a mesh of the same example using a mesh size of 0.05 uol is depicted in Figure 7.10. It can be seen that the chordal error of the mesh is considerably reduced.

One remark to be done is that the assignment of sizes can be done manually or automatically. In this example an automatic size could be assigned to the input boundaries considering the curvature of the model, so no user interaction should be needed.



(a) (b)

Figure 7.9: Mesh of the validation example $VE - T3$ with no mesh size assigned. **(a)** View of the tetrahedra mesh generated. Note that the topology of the model is not preserved in this case. **(b)** View (aligned with a Cartesian direction) of the octree merged with the input boundaries.



(a) (b)

Figure 7.10: Zoom views of a mesh of the validation example $VE - T3$ with a mesh size of 0.05 uol.

7.1.2 Preserving geometrical features

Two validation examples are presented in this section in order to check the preserving of geometrical features: corners and ridges.

Validation example $VE-F1$

This validation example is shown in Figure 7.11, where its characteristic sizes are depicted. It is a synthetic wing profile.

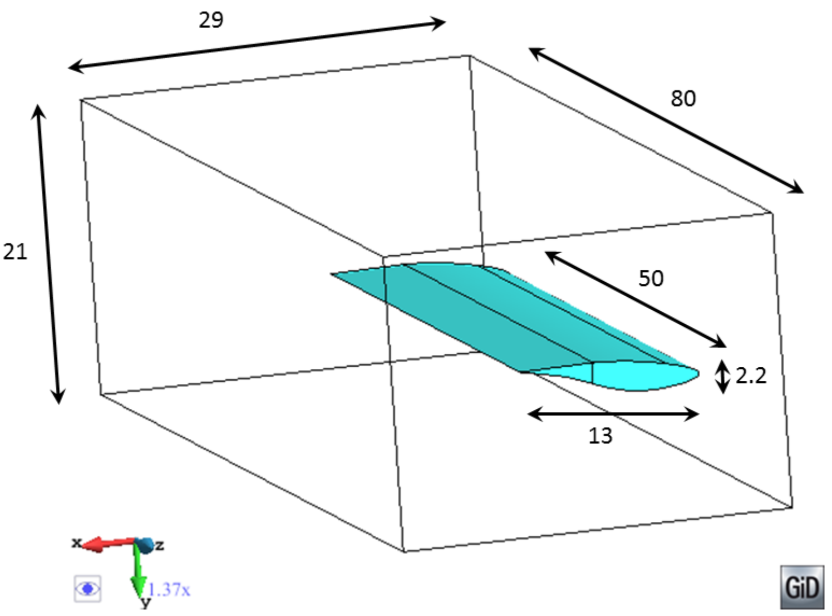


Figure 7.11: Model used in the validation example $VE-F1$. View of the geometry of the model and its characteristic sizes (in uol).

Computational Fluid Dynamic (CFD) simulations of this kind of geometries require to capture the flow behavior near the end part of the wing (the sharp edge), so its preservation is crucial for the mesher. Two views of the input boundaries used for the mesher are depicted in Figure 7.12.

The model has a control volume around the wing profile and two volumes are meshed: the wing itself and its outer part (inside the control volume). The two volumes are watertight. A size of 0.3 uol has been assigned to the surface entities of the wing, and the TF is equal to 0.6. The general mesh size for the control volume is 5 uol.

The tetrahedra mesh generated is shown in Figure 7.13(a). A zoom view of the inner elements is shown in Figure 7.13(b). As it can be seen, the sharp edges have been properly preserved.

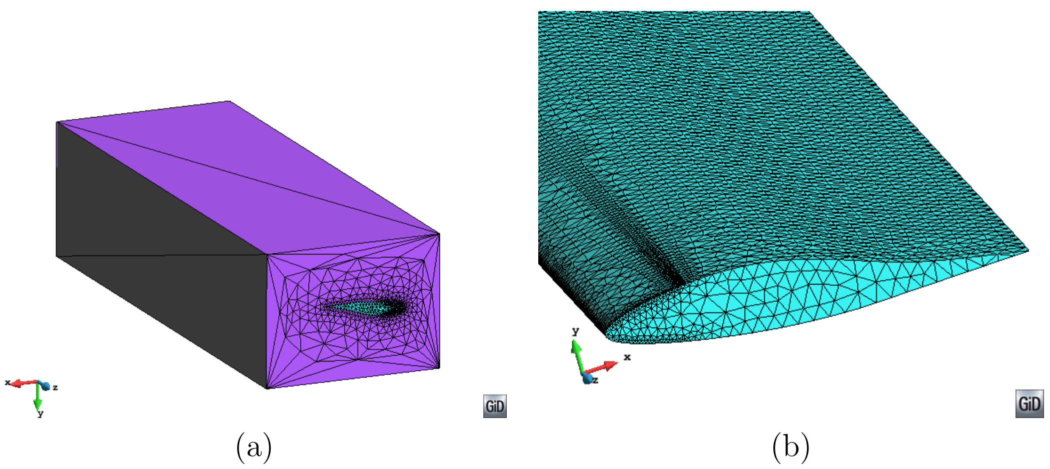


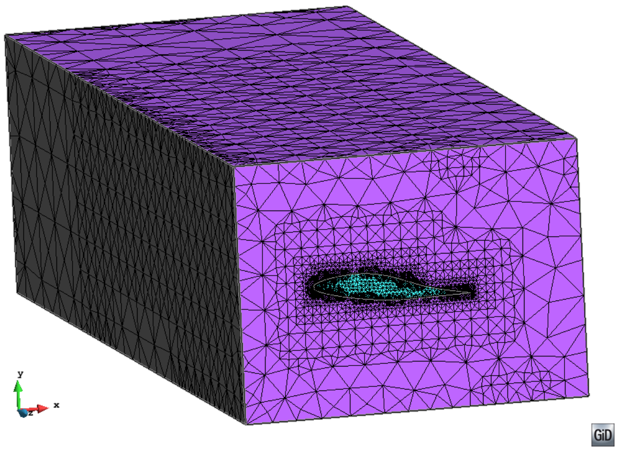
Figure 7.12: Views of the triangle mesh used as the input boundaries for the validation example $VE - F1$. **(a)** View of the the outer part of the control volume. **(b)** Detail of the triangle mesh in the wing profile used as input boundaries.

It also can be appreciated that the size of the skin triangles of the generated tetrahedra is independent from the size of the triangles of the input boundaries. The independence of sizes between the input and the final mesh is an advantage of the presented meshing algorithm: the mesh of the input boundary only has to take care on criteria for representing precisely the shape of the domain, without considering the required size for the volume mesh. The same input boundary mesh can be used for generating volume meshes with different sizes. In advancing front-based meshers, for instance, the surface mesh has to be generated for each new size required for the volume.

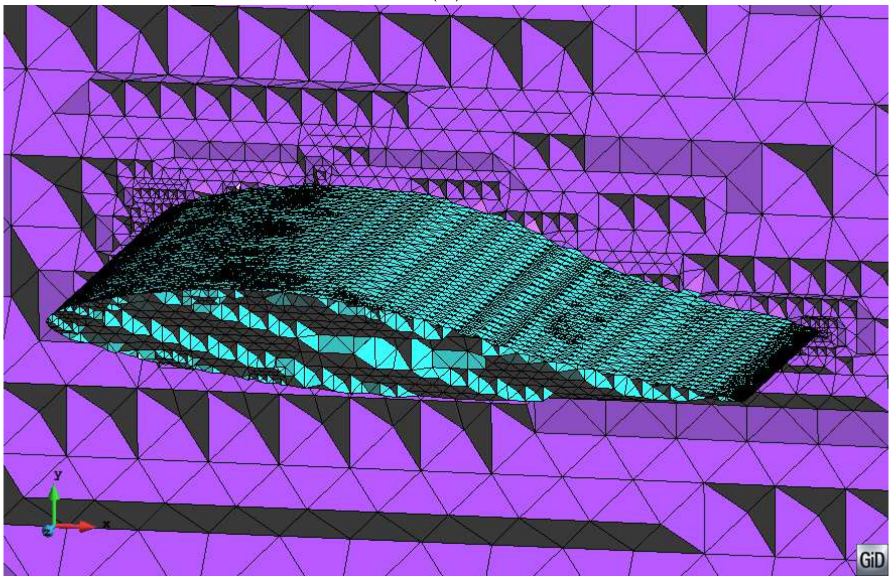
The data of the generated mesh is detailed in Table 7.5.

Number of threads	1
Mesh general size (uol)	5
Mesh size in the wing surface (uol)	0.3
Transition factor	0.6
Number of tetrahedra	1,097,012
Number of nodes	193,400
Minimum dihedral angle (degrees)	2.1
Number of tetrahedra with minimum dihedral angle lower than 5 degrees	3
Time to generate the mesh (seconds)	40.9
Speed (Mtetrahedra per minute)	1.6

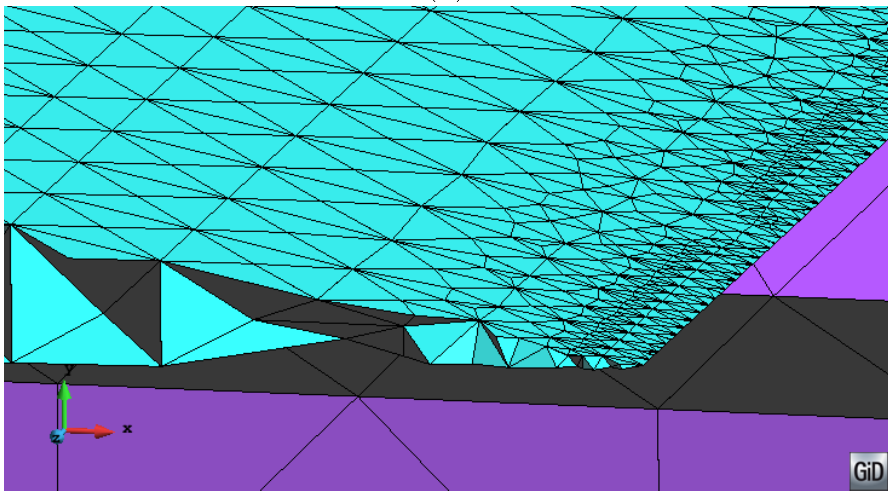
Table 7.5: Data of the validation example $VE - F1$.



(a)



(b)



(c)

Figure 7.13: Results for the validation example $VE - F1$. **(a)** Final tetrahedra mesh. **(b)** and **(c)** Zoom views of internal elements of the final mesh.

It can be appreciated that the minimum dihedral angle of the mesh elements is 2.1 degrees, and there are three elements with a minimum dihedral angle less than 5 degrees. These elements are in the wing volume, in the region where the geometry itself have a very small dihedral angle (the rear part of the wing).

Validation example $VE - F2$

The validation example $VE - F2$ is the model of a mechanical piece (a crankshaft of an automotive) provided by the company *Quantech ATZ* [Qua14]. Its geometrical definition comes from an *.stl* file (that is, a triangle mesh). This triangle mesh representing the input boundaries (depicted in Figure 7.14) is watertight.

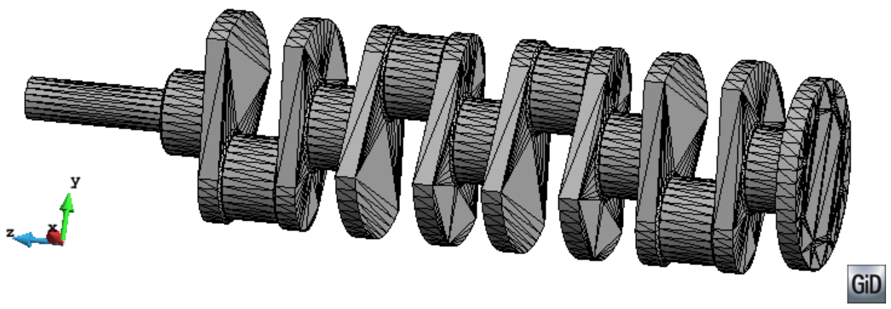


Figure 7.14: Model used in the validation example $VE - F2$. View of the triangle mesh used as the input boundaries of the volume.

The dimensions of the bounding box of the model are around 630 by 150 by 130 uol. The thin parts of the model have approximately 15 uol. A uniform mesh size of 20 uol and a TF equal to 1 has been provided in the input data of the mesher.

The tetrahedra mesh generated is shown in Figure 7.15. As it can be seen, the sharp edges have been properly preserved. Some zoom views of the mesh and the inner elements are depicted in Figure 7.16.

It has to be noted that the input boundaries in this example are defined with very low quality triangles. However, the new mesher does not need a good quality mesh to represent the boundaries of the domain. This is not the case for other meshing algorithms, like the one based in advancing front technique. Using such techniques with this model would require a previous meshing of the boundaries, even when the domain is watertight (like in this case).

The data of the generated mesh is detailed in Table 7.6.

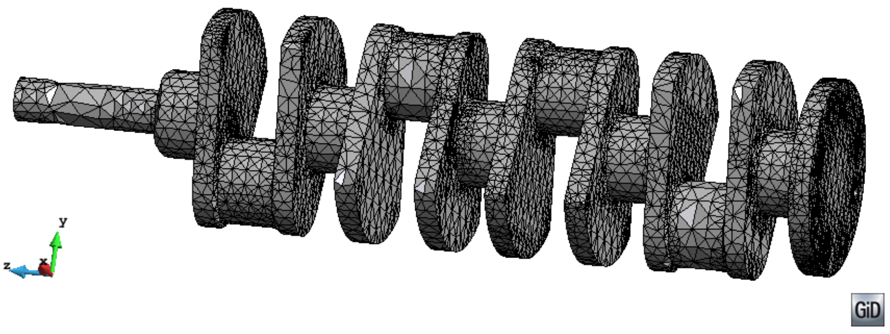


Figure 7.15: Results for the validation example $VE - F2$. Final tetrahedra mesh generated.

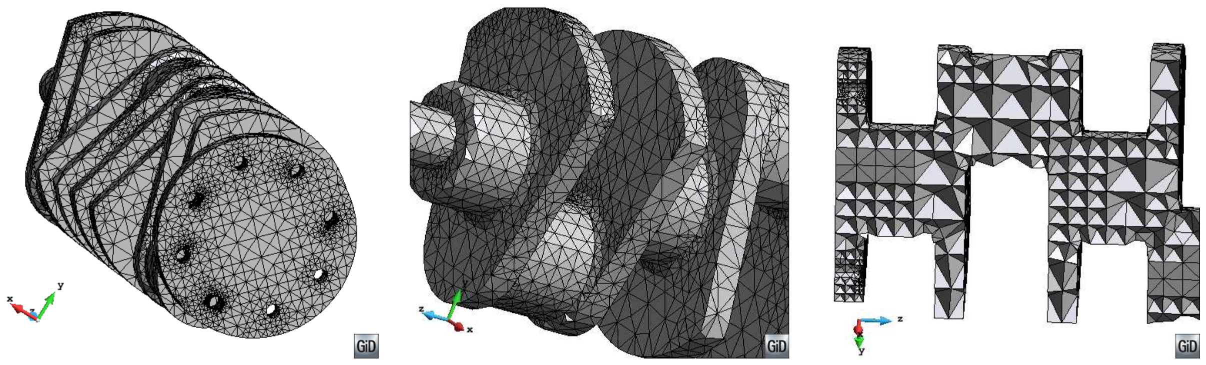


Figure 7.16: Different views of the mesh generated from validation example $VE - F2$ and its inner elements.

Number of threads	1
Mesh general size (uol)	20
Transition factor	1.0
Number of tetrahedra	88,718
Number of nodes	22,549
Minimum dihedral angle (degrees)	2.5
Number of tetrahedra with minimum dihedral angle lower than 5 degrees	1
Time to generate the mesh (seconds)	4.3
Speed (Mtetrahedra per minute)	1.2

Table 7.6: Data of the validation example $VE - F2$.

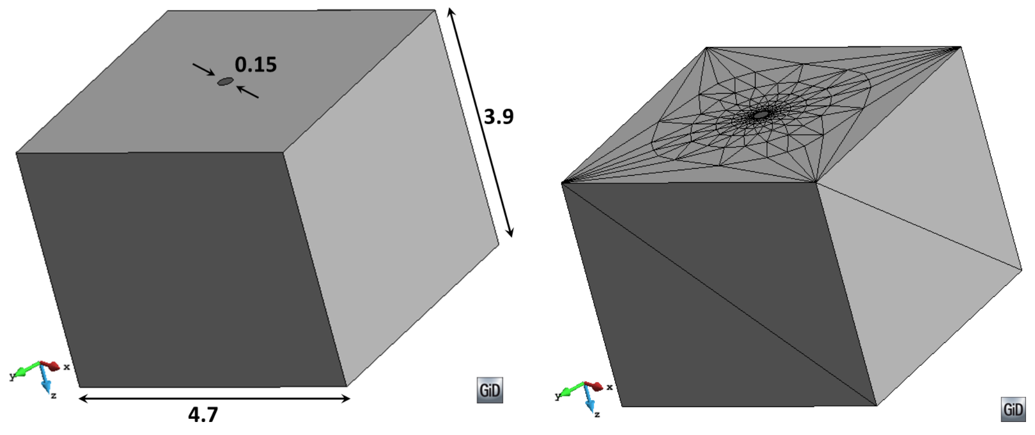
7.1.3 Non-watertight boundaries

Some validation examples are presented in this section with non-watertight volumes. When dealing with non-watertight geometries, it is not guaranteed that any edge of the mesh or any ray of the ray casting operations passes through a gap or a region with overlapping entities. To validate the algorithm, the models used in the first two examples have been created ensuring that some of these situations happens.

It has to be considered that these examples cannot be meshed using advancing front or Delaunay based techniques, as they require a watertight definition of the boundaries of the domain. In these cases, an extra effort must be invested in the CAD cleaning operations, or even to re-gerenate the mesh of the boundary surfaces before applying the volume mesher.

Validation example $VE - W1$

This example consists in a cubic volume with a gap in one of its faces. A view of its geometry with its characteristic sizes and the triangle mesh used as input for the mesher is shown in Figure 7.17.



(a) (b)

Figure 7.17: Model used in the validation example $VE - W1$. **(a)** View of the geometry of the volume and its characteristic sizes in uol. **(b)** View of the triangle mesh used as the input boundaries for the volume mesher.

The position of the gap (centered in the face) ensures that some of the rays used in the ray casting technique, as well as some tetrahedron edge involved in the surface fitting process passes through it.

The model has been meshed with a uniform mesh size of 0.7 uol and a TF equal to 1. It has

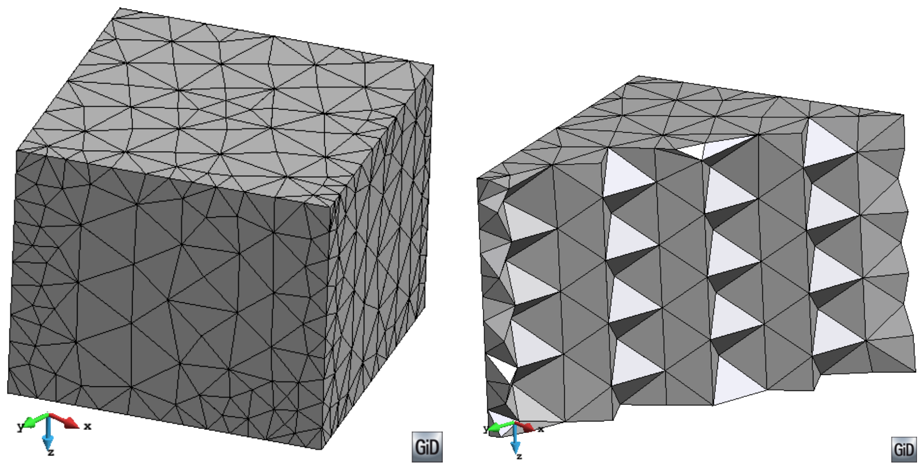


Figure 7.18: Results for the validation example $VE - W1$. **(a)** Final tetrahedra mesh generated. **(b)** View of internal elements of the final mesh in the region where the gap in the input boundary is.

to be noted that, when dealing with input boundaries with gaps, the mesh size in the region of gap must be greater than the gap itself, otherwise the mesher could fail.

The tetrahedra mesh generated is depicted in Figure 7.18. It can be appreciated that the skin of the mesh is completely watertight, closing the gap existing in the input boundaries. In the coloring process, the Cartesian ray passing through the gap has been considered as invalid, but the contributions of the other ones have provided with the right information in order to color all the nodes. In the surface fitting process some GIPs (Section 3.4.1) have been generated, as some of the edges of the tetrahedra mesh passed through the gap.

The data of the generated mesh is detailed in Table 7.7.

Number of threads	1
Mesh general size (uol)	0.7
Transition factor	1.0
Number of tetrahedra	3,823
Number of nodes	978
Minimum dihedral angle (degrees)	5.9
Time to generate the mesh (seconds)	0.1
Speed (Mtetrahedra per minute)	2.6

Table 7.7: Data of the validation example $VE - W1$.

Validation example $VE - W2$

This validation example aims to evaluating the mesher in non-watertight geometries having overlapping entities in its contours.

For this purpose, a model of an sphere has been build from two different skin meshes of the same sphere with a slight difference in its mesh size. Some elements of the meshes have been deleted, and then both triangle meshes have been merged.

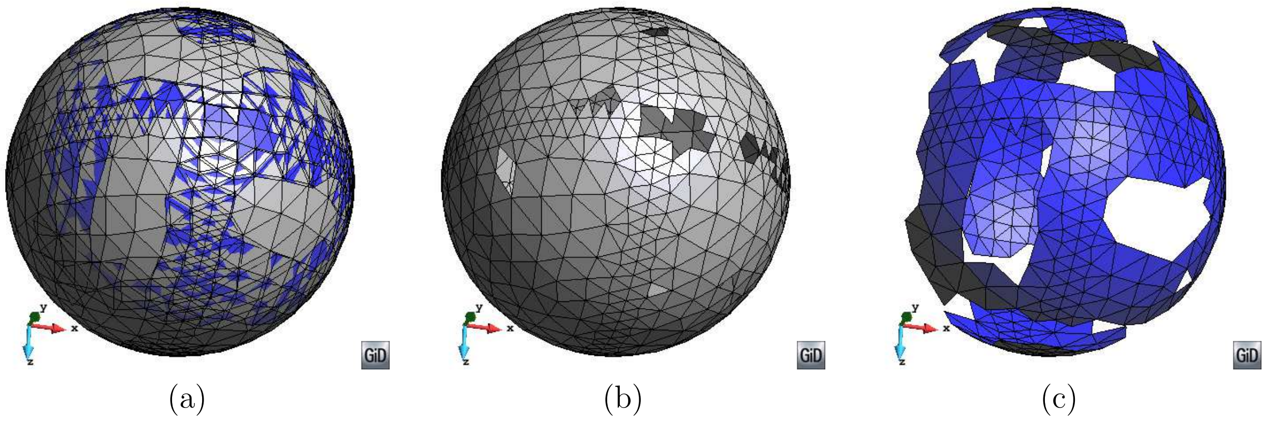


Figure 7.19: Model used in the validation example $VE - W2$. **(a)** The input boundaries are built merging two different parts of triangles meshes of a sphere (shown in **(b)** and **(c)**), so it has overlapping surface entities.

The two different meshes used (depicted in different colors), as well as the resulting boundary of the sphere are shown in Figure 7.19. It can be appreciated the overlapping regions in some parts of the boundaries. Actually, there are regions where one of the sphere skins is in the outer part of the domain, and regions were it is in the inner part. It has to be noted that the two skin meshes are not intersected: they have no topological relationship although they are occupying the same position in the 3D space.

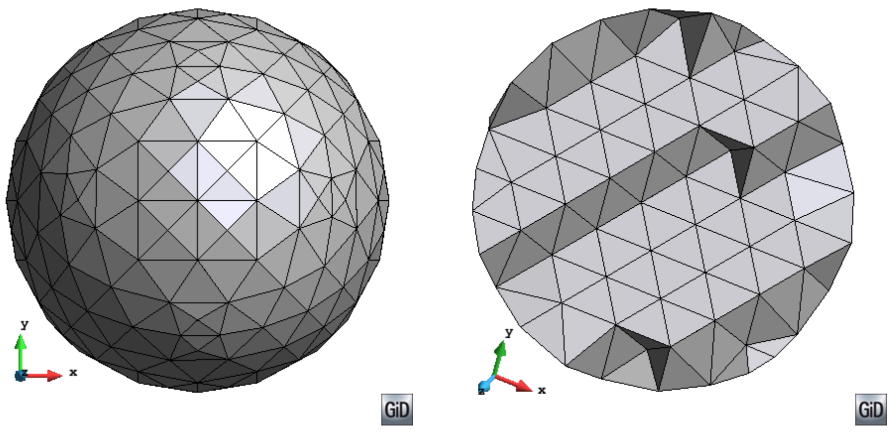
The radius of the sphere is 8.5 uol, and the maximum distance between overlapping entities is around 0.05 uol. The mesh has been generated with a uniform size of 1.0 uol.

The tetrahedra mesh generated is depicted in Figure 7.20. It can be appreciated that the skin of the mesh is completely watertight and enclose the volume of the sphere correctly.

The data of the generated mesh is detailed in Table 7.8.

Number of threads	1
Mesh general size (uol)	1.0
Transition factor	1.0
Number of tetrahedra	11,946
Number of nodes	2,557
Minimum dihedral angle (degrees)	3.1
Number of tetrahedra with minimum dihedral angle lower than 5 degrees	2
Time to generate the mesh (seconds)	0.36
Speed (Mtetrahedra per minute)	2.0

Table 7.8: Data of the validation example $VE - W2$.



(a) (b)

Figure 7.20: Results for the validation example $VE - W2$. **(a)** Final tetrahedral mesh. **(b)** View of internal elements of the mesh.

Validation example $VE - W3$

The model used for this validation example is a mechanical part (one volume) provided by the company *Quantech ATZ* [Qua14]. Its definition comes directly from a CAD system in mesh format (.stl) as it is provided in some real industry applications. As it is common in these cases, the input boundaries are non-watertight, although the geometrical definition (with NURBS surfaces) was watertight in the original CAD system used to define its shape. It is a very common case which illustrates the need of CAD cleaning operations before the meshing process when a conventional volume mesher is used. This example evidences that this meshing algorithm works without the need of performing any CAD cleaning operation.

A view of the model is depicted in Figure 7.21. It can be seen that some of the triangles

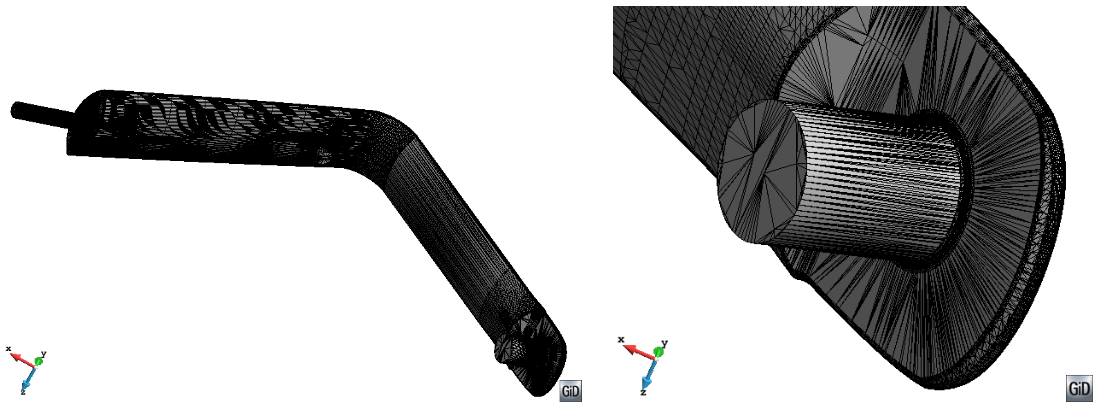


Figure 7.21: Model used in the validation example $VE-W3$. **(a)** Render view of the contours of the volume. **(b)** Zoom of a part of the contours with the sharp edges to be preserved.

used for the definition of the boundaries have a very bad aspect ratio. However, the shape of the volume is represented very well (with very low chordal error). The model boundary is smooth, but some parts of it (the cylindrical parts) present sharp edges to be preserved, as shown in Figure 7.21(b). The largest size of the bounding box of the model is around 200 uol, and the thin parts of it measure around 2 uol. The general desired mesh size used is 1 uol.

Defining the *higher entity* index of an edge as the number of triangles it belongs to, the present model has some edge with higher entity index equal to one and four. It has to be noted that a watertight volume must have all the edges of the triangles defining its contour with a higher entity index equal to two. This indicates that the model used in this example is not watertight. The edges with a value equal to one are related to a gap in the contours.

The higher entity index of the contours of the model is depicted in Figure 7.22. It can be appreciated that the non-watertight parts of the model are localized in very small regions. This evidences the difficulty of the CAD cleaning operations when a conventional mesher is used: apart from the difficulty to close the gaps of the boundary or edit it to avoid overlapping entities, it may be also difficult to detect the problematic regions. Furthermore, the quality of the triangles representing the input boundaries is very bad, so a new contour mesh should be generated for some standard meshing techniques, such as the advancing front.

The edges of the input boundaries selected to be preserved are the ones enclosing a dihedral angle lower than 95 degrees. Actually, not all of them have been considered, because some of the triangles defining the domain are so distorted that it is not possible to compute accurately its normal (this aspect is more detailed in Section 5.3.1). The edges considered to be preserved

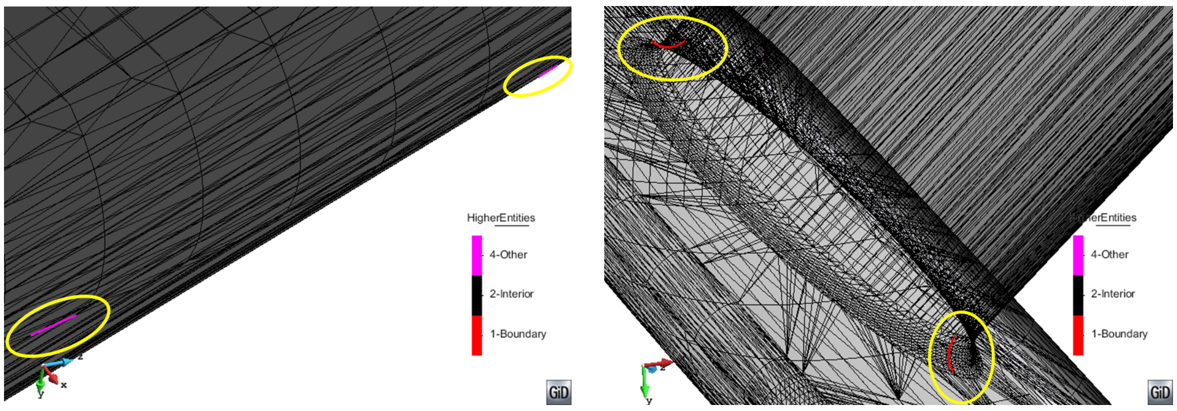
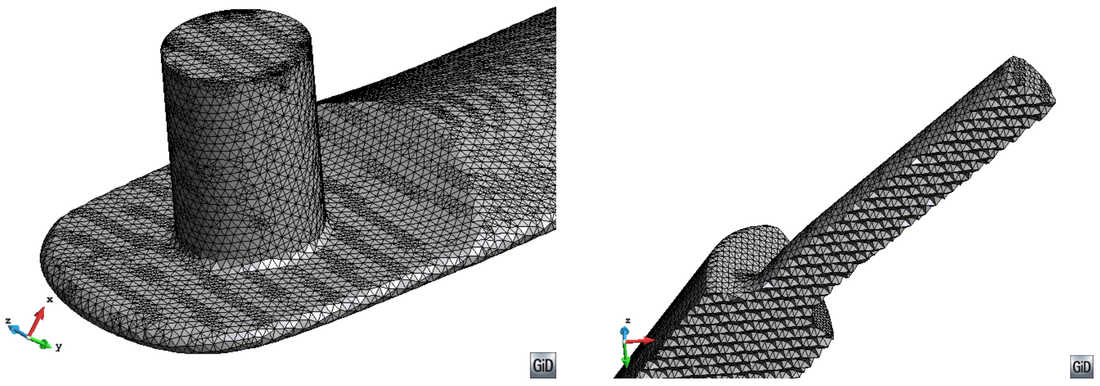


Figure 7.22: Higher entities indices of the edges of the contours defining the model of $VE{-}W3$. Two zoom views of some of the edges with higher entity index different from two.

are the ones of the top of the small cylindrical parts of the volume (Figure 7.21(b)).

A mesh of 1 uol uniform size has been generated. The data of the generated mesh is detailed in Table 7.9, and some views of it are shown in Figure 7.23.



(a) (b)

Figure 7.23: Mesh generated of the model of $VE - W3$. **(a)** Zoom view of a part of the model. **(b)** Zoom of a selection of inner tetrahedra.

7.1.4 Coloring pathological situations

Some validation examples presenting the pathological situations for the ray casting technique detailed in Section 4.2 are shown in this section. The models used are prepared ensuring some of the rays used in the ray casting technique for the nodes coloring intersects the input boundaries in special intersection types (T, P, M or W) creating a MIP (Section 3.4).

Number of threads	1
Mesh general size (uol)	1.0
Transition factor	0.7
Number of tetrahedra	1,045,878
Number of nodes	214,671
Minimum dihedral angle (degrees)	3.6
Number of tetrahedra with minimum dihedral angle lower than 5 degrees	3
Time to generate the mesh (seconds)	65.6
Speed (Mtetrahedra per minute)	1.0

Table 7.9: Data of the validation example $VE - W3$.

Validation example $VE - C1$

The validation example $VE - C1$ is created repeating a symmetric configuration of volumes ensuring some of the rays of the ray casting process intersects the input boundaries in a MIP with multiple interfaces (M type intersection) and in co-planar intersections (P type intersection).

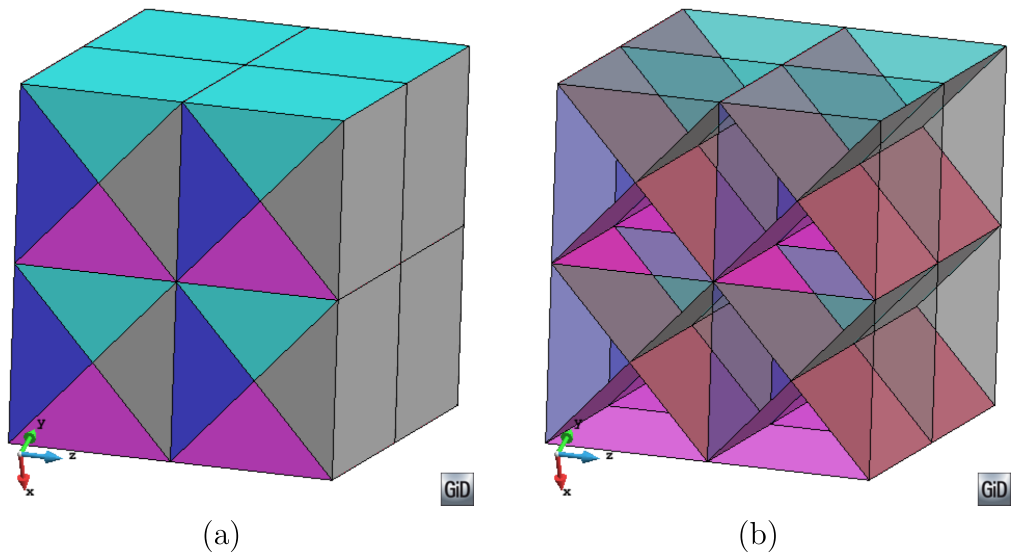
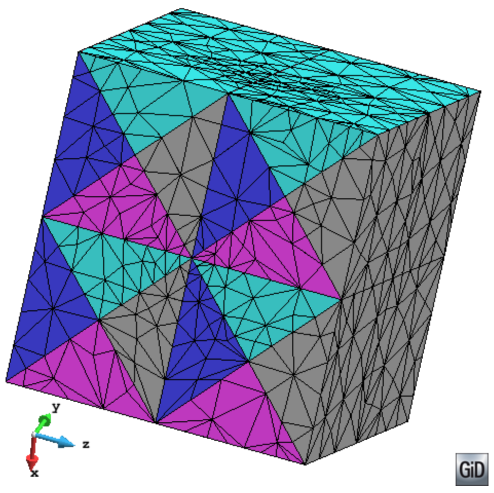
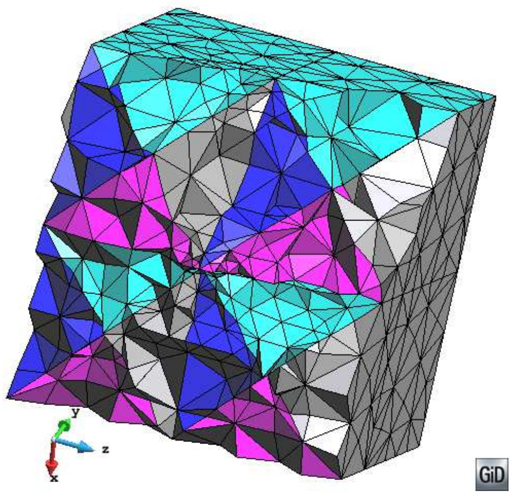


Figure 7.24: Model used in the validation example $VE - C1$. **(a)** Render view of the contours of the volumes. **(b)** Transparent view of the external contours of the model in order to appreciate the inner part of it.

The model is shown in Figure 7.24. It is formed by 32 watertight volumes sharing the contact surfaces between them. The side of the bounding box of the model has 11.3 uol.



(a)



(b)

Figure 7.25: Results for the validation example $VE - C1$. **(a)** Final tetrahedral mesh. **(b)** View of internal elements in the final mesh.

The tetrahedra mesh generated using a general mesh size of 2.0 uol is depicted in Figure 7.25. For the node coloring process, no ray has been declared as invalid, so the strategies used for detecting and solving the pathological configurations of rays have been demonstrated to work properly. In particular, M intersection types have been involved in this model. This kind of intersections forces to create extra rays from the multiple interface intersection, so more rays are used compared to the single intersection types.

It can also be appreciated that the topology of the model has been correctly represented by the final mesh, as each one of the volumes of the model has its tetrahedra mesh.

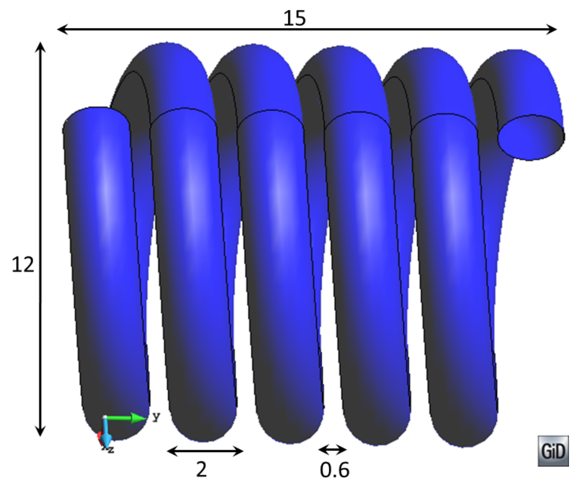
The data of the generated mesh is detailed in Table 7.10.

Number of threads	1
Mesh general size (uol)	2.0
Transition factor	0.7
Number of tetrahedra	49,823
Number of nodes	10,129
Minimum dihedral angle (degrees)	5.3
Time to generate the mesh (seconds)	1.8
Speed (Mtetrahedra per minute)	1.7

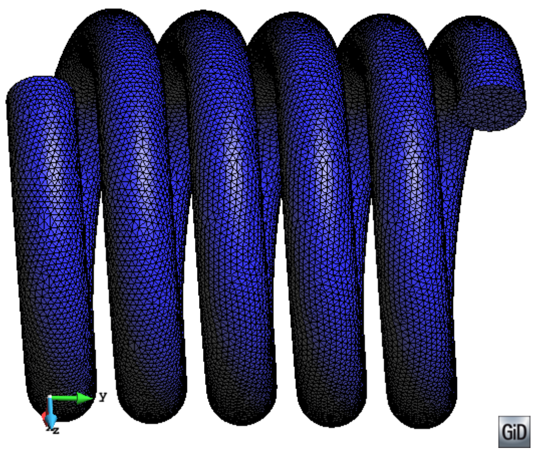
Table 7.10: Data of the validation example $VE - C1$.

Validation example $VE - C2$

A coil shaped volume has been used for this example. The model is formed by the watertight volume shown in Figure 7.26, where its characteristic sizes are depicted.



(a)



(b)

Figure 7.26: Model used in the validation example $VE - C2$. **(a)** View of the geometry of the volume and its characteristic sizes in uol. **(b)** View of the triangle mesh used as the input boundaries for the volume mesher.

A large amount of rays used for the nodes coloring of this model intersect tangentially the input boundaries, so it is a suitable validation example to check if the T intersection types are well detected and solved.

This model is also interesting from the topology preservation point of view, as the same volume is curved, and some of its parts are very near one from each other. This situation is problematic for many octree based meshers, because they tend to join the parts of the input boundaries which are very close one from each other. This is the case of the mesh depicted in Figure 7.28.

The tetrahedra mesh generated using the proposed mesher with a desired mesh size of 1 uol is shown in Figure 7.27. It can be appreciated that all the nodes are colored correctly. Furthermore, the topology of the model is also well captured, thanks to the combination of octree refinement, surface fitting and tetrahedra coloring strategies followed by the presented algorithm.

As in other examples presented, the mesh size chosen in this example is set in order to highlight some feature of the mesher (in this case, the preservation of the topology), but it

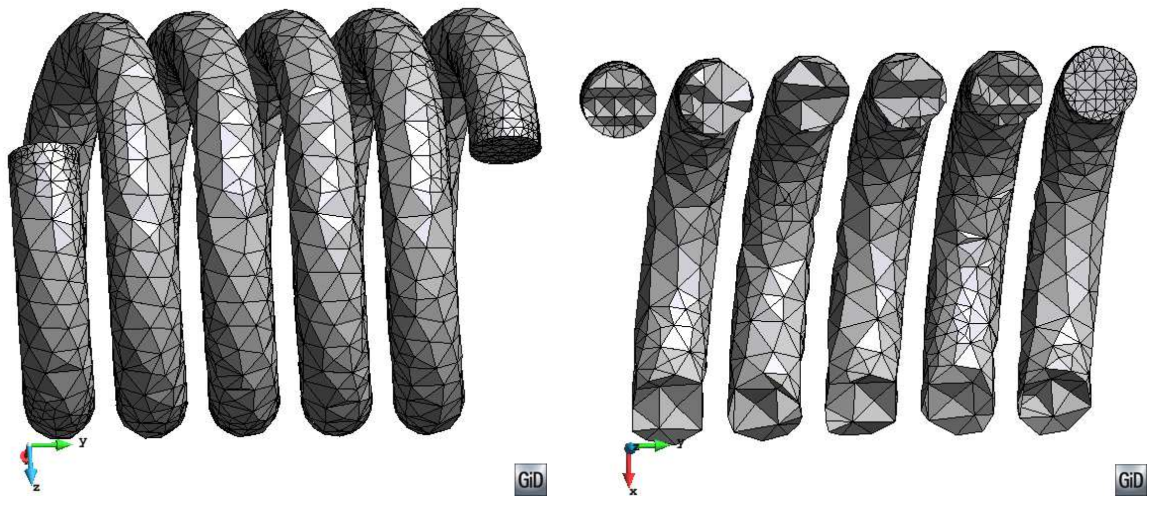


Figure 7.27: Results for the validation example $VE - C2$. **(a)** Final tetrahedral mesh. **(b)** View of internal elements in the final mesh.

yields a mesh with a high chordal error in the boundaries due to their curvature. This problem can be solved by just assigning a lower size in the boundaries, as it is shown in Figure 7.29(b). Some specific make-up and smoothing operations could be also applied to the coarse mesh in order to reduce its chordal error.

The data of the generated mesh is detailed in Table 7.11.

Number of threads	1
Mesh general size (uol)	1.0
Transition factor	1.0
Number of tetrahedra	19,512
Number of nodes	5,888
Minimum dihedral angle (degrees)	5.0
Time to generate the mesh (seconds)	3.5
Speed (Mtetrahedra per minute)	0.3

Table 7.11: Data of the validation example $VE - C2$.

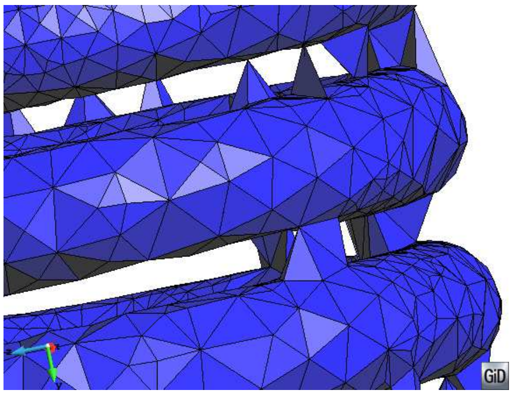
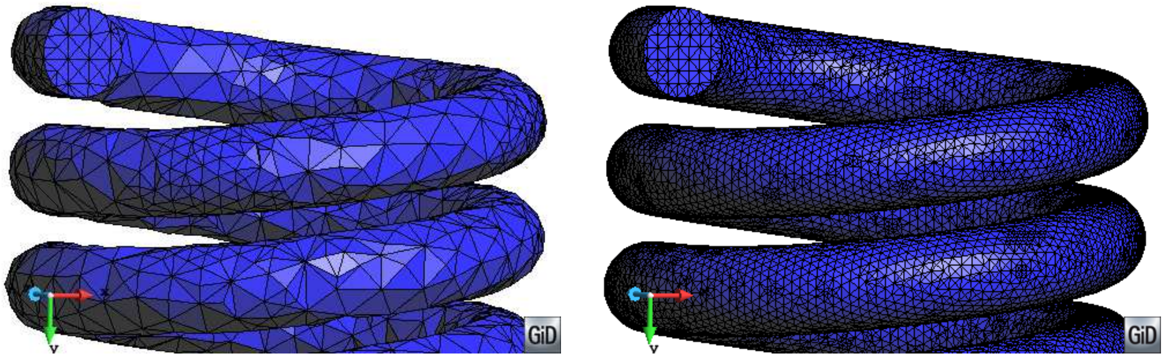


Figure 7.28: View of a mesh generated with a conventional octree based mesher not preserving the topology of the model.



(a) (b)

Figure 7.29: Zoom view of two meshes of the validation example $VE-C2$: **(a)** with a general mesh size of 1.0 uol, and **(b)** with a mesh size of 0.3 uol assigned to the input surface entities. It can be appreciated the lower chordal error using a lower mesh size.

Validation example $VE - C3$

This validation example has been created to evidence the local ray casting technique is necessary. This technique is applied when the three Cartesian rays passing through a node are invalid. The model is a cube of 4.7 uol of side which faces are represented by unconnected triangles in the configuration depicted in Figure 7.30. The gap size in the center part of the faces is approximately of 0.25 uol.

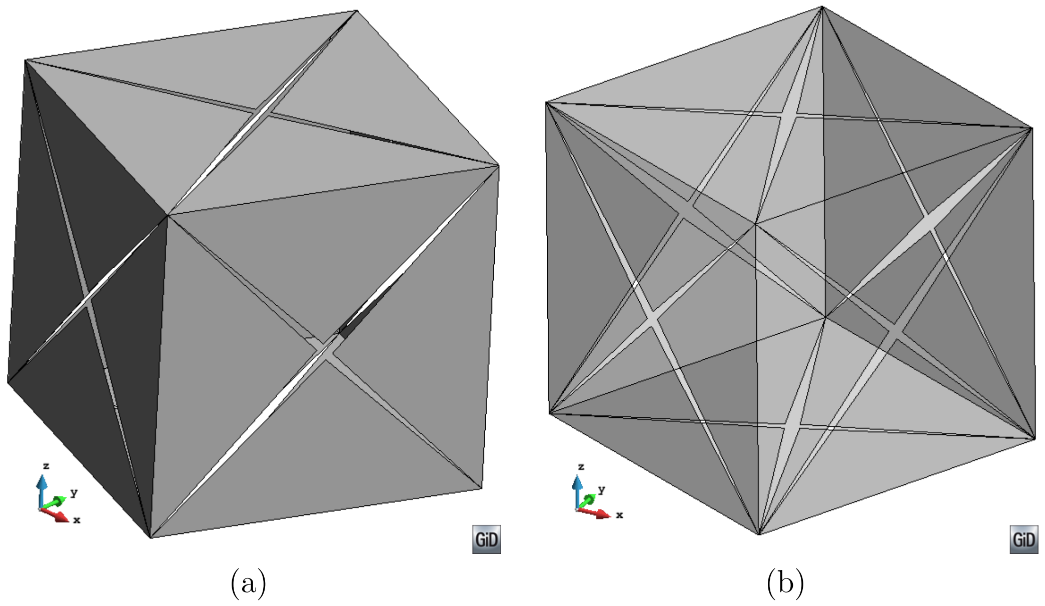


Figure 7.30: Model used in the validation example $VE - C3$. **(a)** Render view of the triangle mesh used as input boundary for the mesher. **(b)** Transparent view in order to appreciate the gaps in the rear faces of the volume.

This configuration ensures the node placed in the center of the cube will have its three rays invalid, so its coloring must be made by the local ray casting technique.

The tetrahedra mesh generated with a mesh size of 1 uol is depicted in Figure 7.31. It can be appreciated that the mesh has been generated successfully, providing with a watertight skin of the tetrahedra. The color of the central node of the cube has been provided using the local ray casting technique, as the three rays passing through it were considered as invalid.

The data of the generated mesh is detailed in Table 7.12.

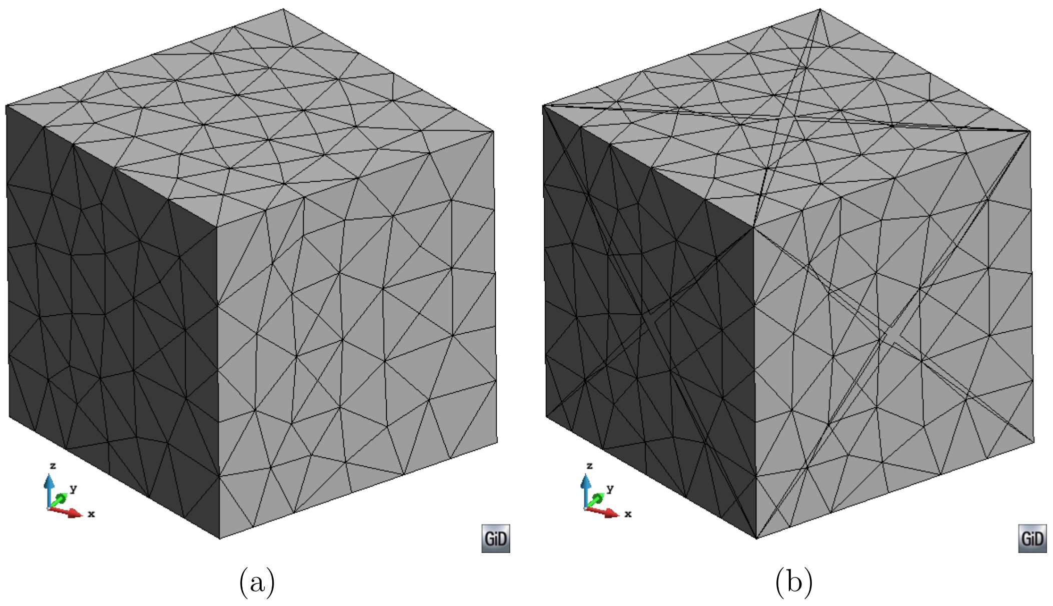


Figure 7.31: Results for the validation example $VE - C3$. **(a)** View of the final tetrahedral mesh. **(b)** The same view of the tetrahedra mesh with the input boundary mesh superposed.

Number of threads	1
Mesh general size (uol)	1.0
Transition factor	1.0
Number of tetrahedra	1,804
Number of nodes	541
Minimum dihedral angle (degrees)	10.9
Time to generate the mesh (seconds)	0.07
Speed (Mtetrahedra per minute)	1.5

Table 7.12: Data of the validation example $VE - C3$.

7.1.5 Surface mesh inner to a volume

In this section an example corresponding to the meshing of surfaces inner to a volume is shown (Section 5.3.10).

Validation example $VE - I1$

This validation example corresponds to a 420 dinghy boat sails set (main, jib and spinnaker). The example includes the mast and other line elements such as the jib halyard, the spinnaker pole and the topping lift. Other elements of the rig such as the shrouds and spreaders have been omitted.

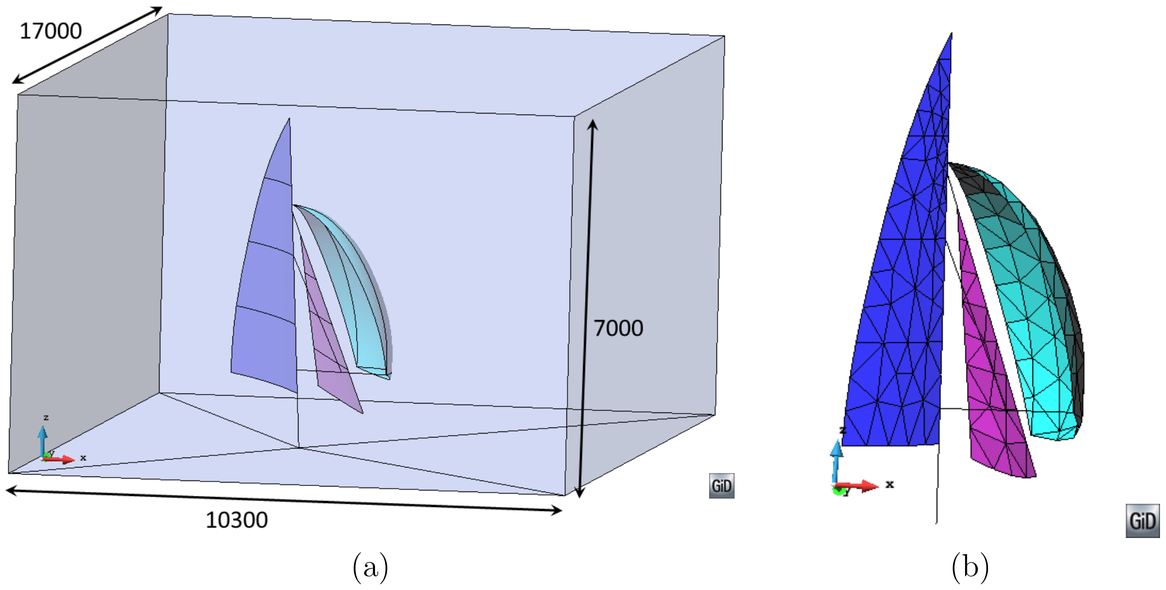


Figure 7.32: Model used in the validation example $VE - I1$. **(a)** Render view of the model with transparencies in the outer surfaces of the control volume. Characteristic sizes of the model are depicted in uol. **(b)** Zoom view of the triangle and linear meshes (used as input boundaries for the mesher) of the sails and the inner line entities to be preserved.

From the mesher point of view, the interest of this model is to reach a volume mesh surrounding the sails, and topologically connected to them. Furthermore, the tetrahedra mesh must be also conformal to the line elements (as the mast) present in the model. This means some of the tetrahedra faces should be triangles of the sails, and some of the tetrahedra edges must be 1D linear elements of the model. For this reason a control volume has been built around the model. A view of the model is depicted in Figure 7.32. From the topological point

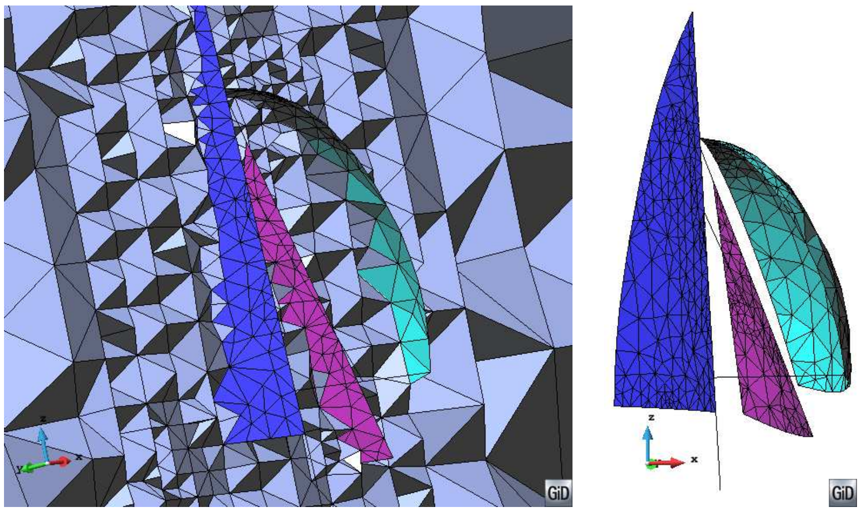


Figure 7.33: Results for the validation example $VE-I1$. **(a)** View of some internal tetrahedra with conformal to the triangle mesh of the sails. **(b)** Triangle and linear meshes of the model.

of view it is interesting to note that the surface entities representing the sails and the line entities representing the other elements are not part of the boundaries of the control volume. They are inside the volume, but with no topological connection to it.

The mesh has been generated with 5000 uol as general mesh size, and no specific size assigned to the boundaries. Two different views of the mesh generated are depicted in Figure 7.33. A view of some internal tetrahedra and the triangles of the sails is shown in Figure 7.33(a), where it can be appreciated that the tetrahedra are conformal with the triangles of the sails. A view of the triangle and linear elements of the model is depicted in Figure 7.33(b).

The data of the generated mesh of this validation example is detailed in Table 7.13.

Number of threads	1
Mesh general size (uol)	5,000
Transition factor	0.7
Number of tetrahedra	21,630
Number of nodes	4,156
Minimum dihedral angle (degrees)	5.2
Time to generate the mesh (seconds)	0.38
Speed (Mtetrahedra per minute)	3.5

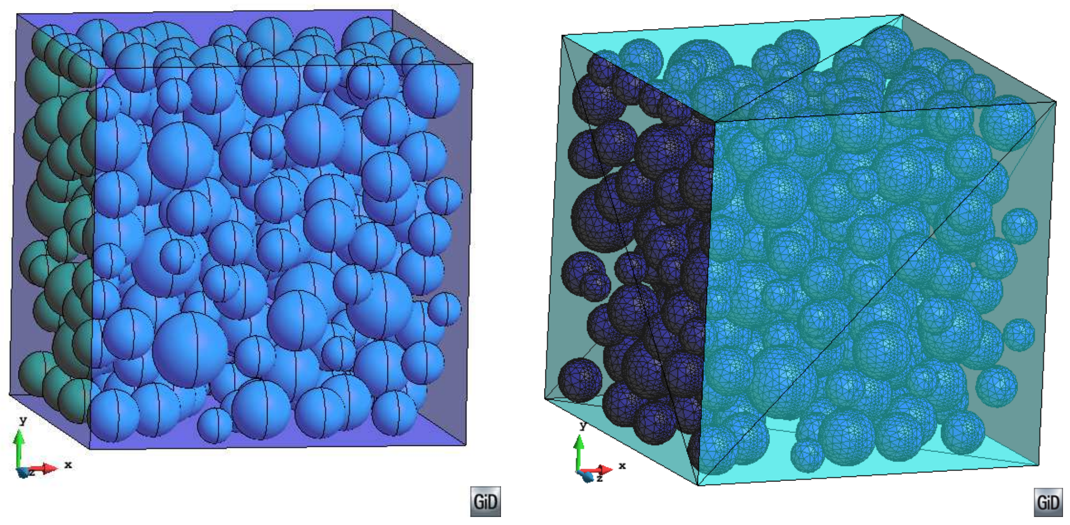
Table 7.13: Data of the validation example $VE - I1$.

7.1.6 Embedded mesh

In this section, a validation example of an embedded mesh is presented. As explained in previous sections, an embedded mesh is not body fitted, so its generation is much more faster.

Validation example $VE - E1$

Validation example $VE - E1$ is a representative volume element (RVE) of a synthetic porous material. A view of the model is depicted in Figure 7.34.



(a) (b)

Figure 7.34: Model used in the validation example $VE - E1$. **(a)** View of the geometry of the model with transparencies in its external surfaces in order to appreciate the internal voids of it. **(b** View of the input mesh used for the mesher.

The model is a watertight volume with spherical voids (holes) representing the porous. The voids are made with randomnly placed spheres of different diameters. The side of the cube is 10 uol, and the diameters of the spheres goes from 0.9 to 2.0 uol. The minimum distance between the spheres and the boundaries of the cube (or between different spheres) is around 0.2 uol. In order to capture better the interafces between the voids and the material, a size of 0.1 uol has been assigned to the input boundaries, and a general mesh size of 0.2 uol has been assigned to the hole model. The TF is equal to 1.

The triangle mesh used as the input boundaries for the mesher is shown in Figure 7.34(b).

It can be appreciated that the spherical voids are represented with a fine mesh in order to capture well their shape, as the external surfaces of the cube are represented only by two triangles each one of them. As they are planar, its shape is perfectly captured in this way.

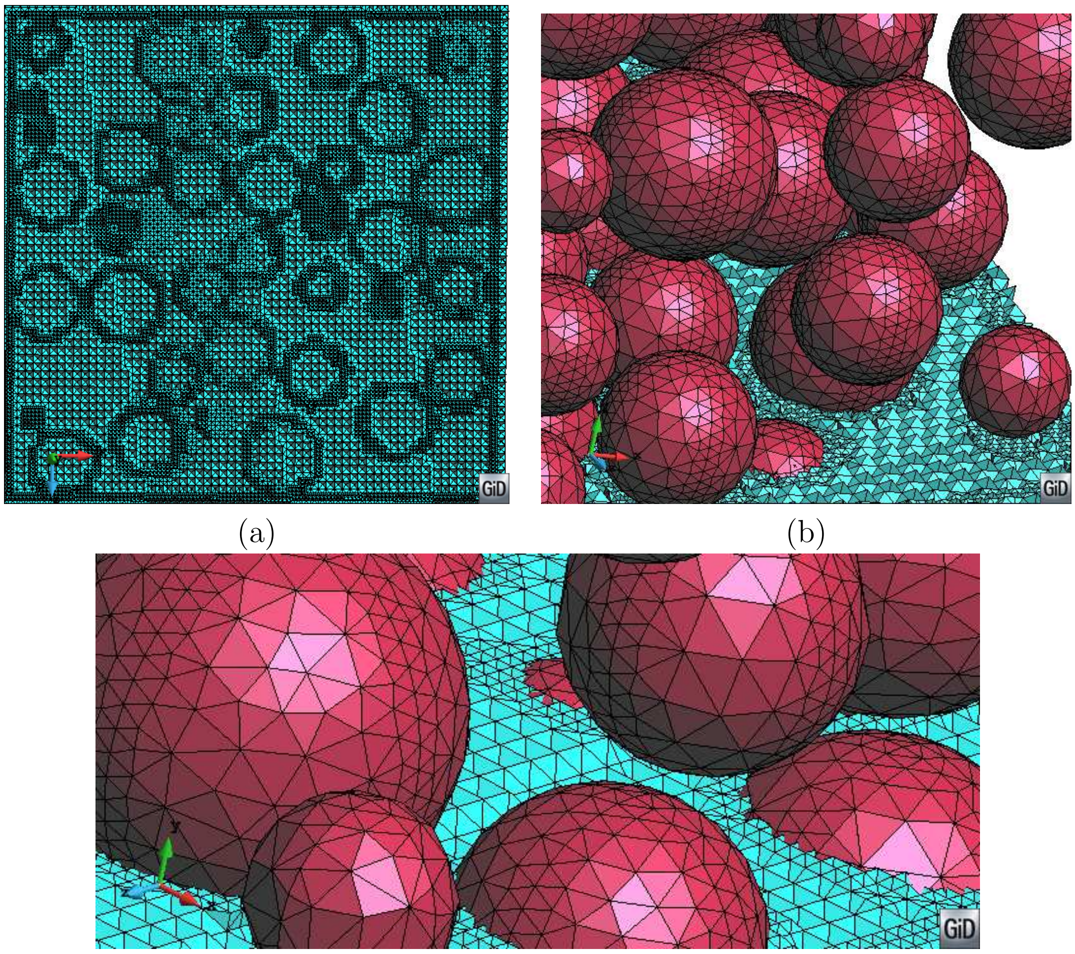

(a)
(b)
(c)

Figure 7.35: Results for the validation example $VE-E1$. **(a)** View of the inner part of the final tetrahedra mesh. **(b)** and **(c)** Details of the inner part of the final tetrahedra mesh together with the triangle mesh used as the input boundaries.

A view of the inner part of the generated mesh is shown in Figure 7.35. As it is a non body-fitted mesh, all the tetrahedra come directly from the tetrahedra pattern applied to the octree cells, with no distortion. The level of refinement near the input boundaries due to the smaller desired size applied onto it can be appreciated. In Figure 7.35(b) a detail of the mesh is depicted together with the input mesh for the mesher. It can be seen that the tetrahedra does not fit with the input boundaries.

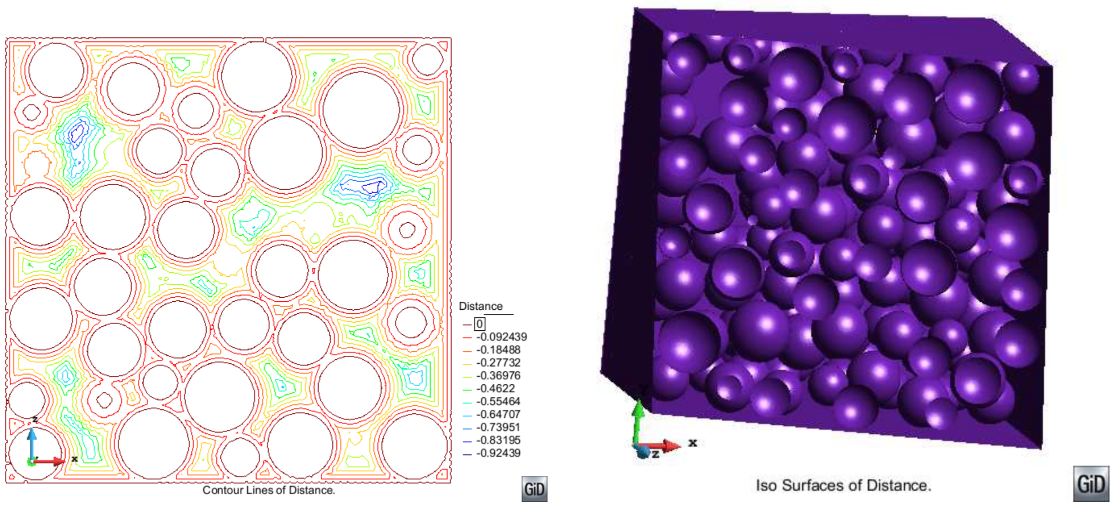


Figure 7.36: Results for the validation example $VE - E1$. **(a)** View of a cut of the model with the isolines of distance. **(b)** View of the iso-surface of distance 0 in the inner part of the volume.

A view of the isolines of distance within a cut of the tetrahedra mesh is shown in Figure 7.36(a), and an inner view of the iso-surface of distance equal to 0 is shown in Figure 7.36(b). It can be appreciated that the distances on the nodes are well computed.

The data of the generated mesh is detailed in Table 7.14.

Number of threads	1
Mesh general size (uol)	0.2
Mesh size in input boundaries (uol)	0.1
Transition factor	1.0
Number of tetrahedra	11,511,840
Number of nodes	1,967,445
Time to generate the mesh (seconds)	17.6
Speed (Mtetrahedra per minute)	39.3

Table 7.14: Data of the validation example $VE - E1$.

7.1.7 Performance

The validation examples in this section aim to evaluating the efficiency inofhe implementation of the algorithm. For this purpose several meshes of the same models are generated with different sizes in order to get an idea of the performance of the mesher.

It has to be considered that each model has its own particularities, making difficult to extrapolate the speed of the mesher from a model to another. However, knowing the characteristics of the model can give us an idea of the behavior of the mesher for other ones.

As some parts of the algorithm have been implemented using shared memory parallelism, the examples in this section are run with different number of threads (from 1 to 4) in order to evaluate its scalability.

For sake of the performance analysis, the following main parts of the algorithm are distinguished:

- *Refinement.* This relates to all the octree refinement processes and the operations they involve. In case of topology refinement criteria, for instance, the subdivision of a cell implies the creation of the linear octree nodes of its children, as well as their coloring and the determination of their closest point in boundary in case they come from an interface cell.
- *Nodes and tetrahedra generation.* This includes three main parts:
  - The creation of the linear octree nodes just before the coloring process. In this step almost all the octree nodes are created, except the ones created during the topology refinement criteria. The creation of an octree node implies:
    - The creation and storage of the node itself.
    - The assignation of the corresponding octree position in the cells surrounding the node.
    - The search and assignment of its closest position onto the input boundaries. This is done only to the nodes belonging to an interface cell.
  - Creation of the tetrahedra from the interface cells applying the tetrahedra pattern. Within the creation of these tetrahedra, the information of the tetrahedra surrounding each node is also stored.
  - Creation of tetrahedra from the inner cells following the tetrahedra pattern. These tetrahedra are created at the end of the meshing process, as they are directly

part of the final mesh (no mesh edition is performed on them). In this case no neighboring information is stored in the nodes.

- *Coloring.* This part corresponds to the node coloring process, following the ray casting technique (Section 4). It has to be considered that the majority of the nodes are colored in this step, but not all of them. The coloring of the octree nodes created during the topology refinement criteria is performed inside the refinement part.
- *Surface fitting.* This part corresponds exactly to the surface fitting process defined in Section 5.3.6.
- *Tetrahedra coloring.* All the tetrahedra coloring operations, as well as the creation of skin triangles of tetrahedra of the same color, are included in this part. It has to be noted that the process of coloring an undetermined tetrahedra may involve the coloring of some positions in space (points onto some tetrahedra face).
- *Make-up and smoothing.* This part consists in the following processes:
  - Collapse of small edges performed just after the surface fitting process.
  - Tetrahedra swapping.
  - Smoothing process involving movement of nodes.
- *Others.* All the other processes of the meshing algorithm not present in the previous ones are included in this part. It is the case, for instance, of the operations related to the preservation of geometric features. It is not treated apart because it is really low time consuming.

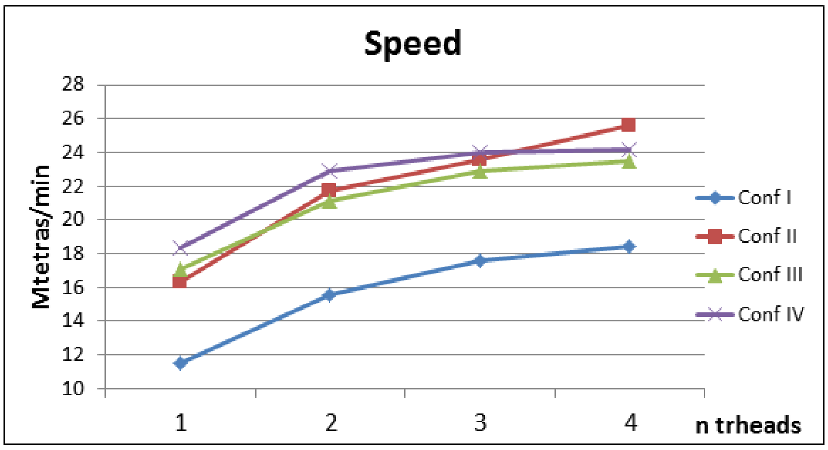
Validation example $VE-S1$

The most favorable case for the presented mesher is a model as much massive as possible (maximum sphericity), watertight, with no geometrical features to be preserved, and with a uniform mesh size. The validation example $VE-S1$ is a sphere of 10 uol of diameter, which fits with these characteristics. The results of this validation test case should provide with an upper limit for assessing the performance of the mesher.

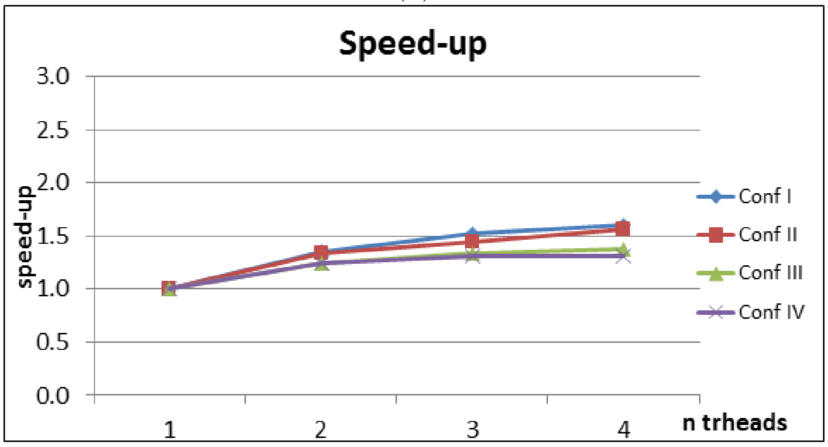
Four different configurations of the sphere have been run, corresponding to 4 different uniform mesh sizes. Each one of the configurations has used a triangle mesh of the same size as the one of the volume as the input boundary.

The minimum dihedral angle of the tetrahedra generated is higher than 15.0 degrees in all the configurations.

All the configurations have been run using 1, 2, 3 and 4 threads. The results of speed and speed-up for each configuration are depicted in Figure 7.37.



(a)



(b)

Figure 7.37: Graphs of **(a)** speed (in Mtetrahedra per minute) and speed-up **(b)** corresponding to the four configurations of validation example $VE - S1$.

The characteristics of each configuration are depicted in Table 7.15.

As it can be appreciated, in this example the mesher generates between 11 and 18 millions of tetrahedra per minute (using one thread), depending on the configuration. Increasing the number of threads results in a low speed-up, arriving at around 1.5 when 4 threads are used. The speed-up is far from the optimal one because of the implementation of the algorithm. The current implementation has been focused in parallelizing the naturally parallel parts of the

Configuration	I	II	III	IV
Mesh size	0.18	0.085	0.055	0.035
Number of tetrahedra (millions)	1.1	10.3	38.0	103.5
Number of nodes (millions)	0.2	1.8	6.4	17.5
Memory peak (Gb)	0.2	1.8	6.7	18.8
Memory ratio	6.8	8.1	8.2	8.5

Table 7.15: Data of the different configurations used for the validation example $VE - S1$.

algorithm. The other parts could be also implemented in parallel, reaching higher speed-up values.

The distribution of time consumed by the main parts of the mesher changes depending on the configuration. The reason for this is the node and tetrahedra generation part is basically proportional to the number of tetrahedra and nodes to be created. This is not the case of the other parts of the algorithm, as they are mostly applied only to the tetrahedra near the contours of the volume.

The distribution of computing times corresponding to configuration IV for the different number of threads used is depicted in Figure 7.38.

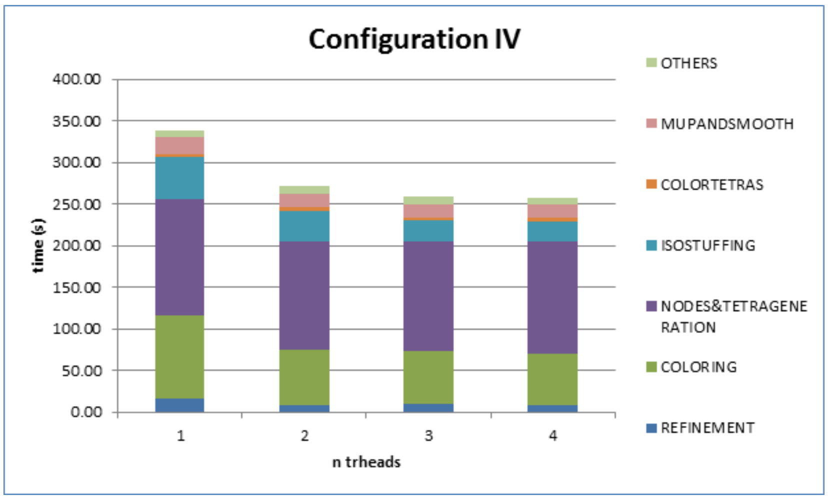


Figure 7.38: Time consumed in the meshing process of configuration IV of validation example $VE - S1$ detailed in the different parts of the algorithm.

It has to be noted that this example (a uniform meshed sphere) is quite particular as some of the main parts of the presented mesher are not representative. The tetrahedra coloring algorithm, for instance, is not applied in this case as there are no undetermined tetrahedra after the surface fitting process. Furthermore this is the model with maximum possible sphericity, so the majority of the tetrahedra (the inner ones) comes directly from applying of the tetrahedra patterns.

Validation example $VE - S2$

This validation example is a synthetic model of a city. A view of the model is depicted in Figure 7.39.

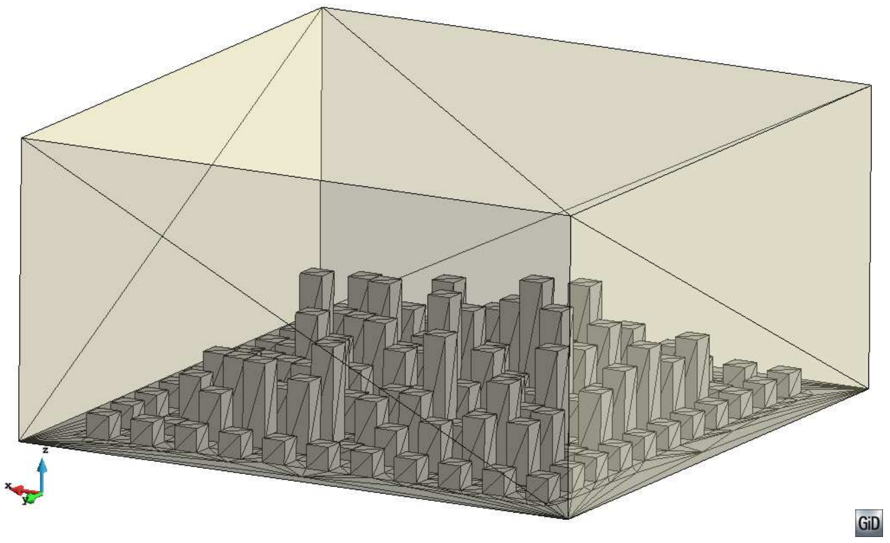


Figure 7.39: Model used in the validation example $VE - S2$.

The model basically consists in prismatic shapes representing buildings, connected to a planar terrain, and enclosed in a unique watertight control volume. The base of the control volume is 1000 by 1000 uol, and its height is 500 uol. All the buildings have a base of 40 by 40 uol (they are also separated this distance one from eachother) and their heights go from 20 to 200 uol. A zoom view of some of the buildings is shown in Figure 7.40.

This model is not so favorable for the mesher, as it contains sharp edges to be preserved, and it will not be meshed with a uniform size, so it could give a more realistic idea of the performance of the mesher applied to real models. The sphericity of the model is 0.54.

In this case also four configurations have been used to check the performance and scalability of the mesher. Each configuration has the same input boundary (shown in Figure 7.39(a)), made by 2036 triangles. A TF of 0.7 is used, and two sizes are assigned: one for the surface entities defining the terrain and the buildings, and a general mesh size for the rest of the domain.

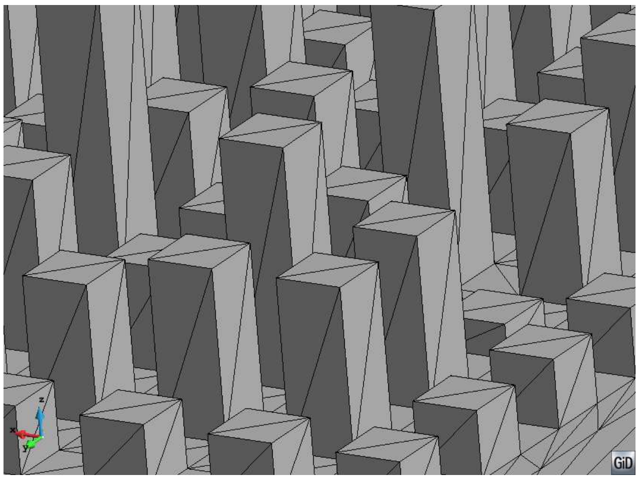


Figure 7.40: Zoom of the buildings of the model used in the validation example $VE - S2$.

The characteristics of each configuration are detailed in Table 7.16. The minimum dihedral angle of the tetrahedra generated is higher than 5 degrees in all the configurations.

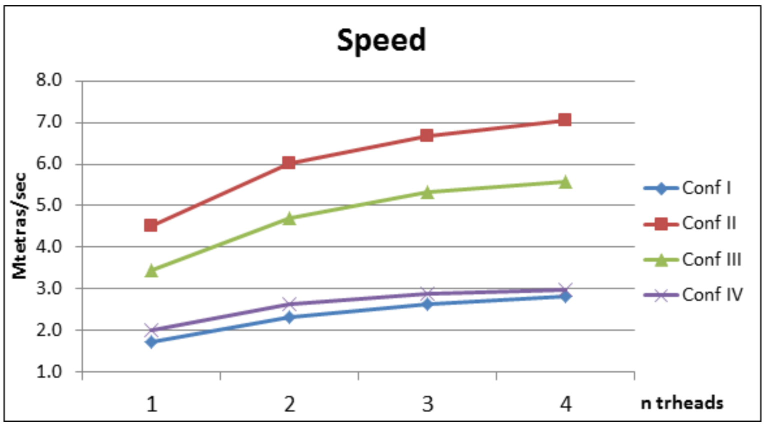
Configuration	I	II	III	IV
Mesh size in buildings	10	3	2	1
Mesh general size	50	10	10	5
Number of tetrahedra (millions)	1.12	14.6	28.1	84.8
Number of nodes (millions)	0.2	2.7	5.5	16.6
Memory peak (Gb)	0.5	3.2	7.1	25.5
Memory ratio	12.4	9.0	10.2	12.0

Table 7.16: Data of different configurations used for the validation example $VE - S2$.

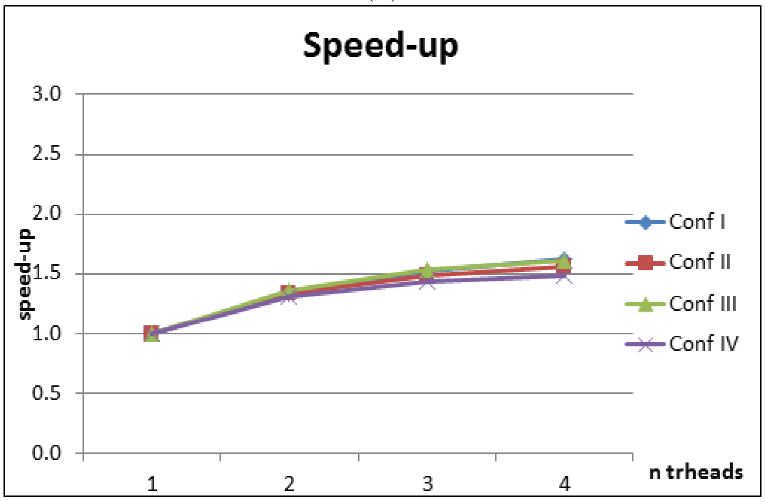
All the configurations have been run using 1, 2, 3 and 4 threads. The results of speed and speed-up for each configuration are depicted in Figure 7.41.

The results obtained in this example show that the mesher generates between 2 and 4.5 millions of tetrahedra per minute (using 1 thread), depending on the configuration. The speed-up reaches 1.6 using 4 threads.

The distribution of computing time in the different parts of the mesher is rather similar in the 4 configurations. The one corresponding to configuration IV using 1 thread is depicted in Figure 7.42. It can be seen that the most consuming parts of the algorithm in this example are the refinement, the mesh improvement operations (make-up and smoothing) and the surface fitting process. Considering the refinement part, more than the half of the time is devoted to



(a)



(b)

Figure 7.41: Graphs of **(a)** speed (in Mtetrahedra per minute) and speed-up **(b)** corresponding to the four configurations of validation example $VE - S2$.

the topology refinement criteria.

The speed-up curves of the most relevant parts of the algorithm are depicted in Figure 7.43. As it can be appreciated, none of them presents an optimal speed-up. This is the reason of the low speed-up reached considering the whole meshing process. The surface fitting process is the one with a higher speed-up, reaching almost 2.5 with 4 threads.

Some aspects of the implementation of the algorithm could be improved in order to reach a higher speed-up for the whole meshing process.

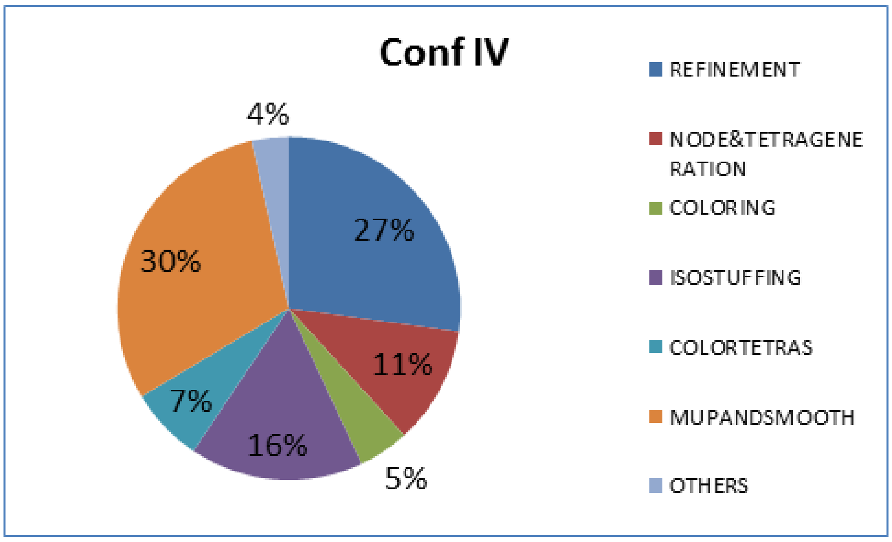


Figure 7.42: Percentages of time consumed in the meshing process of configuration IV of validation example $VE - S2$.

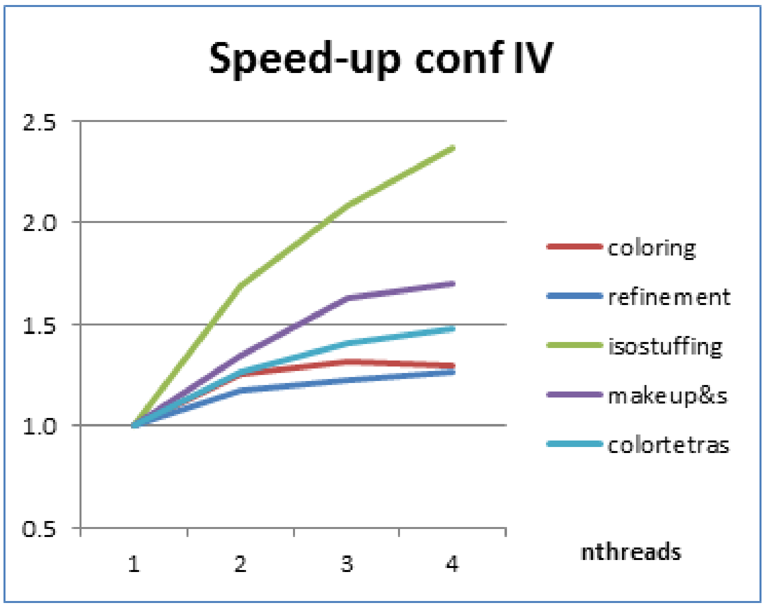


Figure 7.43: Speed-up of the main parts of the algorithm generating the mesh of validation example $VE - S2$.

7.2 Racing car example

The model used in this section corresponds to a racing car. Specifically, the part of interest for a CFD numerical simulation is the air surrounding the car. For this reason a synthetic wind tunnel (a control volume) has to be build around it. For a CFD simulation the control volume must be large enough to avoid any influence of the boundary conditions in the results. In this example only a small control volume has been build, considering the more interesting part from the meshing point of view is the region near its surfaces.

To run a CFD simulation on this model would imply also to generate a boundary layer mesh near the surfaces of the car and the floor. This is a specific kind of mesh with very thin elements in the normal direction of the flow in order to capture well the high velocity gradients present in that regions.

The generation of the boundary layer mesh is out of the scope of this work, but its requirements have been taken into account. There are two main families of boundary layer meshes:

- *A priori*: this approach generates first a boundary layer mesh from the input surface entities towards the inner part of the volume. After a given number of layers of elements, the contours of this mesh are used as the input boundaries to feed an isotropic volume mesher [GS98, AL09].

- *A posteriori*: in this case the isotropic mesh is generated first, and afterwards the boundary layer elements are generated from the contours of the isotropic mesh pushing towards the interior of the volume the existing tetrahedra [IN02]. As it implies the movement of nodes, typically a re-positioning of the nodes near the boundary layer mesh is needed.

As the presented mesher is not constrained (the volume mesh is not conformal to the input surface entities), a boundary layer mesh cannot be generated a priori, so a posteriori method should be used. In the presented example the boundary layer mesh has not been generated, as it is considered as a separated process, and is out of the scope of the thesis.

The model is watertight. Some figures of the geometry of the model with its characteristic sizes are shown in Figure 7.44. Views of the input surface mesh used are depicted in Figure 7.45. It can be appreciated that the quality of the input surface mesh is quite good, so it could be used probably as an input for an advancing front based mesher. For generating the volume mesh of this model using the presented octree mesher a worse quality mesh could be used to define the boundaries, saving the corresponding time on the surface meshing part.

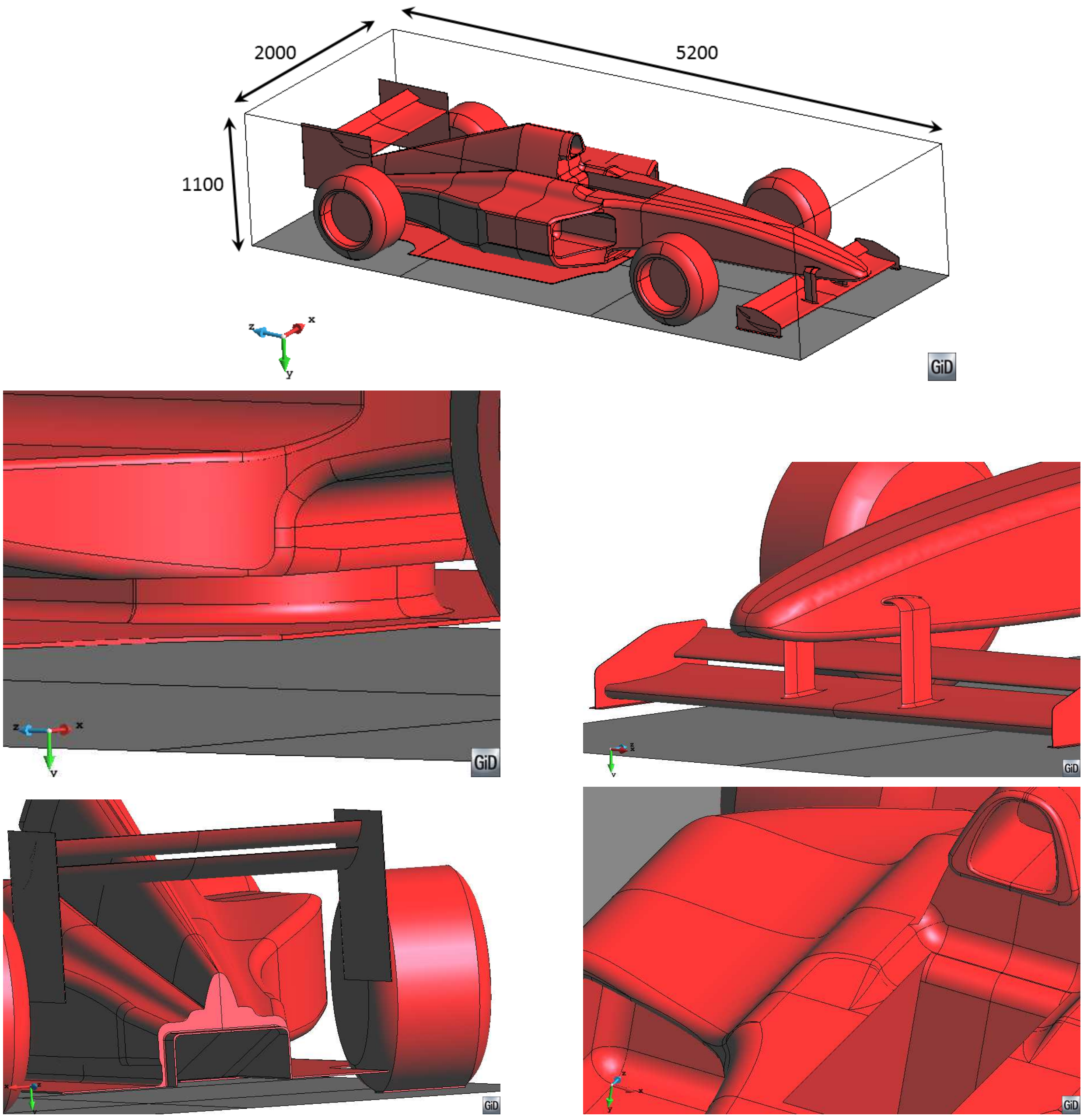


Figure 7.44: Different views of the racing car model to be meshed of in this section.

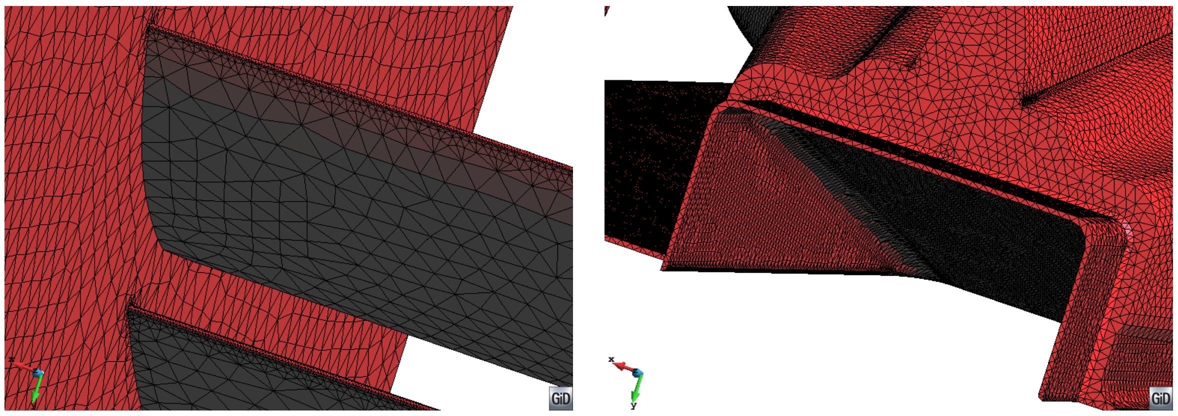


Figure 7.45: Different views of the triangle mesh used as input boundaries for the racing car model.

The mesh has been generated setting a mesh size of 10 uol in the skin of the car, as well as the floor, and a 20 uol in the outlet of the control volume (the surface enclosing the control volume in the rear part of the car). No general mesh size has been assigned, so the element size should grow with the distance to the car and the outlet surface accordingly to the size transition function. A TF of 0.6 has been used.

The data of the generated mesh of this example is detailed in Table 7.17.

Mesh general size (uol)	None
Mesh size in skin of the car and floor (uol)	10.0
Mesh size in outlet surface (uol)	20.0
Transition factor	0.6
Number of tetrahedra (millions)	11.6
Number of nodes (millions)	2.5
Minimum dihedral angle (degrees)	1.02
Number of elements with t min dihedral angle < 4	10

Table 7.17: Data of the racing car example.

Some views of the tetrahedra mesh generated are depicted in Figure 7.47. It can be appreciated the effect of the size transition function. As the TF used in this model is lower than 1, the size of the elements is not growing until a given distance to the skin of the car. However, the octree configuration where the tetrahedra come from force to double the size of the elements at this point, so the size transition is not as gradual as in other meshing methods (like advancing front based ones). A TF equal to 1 would lead to a faster transition governed by the 2 to 1 constraint. zoom views or the skin mesh of the tetrahedra generated is shown in Figure 7.46.

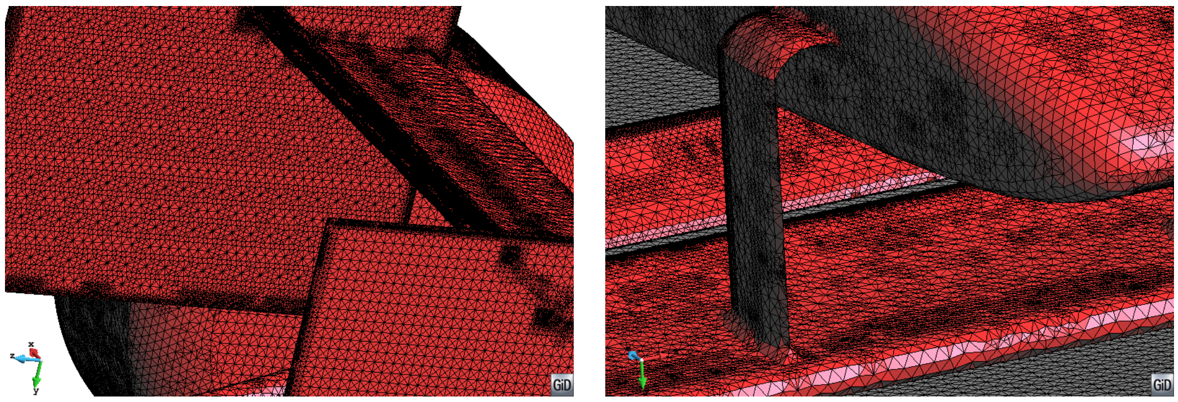


Figure 7.46: Different views of the skin mesh of the tetrahedra generated for the racing car model.

The mesh has been generated using 1, 2, 3 and 4 threads. The results of speed and speed-up for each configuration are depicted in Figure 7.48. As it can be seen, the maximum speed-up reached is 1.6 using 4 threads.

The distribution of CPU time in the different parts of the algorithm using 1 thread is depicted in Figure 7.49. It can be appreciated that in this example the refinement part takes more relevance than in the other examples. Specifically, the octree refinement considering the input mesh sizes and their propagation using the size transition function takes around 80% of the refinement time. This is due to the distribution of mesh size entities used in this model: all the triangles from the input mesh of the skin surfaces of the car and the floor are mesh size entities corresponding to 10 uol. Each one of this triangle generates the corresponding generalized mesh size points, from which the size transition function is applied to propagate the minimum and maximum sizes allowed to the surrounding octree cells considering the influence radius.

The tetrahedra coloring part also consumes an important part of the meshing process in this example. Because of the nature of the model (there are very thin parts to be meshed), there are many tetrahedra with all their nodes in the interface (undetermined tetrahedra) that must be colored using the methodology explained in section 5.3.7.

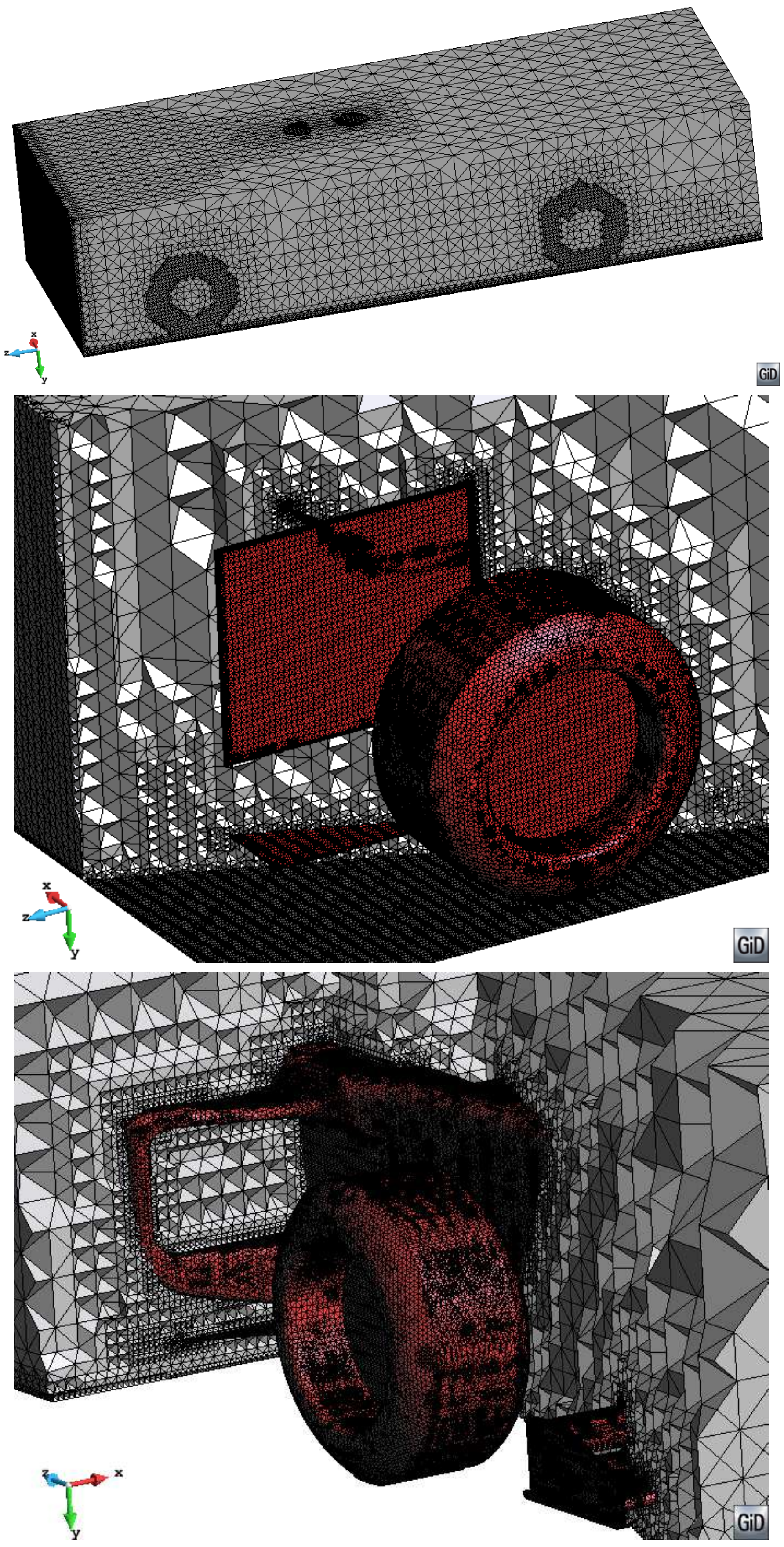
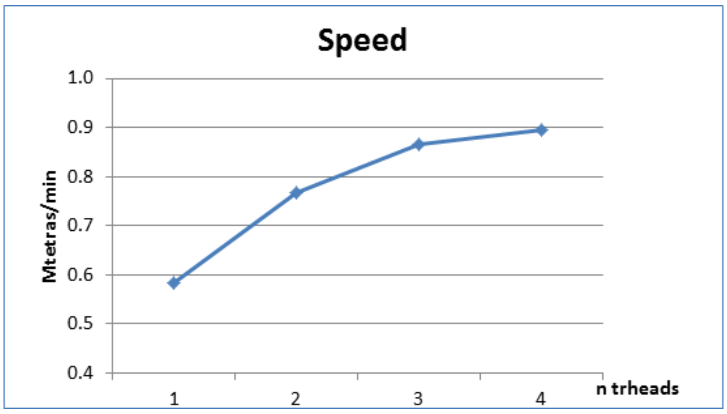
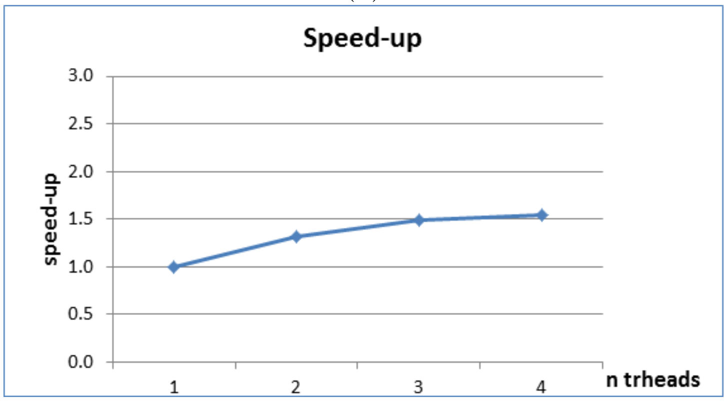


Figure 7.47: Different views of the final tetrahedral mesh for the racing car model.



(a)



(b)

Figure 7.48: Graphs of **(a)** speed (in Mtetrahedra per minute) and speed-up **(b)** corresponding to the racing car example.

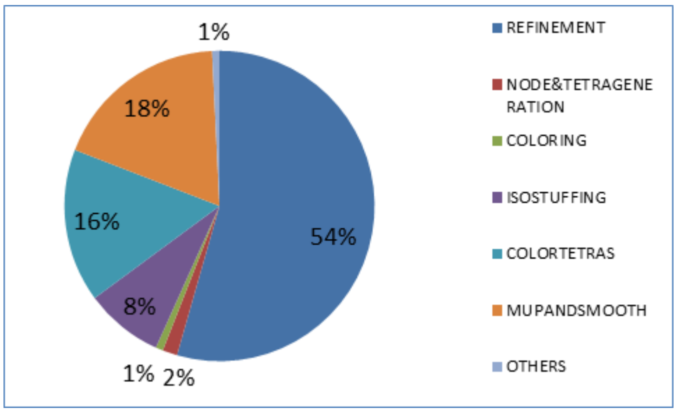
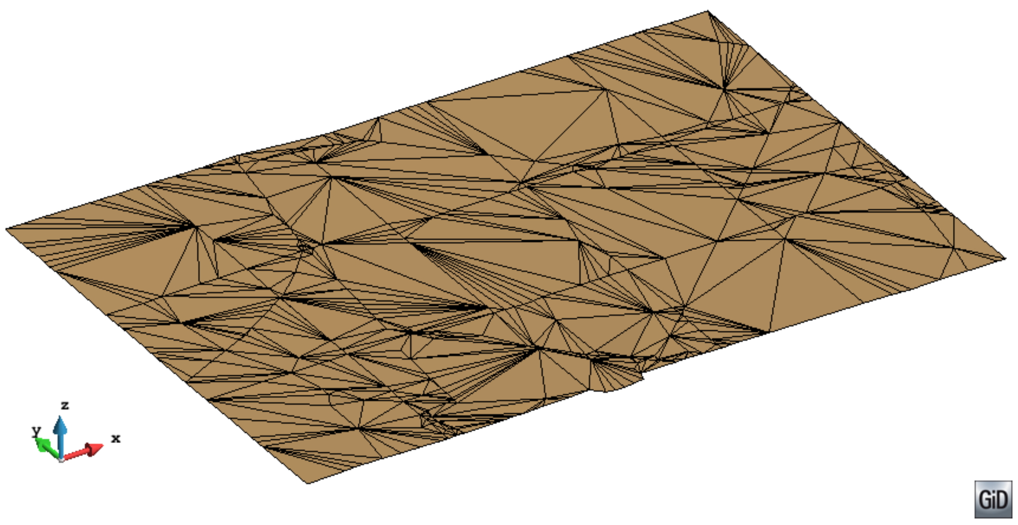


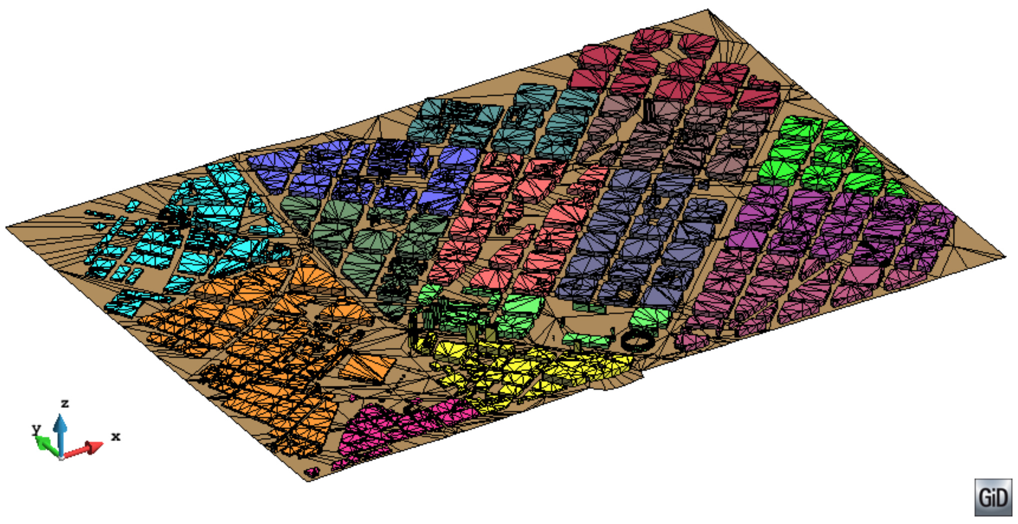
Figure 7.49: Percentages of CPU time consumed in the meshing process for the racing car example.

7.3 Barcelona city model

In this example a model of the city of Barcelona is used to generate an embedded mesh. Actually, the model used is a part of the model of the whole city. The model has been provided by the *Virtual Visualization Lab*, from *Barcelona Media company* [Med14], and consists in the Digital Terrain Model (DTM) and simplified models of the buildings.



(a)



(b)

Figure 7.50: Model of a part of the city of Barcelona used in this example. **(a)** View of the digital terrain model (DTM). **(b)** View of the DTM and the buildings.

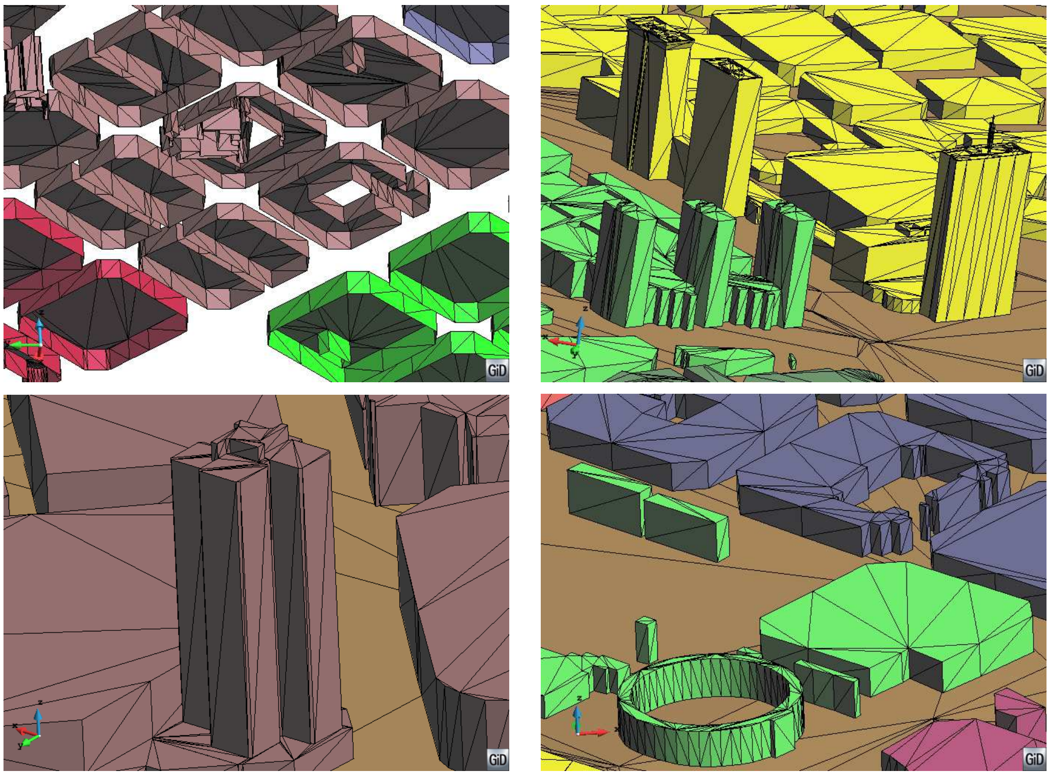


Figure 7.51: Zoom view of some part of the Barcelona city model in order to appreciate the level of detail of the buildings description. In the first figure it can be appreciated that the building models are opened in their lower part.

The scale of the model is 1 : 50 and there is no topological connection between the buildings and the terrain model. A view of the DTM and the buildings can be appreciated in Figure 7.50.

Some zoom views of the model are depicted in Figure 7.51. As it can be seen, the level of definition of the buildings does not capture all their details, but it is accurate enough to perform a CFD simulation of the air flow around them in the urban environment. It can also be appreciated that the buildings models are not represented by a watertight definition, as they have not any surface entity in its lower part. Furthermore, some of them have gaps and overlapping entities.

The presented model is basically a collection of surface entities defining the skin of the city which must act as a barrier for the air flow in an aerodynamic simulation. In order to generate the tetrahedra mesh of the air surrounding the buildings, a volume must be created. Furthermore, as an embedded mesh will be generated, another volume must be created to indicate whether a node is inside the air domain or not. For notation purposes, let us distinguish between the *control volume* and the *embedded volume*:

- The control volume is the one where the tetrahedra for the simulation will be generated.
- The embedded volume gives the topological information needed to indicate if a node is inside or outside the fluid domain (where the air flow is calculated). From the topological point of view, the embedded volume can be considered as a hole in the control volume where the simulation should not be applied.

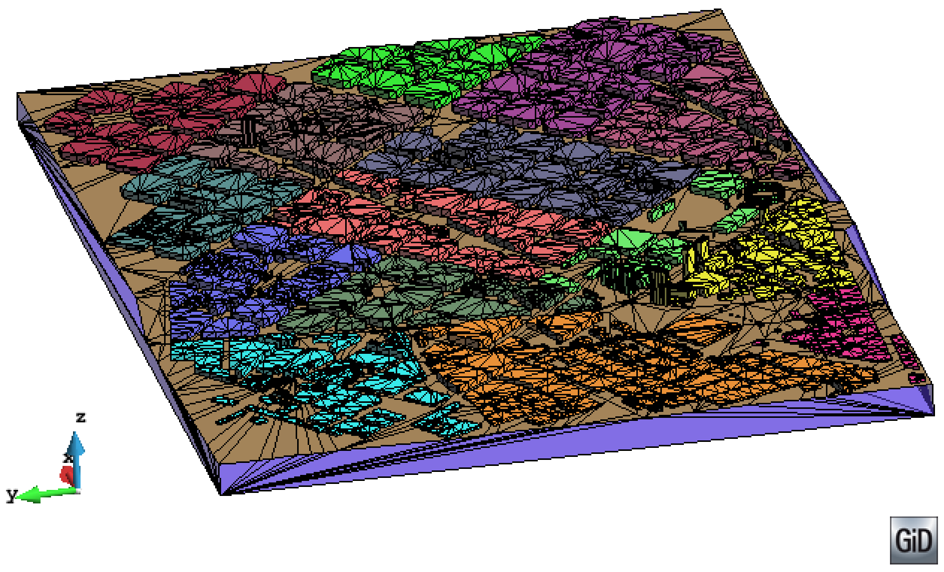
A view of the embedded volume is depicted in Figure 7.52(a), and the control volume is shown in Figure 7.52(b). The control volume is extremely simple; it is just a parallepiped. Its shape has been chosen taken into account the way of assigning the boundary conditions of a numerical simulation. The control volume may overlap partially or completely the embedded volume. However, this overlapping does not require a topological relationship between both volumes, which simplifies much its definition.

The embedded volume is a non-watertight volume, as the DTM and the buildings have gaps and overlapping entities. Some view of the higher entities of the edges of the DTM and the buildings are shown in Figure 7.53. Recall that all the edges should have a higher entity value equal to 2 in a watertight volume.

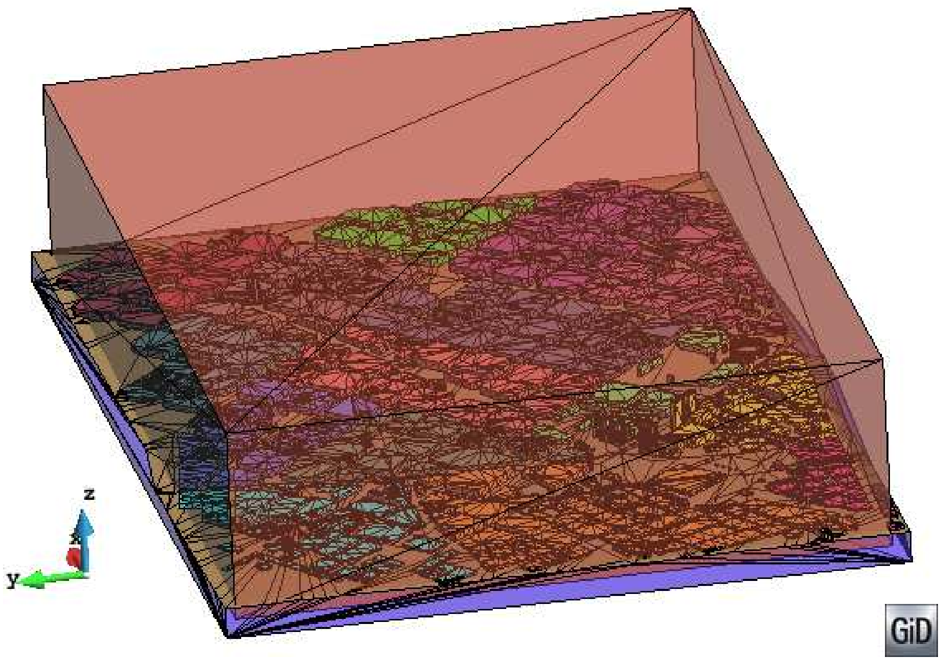
More than non-watertight, this volume is not correctly defined due to the way the buildings are presented with respect to the DTM. 2D schematic embedded surfaces trying to represent this situation are depicted in Figure 7.54. A scheme of a watertight embedded surface including only the terrain is depicted in Figure 7.54(a). A watertight definition of the embedded surface including a building is shown in Figure 7.54(b). A possible valid non-watertight definition of the embedded surface could be the one depicted in Figure 7.54(c), where there is no topological contact between the terrain and the building. Here, the size of the gap and the overlapping entities in the contact region must be small enough to be used in the coloring process.

The configuration given for this example corresponds to the scheme shown in Figure 7.54(d). In this case, the part of the embedded volume inside the building is not well defined, as there is a boundary entity (segment AB in Figure 7.54(d)) which definition indicates is interfacing the volume and its outer part, but in reality, this boundary is totally inner to the volume. This situation may set as invalid some of the rays aligned with Z direction used in the coloring process.

The meshing process of this example is based in two main steps:



(a)



(b)

Figure 7.52: Barcelona city model. Volumes defined to generate the embedded mesh. **(a)** Volume defining the lower part of the city, which will act as a hole of the control volume. **(b)** Control volume where the mesh will be generated.

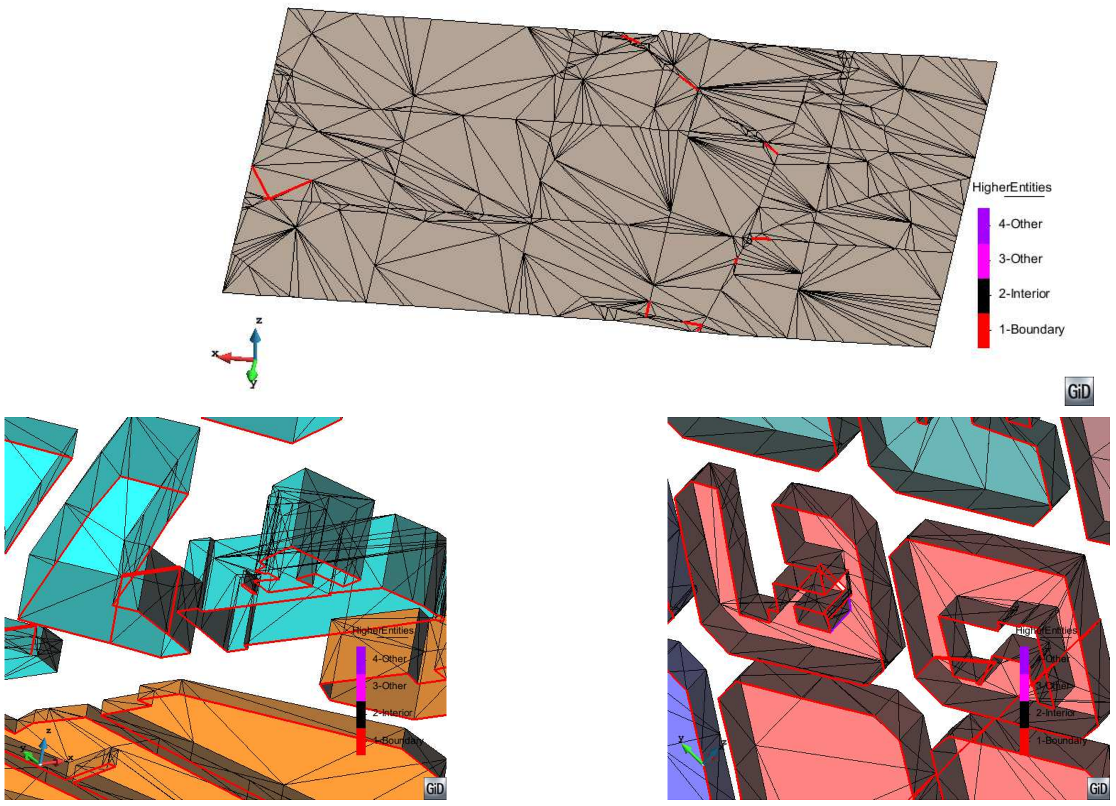


Figure 7.53: Different views of the higher entities of the edges of the DTM (upper figure) and some of the buildings (two figures at the bottom). Red edges are considered as boundaries, as they have higher entity equal to 1.

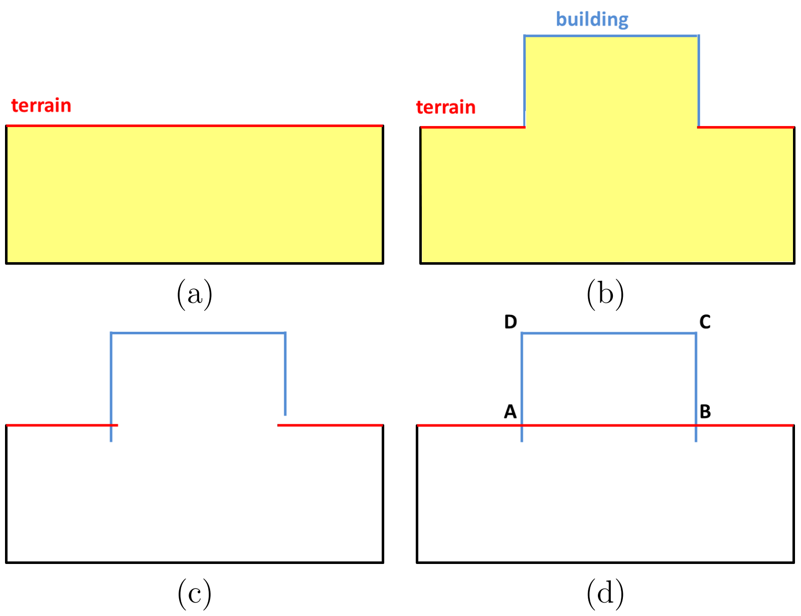


Figure 7.54: Barcelona city model. 2D schemes illustrating the bad definition of the embedded volume. **(a)** Scheme of embedded surface only including the terrain (in red). **(b)** Scheme of a watertight embedded surface including the terrain and a building (in blue). **(c)** Scheme of a valid non-watertight definition of the embedded surface. **(d)** Scheme of a bad definition of the embedded surface including the terrain and a building.

- *Body fitted mesh.* A body fitted mesh of the control volume is generated taking into account the input mesh sizes and not considering the boundaries of the embedded volume.

- *Embedded coloring.* The nodes of the generated body fitted mesh are colored and their distances to the boundaries are calculated, considering only the surface entities contour of the embedded volume.

Although the meshing algorithm accepts non-watertight geometries as an input, some cleaning operations have been made to the original model, in order to be able to generate the mesh:

- First of all, a rectangular shaped portion of the whole city model have been selected, and the rest part of it have been deleted.

- Delete some internal surface entities present in the buildings. This internal surfaces may affect negatively the node coloring process.

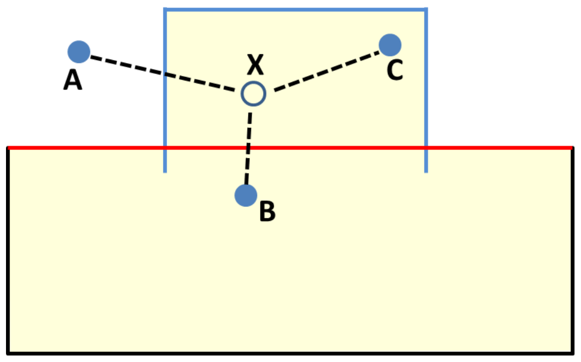


Figure 7.55: Scheme of local rays used to color node X in a bad defined model. Local rays from nodes A and C set node X as an inner node. Local ray from node B provides with bad information, as set the node X as outer.

This CAD cleaning operations are considerably simpler and less time consumer than the required to make the model watertight. The time needed to perform these operations manually is less than one hour.

The bad definition of this model make totally necessary the local ray casting operations applied to the nodes where the three Cartesian rays are invalid. Actually, the Cartesian rays color most of the nodes of the model, setting as uncolored the ones in the bad definitions of the boundaries. In these ones the local ray casting is performed.

An adaptation of the local ray casting process has been made in order to deal with this bad defined model. The model has gaps, nevertheless, the pathological situation of this model is given by a sort of fake interface entity (the segment AB in Figure 7.54(d)). This should lead to erroneous colors provided by some local ray, like the ones depicted in Figure 7.55. In that figure, node X should be colored using local rays from its three surrounding nodes: A, B and C. Node A is outside the volume, and the other two are inside. The ray from node A determines that X is inner to the volume, as it intersects one time the interface (between inner and outer parts), and node A is an outer node. The ray from node C also set X as inner node, as it has no intersections and node C is also an inner node. The problem comes from the ray from node B: it has one intersection and B is inner to the volume, so it should consider X as outer node, which is an error.

The voting process among the local rays used to color a node may take a bad decision depending on the configuration of the neighbors of the node. The adaptation to the local ray casting technique implemented in order to solve this situation is to give priority to the local rays with no intersection. If there are local rays with no intersections, their information is used, otherwise the ones with intersections are considered.

The mesh has been generated with a desired mesh size of 150 uol for the buildings surface entities, 200 uol for the terrain and leaving the mesher to increase the size in the volume accordingly to a transition factor of 1. The data of the generated mesh is detailed in Table 7.18.

Mesh general size (uol)	None
Mesh size in buildings (uol)	150.0
Mesh size in terrain (uol)	200.0
Transition factor	1.0
Number of tetrahedra (millions)	25.1
Number of nodes (millions)	4.3
Minimum dihedral angle (degrees)	5.45

Table 7.18: Data of the Barcelona model example.

A zoom view of the inner tetrahedra of the final mesh together with the input boundaries is depicted in Figure 7.56.

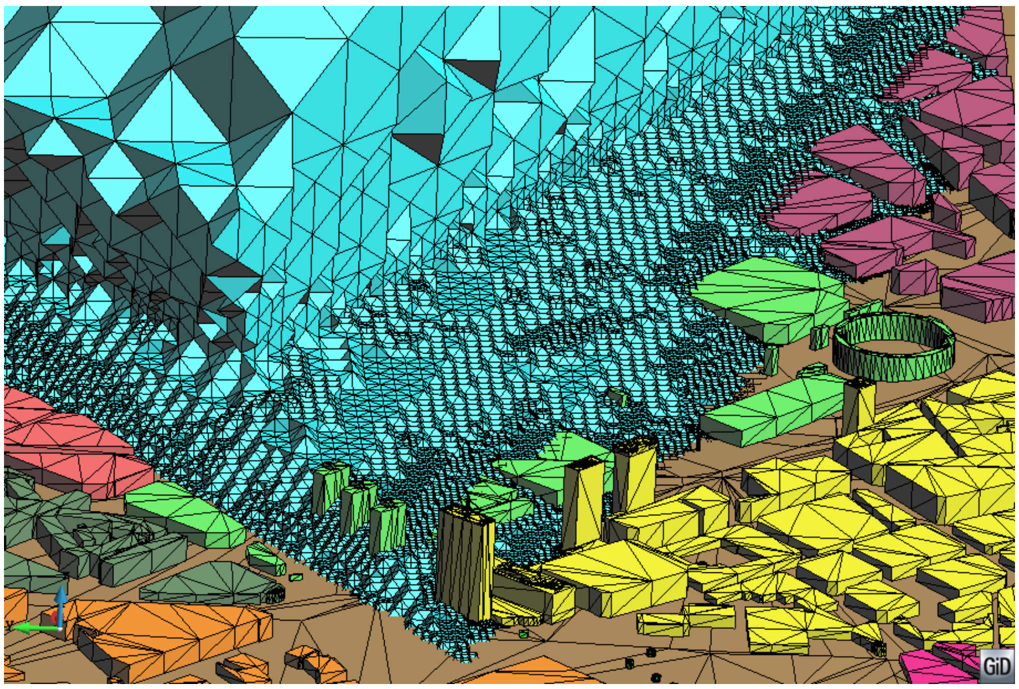


Figure 7.56: Final mesh for the Barcelona city model. Zoom view of the inner part of the mesh together with the input boundaries.

A view of the exterior part of the generated mesh is shown in Figure 7.57(a), and a view of the contourfill of computed distances for embedded method in a cut of the mesh is shown in Figure 7.57(b), together with the isosurface of distance equal to zero extracted from the generated mesh.

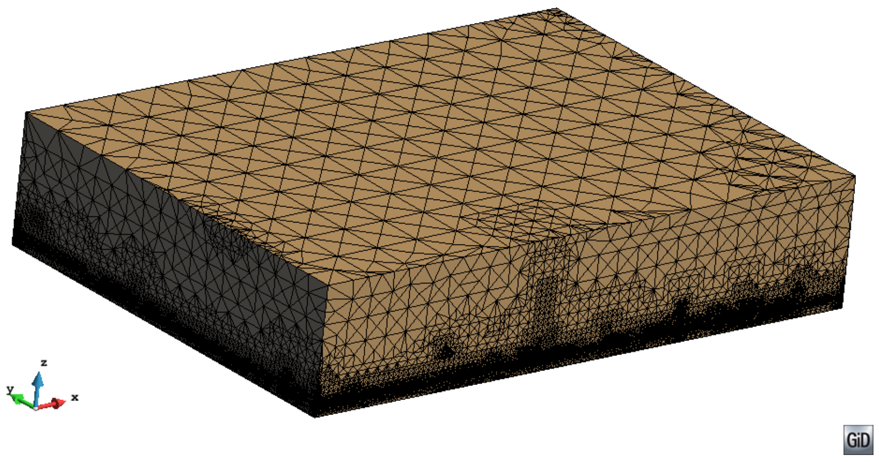
It can be appreciated that the isosurface corresponding to distance zero is not capturing well the sharp edges of the domain. In this work the coloring of the nodes and the distance propagation is done at global level, only considering the nodes. In order to improve the algorithm the distance of the nodes could be treated locally, considering the edges of the mesh and their intersection points with the input boundaries. Depending on the requirements of the simulation to be done it may be important or not to preserve these geometrical features.

The mesh has been generated using 1, 2, 3 and 4 threads. The results of speed for each configuration are depicted in Figure 7.59. As it can be seen, 6.2 Mtetrahedra per minute are generated in serial. The maximum speed-up reached is 1.3 using 4 threads, and the memory peak during the meshing process reach 5.3 Gb.

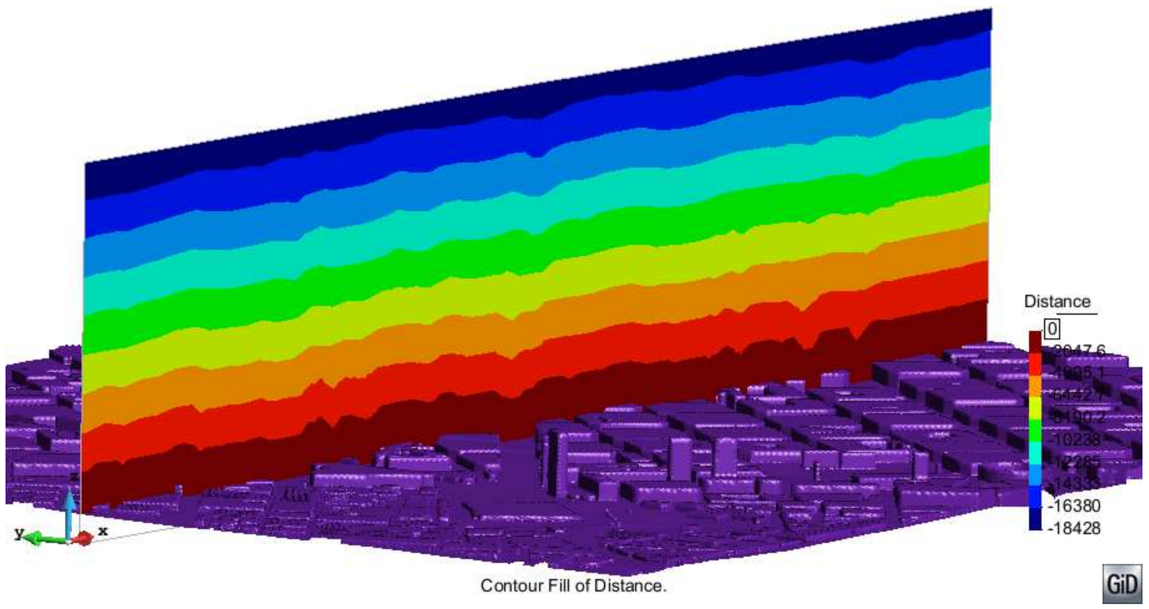
The distribution of times devoted to the two main parts of the algorithm (body fitted mesh of the control volume and embedded coloring) is depicted in Figure 7.60. It has to be considered that the embedded coloring part includes:

- Enter the surface entities of the embedded volume into the octree. This automatically sets the interface cells.
- Compute the exact distance to the boundaries of the nodes belonging to the interface cells.
- Perform the coloring and compute of distances to the nodes using the ray casting technique.

It can be appreciated that (in serial), both parts of the meshing process consumes approximately the same time.

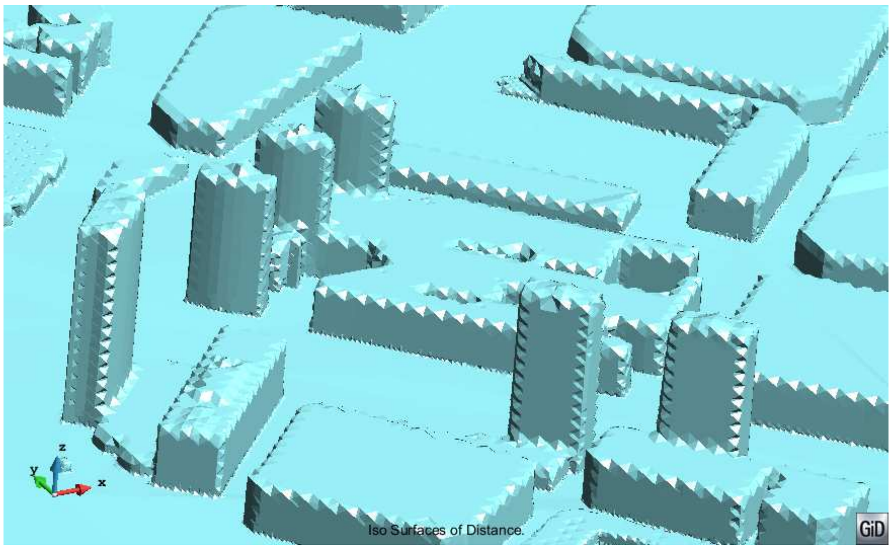


(a)

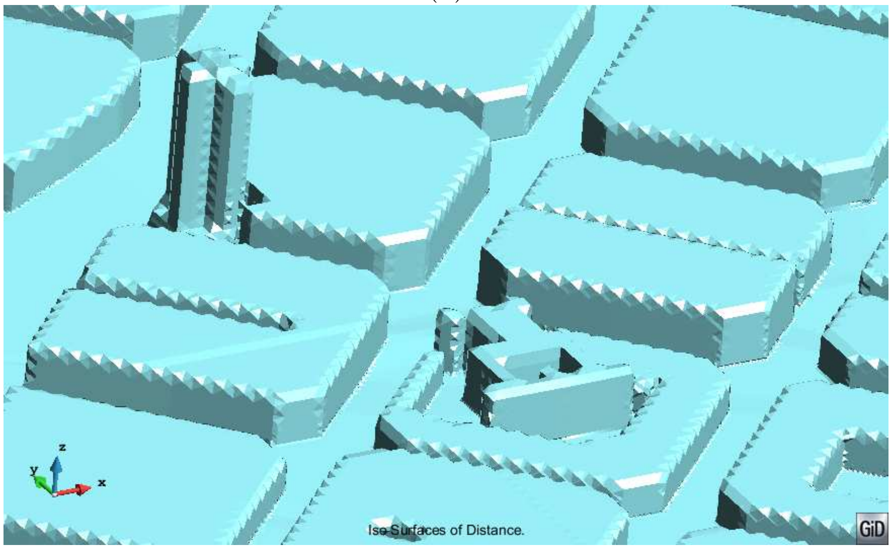


(b)

Figure 7.57: Final mesh for the Barcelona city model. **(a)** View of the external part of the mesh. **(b)** View of the contourfill of distances in a cut plane made on the generated mesh, together with the isosurface corresponding to distance zero.



(a)



(b)

Figure 7.58: Zoom views of the isosurface corresponding to a distance 0 of the mesh generated in Barcelona model.

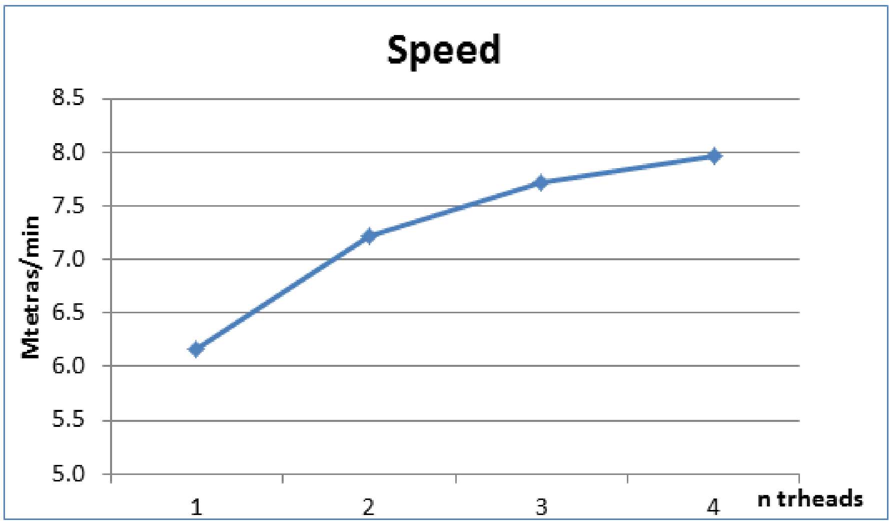


Figure 7.59: Graph of speed (in Mtetrahedra per minute) corresponding to the Barcelona model example.

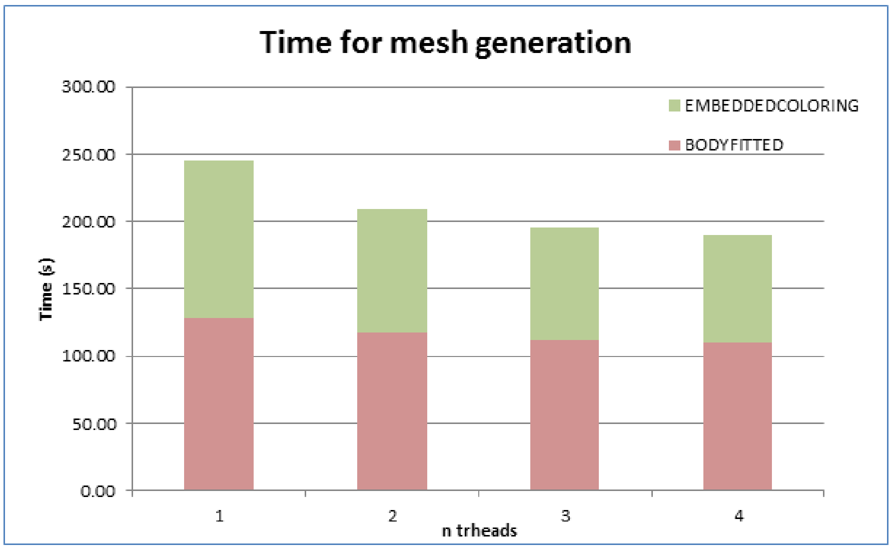


Figure 7.60: Times consumed for the meshing of the Barcelona model example detailed in the two main parts of the algorithm.

7.4 General overview

A general analysis of the body-fitted mesher is done in this section. It is based on the results of the examples with more than one million tetrahedra, which are the following ones:

- Validation example $VE - F1$.
- Validation example $VE - S1$ (the four configurations).
- Validation example $VE - S2$ (the four configurations).
- Racing car example.

The smaller examples are not considered in this section because they should be affected by the time measurements.
Concerning the embedded mesher, as few cases have been run, its analysis is done in the sections of the corresponding examples.

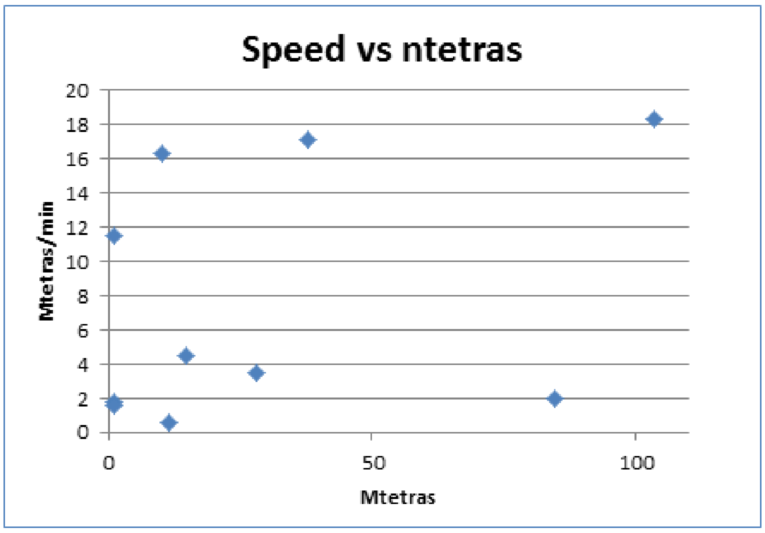
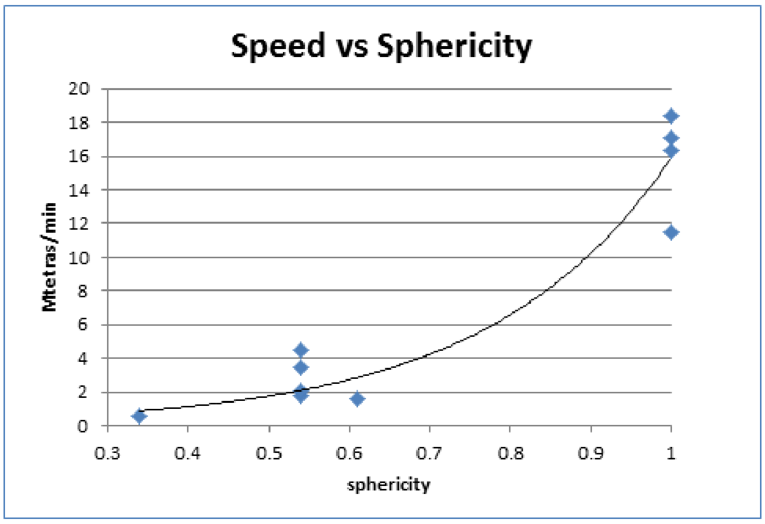


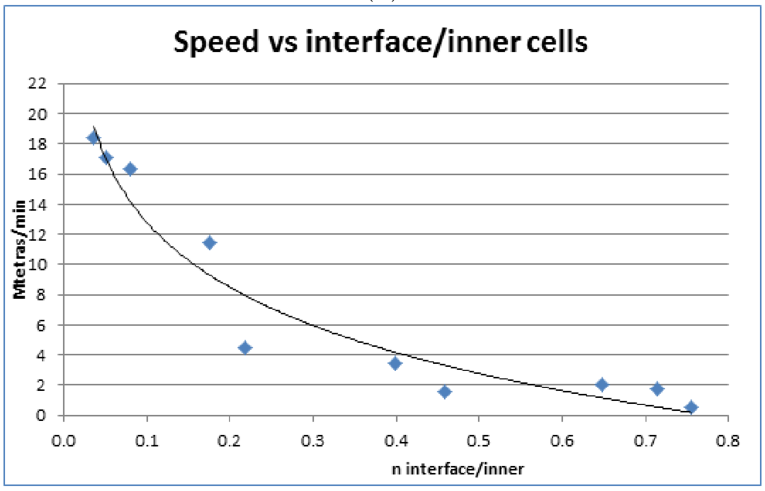
Figure 7.61: Graph relating the speed of generation (Mtetrahedra per minute) and the number of tetrahedra generated.

The speed of the mesher is quite variable depending on the characteristics of the model, going from 600,000 to 18 millions of tetrahedra per minute. Having a look at Figure 7.61, no relationship between the speed of the mesher and the number of tetrahedra generated can be established. This was expected, as in the presented mesher the main part of the effort is put in the tetrahedra near the boundary, while the inner ones are created very fast.

For this reason, in order to evaluate and compare the speed of an octree based mesher is more appropriate to consider the sphericity of the model to be meshed. A graph relating the speed and the sphericity is shown in Figure 7.62(a).



(a)



(b)

Figure 7.62: Speed of tetrahedra generation (Mtetrahedra per minute) versus **(a)** shphericity of the model and **(b)** ratio between number of interface and inner cells. Logarithmic tendency line in black.

It can be seen that the mesher is faster as the model to be meshed is more massive (higher sphericity), independently on the number of tetrahedra generated. However, it can be appreciated that the same model (with the same sphericity) presents different speeds depending on the configuration of the example. The reason for this is that present implementation of the mesher dedicates almost all the effort in the treatment of the tetrahedra coming from the interface cells, so the same model with different sizes assigned to it may have different ratio between the tetrahedra near the interface and the inner ones.

In order to normalize the speed of the mesher considering this effect, a graph relating the speed of the mesher with the ratio between interface and inner cells is depicted in Figure 7.62(b). The behaviour of the mesher is well captured, and the speed increases as the ratio decreases.

Considering the main parts of the algorithm defined in Section 7.1.7 for performance analysis, the percentages of time for each one of these parts are very different in the examples. Depending on the characteristics of the geometrical model and the input parameters, it may be more relevant the octree refinement, the surface fitting process, the tetrahedra coloring or the mesh improvement operations (make-up and smoothing). However, the nodes coloring process is rather fast in all the examples.

It has to be commented that this work has been more focused on the mesh generation than on its improvement, so it is expected that the implementation of make-up and smoothing operations can be improved, and this will affect the percentages of time of the different parts of the algorithm.

Concerning the memory consumed by the mesher, all the examples run have a ratio between the memory peak and the memory needed to store the mesh lower than 15, which represents approximately 0.25 Gb per Mtetrahedra. A graph of the memory peak value for the examples analyzed is depicted in Figure 7.63. It can be appreciated that the memory peak needed to generate a mesh with the presented mesher is rather linear with the number of tetrahedra generated.

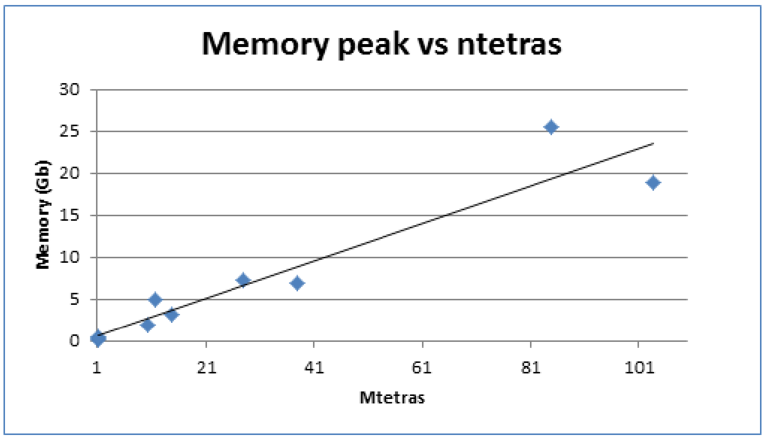


Figure 7.63: Peak memory consumed (Gb) during the meshing process. Linear tendency line in black.

The examples studied show that the implementation done has a low scalability when running in parallel. Values of 1.6 speed-up have been reached using 4 threads, which are far from the optimal ones (4). It has to be commented that this work has been more focused on the algorithm rather than on its implementation: more effort has been put in order to guarantee the mesh is generated than in the parallel implementation of the method. For this reason, it is expected that improvements in the implementation of the algorithm could reach a higher speed-up value. Nevertheless, the implementation of an algorithm always involves a trade-off between the speed and the memory allocated, so an improvement in speed may lead to an increase of memory allocated.

Furthermore the algorithm is based in non-consecutive usage of the memory. Although it has not been measured, a decrease in the cache use efficiency, specially in parallel computations, is expected.

7.4.1 Comparison with other methods

The considerations explained above with regards to the speed of the mesher make it difficult to compare the new mesher with other methods.

Some representative values are presented in Table 7.19 in order to have an order of magnitude. The method is compared with the advancing front based mesher present in GiD [CRP+10a, CRP+10b, CRP+10c] and the Delaunay based mesher implemented in Tetgen [ST10], running in the same computer used in the present work.

Octree-based mesher (this work)	0.6 to 18
Advancing front (GiD)	0.1 to 0.5
Delaunay (Tetgen)	5 to 6

Table 7.19: Order of magnitude of speed of mesh generation of different methods (in Mtetrahedra per minute).

It has to be also considered that the implementation of a mesher can affect drastically its performance, so the values provided should be used only as an order of magnitude. There are implementations of advancing front meshers, for instance, reaching a speed of 1 million of tetrahedra per minute [Löh08] using a similar computer than the used in this work.

Another difficulty at the time of comparing speed of different mesher is the mesh improvement operations (make-up and smoothing) performed after the mesh generation itself. The effort needed for this operations depends strongly on two aspects:

- The quality of the mesh just before the mesh generation.
- The quality required by the simulation to be run using that mesh.

These two aspects are not typically specified, so the speed comparison between methods has to be analyzed carefully.

One consideration to be done when comparing the presented mesher with an advancing front-based one is the requirements they have (in terms of quality) on the input boundaries defining the domain. The new octree-based mesher accepts very bad shaped triangles as input boundaries, returning a good quality triangle mesh as the contours of the tetrahedra generated. Advancing front methods require a very good quality contours mesh, so before the volume mesh generation an extra effort should be added in order to edit (or even to regenerate) the contour mesh. In some cases the CAD cleaning operations to reach the final mesh are more time consuming than the mesh generation itself.

No other octree-based mesher has been used in the present work in order to compare the performance. Most of the octree meshers are based on hexahedral meshing, and not all of them satisfy the main requirements presented in this work (basically geometrical features and topology preservation). However, to have an order or magnitude, speeds between 0.3 and 3 Mtetrahedra per second can be found in the literature [LS07, Mar09] considering octree-based meshers with the same characteristics as the developed in this work. It has to be considered that the computer used is different for this cases: [LS07] used a Mac Pro with 2.66 GHz Intel Xeon processor, and [Mar09] a 2.8 GHz octo-core mac xserve.

Concerning the memory consumed during the meshing process, the presented method consumes approximately the half of the advancing front and Delaunay implementations compared.

Chapter 8

Conclusions and future lines

8.1 Conclusions

A robust and fast tetrahedra octree based mesher has been developed for embedded and body fitted meshes considering the requirements defined in Section 1.2. An innovative algorithm has been proposed to preserve the geometrical features and the topology of the domain accepting non-watertight definitions of the model. To reach these objectives some improvements and adaptations have been done in existing techniques (such as the ray casting method) in order to solve the Point In Polygon problem considering several volumes.

The effectiveness of the algorithm and its implementation has been verified in some validation examples. The mesh generator preserves the geometrical features and the topology of the model. Examples of non-watertight definitions of the domain have been meshed successfully thanks to the coloring algorithm designed based on the ray casting technique.

A summary of key features of the presented work is listed hereafter:

- The body-fitted mesher preserves the geometrical features existing in the domain, such as ridges and corners.
- The topology of the volumes defining the domain is preserved by the body-fitted meshing algorithm.
- The mesher can generate tetrahedra meshes from non-watertight definitions of the volumes. Cases with gaps and overlapping entities in the definition of their boundaries have been meshed successfully.
- The mesher has been adapted to generate embedded and body-fitted meshes.

- The examples run indicate that the mesher is very robust considering different input parameters. Even when no input parameters are introduced, it generates a valid mesh of the domain.
- The mesher is very robust considering the quality of the input mesh. An important advantage of the mesher is that the triangle mesh defining the boundaries is not needed to have good shaped elements. The only requirement for the input mesh is to represent precisely the shape of the contours, which only implies the fulfillment of a given chordal error criterion. For this purpose typical surface render meshes (optimized for visualization purposes) can be used. As these meshes are often provided by the CAD systems, most CAD cleaning operations are saved using this mesher, with the corresponding saving in time. However, if the quality of the input mesh is very low, some pre-treatment may be done in order to detect the sharp edges to be preserved by the mesher. This part is considered external to the mesher.
- The new mesher is independent of the size of the input boundaries. This makes possible to use the same definition of the domain to generate different volume meshes.
- A strategy has been proposed for embedded meshes combining the body fitted mesher for the control volume, and the node coloring and distances propagation considering the contours of the embedded volume.
- The results from the examples studied show that the mesher is fast in comparison with other meshers analyzed. Depending on the characteristics of the model, the mesher generates between 0.6 and 18 millions of tetrahedra per minute on a desktop PC.
- A robust and efficient point coloring algorithm based on a ray casting technique have been designed and implemented in order to detect where the nodes of the mesh are: inside a given volume or laying on an interface between volumes. This coloring algorithm works in domains with several volumes and works with non-watertight definitions of them. A solution has been proposed for solving the pathological cases for ray casting technique. The coloring algorithm has been adapted to compute the distances of the nodes of the mesh to the input boundaries for embedded meshes, using the Manhattan distance as an approximation of the exact one.
- An algorithm has been proposed to color the tetrahedra with all its nodes in interface entities: that is, determine the volume these tetrahedra belongs to. This algorithm ensures the preservation of the topology of the model.

- The mesher generates jointly the meshes of several volumes if they share some line or surface entity, and gets as input data triangle meshes defining the boundaries of the volumes.

- The mesh resulting from the meshing algorithm may have poor quality elements depending on the characteristics of the model. For this reason make-up and smoothing operations have been implemented in order to improve the quality of the mesh.

- The presented mesher tries to fit the final mesh with the desired mesh size, but this cannot always be accomplished precisely. This is because the tetrahedra generated come directly from the octree cells, and their size comes from subdividing the octree root recursively. Size transitions between regions with different desired size are controlled by a size transition function. However, the final mesh cannot reproduce smooth size transitions, as the octree cells sizes changes by a factor of two in each refining process.

- The octree structure has been adapted successfully to the mesh generation, using an efficient implementation of a binary octree.

- The current implementation does not preserve the topology of the model in configurations where no proper input mesh size is provided and consequently, if the domain presents very thin parts in comparison with its characteristic length, the topology may not be preserved.

- The isotropic structure of the octree may lead to excessive refined tetrahedra meshes near the boundaries in comparison to other meshing methods.

- The algorithm has been adapted to generate 3D surface meshes which are inner to a volume, or isolated in space.

- The implementation of the mesher has been done following the OpenMP paradigm for parallelizing some of the parts of the algorithm. Nevertheless, the maximum speed-up reached is 1.6 using 4 threads, which is considered to be improvable.

- The implementation of the mesher generates meshes consuming less than 15 times the memory needed to store the final mesh. This represents approximately 0.25 Gb of peak memory per Mtetrahedra generated. This is less than the other meshing techniques analysed in this work.

- The mesher could be used for CAD cleaning operations, as it can return a watertight mesh from a non-watertight one.

The mesher has been implemented as a static library and it is used by the version 11.1.9d of the pre and post-processor GiD [MCP$^+$13].

8.2 Future research lines

From the design and implementation of the meshing algorithm presented in this work, the following future research lines have been identified:

- Improve the implementation of the algorithm. This work has been focused more on the algorithm rather than on its parallel implementation. The serial part can be parallelized and the parallel part can be improved.

- Implement the meshing algorithm using distributed memory paradigm (MPI). This kind of implementations uses a decomposition of the space in order to treat independently separated parts of it. As the octree structure is a decomposition of the space itself, the implementation could take advantages of its characteristics. However, focusing in one part of the domain, some of the octree refinement criteria need information of other parts, so the implementation of the algorithm is not obvious.

- Improve the make-up and smoothing operations. These operations can be improved in order to get better quality meshes using less time. Furthermore, a coarsening operation could be added in order to avoid the excessive level of refinement present in some parts of the boundaries of the volumes.

- Adapt the mesher to work with 3D images as a definition of the input boundaries (such as medical images (IMR), or tomography ones). In these cases, the boundaries of the volumes involved in the domain are not defined explicitly: they can be extracted from a distance function applied to a 3D regular grid of points. As the presented mesher uses an octree (which a regular grid is an specific case), the information of the distances can be included directly in the algorithm and some of its processes can be adapted to generate the mesh of the different volumes of the domain.

- Adapt the algorithm to generate hexahedra meshes. Actually, the inner cells of the octree are already hexahedra, so they could be used as final elements, defining a given pattern for the cases where two neighbor cells are in different levels (have different size) in order to generate a conformal mesh. The more complicated parts of the algorithm to reach this goal should be the preservation of the geometrical features and the surface fitting process. In the present algorithm, these processes involve the splitting of tetrahedra edges or faces. These operations generate automatically new tetrahedra, but applying them to hexahedra should generate other types of polyhedral. This should require to define a sort of transition element types, or to redefine directly some of the operations involved.

- Adapt the mesher to NURBS curves and surfaces as input for defining the volumes boundaries and forced line entities. The smoother definition of NURBS compared with meshes ensures a better approximation to the shape of the domain. Theoretically the design of the algorithm has been done considering this option, but in practice, the use of geometrical data instead of mesh data as input may lead to some difficulties in terms of implementation. Specially the operations involving intersections. Intersecting an edge with a triangle has a limited number of possible cases, but intersecting it with a NURBS surface is much more complicated, and can present several pathological configurations.

- Implement automatic tools in order to filter and detect the geometrical features to be preserved from the analysis of the input data. When the input boundaries have low quality, the normal of some of the triangles cannot be computed properly, so some edges can be set to be preserved when they are not really sharp edges. Actually, this part can be considered as external to the meshing algorithm, but enables the mesher to be called directly from a CAD system, which may automatize as much as possible the operations to create the input data and saves time at user level.

- Adapt the algorithm to models with specific topology. As seen in the example of the Barcelona city model, adapting slightly the ray casting coloring algorithm, the mesher can deal with bad topological definition of the domain. This can be interesting at the time of trying to adapt the mesher to a set of cases which may be bad defined, but may present known pathological situations. This is the case of most DTM models, as well as in CAD geometries coming from other sources.

- Implement the parts of the topological refinement criteria concerning surface and volume parts. Their implementation should ensure the topological preservation even when there are very thin parts in the model and no mesh size is provided to the mesher.

- For the embedded mesher, the distance from the nodes to the input boundaries can be treated locally, considering each edge of the mesh. This should help to capture the sharp edges of the model by the isosurface of distance zero.

Bibliography

[AEF$^+$95] N Akkiraju, H Edelsbrunner, M Facello, P Fu, E Mücke, and C Varela. Alpha shapes: definition and software. In *Proceedings of the 1st International Computational Geometry Software Workshop*, pages 63–66, 1995.

[AL09] Romain Aubry and Rainald Löhner. Generation of viscous grids at ridges and corners. *International journal for numerical methods in engineering*, 77(9):1247–1289, 2009.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[Bak87] Timothy J Baker. Three dimensional mesh generation by triangulation of arbitrary point sets. *AIAA paper*, 87(1124), 1987.

[Bak89] Timothy J Baker. Developments and trends in three-dimensional mesh generation. *Applied Numerical Mathematics*, 5(4):275–304, 1989.

[BBK06] Daniel K Blandford, Guy E Blelloch, and Clemens Kadow. Engineering a compact parallel delaunay algorithm in 3d. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 292–300. ACM, 2006.

[BEG94] Marshall Bern, David Eppstein, and John Gilbert. Provably good mesh generation. *Journal of Computer and System Sciences*, 48(3):384–409, 1994.

[CC12] Jose J. Camata and Alvaro L. G. A. Coutinho. Parallel implementation and performance analysis of a linear octree finite element mesh generation scheme. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2012.

[CCL02] Juan R. Cebral, Fernando E. Camelli, and Rainald Lohner. A feature-preserving volumetric technique to merge surface triangulations. *International Journal for Numerical Methods in Engineering*, 55(2):177–190, 2002.

[CDS12] Siu-Wing Cheng, Tamal K. Dey, and Jonathan Shewchuk. *Delaunay Mesh Generation.* Chapman & Hall/CRC, 1st edition, 2012.

[CIO03] Nestor Calvo, Sergio R Idelsohn, and Eugenio Oñate. The extended delaunay tessellation. *Engineering Computations*, 20(5/6):583–600, 2003.

[CRP$^+$10a] A. Coll, R. Ribó, M. Pasenau, E. Escolano, J.Suit. Perez, A. Melendo, and A. Monros. *GiD v.11 Customization Manual*, 2010.

[CRP$^+$10b] A. Coll, R. Ribó, M. Pasenau, E. Escolano, J.Suit. Perez, A. Melendo, and A. Monros. *GiD v.11 Reference Manual*, 2010.

[CRP$^+$10c] A. Coll, R. Ribó, M. Pasenau, E. Escolano, J.Suit. Perez, A. Melendo, and A. Monros. *GiD v.11 User Manual*, 2010.

[Far97] Gerald Farin. *Curves and surfaces for computer-aided geometric design - a practical guide (4. ed.).* Computer science and scientific computing. Academic Press, 1997.

[FG00] P.J. Frey and P.L. George. *Mesh Generation: Application to Finite Elements.* Hermès Science Publications, 2000.

[Fry94] Jan Frykestig. *Advancing front mesh generation techniques with application to the finite element method.* PhD thesis, Chalmers University of Technology, 1994.

[Fuc98] Alexander Fuchs. Automatic grid generation with almost regular delaunay tetrahedra. In *IMR*, pages 133–147, 1998.

[GB98] Paul-Louis George and Houman Borouchaki. *Delaunay triangulation and meshing: application to finite elements.* Hermes Paris, 1998.

[GB03] Paul-Louis George and Houman Borouchaki. Back to edge flips in 3 dimensions. In *IMR*, pages 393–402, 2003.

[Geo91] P.L. George. *Automatic mesh generation: application to finite element methods.* Wiley, 1991.

[GHS90] Paul-Louis George, Frédéric Hecht, and Éric Saltel. Fully automatic mesh generator for 3d domains of any shape. *IMPACT of Computing in Science and Engineering*, 2(3):187–218, 1990.

[GS98] Rao V. Garimella and Mark S. Shephard. Boundary layer meshing for viscous flows in complex domains. In *IMR*, pages 107–118, 1998.

[HS97] Chong-Wei Huang and Tian-Yuan Shih. On the complexity of point-in-polygon algorithms. *Comput. Geosci.*, 23(1):109–118, February 1997.

[HYFK98] Jian Huang, R. Yagel, Vassily Filippov, and Y. Kurzion. An accurate method for voxelizing polygon meshes. In *Volume Visualization, 1998. IEEE Symposium on*, pages 119–126, 1998.

[IN02] Yasushi Ito and Kazuhiro Nakahashi. Unstructured mesh generation for viscous flow computations. In *IMR*, pages 367–377, 2002.

[ITN08] Takashi Ishida, Shun Takahashi, and Kazuhiro Nakahashi. Efficient and robust cartesian mesh generation for building-cube method. *Journal of Computational Science and Technology*, 2(4):435–446, 2008.

[JT93] H Jin and RI Tanner. Generation of unstructured tetrahedral meshes by advancing front technique. *International Journal for Numerical Methods in Engineering*, 36(11):1805–1823, 1993.

[KKŽ05] Josef Kohout, Ivana Kolingerová, and Jiří Žára. Parallel delaunay triangulation in¡ i¿ e¡/i¿¡ sup¿ 2¡/sup¿ and¡ i¿ e¡/i¿¡ sup¿ 3¡/sup¿ for computers with shared memory. *Parallel Computing*, 31(5):491–522, 2005.

[Kra86] E.F. Krause. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry.* Dover Publications, 1986.

[KWPH06] Aaron Knoll, Ingo Wald, Steven Parker, and Charles Hansen. Interactive isosurface ray tracing of large octree volumes. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 115–124. IEEE, 2006.

[LC99] Rainald Löhner and Juan R. Cebral. Parallel advancing front grid generation. In *in International Meshing Roundtable, Sandia National Labs*, pages 67–74, 1999.

[LCC$^+$08] Rainald Löhner, Juan R Cebral, Fernando E Camelli, S Appanaboyina, Joseph D Baum, Eric L Mestreau, and Orlando A Soto. Adaptive embedded and immersed unstructured grid techniques. *Computer Methods in Applied Mechanics and Engineering*, 197(25):2173–2197, 2008.

[Lo85] S. H. Lo. A new mesh generation scheme for arbitrary planar domains. *International Journal for Numerical Methods in Engineering*, 21(8):1403–1426, 1985.

[LO98] Rainald Löhner and Eugenio Oñate. An advancing front point generation technique. *Communications in Numerical Methods in Engineering*, 14(12):1097–1108, 1998.

[Löh93] Rainald Löhner. Matching semi-structured and unstructured grids for navier-stokes calculations. *AIAA paper*, 3348:1993, 1993.

[Löh96] Rainald Löhner. Extensions and improvements of the advancing front grid generation technique. *Communications in numerical methods in engineering*, 12(10):683–702, 1996.

[Löh08] Rainald Löhner. *Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods*. Wiley, 2008.

[LP88] Rainald Löhner and Paresh Parikh. Generation of three-dimensional unstructured grids by the advancing-front method. *International Journal For Numerical Methods in Fluids*, 8:1135–1149, 1988.

[LP00] Ming-Chih Lai and Charles S Peskin. An immersed boundary method with formal second-order accuracy and reduced numerical viscosity. *Journal of Computational Physics*, 160(2):705–719, 2000.

[LS07] François Labelle and Jonathan Richard Shewchuk. Isosurface stuffing: fast tetrahedral meshes with good dihedral angles. *ACM Trans. Graph.*, 26(3), July 2007.

[Mar09] Loïc Maréchal. Advances in octree-based all-hexahedral mesh generation: Handling sharp features. In BrettW. Clark, editor, *Proceedings of the 18th International Meshing Roundtable*, pages 65–84. Springer Berlin Heidelberg, 2009.

[MCP$^+$13] A. Melendo, A. Coll, M. Pasenau, E. Escolano, and A. Monros. www.gidhome.com, 2013. [Online; accessed 25-September-2013].

[Med14] Barcelona Media. www.barcelonamedia.org/seccio/bm-labs/laboratori-de-visualitzacio-virtual, 2014. [Online; accessed 1-April-2014].

[MV92] Scott A Mitchell and Stephen A Vavasis. Quality mesh generation in three dimensions. In *Proceedings of the eighth annual symposium on Computational geometry*, pages 212–221. ACM, 1992.

[Nay99] D. J. Naylor. Filling space with tetrahedra. *International Journal for Numerical Methods in Engineering*, 44(10):1383–1395, 1999.

[NRNTfI67] S. Nordbeck, B. Rystedt, and fasc. 1 1967 Nordisk Tidskrift for Informationsbehandling, vol. 7. *Computer Cartography in Polygon Programs, by Stig Nordbeck [and] Bengt Rystedt.* Lund studies in geography. Ser. C. General and mathematical geography, no. 7. 1967.

[NT03] Fakir S. Nooruddin and Greg Turk. Simplification and repair of polygonal models using volumetric techniques. *Visualization and Computer Graphics, IEEE Transactions on*, 9(2):191–205, 2003.

[NVP82] Nguyen-Van-Phai. Automatic mesh generation with tetrahedron elements. *International Journal for Numerical Methods in Engineering*, 18(2):273–289, 1982.

[Oña09] E. Oñate. *Structural Analysis with the Finite Element Method. Linear Statics: Volume 1: Basis and Solids.* Lecture Notes on Numerical Methods in Engineering and Sciences. Springer, 2009.

[Ope13] OpenMP. www.openmp.org, 2013. [Online; accessed 25-September-2013].

[OS00] Steven J Owen and Sunil Saigal. H-morph: an indirect approach to advancing front hex meshing. *International Journal for Numerical Methods in Engineering*, 49(1-2):289–312, 2000.

[Pes02] Charles S Peskin. The immersed boundary method. *Acta numerica*, 11, 2002.

[PVMZ87] J Peraire, M Vahdati, K Morgan, and O.C Zienkiewicz. Adaptive remeshing for compressible flow computations. *Journal of Computational Physics*, 72(2):449 – 466, 1987.

[Qua14] QuantechATZ. www.quantech.es, 2014. [Online; accessed 1-April-2014].

[QZ10] Jin Qian and Yongjie Zhang. Sharp feature preservation in octree-based hexahedral mesh generation for cad assembly models. In *Proceedings of the 19th International Meshing Roundtable*, pages 243–262. Springer, 2010.

[Sam06] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling Series. Elsevier Science, 2006.

[Sch96] R. Schneiders. A grid-based algorithm for the generation of hexahedral element meshes. *Engineering with Computers*, 12:168–177, 1996.

[Sch08] Stefan Schirra. How reliable are practical point-in-polygon strategies? In *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pages 744–755, Berlin, Heidelberg, 2008. Springer-Verlag.

[SF02] Ronald N.Perry Sarah F.Frisken. Simple and efficient traversal methods for quadtrees and octrees. Technical report, MITSUBISHI ELECTRIC RESEARCH LABORATORIES, 2002.

[SG91] Mark S. Shephard and Marcel K. Georges. Automatic three-dimensional mesh generation by the finite octree technique. *International Journal for Numerical Methods in Engineering*, 32(4):709–749, 1991.

[She12] Jonathan Richard Shewchuk. *Combinatorial Scientific Computing*, chapter 10 - Unstructured Mesh Generation, pages 259–285. Chapman and Hall CRC, 2012.

[Som23] DMY Sommerville. Space-filling tetrahedra in euclidean space. In *Proc. Edinburgh Math. Soc*, volume 41, pages 49–57. Cambridge Univ Press, 1923.

[SSA+07] Hari Sundar, Rahul S Sampath, Santi S Adavani, Christos Davatzikos, and George Biros. Low-constant parallel algorithms for finite element simulations using linear octrees. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 25. ACM, 2007.

[ST10] Hang Si and A TetGen. A quality tetrahedral mesh generator and a 3d delaunay triangulator. *UR L http://tetgen. berlios. de*, 2010.

[TOG05] TIANKAI TU, DAVID R. O'HALLARON, and OMAR GHATTAS. Scalable parallel octree meshing for terascale applications. In *Proceedings of the 2005*

ACM/IEEE conference on Supercomputing, SC '05, pages 4–, Washington, DC, USA, 2005. IEEE Computer Society.

[Wea92] Nigel P Weatherill. Delaunay triangulation in computational fluid dynamics. *Computers & Mathematics with Applications*, 24(5):129–150, 1992.

[WK93] S.W. Wang and A.E. Kaufman. Volume sampled voxelization of geometric primitives. In *Visualization, 1993. Visualization '93, Proceedings., IEEE Conference on*, pages 78–84, 1993.

[YS83] M.A. Yerry and M.S. Shephard. A modified quadtree approach to finite element mesh generation. *Computer Graphics and Applications, IEEE*, 3(1):39 –46, jan. 1983.

[YS84] M. A. Yerry and M.S. Shephard. Automatic three-dimensional mesh generation by the modified-octree technique. *International Journal For Numerical Methods in Engineering*, 20:1965–1990, 1984.

[ZP71] OC Zienkiewicz and DV Phillips. An automatic mesh generation scheme for plane and curved surfaces by isoparametricco-ordinates. *International Journal for Numerical Methods in Engineering*, 3(4):519–528, 1971.

[ZTZ05] O.C. Zienkiewicz, R.L. Taylor, and J.Z. Zhu. *The Finite Element Method: Its Basis and Fundamentals: Its Basis and Fundamentals*. Elsevier Science, 2005.

Appendix A

Profiling tables and complete data of examples

This annex compile all the results data from the examples shown in chapter 7.

A.1 Validation example $VE-T1$

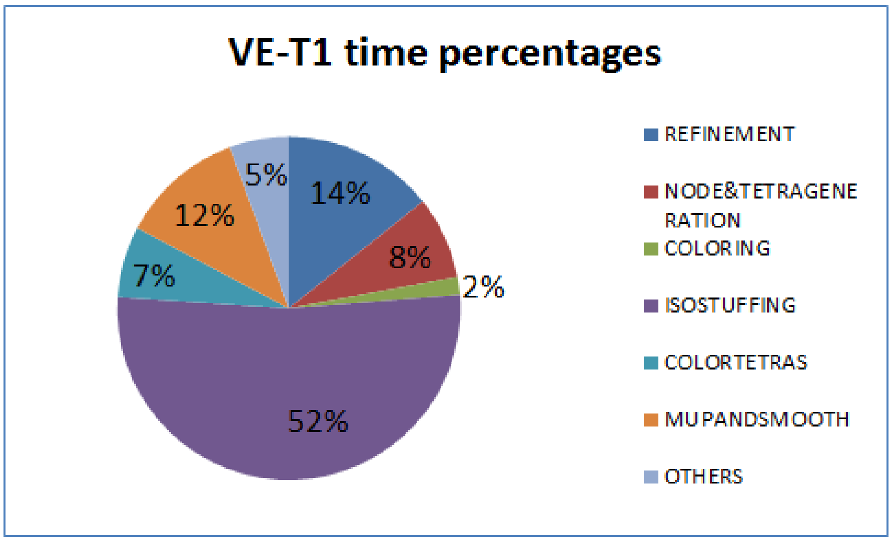


Figure A.1: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE-T1$.

Model data	
Sphericity	0.69
Number of volumes	1
Number of triangles input mesh	18,616
Watertight	*yes*
Mesh size (uol)	1.0
Transition factor	1.0
Result mesh	
Number of tetrahedra	9,474
Number of nodes	2,211
Number of triangles (skin of tetrahedra)	2,150
Number of edges	0
Minimum dihedral angle before make-up and smoothing (degrees)	0.01
Tetrahedra with min dihedral angle < 5 before make-up and smoothing	102
Minimum dihedral angle in final mesh (degrees)	5.68
Memory	
Memory base (Mb)	6.8
Memory peak (Mb)	15.9
Memory ratio	2.3
Algorithm data	
Number of local ray casting operations	0
Number of undetermined tetrahedra	216
Number of tetrahedra from interface cells	25,592
Number of tetrahedra after make-up and smoothing	8,850
Number of octree cells (refining the whole root)	2,605
Number of outer cells (refining the whole root)	1,435
Number of octree cells in the current implementation	2,605
Number of interface octree cells	818
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	0.75
Speed (Mtetrahedra per minute)	0.8

Table A.1: Data of the validation example $VE-T1$.

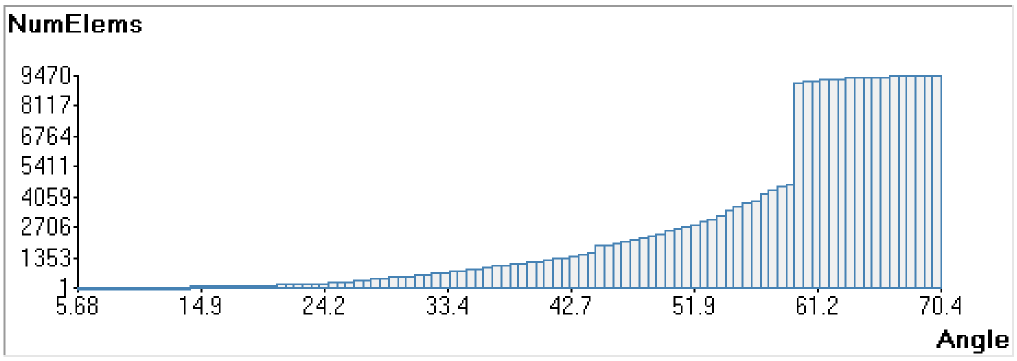


Figure A.2: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - T1$.

A.2 Validation example $VE - T2$

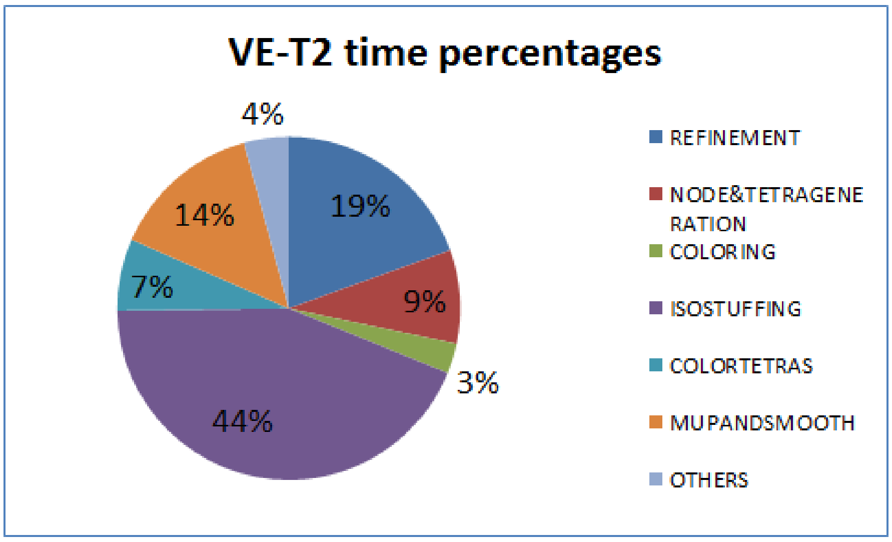


Figure A.3: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE - T2$.

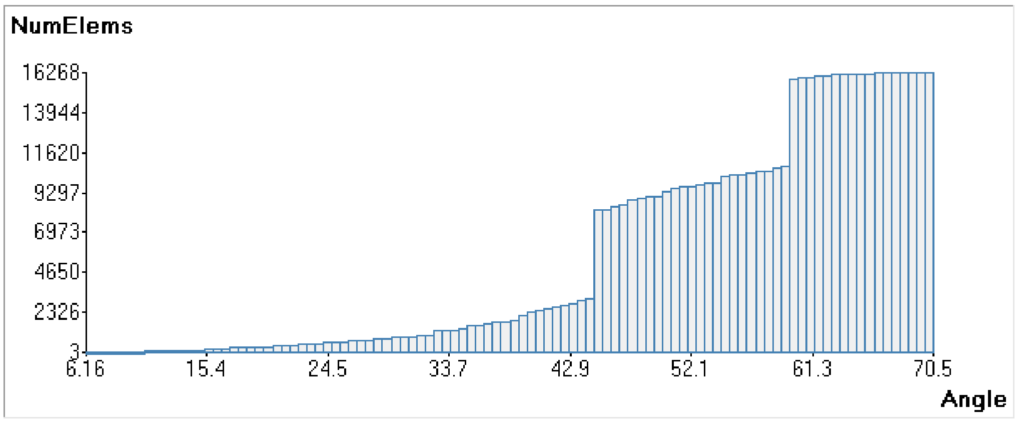


Figure A.4: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - T2$.

Model data	
Sphericity	0.58
Number of volumes	4
Number of triangles input mesh	764
Watertight	*yes*
Mesh size (uol)	$None$
Transition factor	1.0
Result mesh	
Number of tetrahedra	16,304
Number of nodes	3,830
Number of triangles (skin of tetrahedra)	3,883
Number of edges	377
Minimum dihedral angle before make-up and smoothing (degrees)	0.81
Tetrahedra with min dihedral angle < 5 before make-up and smoothing	32
Minimum dihedral angle in final mesh (degrees)	6.16
Memory	
Memory base (Mb)	6.62
Memory peak (Mb)	15.3
Memory ratio	2.32
Algorithm data	
Number of local ray casting operations	0
Number of undetermined tetrahedra	162
Number of tetrahedra from interface cells	29,742
Number of tetrahedra after make-up and smoothing	13,383
Number of octree cells (refining the whole root)	3,116
Number of outer cells (refining the whole root)	1,697
Number of octree cells in the current implementation	2,822
Number of interface octree cells	814
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	0.37
Speed (Mtetrahedra per minute)	2.6

Table A.2: Data of the validation example $VE - T2$.

A.3 Validation example $VE - T3$

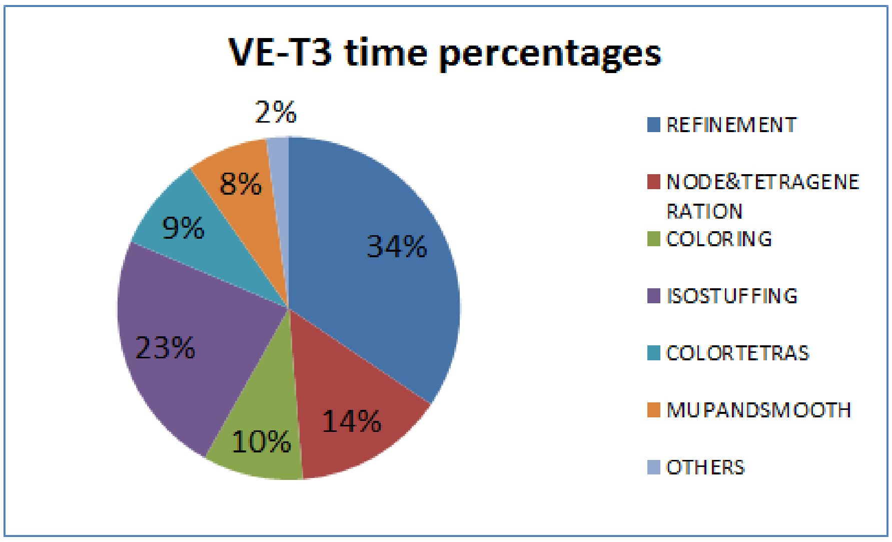


Figure A.5: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE - T3$.

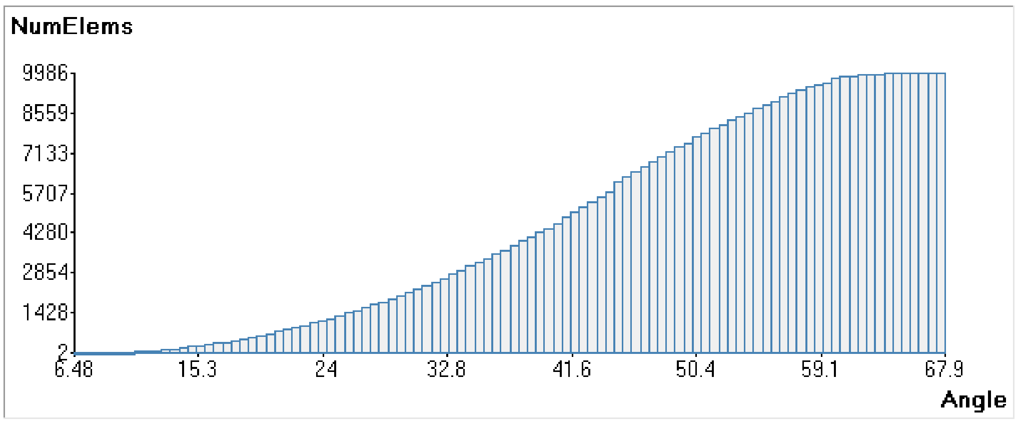


Figure A.6: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - T3$.

Model data	
Sphericity	0.22
Number of volumes	1
Number of triangles input mesh	12,082
Watertight	*yes*
Mesh size (uol)	0.1
Transition factor	1.0
Result mesh	
Number of tetrahedra	9,986
Number of nodes	3,733
Number of triangles (skin of tetrahedra)	6,368
Number of edges	36
Minimum dihedral angle before make-up and smoothing (degrees)	0.0
Tetrahedra with min dihedral angle < 5 before make-up and smoothing	994
Minimum dihedral angle in final mesh (degrees)	6.48
Memory	
Memory base (Mb)	8.0
Memory peak (Mb)	72.5
Memory ratio	9.07
Algorithm data	
Number of local ray casting operations	0
Number of undetermined tetrahedra	2,167
Number of tetrahedra from interface cells	59,437
Number of tetrahedra after make-up and smoothing	9,986
Number of octree cells (refining the whole root)	131,580
Number of outer cells (refining the whole root)	130,046
Number of octree cells in the current implementation	1,512
Number of interface octree cells	8,786
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	2.5
Speed (Mtetrahedra per minute)	0.2

Table A.3: Data of the validation example $VE - T3$.

A.4 Validation example $VE - F1$

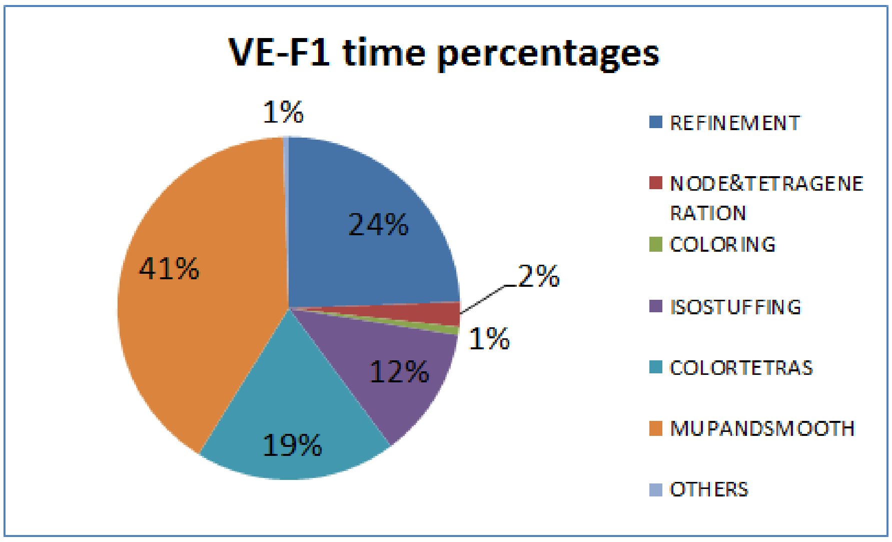


Figure A.7: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE - F1$.

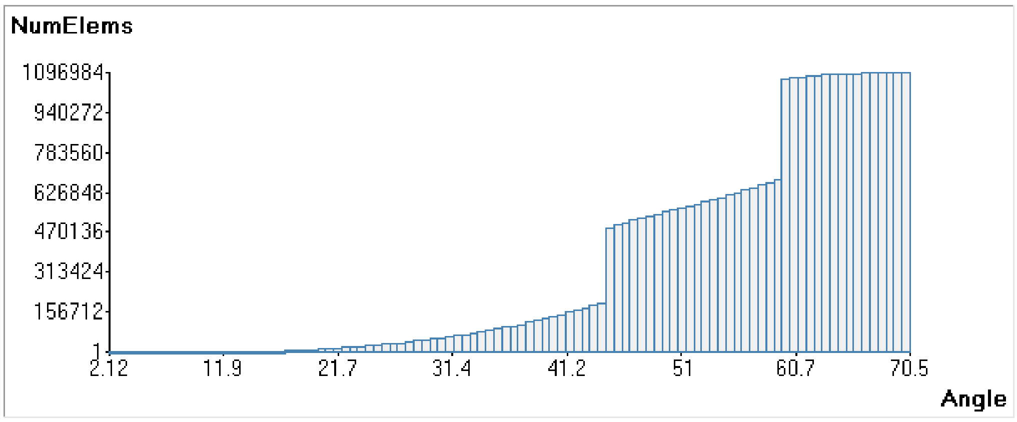


Figure A.8: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - F1$.

Model data	
Sphericity	0.61
Number of volumes	2
Number of triangles input mesh	40,285
Watertight	*yes*
General mesh size (uol)	5.0
Mesh size in the wing surfaces (uol)	0.3
Transition factor	0.6
Result mesh	
Number of tetrahedra	1,097,012
Number of nodes	193,400
Number of triangles (skin of tetrahedra)	99,419
Number of edges	1,716
Minimum dihedral angle before make-up and smoothing (degrees)	0.04
Tetrahedra with min dihedral angle < 5 before make-up and smoothing	3,876
Minimum dihedral angle in final mesh (degrees)	2.12
Tetrahedra with min dihedral angle < 5	3
Memory	
Memory base (Mb)	39.1
Memory peak (Mb)	312.7
Memory ratio	8.0
Algorithm data	
Number of local ray casting operations	0
Number of undetermined tetrahedra	19,038
Number of tetrahedra from interface cells	755,149
Number of tetrahedra after make-up and smoothing	833,117
Number of octree cells (refining the whole root)	73,676
Number of outer cells (refining the whole root)	3,924
Number of octree cells in the current implementation	73,172
Number of interface octree cells	21,928
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	40.9
Speed (Mtetrahedra per minute)	1.6

Table A.4: Data of the validation example $VE - F1$.

A.5 Validation example $VE - F2$

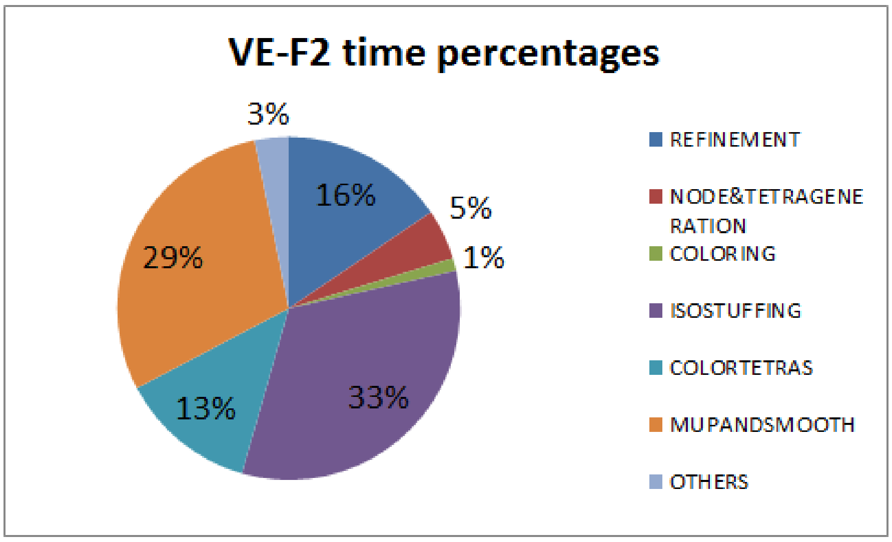


Figure A.9: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE - F2$.

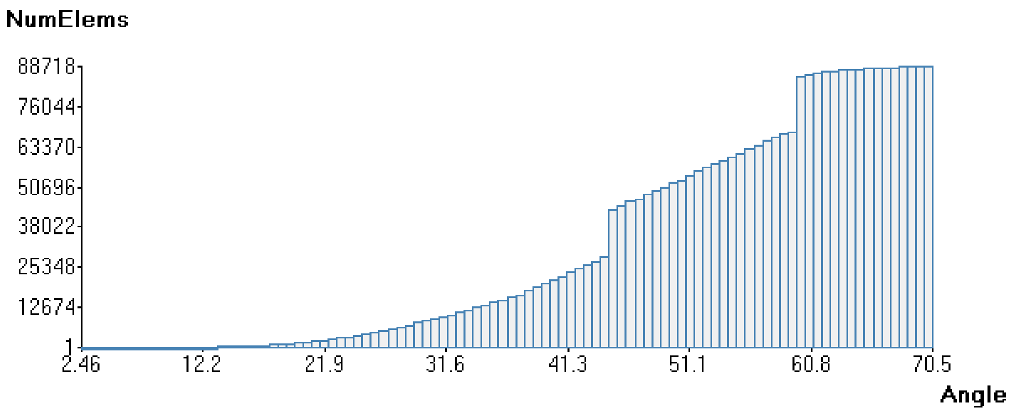


Figure A.10: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - F2$.

Model data	
Sphericity	0.3
Number of volumes	1
Number of triangles input mesh	4,580
Watertight	*yes*
Mesh size (uol)	20.0
Transition factor	1.0
Result mesh	
Number of tetrahedra	88,718
Number of nodes	22,549
Number of triangles (skin of tetrahedra)	26,456
Number of edges	2,833
Minimum dihedral angle before make-up and smoothing (degrees)	0.04
Tetrahedra with min dihedral angle < 5 before make-up and smoothing	714
Minimum dihedral angle in final mesh (degrees)	2.46
Tetrahedra with min dihedral angle < 5	1
Memory	
Memory base (Mb)	10.1
Memory peak (Mb)	78.0
Memory ratio	7.7
Algorithm data	
Number of local ray casting operations	10
Number of undetermined tetrahedra	4,192
Number of tetrahedra from interface cells	196,690
Number of tetrahedra after make-up and smoothing	87,073
Number of octree cells (refining the whole root)	18,754
Number of outer cells (refining the whole root)	10,930
Number of octree cells in the current implementation	16,941
Number of interface octree cells	5,357
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	4.3
Speed (Mtetrahedra per minute)	1.2

Table A.5: Data of the validation example $VE - F2$.

A.6 Validation example $VE - W1$

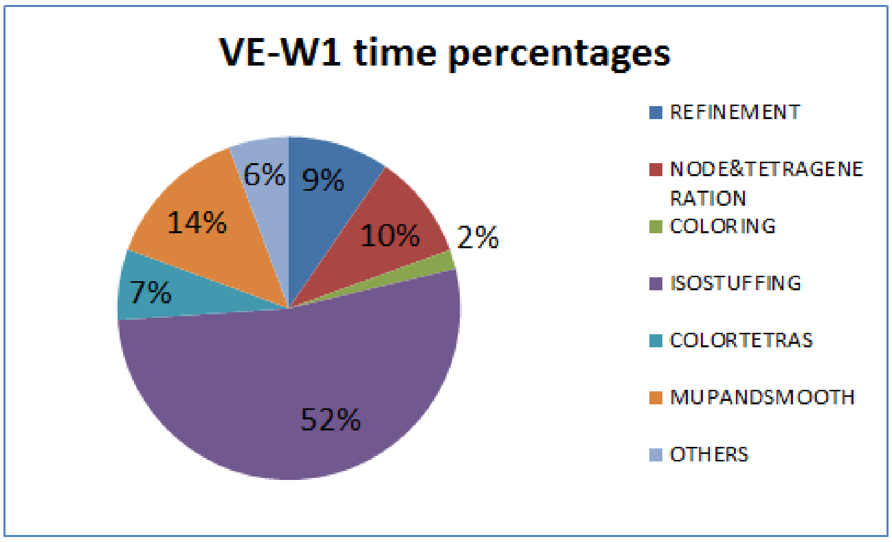


Figure A.11: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE - W1$.

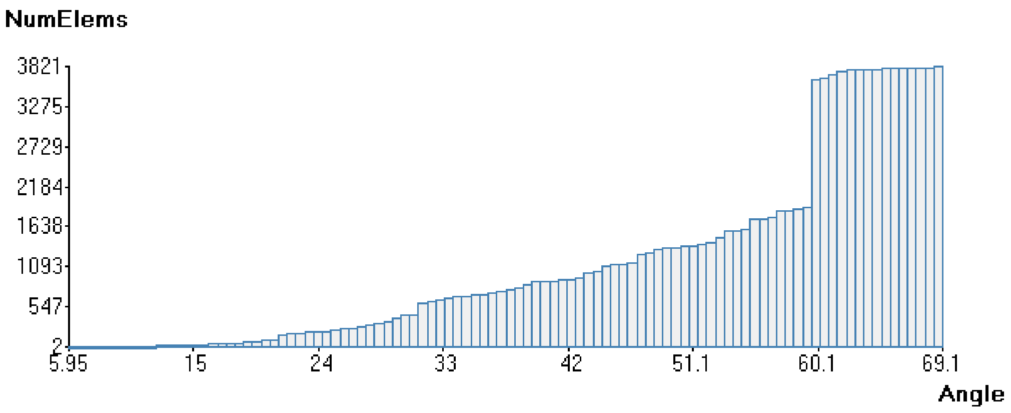


Figure A.12: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - W1$.

Model data	
Sphericity	0.8
Number of volumes	1
Number of triangles input mesh	314
Watertight	*no*
Mesh size (uol)	0.7
Transition factor	1.0
Result mesh	
Number of tetrahedra	3,823
Number of nodes	978
Number of triangles (skin of tetrahedra)	1,130
Number of edges	172
Minimum dihedral angle before make-up and smoothing (degrees)	4.12
Tetrahedra with min dihedral angle < 5 before make-up and smoothing	3
Minimum dihedral angle in final mesh (degrees)	5.95
Memory	
Memory base (Mb)	5.46
Memory peak (Mb)	7.67
Memory ratio	1.4
Algorithm data	
Number of local ray casting operations	0
Number of undetermined tetrahedra	82
Number of tetrahedra from interface cells	9,290
Number of tetrahedra after make-up and smoothing	3,631
Number of octree cells (refining the whole root)	1,534
Number of outer cells (refining the whole root)	1,150
Number of octree cells in the current implementation	1,142
Number of interface octree cells	272
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	0.1
Speed (Mtetrahedra per minute)	2.6

Table A.6: Data of the validation example $VE - W1$.

A.7 Validation example $VE - W2$

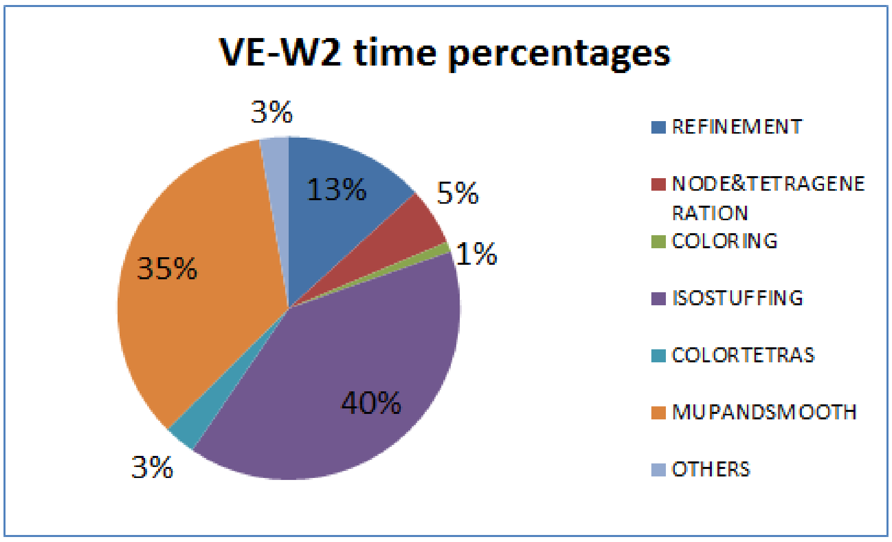


Figure A.13: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE - W2$.

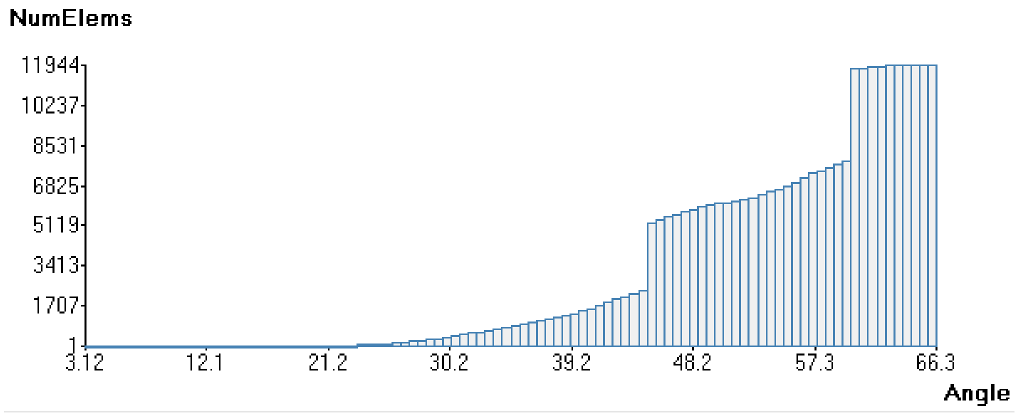


Figure A.14: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - W2$.

Model data	
Sphericity	1.0
Number of volumes	1
Number of triangles input mesh	1,858
Watertight	*no*
Mesh size (uol)	1.0
Transition factor	1.0
Result mesh	
Number of tetrahedra	11,946
Number of nodes	2,557
Number of triangles (skin of tetrahedra)	1,892
Number of edges	0
Minimum dihedral angle before make-up and smoothing (degrees)	4.72
Tetrahedra with min dihedral angle < 5 before make-up and smoothing	1
Minimum dihedral angle in final mesh (degrees)	3.12
Tetrahedra with min dihedral angle < 5	2
Memory	
Memory base (Mb)	6.1
Memory peak (Mb)	13.4
Memory ratio	2.2
Algorithm data	
Number of local ray casting operations	1
Number of undetermined tetrahedra	2
Number of tetrahedra from interface cells	26,199
Number of tetrahedra after make-up and smoothing	11,552
Number of octree cells (refining the whole root)	2,080
Number of outer cells (refining the whole root)	844
Number of octree cells in the current implementation	2,080
Number of interface octree cells	770
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	0.36
Speed (Mtetrahedra per minute)	2.0

Table A.7: Data of the validation example $VE - W2$.

A.8 Validation example $VE - W3$

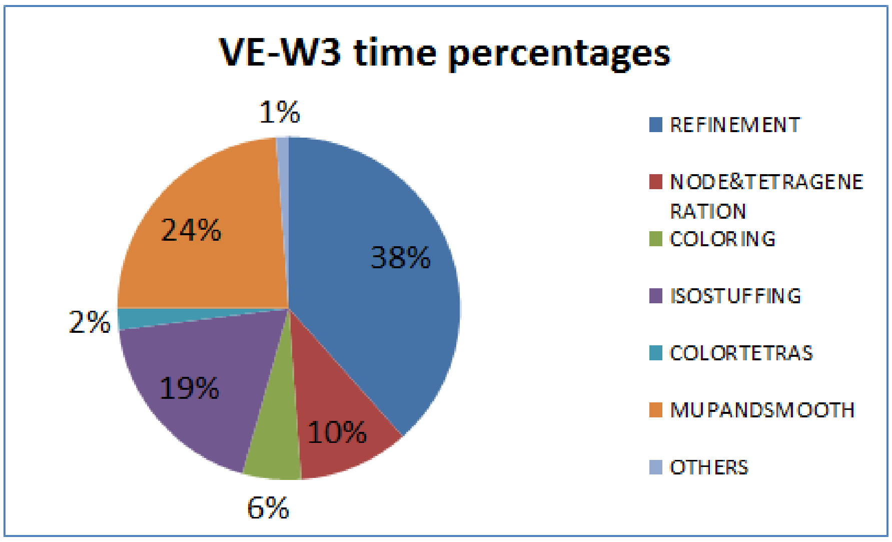


Figure A.15: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE - W3$.

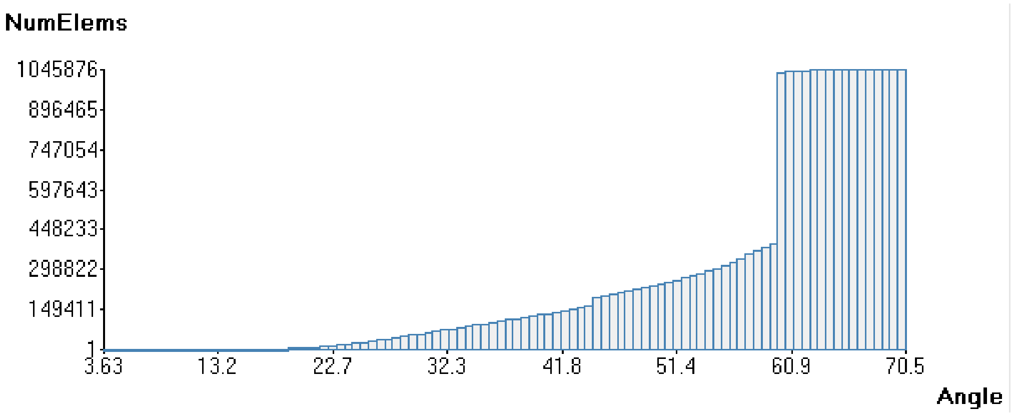


Figure A.16: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - W3$.

Model data	
Sphericity	1.0
Number of volumes	1
Number of triangles input mesh	37,884
Watertight	*yes*
Mesh size (uol)	1.0
Transition factor	0.7
Result mesh	
Number of tetrahedra	1,045,878
Number of nodes	214,671
Number of triangles (skin of tetrahedra)	146,198
Number of edges	265
Minimum dihedral angle before make-up and smoothing (degrees)	0.0
Minimum dihedral angle in final mesh (degrees)	18,751
Memory	
Memory base (Mb)	45.2
Memory peak (Mb)	80.1
Memory ratio	17.7
Algorithm data	
Number of local ray casting operations	0
Number of undetermined tetrahedra	931
Number of tetrahedra from interface cells	1,381,308
Number of tetrahedra after make-up and smoothing	734,994
Number of octree cells (refining the whole root)	1,375,830
Number of outer cells (refining the whole root)	1,276,251
Number of octree cells in the current implementation	175,820
Number of interface octree cells	42,295
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	65.6
Speed (Mtetrahedra per minute)	1.0

Table A.8: Data of the validation example $VE - W3$.

A.9 Validation example $VE - C1$

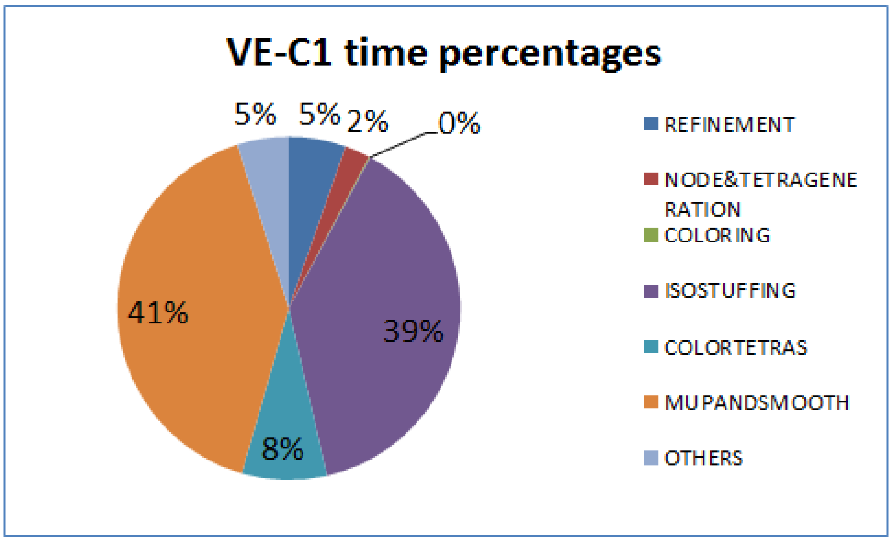


Figure A.17: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE - C1$.

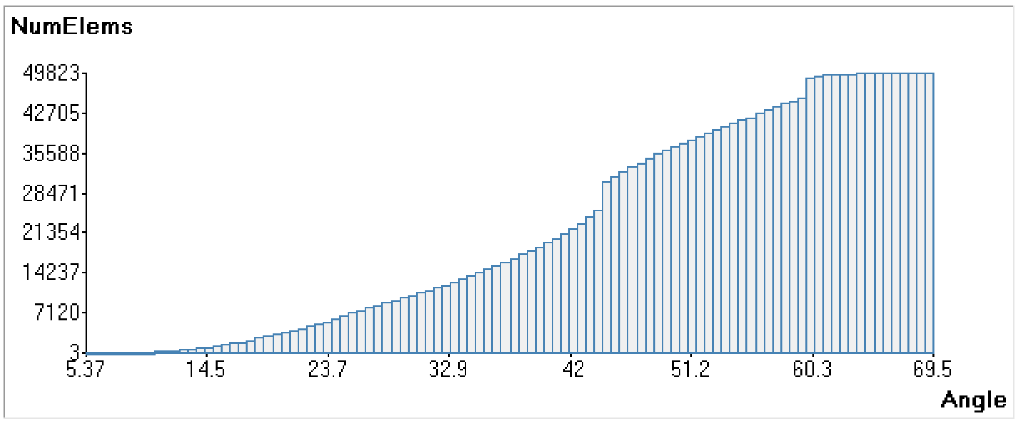


Figure A.18: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - C1$.

Model data	
Sphericity	0.33
Number of volumes	32
Number of triangles input mesh	272
Watertight	*yes*
Mesh size	2
Transition factor	0.7
Result mesh	
Number of tetrahedra	49,823
Number of nodes	10,129
Number of triangles (skin of tetrahedra)	15,880
Number of edges	1,218
Minimum dihedral angle before make-up and smoothing (degrees)	1.54
Tetrahedra with min dihedral angle < 5 before make-up and smoothing	166
Minimum dihedral angle in final mesh (degrees)	5.37
Memory	
Memory base (Mb)	8.8
Memory peak (Mb)	30.0
Memory ratio	3.4
Algorithm data	
Number of local ray casting operations	0
Number of undetermined tetrahedra	143
Number of tetrahedra from interface cells	48,511
Number of tetrahedra after make-up and smoothing	49,823
Number of octree cells (refining the whole root)	3,711
Number of outer cells (refining the whole root)	1,245
Number of octree cells in the current implementation	3,711
Number of interface octree cells	2,262
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	1.8
Speed (Mtetrahedra per minute)	1.7

Table A.9: Data of the validation example $VE - C1$.

A.10 Validation example $VE - C2$

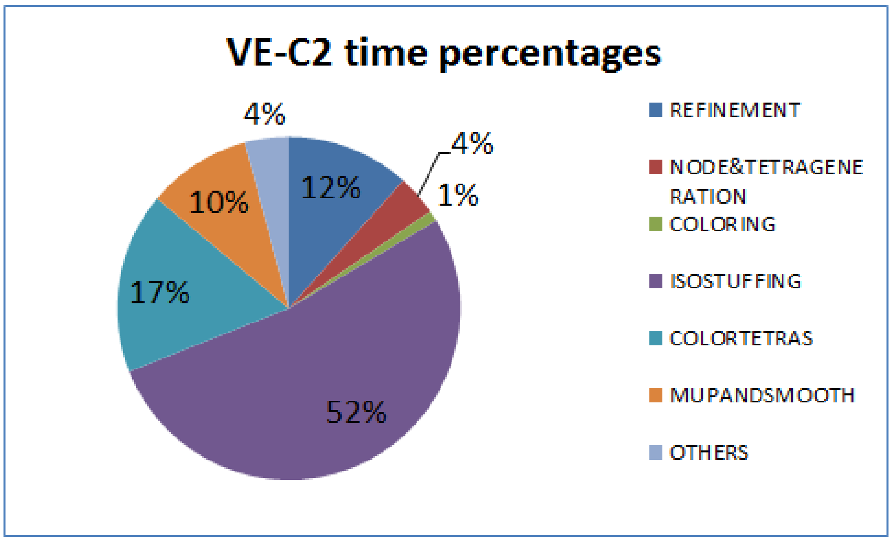


Figure A.19: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE - C2$.

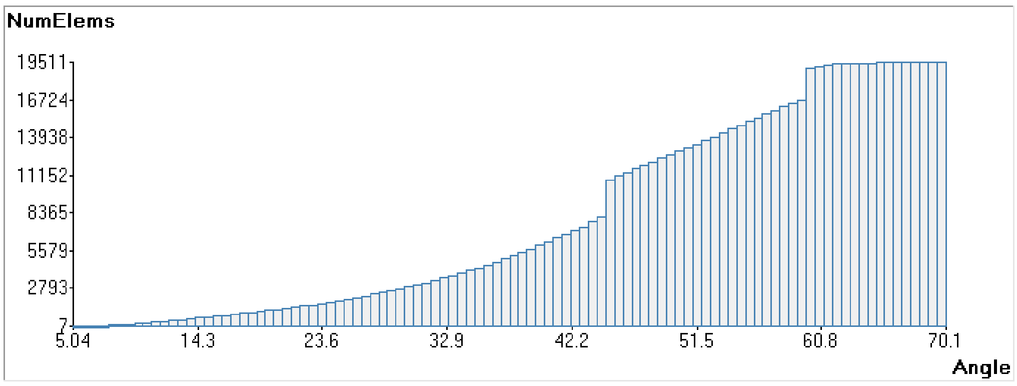


Figure A.20: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - C2$.

Model data	
Sphericity	0.29
Number of volumes	1
Number of triangles input mesh	54,488
Watertight	*yes*
Mesh size	1.0
Transition factor	1.0
Result mesh	
Number of tetrahedra	19,512
Number of nodes	5,888
Number of triangles (skin of tetrahedra)	8,466
Number of edges	89
Minimum dihedral angle before make-up and smoothing (degrees)	0.01
Tetrahedra with min dihedral angle < 5 before make-up and smoothing	1,150
Minimum dihedral angle in final mesh (degrees)	5.04
Memory	
Memory base (Mb)	7.7
Memory peak (Mb)	36.2
Memory ratio	4.7
Algorithm data	
Number of local ray casting operations	0
Number of undetermined tetrahedra	1,478
Number of tetrahedra from interface cells	52,291
Number of tetrahedra after make-up and smoothing	19,512
Number of octree cells (refining the whole root)	4,747
Number of outer cells (refining the whole root)	2,577
Number of octree cells in the current implementation	4,677
Number of interface octree cells	1,895
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	3.5
Speed (Mtetrahedra per minute)	0.3

Table A.10: Data of the validation example $VE - C2$.

A.11 Validation example $VE - C3$

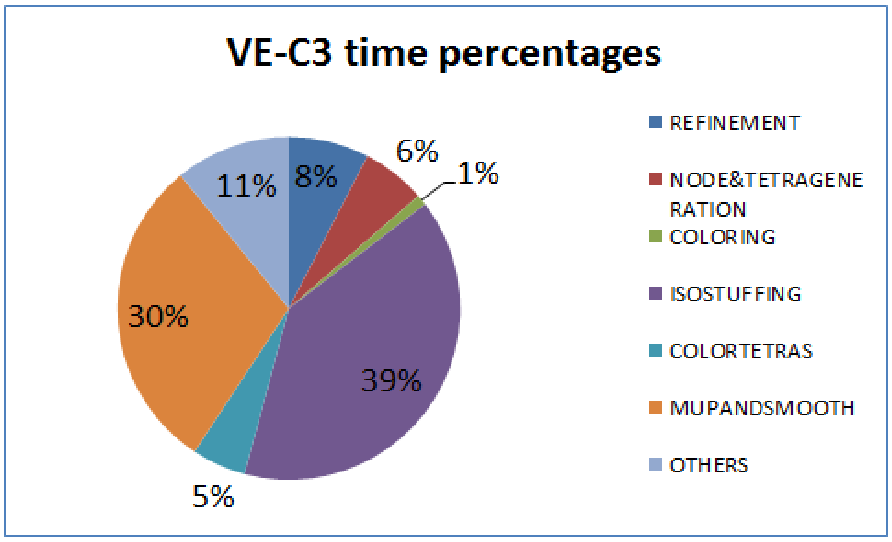


Figure A.21: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE - C3$.

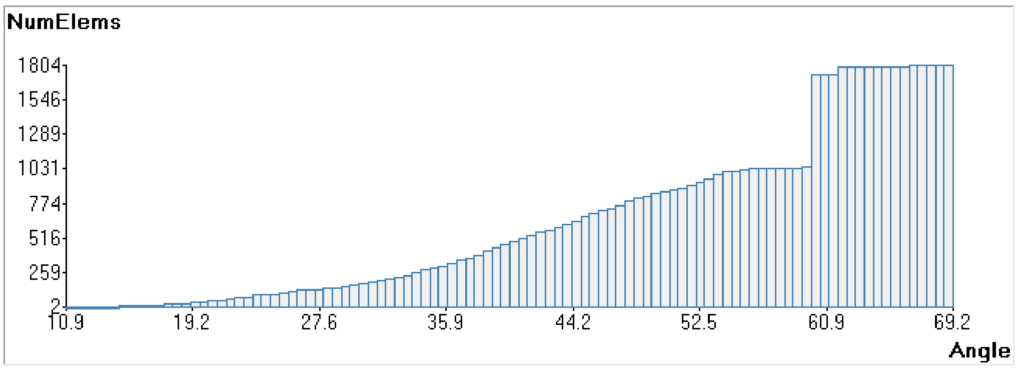


Figure A.22: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - C3$.

Model data	
Sphericity	0.81
Number of volumes	1
Number of triangles input mesh	24
Watertight	*no*
Mesh size	1.0
Transition factor	1.0
Result mesh	
Number of tetrahedra	1,828
Number of nodes	501
Number of triangles (skin of tetrahedra)	652
Number of edges	110
Minimum dihedral angle before make-up and smoothing (degrees)	2.9
Tetrahedra with min dihedral angle < 5 before make-up and smoothing	18
Minimum dihedral angle in final mesh (degrees)	10.1
Memory	
Memory base (Mb)	5.7
Memory peak (Mb)	7.1
Memory ratio	1.23
Algorithm data	
Number of local ray casting operations	7
Number of undetermined tetrahedra	41
Number of tetrahedra from interface cells	6,608
Number of tetrahedra after make-up and smoothing	1,828
Number of octree cells (refining the whole root)	589
Number of outer cells (refining the whole root)	343
Number of octree cells in the current implementation	589
Number of interface octree cells	198
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	0.07
Speed (Mtetrahedra per minute)	1.5

Table A.11: Data of the validation example $VE - C3$.

A.12 Validation example $VE - I1$

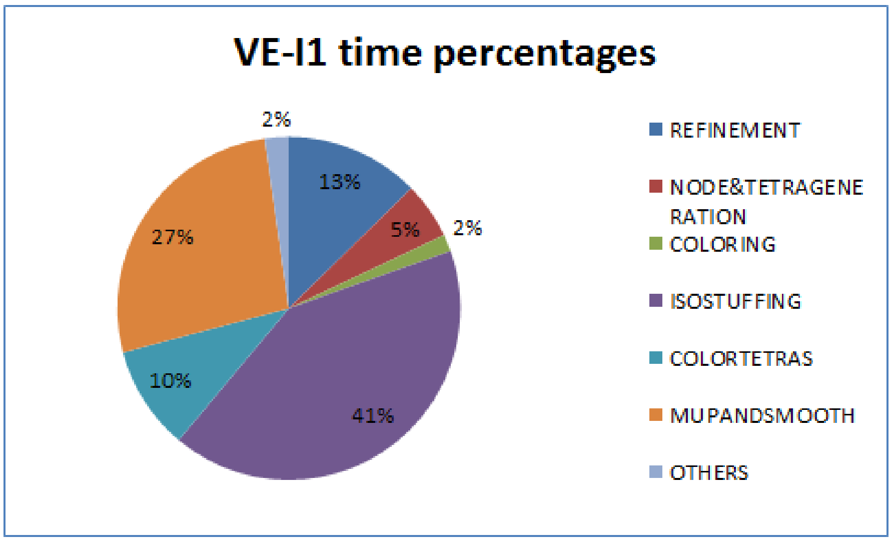


Figure A.23: Time percentages of different parts of the algorithm for generating the mesh of validation example $VE - I1$.

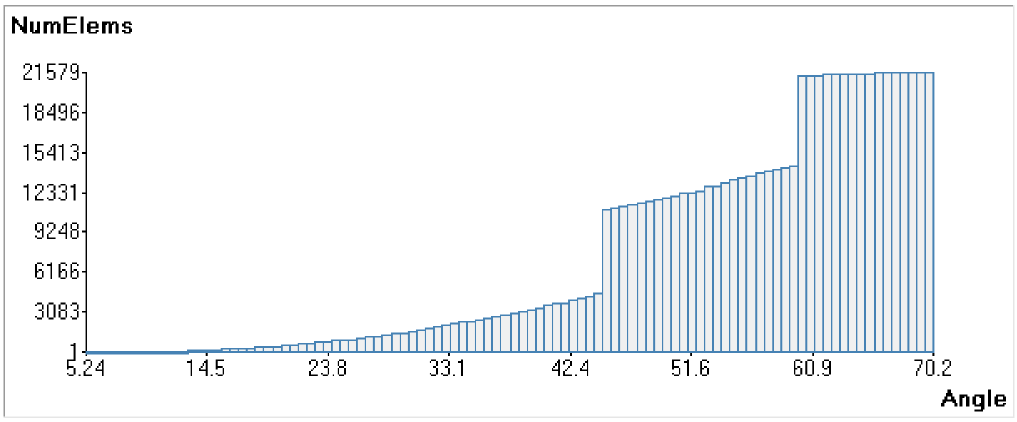


Figure A.24: Distribution of minimum dihedral angles in the mesh generated in the validation example $VE - I1$.

Model data	
Sphericity	0.74
Number of volumes	1
Number of triangles input mesh	368
Watertight	*yes*
Mesh size (uol)	5,000
Transition factor	0.7
Result mesh	
Number of tetrahedra	21,630
Number of nodes	4,156
Number of triangles (skin of tetrahedra)	2,388
Number of edges	362
Minimum dihedral angle before make-up and smoothing (degrees)	0.24
Tetrahedra with min dihedral angle < 5 before make-up and smoothing	44
Minimum dihedral angle in final mesh (degrees)	5.24
Memory	
Memory base (Mb)	8.2
Memory peak (Mb)	14.3
Memory ratio	1.75
Algorithm data	
Number of local ray casting operations	0
Number of undetermined tetrahedra	96
Number of tetrahedra from interface cells	21,802
Number of tetrahedra after make-up and smoothing	19,456
Number of octree cells (refining the whole root)	2,017
Number of outer cells (refining the whole root)	621
Number of octree cells in the current implementation	1,884
Number of interface octree cells	533
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	0.38
Speed (Mtetrahedra per minute)	3.5

Table A.12: Data of the validation example $VE - I1$.

A.13 Validation example $VE - E1$

Model data	
Number of volumes	1
Number of triangles input mesh	444498
Watertight	*yes*
General mesh size (uol)	0.2
Mesh size in the boundaries (uol)	0.1
Transition factor	1.0
Result mesh	
Number of tetrahedra	11,511,840
Number of nodes	1,967,445
Memory	
Memory base (Mb)	263.6
Memory peak (Mb)	2,194
Memory ratio	8.32
Algorithm data	
Number of local ray casting operations	0
Number of octree cells in the current implementation	844,341
Number of interface octree cells	400,060
Time and speed	
Number of threads	1
Total time for generating the mesh (s)	17.6
Speed (Mtetrahedra per minute)	39.3

Table A.13: Data of the validation example $VE - E1$.

A.14 Validation example $VE - S1$

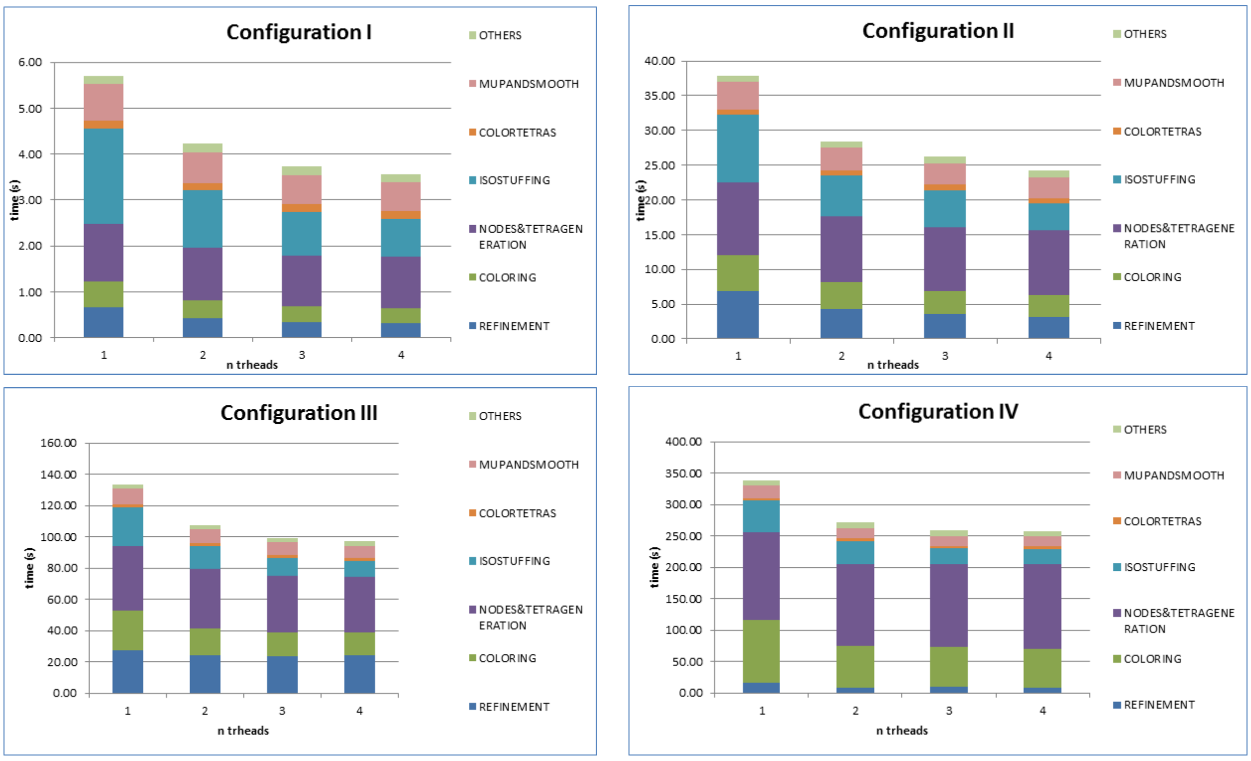


Figure A.25: Times consumed in the different parts of the meshing algorithm for the validation example $VE - S1$.

Configuration	I	II	III	IV
Model data				
Sphericity	1.0	1.0	1.0	1.0
Number of volumes	1	1	1	1
Number of triangles input mesh	21608	98838	234308	578968
Watertight	*yes*	*yes*	*yes*	*yes*
Mesh size	0.18	0.085	0.055	0.035
Transition factor	1.0	1.0	1.0	1.0
Result mesh				
Number of tetrahedra (millions)	1.1	10.3	38.0	103.5
Number of nodes (millions)	0.2	1.8	6.4	17.5
Number of triangles (skin of tetrahedra)	37,496	171,854	414,000	807,386
Number of edges	0	0	0	0
Min angle (mda) before smoothing (deg)	17.2	16.6	12.9	15.3
Tetrahedra with mda < 5 before smoothing	0	0	0	0
Min dihedral angle in final mesh (deg)	17.2	16.6	16.8	12.3
Memory				
Memory base (Mb)	30	228	810	2,190
Memory peak (Mb)	211	1,850	6,700	18,800
Memory ratio	6.7	8.1	8.2	8.5
Algorithm data				
Num of local ray casting operations	0	0	0	0
Num of undetermined tetrahedra	0	0	0	0
Num of tetrahedra from interface cells	483,552	2,208,256	5,282,880	10,321,968
Num of tetrahedra after smoothing	258,856	1,218,400	2,927,026	5,758,981
Num of octree cells (refining the whole root)	186,376	1,774,088	6,371,268	16,777,216
Num of outer cells (refining the whole root)	89,220	889,128	3,145,340	8,026,224
Num of octree cells current implementation	123,096	1,000,560	3,493,596	9,266,720
Num of interface octree cells	14,516	65,216	155,748	304,376

Table A.14: Data of the validation example $VE - S1$.

A.15 Validation example $VE - S2$

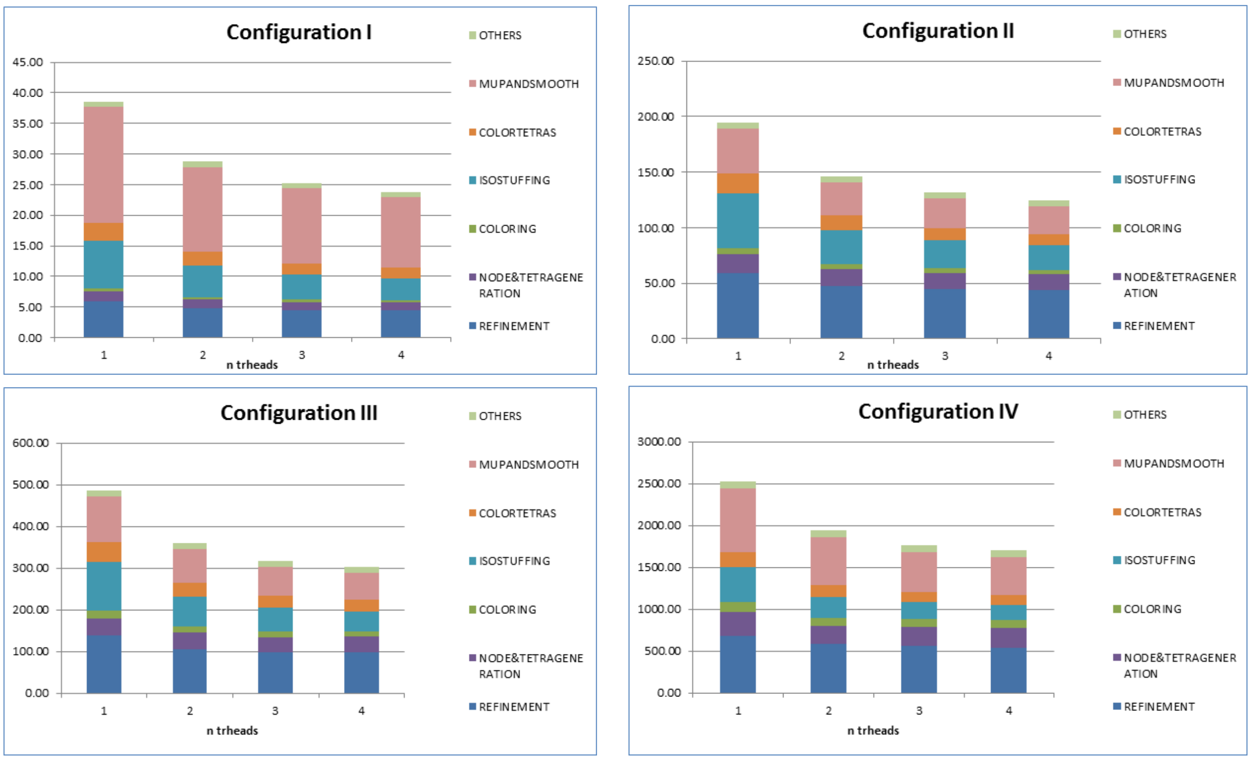


Figure A.26: Times consumed in the different parts of the meshing algorithm for the validation example $VE - S2$.

Configuration	*I*	*II*	*III*	*IV*
Model data				
Sphericity	0.54	0.54	0.54	1.0
Number of volumes	1	1	1	1
Number of triangles input mesh	54260	684810	1475114	5948644
Watertight	*yes*	*yes*	*yes*	*yes*
General mesh size (uol)	50.0	10.0	10.0	5.0
Mesh size in buildings (uol)	10.0	3.0	2.0	1.0
Transition factor	0.7	0.7	0.7	0.7
Result mesh				
Number of tetrahedra (millions)	1.1	14.6	28.1	84.7
Number of nodes millions)	0.2	2.7	5.4	1.7
Number of triangles (skin of tetrahedra)	182,886	991,532	2,644,132	8,484,054
Number of edges	20,380	51,911	88,855	171,960
Min d. angle (mda) before smoothing (deg)	2.8	1.4	1.5	1.7
Tetrahedra with mda < 5 before smoothing	99	167	216	512
Min dihedral angle in final mesh (deg)	6.4	5.1	5.0	5.0
Memory				
Memory base (Mb)	40	353	697	2,112
Memory peak (Mb)	492	3,200	7,120	25,500
Memory ratio	12.4	9.0	10.2	12.0
Algorithm data				
Num of local ray casting operations	0	0	0	0
Num of undetermined tetrahedra	15571	37132	72463	146436
Num of tetrahedra from interface cells (M)	1.3	6.8	20.3	77.9
Num of tetrahedra after smoothing	851,720	4,149,022	13,040,530	55,564,706
Num of octree cells (refining the whole root)	116,600	1,476,735	2,875,503	9,045,604
Num of outer cells (refining the whole root)	21,170	242,640	454,793	2,112,866
Num of octree cells current implementation	113,709	1,422,660	2,763,293	8,615,832
Num of interface octree cells	39,750	220,517	690,257	2,727,568

Table A.15: Data of the validation example $VE - S2$.

A.16 Racing car

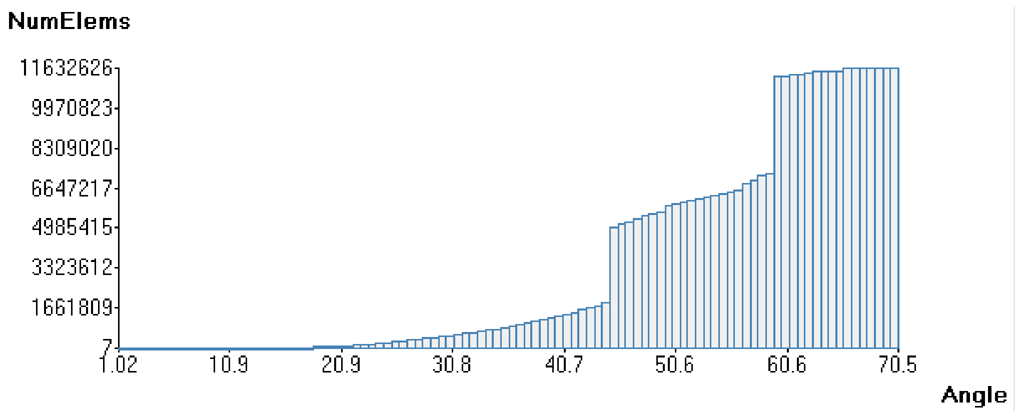


Figure A.27: Distribution of minimum dihedral angles in the mesh generated in the racing car example.

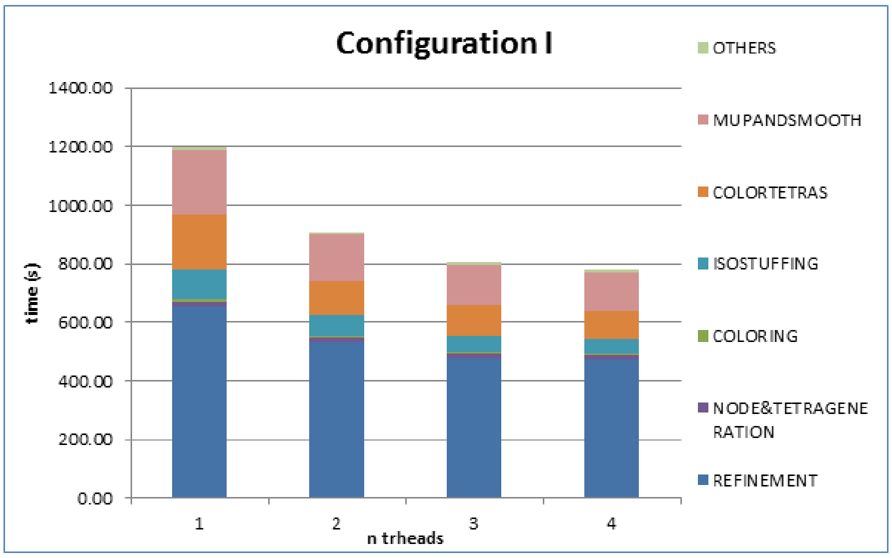


Figure A.28: Times consumed in the different parts of the meshing algorithm for the racing car example.

Model data	
Sphericity	0.34
Number of volumes	1
Number of triangles input mesh	1,076,114
Watertight	*yes*
Mesh general size (uol)	None
Mesh size in skin of the car and floor (uol)	10.0
Mesh size in outlet surface (uol)	20.0
Transition factor	0.6
Result mesh	
Number of tetrahedra (millions)	11.6
Number of nodes (millions)	2.5
Number of triangles (skin of tetrahedra)	1,831,916
Number of edges	34,658
Minimum dihedral angle before make-up and smoothing (degrees)	0.0
Tetrahedra with min dihedral angle < 5 before mesh improvement	56,281
Minimum dihedral angle in final mesh (degrees)	1.02
Tetrahedra with min dihedral angle < 5	424
Memory	
Memory base (Mb)	326
Memory peak (Mb)	4,880
Memory ratio	14.8
Algorithm data	
Number of local ray casting operations	0
Number of undetermined tetrahedra	310,388
Number of tetrahedra from interface cells	12,768,613
Number of tetrahedra after make-up and smoothing	8,528,078
Number of octree cells (refining the whole root)	1,233,352
Number of outer cells (refining the whole root)	282,364
Number of octree cells in the current implementation	1,202,265
Number of interface octree cells	409,315

Table A.16: Data of the racing car example.

A.17 Barcelona model

Model data	
Sphericity	1
Number of volumes	1
Number of triangles input mesh	20,415
Watertight	*no*
General mesh size (uol)	*None*
Mesh size in buildings (uol)	150
Mesh size in terrain (uol)	200
Transition factor	1
Result mesh	
Number of tetrahedra (millions)	25,145,275
Number of nodes	4,354,677
Number of triangles (skin of tetrahedra)	200,020
Number of edges	1,872
Minimum dihedral angle before make-up and smoothing (degrees)	1,111
Minimum dihedral angle in final mesh (degrees)	1,111
Memory	
Memory base (Gb)	0.58
Memory peak (Gb)	5.3
Memory ratio	9.1
Algorithm data	
Number of local ray casting operations	568,557
Number of undetermined tetrahedra	1,443
Number of tetrahedra from interface cells	2,519,564
Number of tetrahedra after make-up and smoothing	1,772,908
Number of octree cells (refining the whole root)	1,877,492
Number of outer cells (refining the whole root)	128,639
Number of octree cells in the current implementation	1,819,392
Number of interface octree cells	44,422

Table A.17: Data of the Barcelona model example.