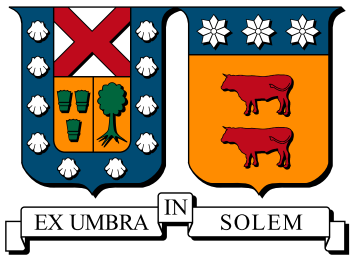


REPAIRING OCTREE BOUNDARY TRANSITION REGIONS COMPOSED OF DIFFERENT TYPES OF ELEMENTS

Tesis de Grado presentada por

Esteban Felipe Manuel Daines Ostria

como requisito parcial para optar al grado de

Magíster en Ciencias de la Ingeniería Informática

Director de Tesis:

Claudio Lobos

NOVEMBER 2018

Acknowledgements

My most sincere gratitude to all the people that supported me in all my years of study.

To my Father for being my role model. I will always be proud of being your son.

To my Mother for the all support and caring, for helping me even in the simplest of problems. Love you Mom.

To all my Friends. Specially Cristopher, Natalia and Roberto, for all these years of company and fun in our office.

And last but not least, to María José who has been there with me all these years, in the good and bad times. Thanks for all the beautiful moments we experienced together.

Resumen

Para realizar simulaciones utilizando métodos numéricos, como volúmenes finitos y elementos finitos, se requiere de una discretización del espacio o malla geométrica. Las mallas geométricas pueden ser generadas con distintos tipos de elementos (tetraedros, pirámides, prismas, hexaedros, entre otros).

Algoritmos basados en octree dividen recursivamente el espacio en 8 o 27 hexaedros (Octantes). El numero de veces que se aplica el proceso de división a un octantes se denomina su Nivel de Refinamiento (*RL*). Cuando una malla presenta octantes con diferentes *RL* es requerido manejar la transición entre regiones finas y gruesas de la malla. Para realizar esto, se aplica patrones de transición a octantes con vecinos de diferentes *RL*.

Cuando usamos el proceso de división en 27, los patrones de transición se pueden generar utilizando solo hexaedros. En el caso de la división en 8, los patrones se deben generar usando diferentes tipos de elementos (elementos mixtos).

Se asegura la validez de los elementos internos de una transición cuando el octante al que se aplica consiste de un hexaedro. Sin embargo, esto puede no ser el caso cuando el octante se encuentra en el borde del dominio, especialmente en regiones cóncavas.

En este trabajo introducimos una nueva técnica de proyección de nodos para reparar elementos de regiones de transición que se encuentran en el borde de la malla. Experimentos en diferentes instancias con variadas métricas de calidad son requeridos para validar la técnica propuesta.

Abstract

To be able to perform simulations with numerical methods, like finite volumes and finite elements, a space discretization or mesh is required. These meshes can be generated using different type of elements (tetrahedra, pyramids, prisms, hexahedra, among others).

Octree–based algorithms recursively divide the space in 8 or 27 hexahedra (octants). The number of times the split process is applied over an octant is the Refinement Level (*RL*). When a mesh presents octants of different *RL*, it is required to manage the transition between fine and coarse regions of the mesh. To do this, transition patterns are applied over octants with neighbors of different *RL*.

When using the 27–split process this can be done only using hexahedra, however in the case of 8–split process this must be done using different types of elements (mixed–elements).

The validity of the elements in the transition is ensured when the octant is a regular hexahedron. However, this may not be true when the octant is at the boundary of the domain, specially in concave regions.

In this work we introduce a novel node projection technique in order to repair the boundary elements of transition regions in the mesh. Tests on different instances with varied element metrics is required to validate the proposed technique.

Table of Contents

List of Tables

List of Figures

Glossary

Conformal Mesh	Mesh where all nodes, edges and faces are consistent for all adjacent elements.
Domain	Space represented by a surface mesh.
Element Quality	Measure of how deformed an element is in comparisson with its most regular representation.
Mesh	Representation of the volume of a 3D object. Is a set of polyhedra.
Node	Vertex in a mesh.
Node Projection	Orthogonal projection of a node onto the domain boundary.
Non–conformal Mesh	Mesh that does not fulfill a conformal mesh description.
Refinement Level	A number that denotes the quantity of elements generated on a mesh. In recursive division meshing techniques, it denotes the number of divisions done to an element.
Surface Mesh	Outside representation of a 3D object. It is a set of edges, vertex and faces.
Transition Pattern	Set of element that is used to maintain conformity zones with different refinement levels.

Chapter 1

Introduction

Computational simulations can be used in medicine to support the understanding of complex biological systems using the information obtained from magnetic resonance images or computer tomographies.

To simulate physical phenomena methods, like finite volumes and finite elements, are used. This two methods require a discretization of the space. This discretization is called a mesh. Choosing the correct mesh can have an important impact in the simulation, like an increment of precision or lower computational times.

There are multiple techniques for mesh generation. The types of elements in mesh generation can vary from one technique to another. The type of element(s) used have impact on the number of nodes and connectivity level among them. These two variables will affect the time performance of simulations.

Mesh refinement refers to the quantity of nodes that the mesh has. Higher mesh refinement means the detail of the elements is higher and the representation could better approximate the domain. Since more nodes on the mesh affect the time performance of simulations, some mesh generation techniques use partial refinement. This means that the regions of the mesh that are more important for the simulations will have a higher concentration of nodes than the rest. An example of partial refinement can be seen in Figure 1.1.

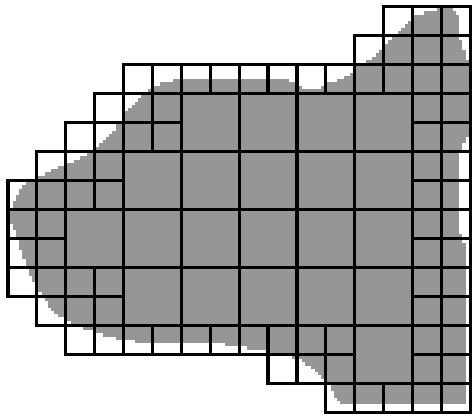


Figure 1.1: Partial refinement.

A conformal mesh has all the nodes of its edges or faces consistent for all adjacent elements. A conformal mesh is continuous in all its nodes. A 2D example of a non–conformal mesh is shown in Figure 1.1. In Figure 1.2a we show a three dimensional example of a non–conformal mesh. When a simulation is done in a non–conformal mesh some inconsistencies and errors could happen, as shown in Figure 1.2b.

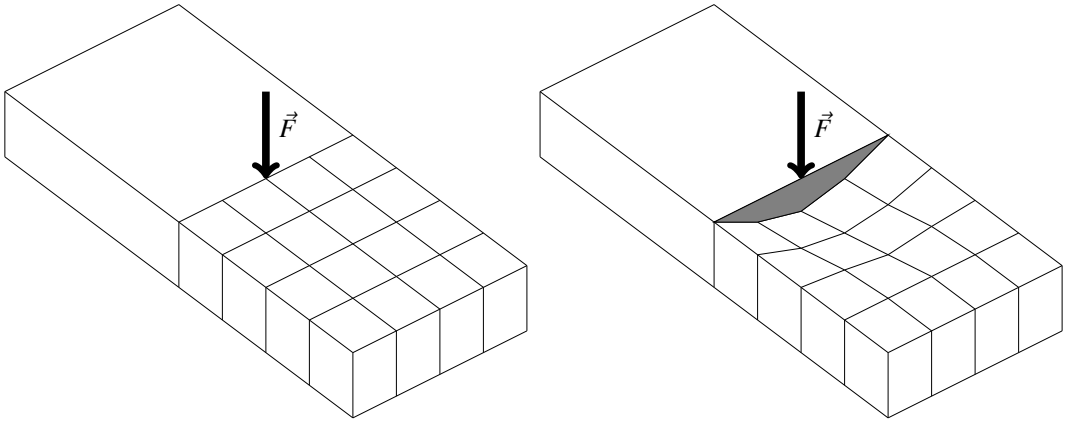
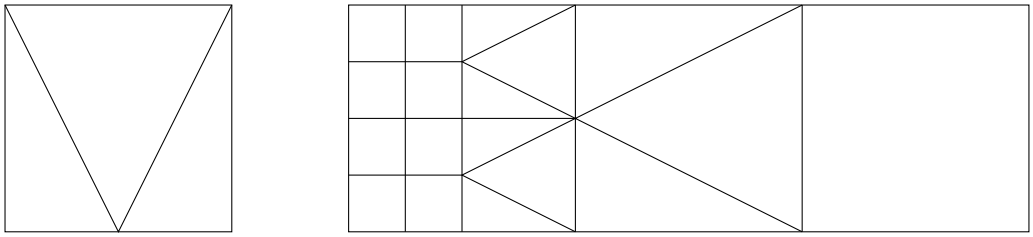


Figure 1.2: Continuity problem with non–conformal mesh.

To make conformal meshes with partial refinement we can use special elements to make transitions between regions of different refinement level. This set of elements are called a transition pattern. A 2D example of a transition pattern is shown in Figure 1.3.



(a) Transition pattern **(b)** Transition between regions.

Figure 1.3: 2D example of transition pattern

The meshing technique used in this thesis uses mixed elements to make transition patterns [1, 2, 3]. The elements used are tetrahedra, hexahedra, wedges and pyramids of quadrilateral base. This technique also uses mixed elements for boundary representation by defining a set of elements, called surface patterns, that replace hexahedra found in the boundary [4]. Section 2.3 presents more in–depth description of this technique, transition patterns and surface patterns.

Simulation software needs that all the elements of the mesh are valid to produce reliable results. The validity of an element can be determined by a variety of metrics that measure different characteristics.

In some partial refinement meshes transition between regions of different refinement level can appear in the surface of the mesh. Since surface patterns were not design to be applied to mixed elements, the application of a surface pattern to a transition pattern sometimes generates invalid elements on the mesh.

As far as we know there is no concrete study in repairing invalid elements of octree meshes with mixed elements and partial refinement. In this thesis we propose a novel approach for invalid element repair for these meshes. In Chapter 2 we present an state of art in octree-based techniques with partial refinement and quality measure for mixed elements. In Chapter 3 the objectives of this thesis and our proposed approach for mesh repairing are presented. In Chapter 4 we show the test instances, experiment to validate the proposed approach and the results. Finally in Chapter 5 we conclude our work and determine the future work to be done.

Chapter 2

State of the Art

2.1. Finite Volumes

The finite volume method [5] is a discretization method focused in the numeric simulation of various types of conservation laws. It is used in different areas of engineering like fluid mechanics, heat transfer and mass transfer.

Around every node of the mesh a control volume, that does not superimpose over control volumes of neighboring nodes, is constructed. The total volume of the domain is equal to the sum of all the control volumes. The differential equations to be solved are integrated over every control volume. The final result is a discretized version of the equation.

The control volumes are generated using a Voronoi diagram of the original mesh [6].

Since the variable values are localized in the volume and not in the nodes or surface, this method accepts some meshes that are invalid for the finite element method (see Section 2.2).

2.1.1. Voronoi Diagram

When we have a number of nodes in a plane that are part of a mesh, a Voronoi diagram divides the plane using nearest neighbor rule. Every node in the diagram is associated to the nearest region of the plane (see Figure 2.1). The characteristics of the diagram are:

1. Every point of an edge of the diagram is equidistant to two nodes of the mesh.
2. Every node of the diagram is equidistant to three nodes of the mesh.

This characteristics make the finite volume method accept the cases shown in Figure 2.2. The extra edge shown in Figure 2.2b does not affect the associated Voronoi diagram.

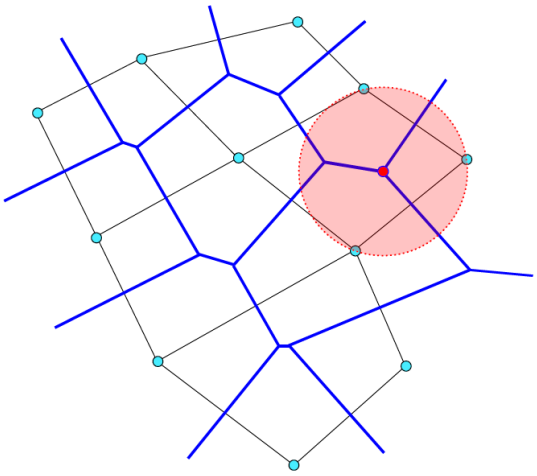
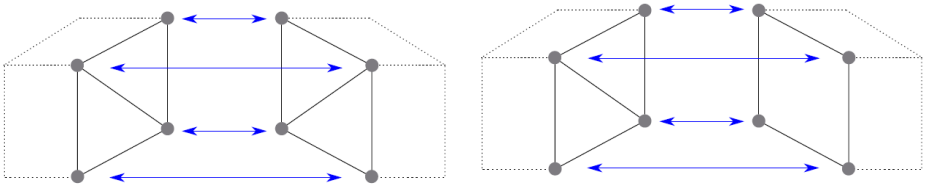


Figure 2.1: 2D mesh and associated Voronoi diagram.
(a) Thick line: Voronoi diagram.
(b) Thin line: 2D mesh.
(c) Light colored points: Nodes of 2D mesh.
(d) Dark colored point: Voronoi diagram point.
(e) Circle: Equidistance to the 2D mesh.



(a) Case 1. **(b)** Case 2.

Figure 2.2: Valid cases for the finite volume method [2].

2.2. Finite Elements

The finite element method [7] is used to fine approximate solutions for differential equations. These approximations are done by creating meshes with a finite number of elements that do not superimpose each other and are continuous, like physical elements. This method is used mainly in the analysis of solids and structures. Other common uses are heat and fluid transfer.

To generate a finite element mesh the domain is divided using lines or surfaces to create subdomains. Subdomains or elements are then combined maintaining mesh conformity. To connect the elements a discrete number of points that are situated in the boundary of the elements are used. We call these points nodes. The displacement of the nodes is used as the main unknown variable for the simulations.

We use the set of functions that define in an unique way the displacement field in each element in relation to the nodal displacement of the element. These displacement functions will define in an unique way the state of deformation within each element. These deformation, with the initial deformations and the proprieties of the material, will define the tension's state in the whole element and its surroundings. It is for this reason that the finite elements method does not accept a mesh that has the case shown in Figure 2.3, which is valid for the finite volumes method.

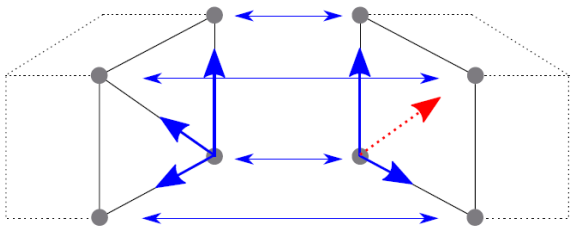


Figure 2.3: Invalid case for the finite elements method [2].

2.3. Octree-Based Meshing Techniques

The octree is a tree data structure in which every tree node is recursively split in a constant number of sub-nodes, called children, typically in 8 or 27. Every tree node represents a hexahedra that is recursively split, when an hexahedra is divided it is replaced by its children in the representation. When using the 27–split strategy, it is possible to produce a pure hexahedral mesh [8, 9, 10]. However, when using the 8–split process on a balanced octree mesh, the transitions between fine and coarse regions can only be managed introducing different types of elements [11, 12, 13, 14]. In the present work we address to this family of algorithms.

Every tree node in an octree is called an octant. Let us say that Ω is the input domain for which a mesh is required. Most octree–based meshing algorithms encapsulate Ω in a primary octant and then proceed to recursively subdivide the octants until a certain constraint is fulfilled (usually a representation error threshold).

The number of recursive subdivision performed over an octant is called refinement level (*RL*). Some algorithms give the option to define different regions over Ω. For each region, a minimum *RL* can be defined and an octant can belong to more than one region. When an octant is divided and some of its children rely completely outside of the domain, their refinement process is halted and they are removed from the representation.

A balanced octree means that any two adjacent octants cannot have refinement level difference greater than one. An octree–based mesh generally require a balanced octree. To do this some octants are refined beyond their original *RL*. A 2D analogous example is shown in Figure 2.4 to illustrate the 8–splitting process.

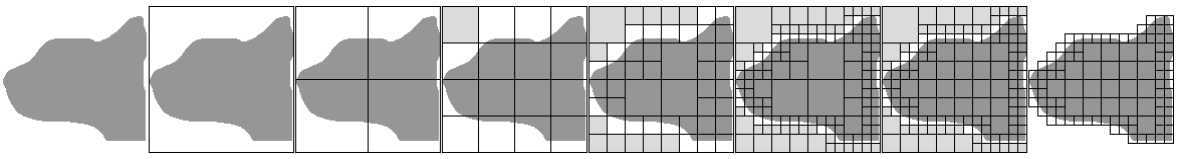


Figure 2.4: 2D analogous example of the generation of a balanced octree mesh using an octree-based algorithm. A region that contains the boundary of Ω was defined with refinement level 4. The rest of the mesh has no minimum refinement level defined, so octants that do not intersect the boundary of Ω have their refinement halted.

A balanced octree is non conformal when there are multiple *RL*. To maintain the continuity of the mesh, a transition pattern must be applied over octants with neighbors of different *RL*. A transition pattern will replace the non conformal hexahedron with different types of elements, so the output mesh is ensured to be conformal. An example of a transition pattern is shown in Figure 2.5.

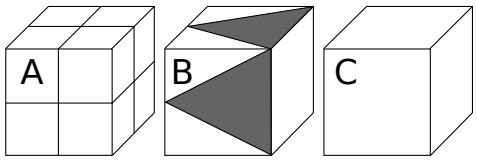


Figure 2.5: Transition pattern example.

For the 27–split transition patterns that only contain hexahedra can be implemented, as shown in Figure 2.6.

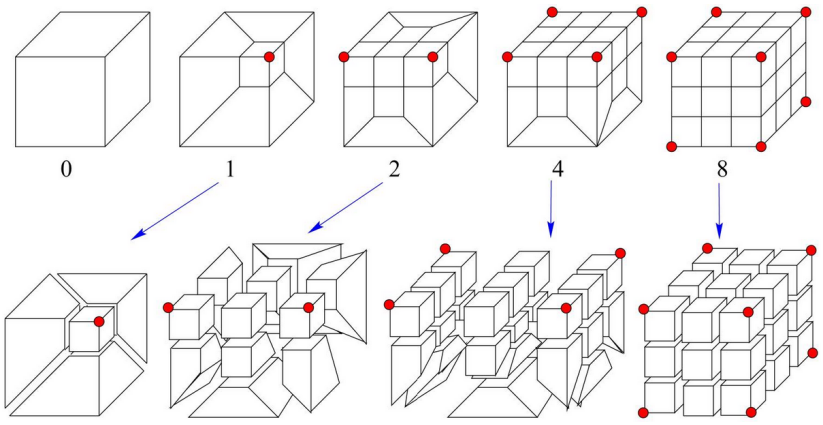
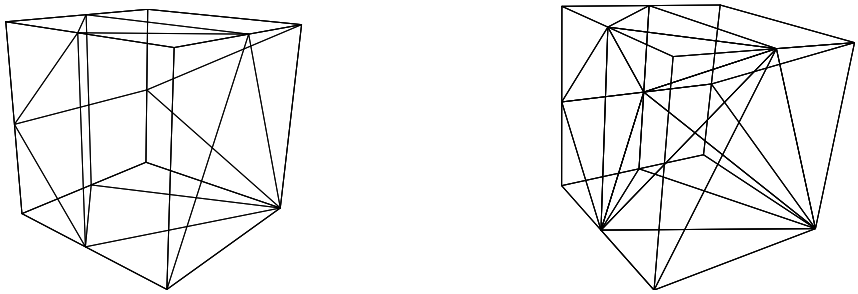


Figure 2.6: Transition patterns for the 27–split octree [9].

For the 8–split octree transition patterns can be designed using different types of elements (Hexahedra, Pyramids, Wedges and Tetrahedra) [3]. Originally 27 transition patterns were designed and implemented in [1]. In a later work, 43 patterns were designed and implemented [2]. In the same work, it was defined that 325 different transition patterns could be made for the 8–split octree. All the patterns were finally designed in a technical report [3]. An example of an external and internal configuration of a transition pattern can be seen in Figure 2.7

Octree–based algorithms must also manage the problem of correct boundary representation,



(a) Surface of a transition pattern. **(b)** Transition pattern with inside elements.

Figure 2.7: Transition pattern with different types of elements [3].

which is not an easy task especially in concave domains. Let us say that a boundary octant is the one that intersects the boundary of Ω, meaning that it will have nodes inside and outside of Ω.

When the boundary octant only contains a hexahedron a surface pattern can be employed [4]. A surface pattern is a set of different types of elements that replace the hexahedron in order to reduce the chances of producing invalid elements. Depending on inside/outside node configuration of the octant a different surface pattern will be employed. A simple 2d–analogous example of a boundary octant handling can be seen in Figure 2.8.

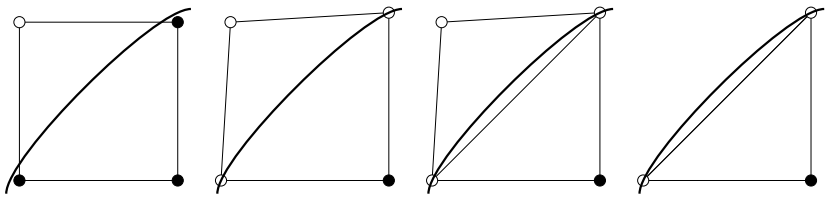


Figure 2.8: Example of boundary octant handling in $2D$. Inside nodes are depicted in black and outside nodes are depicted in white. Nodes on the boundary of Ω are considered outside nodes.

2.3.1. Meshing Algorithm with Different Types of Elements

An octree–based meshing technique, involving both surface and transition patterns, is described in [14]. This technique allows to define different RL for Ω. An explanation of all three RL allowed by the technique is shown in Figure 2.9.

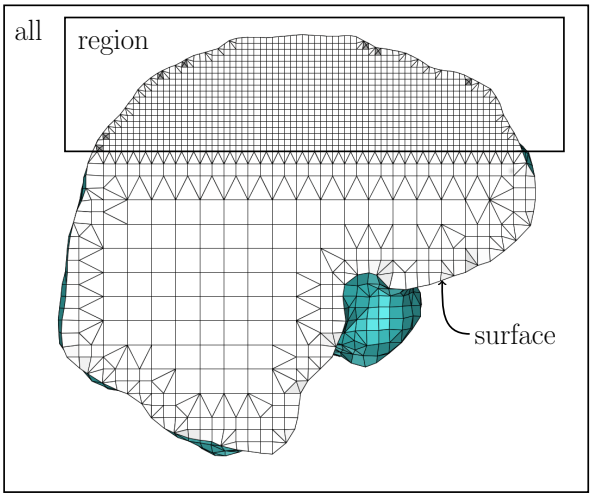


Figure 2.9: Mesh generated with the algorithm described in [14] using all three different *RL* allowed. A *RL* for boundary octants (*surface*), an indepent *RL* that takes effect over all the octants (*all*) and finally yet another *RL* for a particular Region of Interest (*region*), meaning that only octants that are inside Ω and the region of interest (ROI) (another input domain) will be refined to that level. Since an octant may belong to several of these regions, its *RL* will be the larger one.

The main considerations to describe this algorithm are as follow:

- The algorithm's inputs are a triangular surface mesh or domain (Ω) and a set of refinement level constraint for the *RL* allowed (*RLC*).
- The first step of the algorithm is to generate a balanced octree mesh for Ω taking into account the *RLC*. To do this we use the function GENERATE_BALANCED_OCTREE(*RLC*,Ω).
- The second step is to apply transition patterns to octants that neighbor others with a greater *RL*. The function APPLY_TRANSITION_PATTERNS(*mesh*) receives a balanced octree (*mesh*) and iterates over every octant of it. If an octant is adjacent to another with a higher *RL* (it cannot have a difference higher than one since is a balanced octree), the octant's nodes are analyzed and a corresponding transition pattern is applied. The function returns the resulting mesh.
- There is a function GET_BOUNDARY_NODES(*mesh*,Ω) that returns a set of all the nodes that are part of boundary octants, that is all the octants that are part of *mesh* and

intersect Ω. This is used in the third and fifth steps so the algorithm does not iterate over all nodes, just the ones that could be moved on those steps.

- The third step is to project all nodes of boundary octants that are close enough to the surface. This is to enhance the boundary representation and avoid deformed elements. The function PROJECT_INSIDE_NODES(*mesh*,Ω) depicted in Algorithm 1 does this process.

Algorithm 1 Algorithm for inside node projection.

Input: Volumetric mesh *mesh*, triangular surface mesh Ω.

Output: A copy of *mesh* with all inside nodes close enough to Ω projected onto it.

```
1: procedure PROJECT_INSIDE_NODES(mesh,Ω)
2:     bnodes ← GET_BOUNDARY_NODES(mesh,Ω)
3:     for each node in bnodes do
4:         if node is inside of Ω and node is close to Ω then
5:             node.PROJECT_ONTO(Ω)
6:     return mesh
```

- The fourth step is to apply surface patterns to boundary octants. The function APPLY_SURFACE_PATTERNS(*mesh*) recieves a mesh (*mesh*) and iterates over every boundary octant of it. The octant's inside/outside node configuration are analyzed and a corresponding surface pattern is applied. The function returns the resulting mesh.

- In the last step all nodes that remain in the representation but are outside of Ω are projected onto it. To do this process we use the function PROJECT_OUTSIDE_NODES(*mesh*,Ω) depicted in Algorithm 2.

- After the second, the third and the fourth steps, it is checked if an element has all of its nodes outside of Ω. If that is the case, the element is eliminated from the mesh. Nodes on the boundary of Ω are labeled as outside of it, so octants with outside and on surface nodes will be removed too. We use the function ELIMINATE_OUTSIDE_ELEMENTS(*mesh*) for this purpose.

- The algorithm with all its steps is shown in Algorithm 3.

Algorithm 2 Algorithm for outside node projection.

Input: Volumetric mesh *mesh*, triangular surface mesh Ω.

Output: A copy of *mesh* with all outside nodes projected onto Ω.

```
1: procedure PROJECT_OUTSIDE_NODES(mesh,Ω)
2:     bnodes ← GET_BOUNDARY_NODES(mesh,Ω)
3:     for each node in bnodes do
4:         if node is outside of Ω then
5:             node.PROJECT_ONTO(Ω)
6:     return mesh
```

Algorithm 3 Algorithm to generate octree–based meshes with mixed elements and various refinement levels.

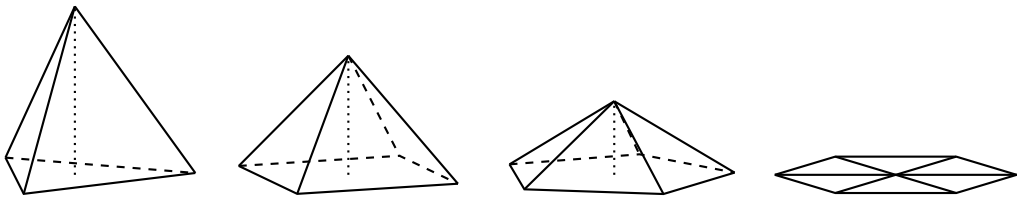
Input: Refinement level constraints *RLC*, triangular surface mesh Ω.

Output: A volumetric mesh of Ω that meets *RLC*.

```
1: procedure GENERATE_MESH(RLC,Ω)
2:     mesh ← GENERATE_BALANCED_OCTREE(RLC,Ω)
3:     mesh ← APPLY_TRANSITION_PATTERNS(mesh)
4:     mesh ← ELIMINATE_OUTSIDE_ELEMENTS(mesh)
5:     mesh ← PROJECT_INSIDE_NODES(mesh,Ω)          // See Algorithm 1
6:     mesh ← ELIMINATE_OUTSIDE_ELEMENTS(mesh)
7:     mesh ← APPLY_SURFACE_PATTERNS(mesh)
8:     mesh ← ELIMINATE_OUTSIDE_ELEMENTS(mesh)
9:     mesh ← PROJECT_OUTSIDE_NODES(mesh,Ω)         // See Algorithm 2
10:    return mesh
```

2.4. Quality Measures for Different Types of Elements

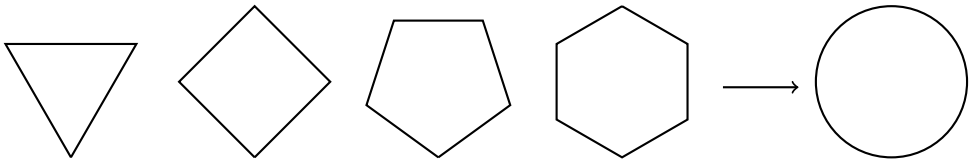
In general, element quality is determined by the level of deformation in comparison to the element's most regular geometric representation or perfect representation. For the hexahedron, tetrahedron and wedge the regular variation is used. Unfortunately a regular variation does not exist for all pyramids. For instance, let us consider the quadrilateral base pyramid. We could suspect that the best pyramid would come when lateral triangular faces are equilateral. If we continue this approach, the perfect hexagonal base pyramid would be a flat element, as seen in Figure 2.10. Another approach to define a perfect pyramid is needed.



(a) 3 **(b)** 4 **(c)** 5 **(d)** 6

Figure 2.10: Increasing number of sides in base of equilateral pyramids.

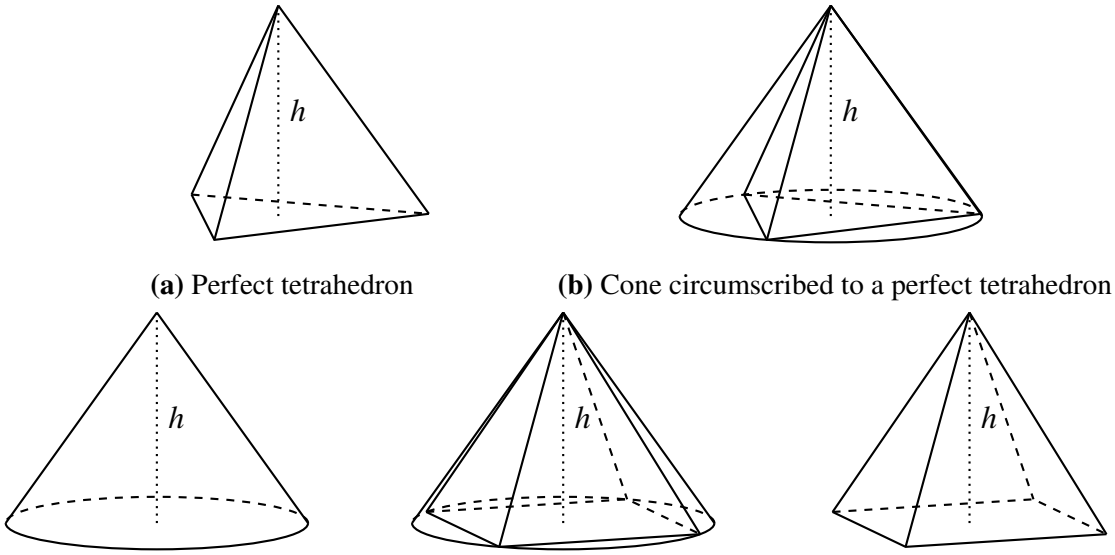
The regular variation element for the pyramid of triangular base (tetrahedron) is defined as the perfect element. As the number of sides of the pyramid's base are increased, it tends to a circular base (As seen in Figure 2.11).



(a) 3 **(b)** 4 **(c)** 5 **(d)** 6 **(e)** ∞

Figure 2.11: Increasing number of sides of equilateral polygons.

All pyramids should have a similar form. With this in mind, the cone that has a regular tetrahedron inscribed is defined as the perfect one. Any pyramid inscribed in the perfect cone is the perfect pyramid. The process of deriving the perfect configuration of a quadrilateral base pyramid from a perfect tetrahedron is shown in Figure 2.12. For the rest of this thesis we call the quadrilateral base pyramid as pyramid.



(a) Perfect tetrahedron **(b)** Cone circumscribed to a perfect tetrahedron

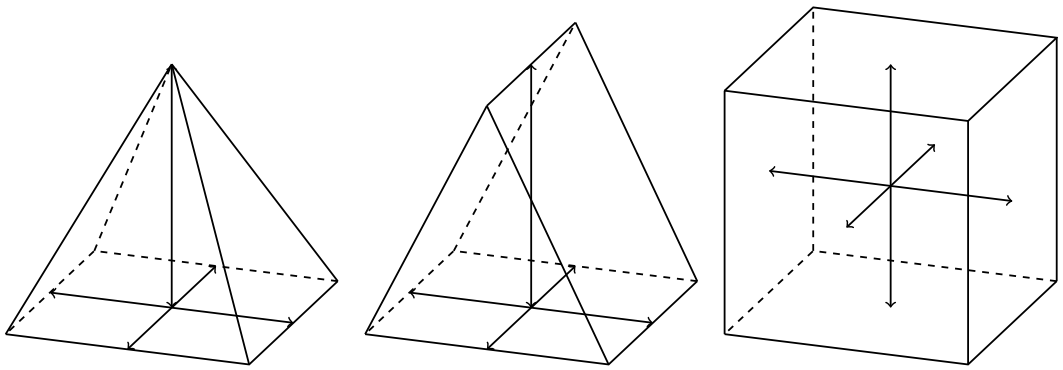
(c) Perfect cone **(d)** Pyramid inscribed in perfect cone **(e)** Perfect pyramid

Figure 2.12: Derivation of a perfect pyramid from perfect tetrahedron. All h in the subfigures have the same value.

There are multiple measures for element quality, each measuring varied types of deformations. A measure commonly used is the Aspect Ratio (AR). In general, the AR of an element is the ratio between its shortest and longest edge. There are different variations of the AR, which use other distances within the element. The lengths used for the AR in this thesis are shown in Figure 2.13 and described as follows:

- **Pyramid:** Distances between opposite edges centers in the base and the height of the pyramid. See Figure 2.13a
- **Wedge:** Distances between opposite edges centers in quadrilateral faces and distances between the center of quadrilateral faces and the opposite edge's center. See Figure 2.13b
- **Hexahedron:** Distances between opposite faces centers. See Figure 2.13c

Another variation defined for the tetrahedra is the Aspect Ratio Gamma (ARG) [15]. The ARG penalizes more the deformation of a tetrahedron in comparison with other AR variants. To calculate the ARG of an element Equation 2.1 is used. Where l_i, $\forall i = \{0, \ldots, 5\}$ are the



(a) Pyramid **(b)** Wedge **(c)** Hexahedron

Figure 2.13: Distances used in this thesis to calculate the *AR* of mixed elements.

edge lengths and V is the volume of the tetrahedron. To range it in $[-1, 1]$, the final quality is $ARG = Q^{-1}$.

$$R = \left(\frac{1}{6} \sum_{i=0}^{5} \|l_i\|^2 \right)^{\frac{1}{2}} \qquad \Rightarrow \qquad Q = \frac{R^3 \sqrt{2}}{12 \cdot V} = \frac{R^3}{8.48528 \cdot V} \tag{2.1}$$

To measure the distortion of a node in contrast with its neighbors we can use the Jacobian. Being J^a the Jacobian of node a, and $\vec{d}_i$ the vector from the node a to the node i. Using the numbers for the nodes depicted in Figure 2.14, we can calculate the Jacobian of node 0 as $J^0 = \vec{d}_3 \cdot (\vec{d}_1 \times \vec{d}_2)$.

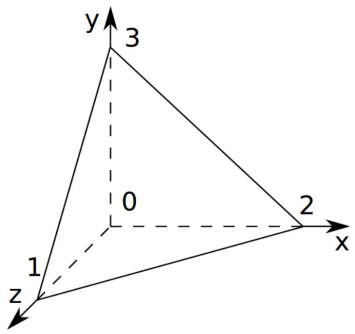


Figure 2.14: Node numeration for tetrahedron.

The problem with the Jacobian is that it depends on the distance between the nodes, so two elements with the same internal angles but different edge sizes get different Jacobian values.

In the case of the hexahedron, an approach to solve this issue is to normalize the vectors used in the Jacobian calculation, so we replace $\vec{d}_i$ for $\hat{d}_i = \vec{d}_i / \|\vec{d}_i\|$. The result is called an Scaled Jacobian (J_S), the values of J_S are in the range of $[-1, 1]$.

The J_S of an hexahedron is the worst J_S of its nodes. It is defined that a $J_S < 0$ means the element is inverted, a good quality is when the $J_S \geq 0.2$ and a perfect hexahedron when $J_S = 1$. Elements with $J_S \in [0, 0.2[$ are called questionable for calculations [16]. Some simulations programs, like ANSYS, consider hexahedra with $J_S \in [0, 0.03[$ as invalid for calculations, since they are almost flat [17]. In this thesis we will also consider elements in that interval as invalid.

The problem of extending the use of J_S to other type of elements is that the perfect quality can only be measured if all of the vectors are orthogonal. Since the perfect element for wedge, tetrahedron and pyramid do not have all of their edges orthogonal to each other, a jacobian normalization per element type is needed.

The Element Normalized Scaled Jacobian (J_{ENS}) propose such normalization [18], by calculating the J_S of the perfect element k^e, where e denotes the corresponding elements, the values for the different elements used in this work are shown in Equation 2.2.

$$\begin{aligned} k^T &= \tfrac{\sqrt{2}}{2} \rightarrow Tetrahedron \\ k^P &= \tfrac{\sqrt{6}}{3} \rightarrow Pyramid \\ k^W &= \tfrac{\sqrt{3}}{2} \rightarrow Wedge \end{aligned} \tag{2.2}$$

With this values we can calculate the J_{ENS} of any node of this elements replacing the value of the corresponding constant in Equation 2.3. Similar to the J_S to calculate the J_{ENS} of an element we use Equation 2.4, where J^e_{ENS} is the J_{ENS} of element e and i are the nodes of element e.

$$J^n_{ENS} = \begin{cases} (1 + k^e) - J_S, & J_S > k^e \\ J_S / k^e, & -k^e \leq J_S \leq k^e \\ -(1 + k^e) - J_S, & J_S < -k^e \end{cases} \tag{2.3}$$

$$J^e_{ENS} = \begin{cases} \min\{J^i_{ENS}\}, & \forall\ i, J^i_{ENS} > 0 \\ \max\{J^i_{ENS}\} : J^i_{ENS} < 0, & \exists\ i : J^i_{ENS} < 0 \end{cases} \tag{2.4}$$

For the J_{ENS} the same intervals as the J_S to denote elements are defined: $[-1, 0[$, inverted; $[0, 0.03[$, invalid; $[0.03, 0.2[$, questionable; $[0.2, 1]$, good.

Chapter 3

Approach

3.1. Hypothesis

For hexahedral meshes ANSYS considers invalid an element with J_S less than 0.03 [17]. Since the J_{ENS} is a normalization of the J_S for different types of elements, we consider an elements invalid if its J_{ENS} is less than 0.03.

When transition between fine and coarse regions appear at the boundary domain, the octree-based meshing algorithm with different types of elements can sometimes generate some invalid or inverted elements . If one invalid element exists in the mesh, a simulation cannot be performed.

One approach to repair element invalidity is by employing node relaxation techniques, which displace nodes within the domain to improve the quality of an element. The displacement is guided by a metric, usually the quality of an element. However, these techniques do not remove the elements from the mesh, so element validity could be achieved by causing a distortion on neighboring elements. Since invalid elements have a high level of distortion, achieving validity could invalidate neighbor elements.

Another approach more specific to this problem is needed. With this in mind we propose the following:

"It is possible to repair invalid elements in an octree–based meshing algorithm with mixed elements, surface patterns and different refinement levels."

3.2. Objectives

3.2.1. General Objectives

- Analise octants that present invalid elements.
- Define a technique to repair meshes with invalid elements.
- Test the proposed technique.

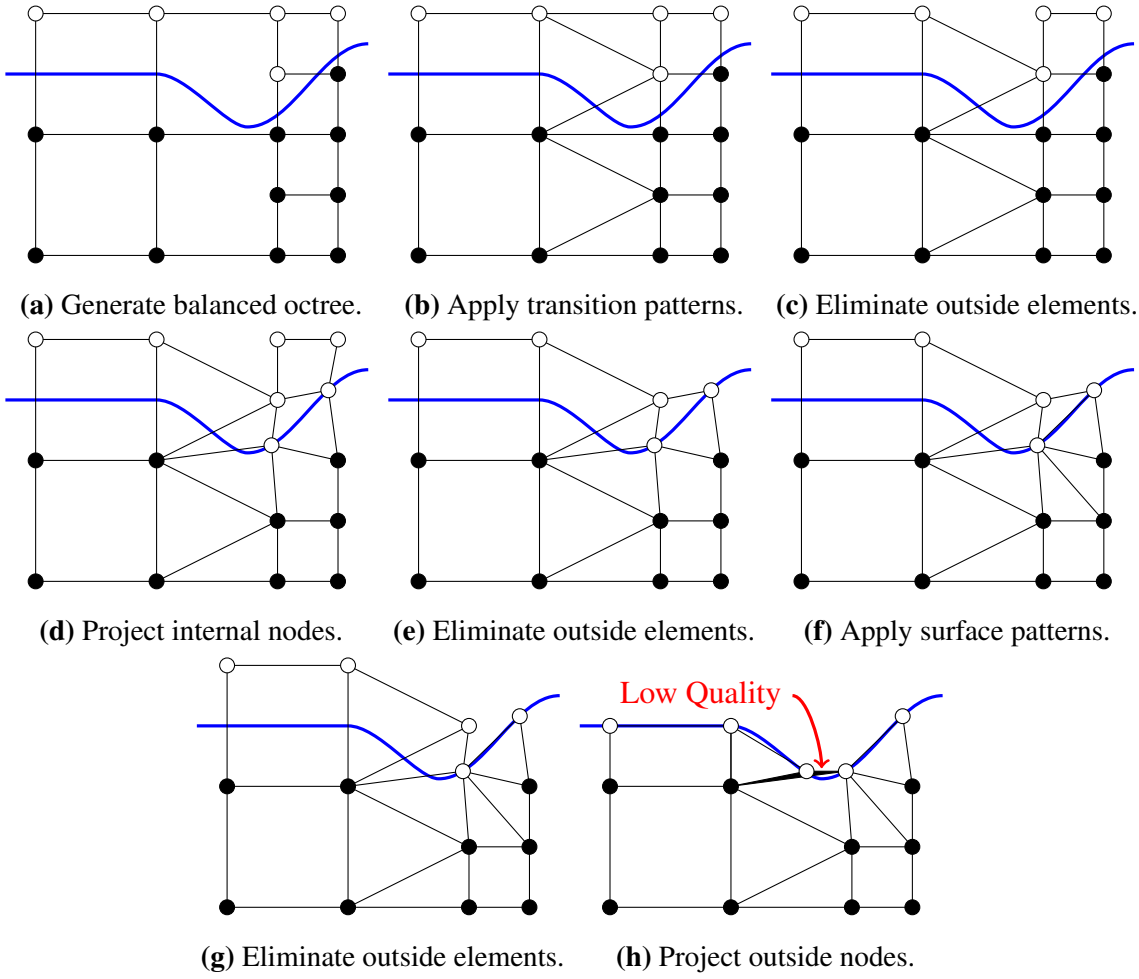
3.2.2. Specific Objectives

- Identify octants with invalid elements.
- Analise node configuration of invalid elements.
- Define technique that eliminates invalid elements from the mesh.
- Implement the proposed technique.
- Define test domains and instances with different regions of interests.
- Validate the proposed technique with tests varying the *RL*.

3.3. Preliminary Study of Invalid Elements

To define the proposed approach we analyzed in which regions of the mesh the elements presented a invalidity or great deformation. Initial results show that all the elements with invalid quality appear on boundary octant. Further more, 279 of the 281 octants with invalid elements presented in the test instances had transition patterns applied to them (see Section 4.2.1 for more details). The other two octants were adjacent to transition patterns.

Transition patterns were not design to appear on the boundary. A 2D analogous example of an invalid element generated by the algorithm presented in Section 2.3.1 when a transition pattern appears on the boundary is shown in Figure 3.1.



(a) Generate balanced octree. **(b)** Apply transition patterns. **(c)** Eliminate outside elements.

(d) Project internal nodes. **(e)** Eliminate outside elements. **(f)** Apply surface patterns.

(g) Eliminate outside elements. **(h)** Project outside nodes.

Figure 3.1: 2D example of problem with transition pattern on the boundary of mesh.

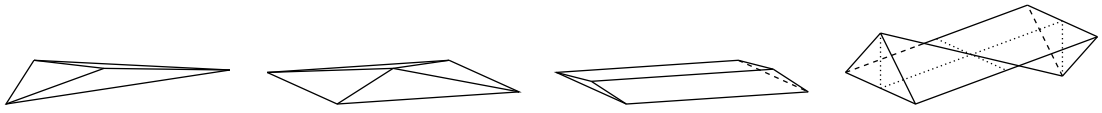
With the appearance of transition patterns on the boundary surface patterns for the wedge and pyramid were design. Mesh conformity was maintain by applying the same conventions for quadrilateral face division as the hexahedron. Since the conventions were design for the hexahedron they do not take to account some configurations that the other types of elements have that could generate invalid elements. So applying surface patters to octants that have transition patterns could increase the chance of generation of invalid elements, since both approach were initially design to only work on base octants (Hexahedron).

It is important to note that the number of invalids elements is less than 0.1% of the total elements of the mesh (as seen in Section 6.2.2). However, as noted before, if one invalid element exists in the mesh, a simulation cannot be performed.

3.4. Proposed Approach for Mesh Repairing

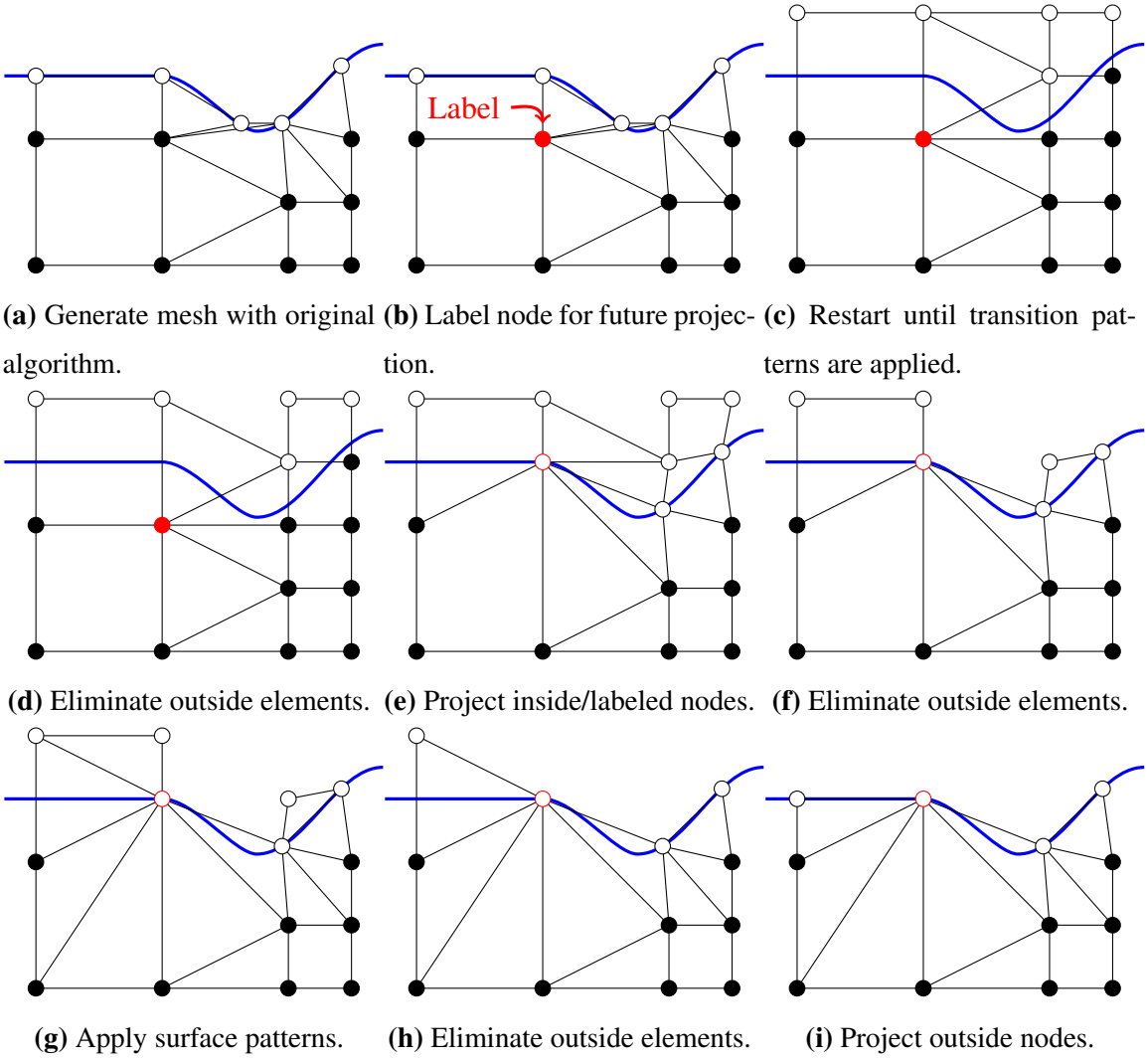
By design, the algorithm eliminates from the mesh any element that has all of its nodes on the boundary of Ω or outside of it. In the study mentioned on the previous section, our study showed that invalid elements appear on boundary octants, this means that at least one of its node is on the boundary of Ω or outside of it.

The approach that we propose is to project the inside nodes of invalid elements onto the boundary so these elements are eliminated from the mesh. An invalid element is considered almost, if not, flat or inverted (See Figure 3.2 for examples of invalid elements). Since some of its nodes are outside of Ω, projecting its inside nodes onto the boundary of Ω should not produce significant deformations to its neighbors. A 2D application of the proposed approach to the mesh generated in Figure 3.1 is shown in Figure 3.3.



(a) Flat tetrahedron. **(b)** Flat pyramid. **(c)** Flat wedge. **(d)** Inverted wedge.

Figure 3.2: Examples of invalid elements.



(a) Generate mesh with original algorithm. **(b)** Label node for future projection. **(c)** Restart until transition patterns are applied.

(d) Eliminate outside elements. **(e)** Project inside/labeled nodes. **(f)** Eliminate outside elements.

(g) Apply surface patterns. **(h)** Eliminate outside elements. **(i)** Project outside nodes.

Figure 3.3: Proposed approach applied to 2D example shown in Figure 3.1.

The node projection proposed in the previous paragraph could make some of the adjacent elements invalid in the process, if this happens these new invalid elements should be projected onto the boundary too. The process is repeated until a set quality threshold for the mesh is attained or a maximum number of iterations is achieved.

To design our approach we follow the considerations detailed in Section 2.3.1 and expand them as follows:

- To handle the forced projection of nodes in the meshing algorithm we redefine the inside node projection algorithm presented in Algorithm 1 so it can receive a set of

nodes to be projected. This redefinition is shown in Algorithm 4.

Algorithm 4 Algorithm for inside node projection that handles force projection.

Input: Volumetric mesh *mesh*, triangular surface mesh Ω, set of nodes labeled for projection *L*.

Output: A copy of *mesh* with all inside nodes close enough to Ω projected onto it.

```
1: procedure PROJECT_INSIDE_NODES(mesh,Ω,L)
2:     bnodes ← GET_BOUNDARY_NODES(mesh,Ω)
3:     for each node in bnodes do
4:         if node ∈ L or (node is inside of Ω and node is close to Ω) then
5:             node.PROJECT_ONTO(Ω)
6:     return mesh
```

- A variation on the meshing algorithm shown in Algorithm 3 that handles force projection is defined in Algorithm 5. This algorithm receives the list of nodes to be forcefully projected, so node labeling is to be handled outside of it.

Algorithm 5 Octree–based meshing technique with forced projection

Input: Refinement level constraints *RLC*, triangular surface mesh Ω, set of nodes labeled for projection *L*.

Output: A volumetric mesh of Ω that meet *RLC* and has labeled nodes on its boundary.

```
1: procedure GENERATE_MESH(RLC,Ω,L)
2:     mesh ← GENERATE_BALANCED_OCTREE(RLC,Ω)
3:     mesh ← APPLY_TRANSITION_PATTERNS(mesh)
4:     mesh ← ELIMINATE_OUTSIDE_ELEMENTS(mesh)
5:     mesh ← PROJECT_INSIDE_NODES(mesh,Ω,L)          // See Algorithm 4
6:     mesh ← ELIMINATE_OUTSIDE_ELEMENTS(mesh)
7:     mesh ← APPLY_SURFACE_PATTERNS(mesh)
8:     mesh ← ELIMINATE_OUTSIDE_ELEMENTS(mesh)
9:     mesh ← PROJECT_OUTSIDE_NODES(mesh,Ω)           // See Algorithm 2
10:    return mesh
```

- To obtain a set of nodes that are part of elements with quality below a given threshold

we defined the function GET_INVALID_NODES(M,T) that is shown in Algorithm 6. Its input parameters are a volumetric mesh with mixed elements and a quality threshold. This functions uses J_{ENS} as element quality measure. The function CALCULATE_JENS(*element*) returns the J_{ENS} of *element*.

Algorithm 6 Label Nodes to Projection

Input: Volumetric mesh with mixed elements M, quality threshold T.

Output: Set of *nodes* that are part of invalid elements.

```
1: procedure GET_INVALID_NODES(M,T)
2:     l ← empty set
3:     for each element in M do
4:         q ← CALCULATE_JENS(element)
5:         if q < T then
6:             for each node in element do
7:                 l.ADD(node)
8:     return l
```

Using Algorithm 5 and Algorithm 6 we design a first version of our proposed approach, which is detailed in Algorithm 7. This algorithm generates the mesh from scratch in every iteration updating a list of labeled nodes. The labeling of nodes between different instances of the mesh generator is possible because it is a deterministic algorithm, so it generates the same nodes from the same initial parameters.

The problem with the first version is that since in every iteration the mesh is generated from scratch, it is time consuming. The generation of the original balanced octree and the application of the transition patterns are done before the inside node projection step. Since the step changed in every iteration is this one, we can save the state of the mesh just before it (Balanced octree with transition patterns) and start every iteration from that state. The proposed algorithm second version is described in Algorithm 8.

Algorithm 7 Proposed Repair First Version

Input: Refinement level constraints *RLC*, triangular surface mesh Ω, quality threshold T, maximum number of iterations It.

Output: If successful, volumetric mesh of Ω that meet the *RLC* with quality greater than T.

```
1:  procedure PROPOSED_REPAIR_V1(RLC,Ω,T,It)
2:      l ← empty set
3:      for 1 to It do
4:          mesh ← GENERATE_MESH(RLC,Ω,l)                    // See Algorithm 5
5:          ln ← GET_INVALID_NODES(mesh,T)                   // See Algorithm 6
6:          if ln is empty then
7:              return mesh
8:          for each node in ln do
9:              l.ADD(node)
10:     return null
```

Algorithm 8 Proposed Repair Second Version

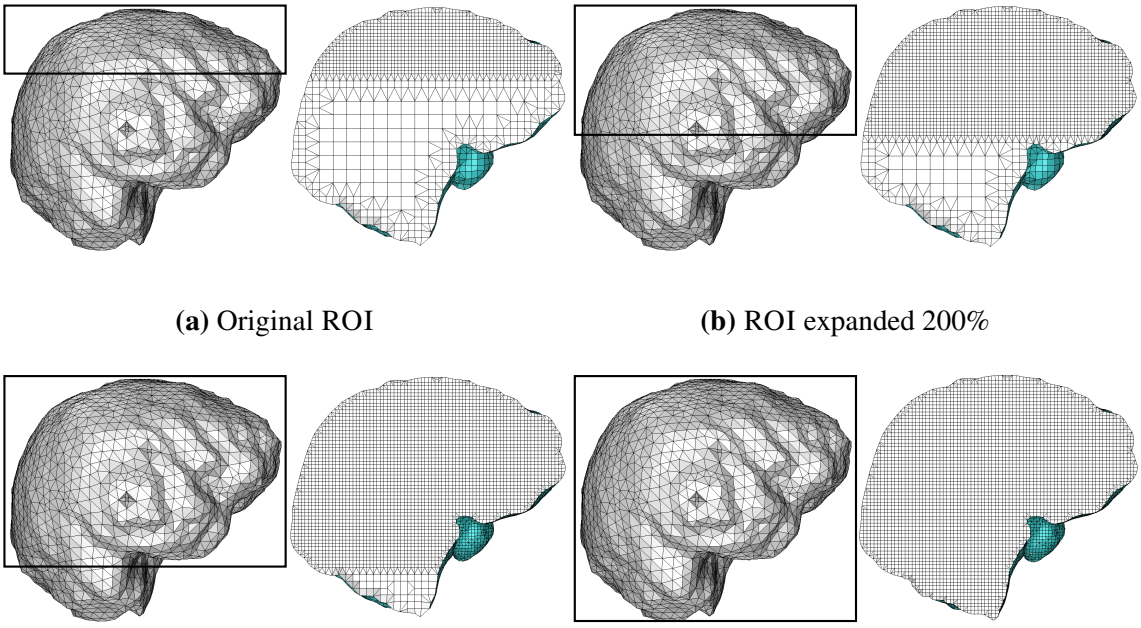
Input: Refinement level constraints *RLC*, triangular surface mesh Ω, quality threshold T, maximum number of iterations It.

Output: If mesh quality threshold is surpased within the number of iterations, a volumetric mesh of Ω that meet the *RLC* with quality greater than T.

```
1:  procedure PROPOSED_REPAIR_V2(RLC,Ω,T,It)
2:      imesh ← GENERATE_BALANCED_OCTREE(RLC,Ω)
3:      imesh ← APPLY_TRANSITION_PATTERNS(imesh)
4:      imesh ← ELIMINATE_OUTSIDE_ELEMENTS(imesh)
5:      l ← empty set
6:      for 1 to It do
7:          mesh ← imesh
8:          mesh ← PROJECT_INSIDE_NODES(mesh,Ω,L)          // See Algorithm 4
9:          mesh ← ELIMINATE_OUTSIDE_ELEMENTS(mesh)
10:         mesh ← APPLY_SURFACE_PATTERNS(mesh)
11:         mesh ← ELIMINATE_OUTSIDE_ELEMENTS(mesh)
12:         mesh ← PROJECT_OUTSIDE_NODES(mesh,Ω)           // See Algorithm 2
13:         ln ← GET_LABELED_NODES(mesh,T)                 // See Algorithm 6
14:         if ln is empty then
15:             return mesh
16:         for each node in ln do
17:             l.ADD(node)
18:     return null
```

3.4.1. Complementary Approach for Mesh Repairing

Both versions of the proposed algorithm will be implemented and validated for this thesis. If the algorithm is not able to eliminate invalid elements we propose an stretching of the original ROI, as shown in Figure 3.4. This process should be done iteratively until the ROI encompass the whole domain, so every octant of the mesh is refined to the *RL* defined by the ROI.



(a) Original ROI

(b) ROI expanded 200%

(c) ROI expanded 300%

(d) ROI expanded 400%

Figure 3.4: Stretching of the ROI by 100% of the original.

In Figure 3.4 we show a stretching of the ROI by 100% of the original one's size. Axis that encompass the domain should be halted. And when the ROI encompass the whole mesh, the process is ended.

To validate this approach and define a good stretching ratio a significant number of test are required, which will be outside the scope of this thesis. We will only measure the worst case for the stretching of the ROI, which is a ROI that encompass the whole mesh.

Chapter 4

Tests and Results

4.1. Test Instances

The domains used for the testing the proposed approach are a liver and a brain cortex, both shown in Figure 4.1.

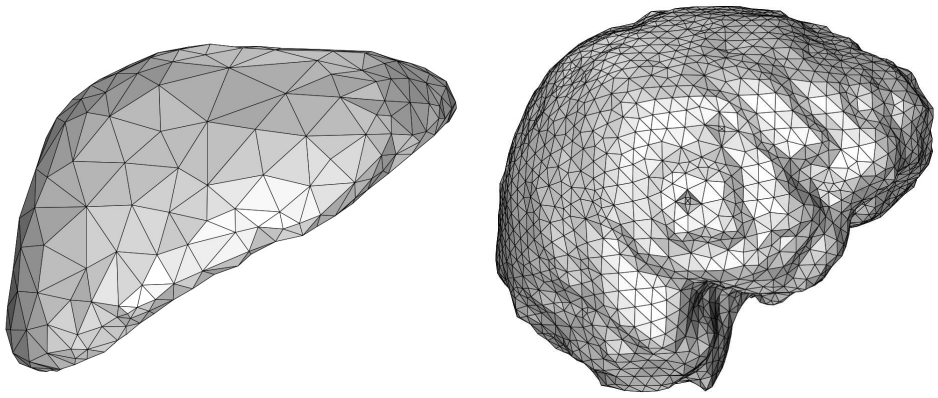


Figure 4.1: Input domains used for testing: liver and cortex.

For testing we use the three different types of refinement that the algorithm allows and that were described in Section 2.3.1 and Figure 2.9: *region*, $surface$, *all*. Over both domains we define different Regions Of Interest (ROI) for testing. The ROI for the liver are shown in Figure 4.2 and for the cortex in Figure 4.3.

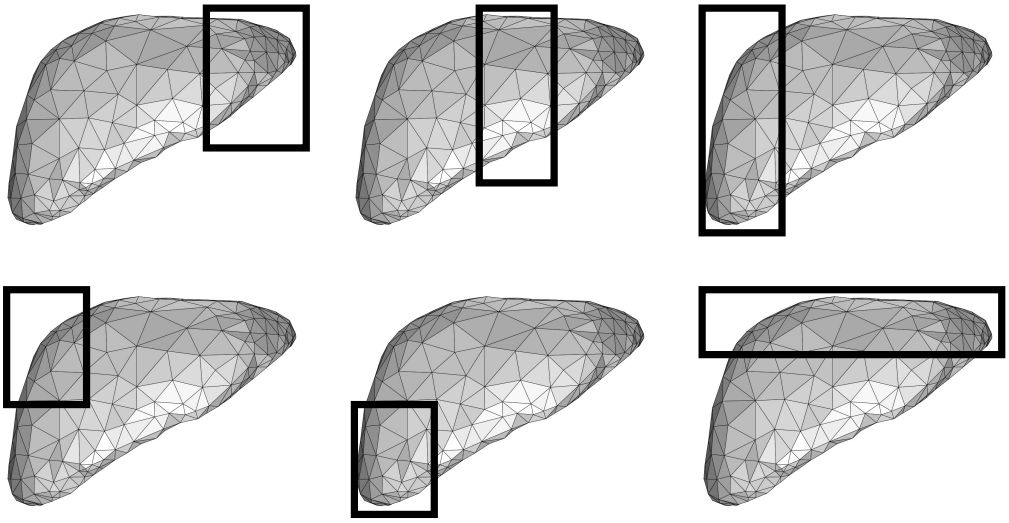


Figure 4.2: ROIs used in liver instances. From left to right and top to bottom: ROI 0 to 5.

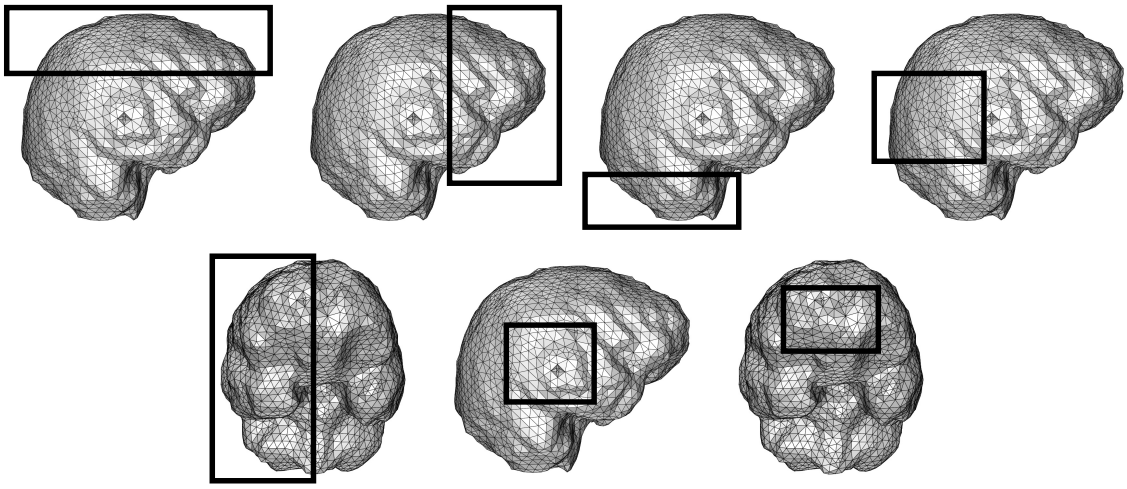


Figure 4.3: ROIs used in cortex instances. From left to right and top to bottom: ROI 0 to 6.

In order to ensure transitions were presented in the final mesh, we used a $RL = n$ in the ROI, a $RL = n - 1$ in the boundary of Ω and a $RL = n - 2$ in the rest of the mesh. We named the instances in the form "*domain_ref_reg*", where *domain* is the *liver* or the *cortex*, *ref* the *RL* used for the ROI (5, 6 or 7), and *reg* the ROI used denoted by the list of regions shown in Figure 4.2 and Figure 4.3. For example "*cortex_5_6*" is the instance where the cortex domain was used with the 7^{th} (numeration starts at 0) ROI of Figure 4.3, and the *RL* was 5 for the *region*, 4 for $surface$ octants and 3 for *all* the rest.

4.2. Experiments and Results

4.2.1. Analysis of Octants with Invalid Elements

Using all the test instances we analyzed all the octants that presented invalid elements. We use the numeration defined in [3] to denote transition patterns. We detected that the pattern 217 was present in 261 invalid octants and the pattern 206 was present in 18. Additionally two base octants were found to have invalid quality, these were boundary octants adjacent to ones with transition patterns applied to them.

The pattern 217 was the one present in the majority of the cases. This is the same pattern that had problems that could not be repaired in a previous work [19].

A 2D plane that shows the face configuration of the pattern 217 is in Figure 4.4. This pattern is the one normally applied to an octant that connect two regions with one *RL* of difference.

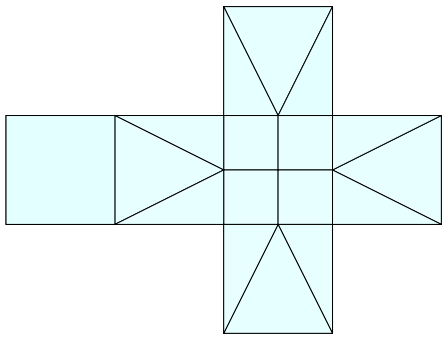


Figure 4.4: Face configuration of pattern 217.

An analysis of the nodes present in the octants with the pattern 217 and invalid elements was done. We detected the nodes inside the mesh, the ones on the surface and the ones that were part of the original octree created by the algorithm but were eliminated form the representation, as no element that contains them remains in it.

Different configurations were found, as shown in Figure 4.5. The number of times each configuration was found is seen in Table 4.1.

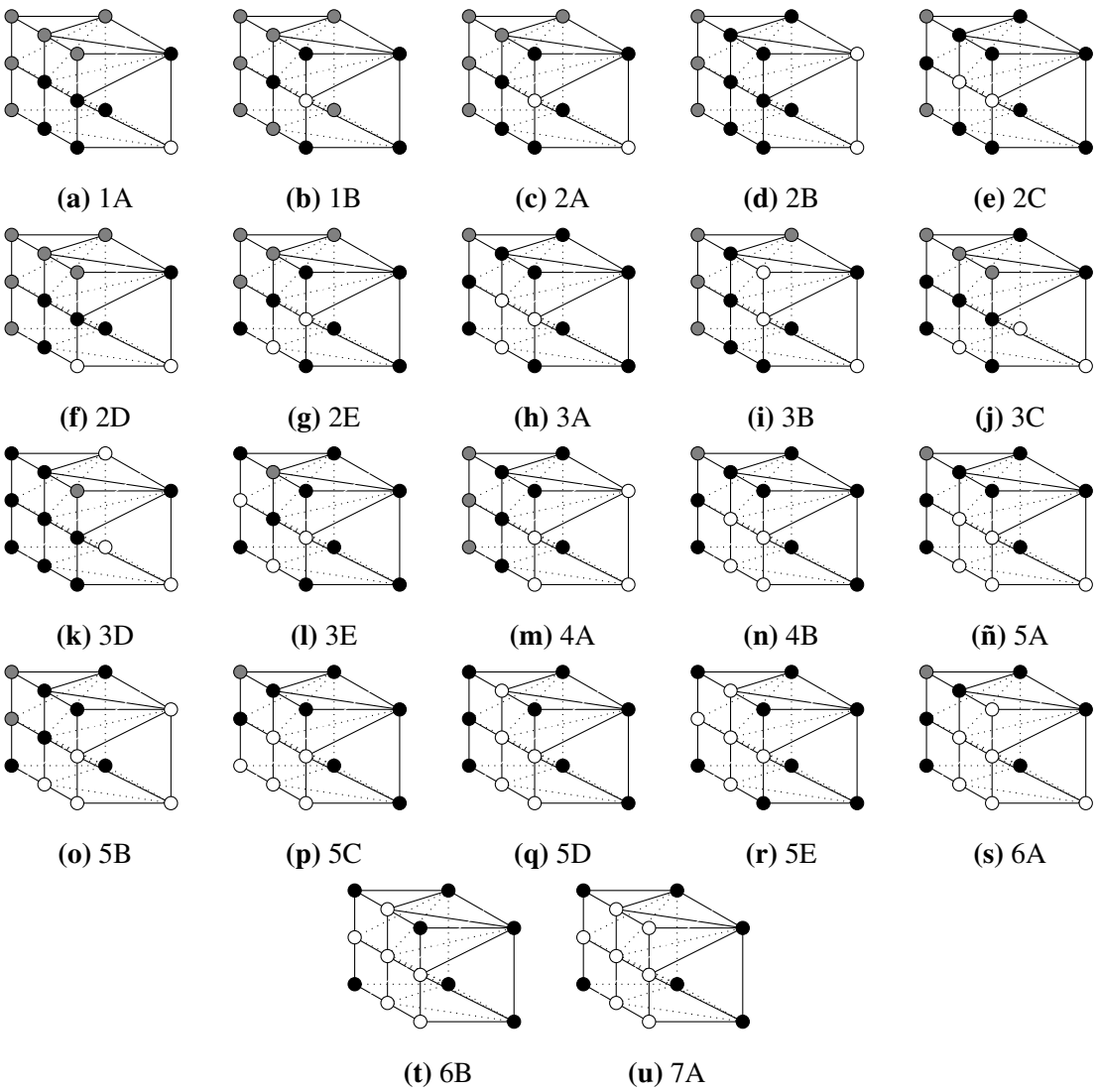


Figure 4.5: Node configuration of octants with pattern 217 and invalid elements. White: Inside nodes. Black: Surface nodes. Gray: Nodes removed from the representation.

Configuration	1A	1B	2A	2B	2C	2D	2E	3A	3B	3C	3D
Quantity	49	5	32	6	1	14	4	2	1	17	1
Configuration	3E	4A	4B	5A	5B	5C	5D	5E	6A	6B	7A
Quantity	1	30	4	34	30	3	5	1	8	5	8

Table 4.1: Number of times a node configuration is present in the test instances.

4.3. Proposed Algorithm's Tests

An implementation of the first version of the proposed algorithm was compared with the original mesh generation algorithm using experiments performed to all test instances. The experiments were conducted with a maximum number of 5 repairing iterations.

The results shown in this section are the ones obtained from the cortex instances with *RL* of 5 in the ROI and all the ROIs of Figure 4.3. **The experiments on the other instances had similar results**. The tables with all the results can be found in Section 6.2.

For some test results three rows are shown per instance in the tables of this section and Section 6.2. The first row correspond to the original algorithm, the second and the third rows correspond to the proposed algorithm with a quality threshold of 0.03 and 0.05 respectively.

A comparison of the final quality of the mesh and the run time was performed. The results are shown in Table 4.2. The quality was computed using J_{ENS} and the time was measured in **seconds**. We define the "time per iteration" as the average time of every iteration after the first one (see Equation 4.1). Therefore, the time per iteration cannot be computed if the algorithm performs only one iteration. The computational time of the first iteration is approximately the run time of the original algorithm.

Let T_{OA} be the original algorithm's runtime, T_{PA} the proposed algorithm's runtime and I_{PA} the proposed algorithm's number of iterations. We define the proposed algorithm's time per iteration (TPI) as seen in Eq. 4.1.

$$TPI = \frac{T_{OA} - T_{PA}}{I_{PA} - 1} \tag{4.1}$$

Another documented result is the number of elements per quality interval. The intervals defined were: Inverted elements ($[-1, 0[$); invalid elements ($[0, 0.03[$); $[0.03, 0.05[$; $[0.05, 0.1[$; $[0.1, 0.15[$; $[0.15, 0.2[$; and good elements $[0.2, 1]$. The results can be seen in Table 4.3.

The proposed approach projects the nodes of an element. This node movement could stretch adjacent elements. The J_{ENS} is used to measure the distortion of element angles, however it

Table 4.2: Comparison of quality and time for the original algorithm and the proposed algorithm's first version.

Instance	Quality[a]	Time[a]	Quality[a]	Time[a]	It[bc]	T/It[bd]
cortex_5_0	-0.009743	2.312	0.037720	4.415	2	2.103
cortex_5_1	-0.000639	2.111	0.041439	4.031	2	1.920
cortex_5_2	0.002585	1.776	0.030672	3.387	2	1.611
cortex_5_3	-0.999649	2.060	0.041507	3.993	2	1.933
cortex_5_4	-0.222353	2.819	0.042440	5.492	2	2.673
cortex_5_5	-0.100899	1.813	0.072093	3.506	2	1.693
cortex_5_6	-0.491747	1.800	-0.981794	8.702	5	1.726

[a] Original Algorithm [b] Proposed Algorithm V1

[c] Numbre of Iterations [d] Time per Iteration

does not take into account element stretching. To measure the stretching we used the *AR*. In the case of tetrahedra we measured the *ARG* as its *AR*. The lowest aspect ratio separated by element type is shown in Table 4.4. A separation for quality intervals is shown in Table 4.5 and Table 4.6.

Projecting elements to the boundary could affect the representation of Ω. An analysis of the mesh with the original algorithm and with our approach was also performed. An example of one of the comparison is shown in Figure 4.6. Nodes marked for projection or projected appear in white, other nodes are in black. The rest of the results of this analysis were similar. Other image comparisons for the cortex instances appear in Section 6.1.

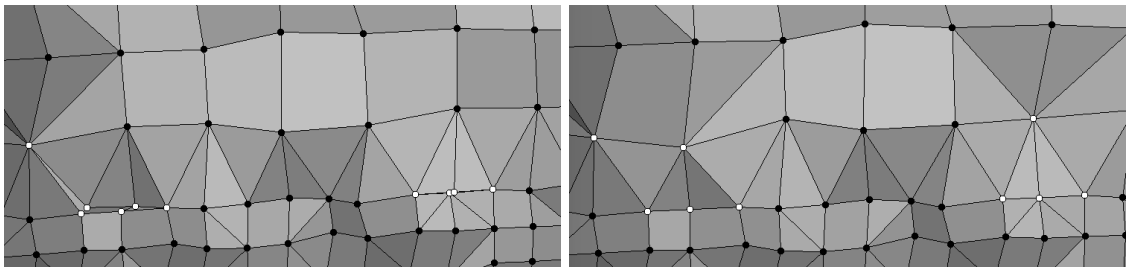


Figure 4.6: Domain representation comparison between original (left) and proposed (right) algorithms.

Table 4.3: Number of elements per J_{ENS} interval.

Instance	Mesh J_{ENS}	Inv[a]	Inv[b]	Que[c]	Que[d]	Que[e]	Que[f]	Good[g]	Total[h]
cortex_5_0	-0.009743	1	8	8	14	35	31	12798	12895
	0.037720	0	0	7	12	35	34	12803	12891
	0.053261	0	0	0	7	35	42	12791	12875
cortex_5_1	-0.000639	1	7	2	9	18	29	11939	12005
	0.041439	0	0	2	9	18	24	11930	11983
	0.058775	0	0	0	9	18	25	11936	11988
cortex_5_2	0.002585	0	3	2	13	22	36	8253	8329
	0.030672	0	0	4	13	21	36	8253	8327
	0.059482	0	0	0	13	22	38	8217	8290
cortex_5_3	-0.999649	3	0	1	11	44	86	11757	11902
	0.041507	0	0	2	12	47	79	11763	11903
	0.000000	0	2	2	22	48	84	11728	11886
cortex_5_4	-0.222353	2	2	1	9	34	63	18402	18513
	0.042440	0	0	1	7	32	64	18402	18506
	0.056449	0	0	0	7	31	67	18390	18495
cortex_5_5	-0.100899	2	2	0	8	30	55	9769	9866
	0.072093	0	0	0	7	33	57	9774	9871
	0.072093	0	0	0	7	33	57	9774	9871
cortex_5_6	-0.491747	9	0	0	2	47	71	9506	9635
	-0.981794	11	1	3	19	55	79	9431	9599
	-0.981794	11	1	3	19	55	79	9431	9599

[a] Inverted Elements: [-1,0[

[b] Invalid Elements: [0,0.03[

[c] Questionable Interval 1: [0.03,0.05[

[d] Questionable Interval 2: [0.05,0.1[

[e] Questionable Interval 3: [0.1,0.15[

[f] Questionable Interval 4: [0.15,0.2[

[g] Good Quality: [0.2,1[

[h] Total Elements

Table 4.4: Mesh J_{ENS}, Mesh *AR* y *AR* per type of element.

Instance	J_{ENS} [a]	AR [b]	Hex [c]	Tet [d]	Pyr [e]	Wed [f]
cortex_5_0	-0.009743	0.001485	0.353553	0.001485	0.149989	0.323838
	0.037720	0.087566	0.353553	0.087566	0.225299	0.324094
	0.053261	0.140746	0.353553	0.140746	0.253287	0.309168
cortex_5_1	-0.000639	0.000218	0.522315	0.000218	0.232392	0.294145
	0.041439	0.108655	0.522315	0.108655	0.232392	0.294145
	0.058775	0.141132	0.522315	0.141132	0.238242	0.294145
cortex_5_2	0.002585	0.005723	0.536592	0.005723	0.205097	0.323838
	0.030672	0.042106	0.536592	0.042106	0.240023	0.323838
	0.059482	0.081617	0.536592	0.081617	0.240023	0.323838
cortex_5_3	-0.999649	0.101051	0.353553	0.101051	0.233464	0.316153
	0.041507	0.101051	0.353553	0.101051	0.259568	0.316153
	0.000000	0.078139	0.353553	0.078139	0.174202	0.189164
cortex_5_4	-0.222353	0.003869	0.353553	0.003869	0.226341	0.311163
	0.042440	0.131622	0.353553	0.131622	0.226341	0.300113
	0.056449	0.076700	0.353553	0.076700	0.274342	0.300113
cortex_5_5	-0.100899	0.001395	0.487193	0.001395	0.299254	0.316153
	0.072093	0.067101	0.487193	0.067101	0.299254	0.316153
	0.072093	0.067101	0.487193	0.067101	0.299254	0.316153
cortex_5_6	-0.491747	0.010296	0.353553	0.010296	0.326315	0.323838
	-0.981794	0.004275	0.151413	0.004275	0.162811	0.172722
	-0.981794	0.004275	0.151413	0.004275	0.162811	0.172722

[a] Lowest J_{ENS} in the Mesh
[b] Lowest *AR* in the Mesh
[c] Lowest *AR* between Hexahedra
[d] Lowest *AR* between Tetrahedra
[e] Lowest *AR* between Pyramids
[f] Lowest *AR* between Wedges

Table 4.5: Number of elements per *AR* interval.

Instance	Mesh *AR*	[0,0.2]	]0.2,0.4]	]0.4,0.6]	]0.6,0.8]	]0.8,1]
cortex_5_0	0.001485	23	358	4743	6611	1160
	0.087566	11	367	4761	6594	1158
	0.140746	5	379	4760	6582	1149
cortex_5_1	0.000218	16	343	4347	6080	1219
	0.108655	7	346	4338	6081	1211
	0.141132	6	351	4341	6083	1207
cortex_5_2	0.005723	10	286	3425	3562	1046
	0.042106	10	284	3435	3557	1041
	0.081617	9	289	3422	3538	1032
cortex_5_3	0.101051	13	345	4394	6001	1149
	0.101051	12	354	4389	5998	1150
	0.078139	25	364	4355	5989	1153
cortex_5_4	0.003869	8	423	5403	11444	1235
	0.131622	4	428	5397	11441	1236
	0.076700	6	425	5402	11432	1230
cortex_5_5	0.001395	12	234	4091	4253	1276
	0.067101	9	239	4089	4259	1275
	0.067101	9	239	4089	4259	1275
cortex_5_6	0.010296	13	285	3944	4273	1120
	0.004275	23	343	3880	4214	1139
	0.004275	23	343	3880	4214	1139

Table 4.6: Number of elements in AR's lower intervals.

Instance	[0,0.04]	[0.04,0.08]	[0.08,0.12]	[0.12,0.16]	[0.16,0.2]
cortex_5_0	6	3	3	6	5
	0	0	3	5	3
	0	0	0	2	3
cortex_5_1	6	2	1	3	4
	0	0	1	2	4
	0	0	0	2	4
cortex_5_2	1	0	1	5	3
	0	2	1	5	2
	0	0	2	4	3
cortex_5_3	0	0	3	5	5
	0	0	3	5	4
	0	1	2	11	11
cortex_5_4	2	0	0	4	2
	0	0	0	3	1
	0	1	1	3	1
cortex_5_5	3	3	1	4	1
	0	2	1	4	2
	0	2	1	4	2
cortex_5_6	1	0	3	3	6
	2	0	6	5	10
	2	0	6	5	10

The second version of the proposed algorithm generates the same results of the first version in terms of quality of the final mesh and number of elements for quality intervals.

The idea behind the design of the second version was to improve running time. A comparison of running time, number of iterations and time per iteration was done. Time improvement of time per iteration between the two proposed versions was computed. The results are shown in Table 4.7. The table contains the original algorithms run time as reference.

Table 4.7: Total Computational Time and Time per iteration comparison between proposed algorithm's versions 1 and 2.

Instance	Time[a]	Time[b]	It[bd]	T/It[be]	Time[c]	It[cd]	T/It[ce]	Imp[cf]
cortex_5_0	2312	4.415	2	2.103	2917	2	0.605	71.2%
cortex_5_1	2111	4.031	2	1.920	2698	2	0.587	69.4%
cortex_5_2	1776	3.387	2	1.611	2247	2	0.471	70.8%
cortex_5_3	2060	3.993	2	1.933	2595	2	0.535	72.3%
cortex_5_4	2819	5.492	2	2.673	3637	2	0.818	69.4%
cortex_5_5	1813	3.506	2	1.693	2275	2	0.462	72.7%
cortex_5_6	1800	8.702	5	1.726	3637	5	0.459	73.4%

[a] Original Algorithm
[b] Proposed Algorithm V1
[c] Proposed Algorithm V2
[d] Number of Iterations
[e] Time per Iteration
[f] Time per Iteration improvement

The complementary approach proposed in Section 3.4.1 has its worst case when the ROI encompass the whole mesh. A measure of the J_{ENS} and AR of the meshes that only had one RL in the whole mesh was done. The meshes were obtained using the original algorithm, the test Ω, and refinements of 5, 6 and 7. The results are shown in Table 4.8.

Table 4.8: Quality of instances with ROI that encompass the whole mesh.

Instance	Mesh J_{ENS}	Mesh AR	Instance	Mesh J_{ENS}	Mesh AR
Liver_5	0.214631	0.323139	Cortex_5	0.111649	0.294145
Liver_6	0.188560	0.315043	Cortex_6	0.129491	0.172594
Liver_7	0.178247	0.308191	Cortex_7	0.176914	0.308087

Chapter 5

Conclusions

In this thesis we analyzed the octree–based technique with mixed elements proposed by Claudio Lobos (see Section 2.3.1). We found that when using transition patterns at the domain's boundary this technique could generate invalid elements. We proposed that a specific repairing technique for this problem could be designed.

To propose a repairing technique, first we needed to analyze the cases in which the algorithm generates invalid elements. We used two different input domains for the tests. To make our tests encompass different cases of element generations we define varied regions of interest for the input domains (see Figure 4.2 and Figure 4.2). These regions were designed so transitions appear in different depths in the mesh, specially in the cortex domain.

The initial study found that in most of the cases, if not all, the invalid elements were generated in octants that had a transition pattern applied or were adjacent to one, and where part of the boundary octants. As boundary octants, they had surface patterns also applied to them.

Further analysis shows that one of the transition patterns was present in most of the cases. This pattern is the one that bridge two balanced regions. Since the refinement region's axis are aligned with the octant division this is the pattern that most likely will be applied to boundary octants.

Analysis of node configuration of this transition pattern was done as shown in Figure 4.5 and

Table 4.1. The data shows that some configurations have a tendency to appear more in the mesh. This findings were not fully used in the technique proposed in this thesis, so further work could be made with them. One idea is to define surface–transition patterns that handle these specific cases.

All the invalid elements that appeared in the test instances where part of the boundary octants. Since invalid elements are almost flat or inverted a **novel node projection technique was proposed**. This technique projects all the nodes of invalid elements to eliminate them from the mesh. This projection could affect adjacent elements and make them invalid, so the algorithm was made iterative.

When trying to eliminate invalid elements (quality threshold of 0.03), only one of the 37 test instances failed. We can conclude that the proposed technique is able to eliminate invalid elements in most of the cases. Further more, since the instance that failed was one of the instances with low refinement level, and therefore bigger elements, forceful projection of nodes will have a bigger impact on adjacent elements. We suggest that with bigger refinement levels our technique is less prompt to error, but further testing may be required.

The original technique only projects internal nodes that are sufficiently near the domain boundary, this is done so element stretching is avoided. Since the proposed technique forcefully projects some nodes, stretching of elements could happen. To measure element stretching we used the AR. We were able to eliminate invalid elements from most of the cases, without a significant stretching of neighbor elements, as shown by the Aspect Ratio measurement in Table 4.4, Table 4.5 and Table 4.6.

When a bigger quality threshold was used (0.05) a greater number of the instances failed. Further more, in some of the cases that were a success with this threshold we noted a little degradation of the AR for hexahedra and wedges. This can be explained as elements in quality range $[0.03, 0.05]$ are not as flat as elements bellow 0.03 quality, so projection of nodes has a bigger impact on neighbor elements. We conclude that the introduced algorithm works most of the time for achieving element validity throughout the mesh. However, it should not be used for further mesh quality improvement, since it can drastically affect adjacent elements of acceptable quality.

After analysis of test results and further experimentation using a bigger threshold (0.05), we can recommend that this algorithm should be set to a maximum of 3 iterations, since most of the successful cases were obtained with 2 or 3 iterations. When more iterations were needed the tests normally failed. The second iteration projects onto the boundary the nodes of the original invalid elements and the third also projects those of adjacent elements that were impacted by the second iteration's displacements. If further iterations are needed, this could exponentially propagate the deformations, incrementing the number of impacted elements and possible errors.

We recommend that if the proposed algorithm fails to eliminate invalid elements the region of interest should be redefined in order to be bigger. If this process keeps going, the worst case results in a region of interest that encompass Ω, so the refinement level in every boundary octant is the same, this is the one defined by the region of interest. To our knowledge, this approach rarely produce invalid elements as seen in Table 4.8. Further study on runtime and mesh quality of this approach is needed.

In the first version of the algorithm every extra iteration added near 100% of the first iteration's computational time. This made the approach slow in terms of runtime. Even when we use the limitation of 3 maximum iterations previously proposed, the repair time of the algorithm is 2 times the original mesh generation time. This happens because the first version was a script that run the code and manipulated it from the outside.

Doing a little study of the steps of the meshing algorithm that were not modified by our approach we were able to detect and save a common state in the middle of the mesh generation process from which to start every iteration. It turned out that the generation of the balanced octree mesh and application of transitions patterns took about two thirds of the meshing process. This two steps are the ones that we do not modified with our approach. Starting every iteration from this saved state we reduced the extra iteration to about 30% of the first iteration. So, if the maximum number of iterations is set to 3, the repair time should never be more than the original mesh generation time.

With an analysis and understanding of the algorithm we were able to make an improvement of 70% of the running time, by detecting the common steps in every iteration and excluding

them from the extra iterations.

We believe that octree–based algorithms are powerful and if they produce invalid elements, it will be at domain's boundary. Therefore, we strongly recommend that octree–based algorithms manage element inversion as part of the algorithm. Further quality improvement can be achieved by node relaxation strategies like [20, 21]. However as they do not remove (flat) elements from the mesh, validity will be achieved by causing a major distortion of neighbor elements. Techniques like the one introduced in this thesis can help to increase element quality throughout the entire mesh.

5.1. Future work

In the conclusions we presented some ideas for future work that derived from our work or are possible with the information found in this thesis. In this section we describe in more detail this ideas so they could be used in other studies.

As noted in the previous section, the analysis on node configuration of transition patterns with invalid elements was done. A further analysis could be used to define surface patterns specific for transition octants or surface–transition patterns definitions.

Surface patterns define a set of elements to replace a boundary element depending on its node configuration. Originally the surface patterns were designed only for hexahedra. With the addition of transition patterns new surface patterns were defined for the pyramid an wedge. This new patterns follow the rules of the ones defined for the hexahedra. This may be the cause for the generation of invalid elements since the rules for hexahedra division do not translate well to the other types of elements.

One common occurrence in the tests was a division of a pyramid in two tetrahedra where none of the new elements could be eliminated since the apex was still inside of the domain. Since both tetrahedra remain in the mesh, element invalidity could happen when outside node projection onto the domain is performed. Further study of common cases is needed to define new approaches.

Another future work proposed in the previous section is the implementation and testing of the redefinition of the ROI when the algorithm fails. The idea is to make the ROI bigger and generate the mesh using the new one. One variable to consider is the rate of stretching, meaning the percentage in which the ROI will grow. We recommend validation of this approach by measuring runtime and element quality.

Further work in quality improvement after invalid element elimination is also required. Node relaxation is one of the techniques that could be used for this purpose. Implementation and validation of different techniques could be done.

All the approaches described in this section are meant to complement the results found in this thesis. Invalid element elimination is only one step to better the representation done by the octree-based technique with mixed elements.

Chapter 6

Appendix

6.1. Comparison of External Mesh Representation in Zones with Projected Nodes

In this section a comparison of the external representation of the mesh generated by the cortex instances with RL 5 at the ROI are shown.

We compare the mesh representation in the zones that had nodes projected, the first image corresponds to the mesh generated by the original algorithm, while the second showa the same zone in the mesh generated by the proposed algorithm.

Nodes that are labeled for projection or were projected are shown in red.

6.1.1. ROI 0

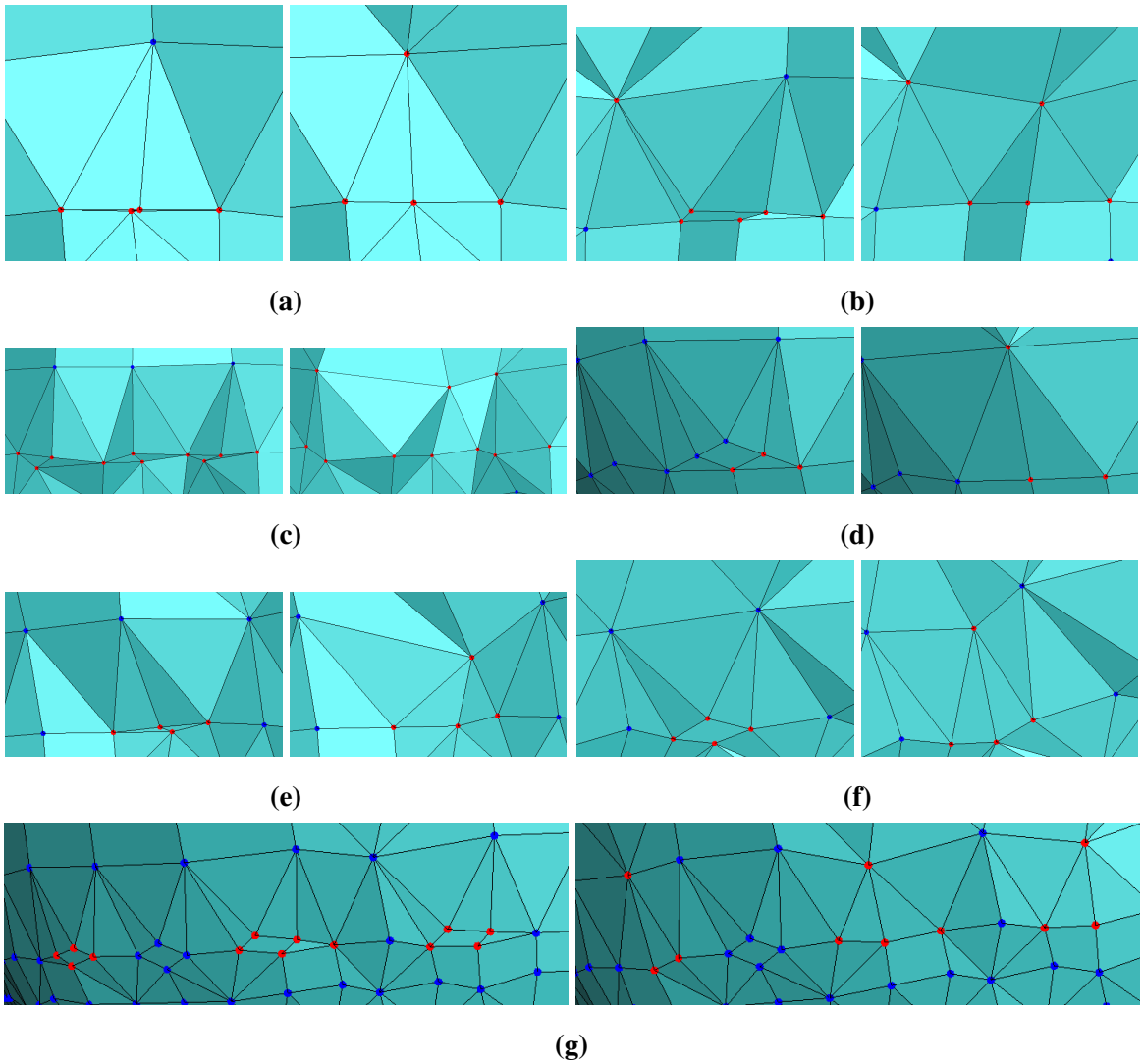


Figure 6.1: Zones with projected nodes in cortex_5_0. In each subfigure: Mesh generated with original algorithm (left) and mesh generated with proposed algorithm (right).

6.1.2. ROI 1

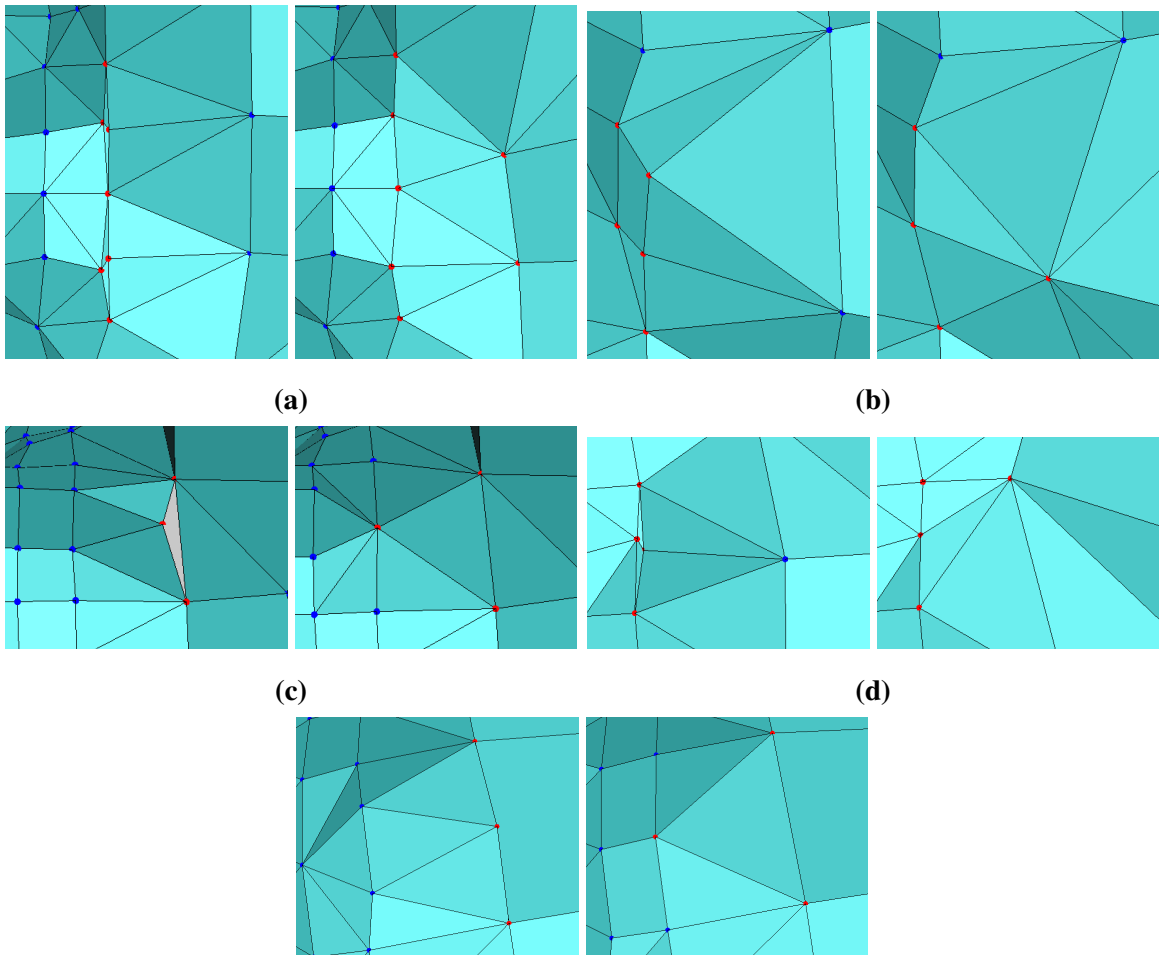
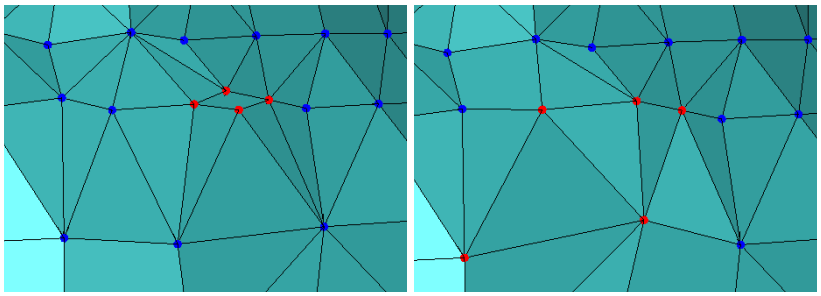
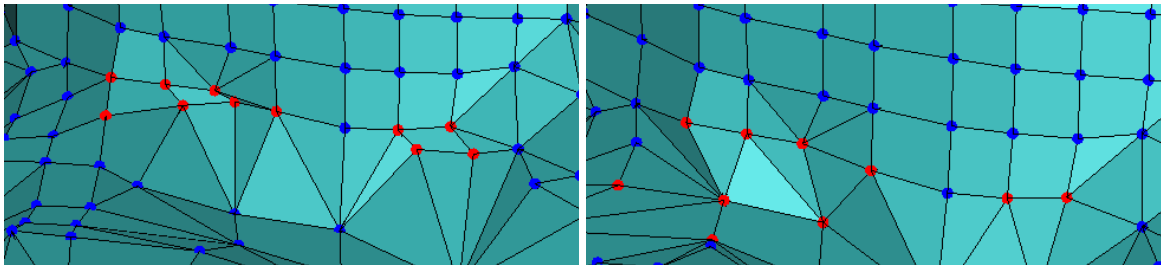


Figure 6.2: Zones with projected nodes in cortex_5_1. In each subfigure: Mesh generated with original algorithm (left) and mesh generated with proposed algorithm (right).

6.1.3. ROI 2



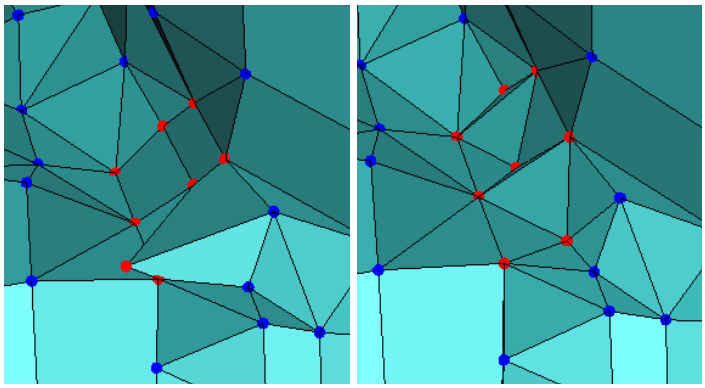
(a)



(b)

Figure 6.3: Zones with projected nodes in cortex_5_2. In each subfigure: Mesh generated with original algorithm (left) and mesh generated with proposed algorithm (right).

6.1.4. ROI 3



(a)

Figure 6.4: Zones with projected nodes in cortex_5_3. In each subfigure: Mesh generated with original algorithm (left) and mesh generated with proposed algorithm (right).

6.1.5. ROI 4

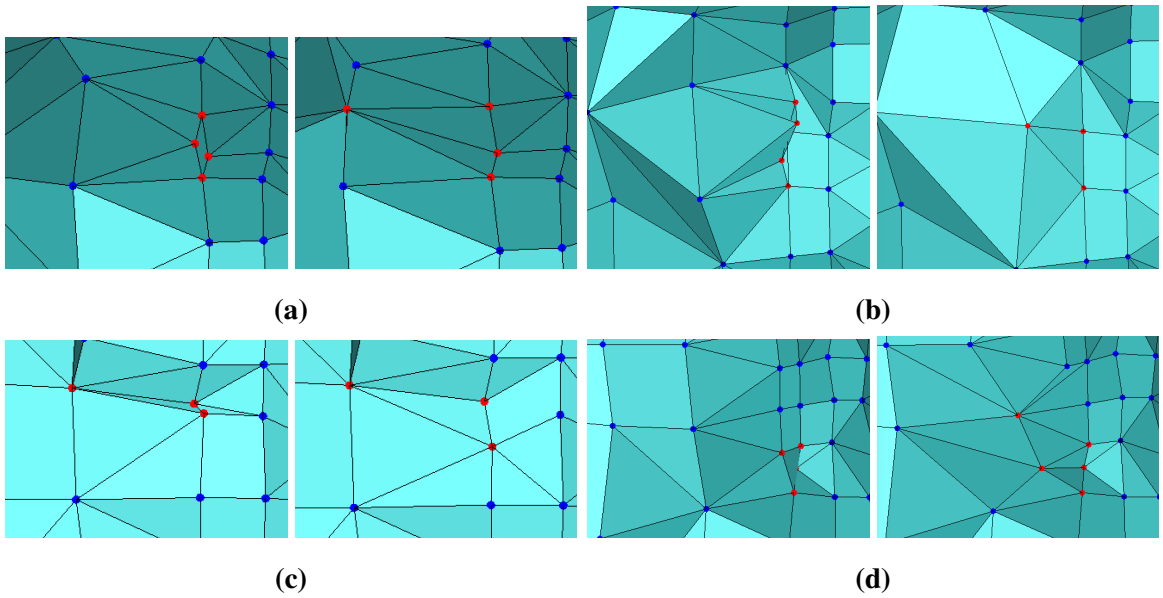


Figure 6.5: Zones with projected nodes in cortex_5_4. In each subfigure: Mesh generated with original algorithm (left) and mesh generated with proposed algorithm (right).

6.1.6. ROI 5

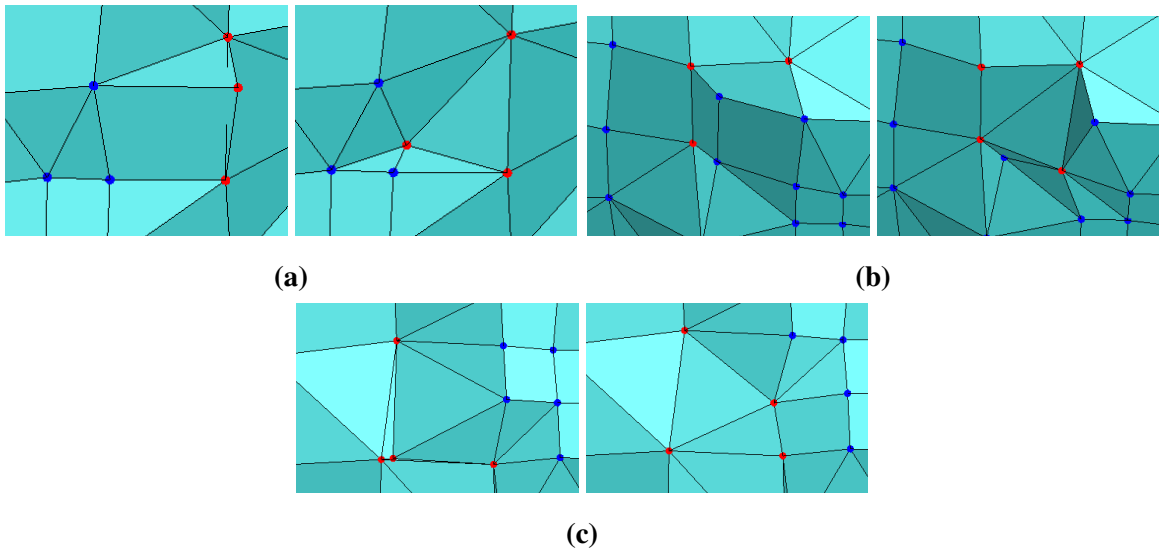


Figure 6.6: Zones with projected nodes in cortex_5_5. In each subfigure: Mesh generated with original algorithm (left) and mesh generated with proposed algorithm (right).

6.2. Tables

6.2.1. Quality and Time: Original Versus Proposed Algorithm

Table 6.1: Comparison of quality and time for the original algorithm and the proposed algorithm's first version for liver instances.

Instance	Quality[a]	Time[a]	Quality[a]	Time[a]	It[bc]	T/It[bd]
liver_5_0	0.020270	0.739	0.042937	1.445	2	0.706
liver_5_1	-0.027474	0.926	0.035620	1.884	2	0.958
liver_5_2	0.000118	1.116	0.032577	2.231	2	1.115
liver_5_3	0.019386	0.935	0.032577	1.854	2	0.919
liver_5_4	0.000118	0.933	0.032577	1.873	2	0.940
liver_5_5	-0.041692	1.127	0.059450	2.208	2	1.081
liver_6_0	-0.008273	3.728	0.036668	11.833	3	4.053
liver_6_1	-0.107687	4.310	0.031186	8.607	2	4.297
liver_6_2	0.024276	6.118	0.031571	18.266	3	6.074
liver_6_3	-0.658588	4.296	0.031571	12.937	3	4.321
liver_6_4	0.035969	5.194	0.035969	5.051	1	
liver_6_5	-0.484636	5.580	0.057798	16.694	3	5.557
liver_7_0	-0.006435	16.248	0.033603	50.066	3	16.909
liver_7_1	-0.045873	23.708	0.032360	71.974	3	24.133
liver_7_2	0.008232	32.832	0.031518	100.269	3	33.719
liver_7_3	0.019535	18.841	0.031518	58.694	3	19.927
liver_7_4	0.008232	29.059	0.033574	59.420	2	30.361
liver_7_5	-0.519822	27.754	0.031121	87.327	3	29.787

[a] Original Algorithm
[b] Proposed Algorithm V1
[c] Numbre of Iterations
[d] Time per Iteration

Table 6.2: Comparison of quality and time for the original algorithm and the proposed algorithm's first version for cortex instances..

Instance	Quality[a]	Time[a]	Quality[a]	Time[a]	It[bc]	T/It[bd]
cortex_5_0	-0.009743	2.312	0.037720	4.415	2	2.103
cortex_5_1	-0.000639	2.111	0.041439	4.031	2	1.920
cortex_5_2	0.002585	1.776	0.030672	3.387	2	1.611
cortex_5_3	-0.999649	2.060	0.041507	3.993	2	1.933
cortex_5_4	-0.222353	2.819	0.042440	5.492	2	2.673
cortex_5_5	-0.100899	1.813	0.072093	3.506	2	1.693
cortex_5_6	-0.491747	1.800	-0.981794	8.702	5	1.726
cortex_6_0	0.003271	9.090	0.031988	17.803	2	8.713
cortex_6_1	-0.448111	7.600	0.037047	22.303	3	7.352
cortex_6_2	-0.085751	5.853	0.036888	17.051	3	5.599
cortex_6_3	0.005869	6.736	0.034045	13.336	2	6.600
cortex_6_4	-0.241382	12.101	0.065137	35.728	3	11.814
cortex_6_5	-0.066082	6.873	0.038331	13.430	2	6.557
cortex_6_6	-0.619333	6.377	0.032520	18.798	3	6.211
cortex_7_0	-0.235302	45.342	0.030904	89.020	2	43.678
cortex_7_1	-0.481033	40.016	0.030665	118.788	3	39.386
cortex_7_2	-0.030257	24.771	0.033142	48.355	2	23.584
cortex_7_3	-0.316319	35.539	0.037141	106.010	3	35.236
cortex_7_4	0.009385	60.464	0.030015	183.344	3	61.440
cortex_7_5	0.000001	31.360	0.038597	62.191	2	30.831
cortex_7_6	-0.726812	28.186	0.034500	83.139	3	27.477

[a] Original Algorithm [b] Proposed Algorithm V1

[c] Numbre of Iterations [d] Time per Iteration

6.2.2. Number of Elements in J_{ENS} Intervals

Table 6.3: Number of elements per J_{ENS} interval for the liver instances with RL 5 at the ROI.

Instance	Mesh J_{ENS}	Inv[a]	Inv[b]	Que[c]	Que[d]	Que[e]	Que[f]	Good[g]	Total[h]
liver_5_0	0.020270	0	1	1	3	14	19	6375	6413
	0.042937	0	0	1	2	12	19	6374	6408
	0.080986	0	0	0	1	12	19	6363	6395
liver_5_1	-0.027474	1	1	5	16	46	74	7990	8133
	0.035620	0	0	5	15	42	76	7979	8117
	0.050222	0	0	0	13	37	79	7955	8084
liver_5_2	0.000118	0	2	7	12	30	68	12081	12200
	0.032577	0	0	7	13	32	75	12068	12195
	0.056075	0	0	0	12	34	72	12067	12185
liver_5_3	0.019386	0	3	7	12	19	34	8139	8214
	0.032577	0	0	6	12	19	35	8151	8223
	0.056075	0	0	0	12	18	37	8153	8220
liver_5_4	0.000118	0	2	4	3	34	77	9422	9542
	0.032577	0	0	4	4	36	84	9409	9537
	0.091241	0	0	0	3	37	82	9402	9524
liver_5_5	-0.041692	1	2	0	27	36	70	10565	10701
	0.059450	0	0	0	25	34	66	10559	10684
	0.059450	0	0	0	25	34	66	10559	10684

[a] Inverted Elements: [-1,0[

[b] Invalid Elements: [0,0.03[

[c] Questionable Interval 1: [0.03,0.05[

[d] Questionable Interval 2: [0.05,0.1[

[e] Questionable Interval 3: [0.1,0.15[

[f] Questionable Interval 4: [0.15,0.2[

[g] Good Quality: [0.2,1[

[h] Total Elements

Table 6.4: Number of elements per J_{ENS} interval for the liver instances with RL 6 at the ROI.

Instance	Mesh J_{ENS}	Inv[a]	Inv[b]	Que[c]	Que[d]	Que[e]	Que[f]	Good[g]	Total[h]
liver_6_0	-0.008273	1	4	8	12	32	87	37142	37286
	0.036668	0	0	8	11	33	86	37153	37291
	0.051381	0	0	0	9	32	86	37143	37270
liver_6_1	-0.107687	1	6	3	20	76	156	47825	48087
	0.031186	0	0	3	19	70	156	47835	48083
	0.064924	0	0	0	19	70	158	47836	48083
liver_6_2	0.024276	0	1	11	17	48	160	77742	77979
	0.031571	0	0	11	18	50	160	77753	77992
	0.050636	0	0	0	17	49	166	77768	78000
liver_6_3	-0.658588	2	6	16	17	31	88	47068	47228
	0.031571	0	0	15	18	33	86	47081	47233
	0.050636	0	0	0	17	32	93	47089	47231
liver_6_4	0.035969	0	0	2	9	66	196	59799	60072
	0.035969	0	0	2	9	66	196	59799	60072
	0.055803	0	0	0	9	65	199	59795	60068
liver_6_5	-0.484636	5	3	0	50	88	139	66990	67275
	0.057798	0	0	0	46	82	149	66962	67239
	0.057798	0	0	0	46	82	149	66962	67239

[a] Inverted Elements: [-1,0[

[b] Invalid Elements: [0,0.03[

[c] Questionable Interval 1: [0.03,0.05[

[d] Questionable Interval 2: [0.05,0.1[

[e] Questionable Interval 3: [0.1,0.15[

[f] Questionable Interval 4: [0.15,0.2[

[g] Good Quality: [0.2,1[

[h] Total Elements

Table 6.5: Number of elements per J_{ENS} interval for the liver instances with RL 7 at the ROI.

Instance	Mesh J_{ENS}	Inv[a]	Inv[b]	Que[c]	Que[d]	Que[e]	Que[f]	Good[g]	Total[h]
liver_7_0	-0.006435	1	9	13	27	53	190	203467	203760
	0.033603	0	0	13	24	47	189	203439	203712
	0.051874	0	0	0	25	47	190	203413	203675
liver_7_1	-0.045873	3	15	7	49	169	387	287257	287887
	0.032360	0	0	7	50	169	402	287203	287831
	0.052098	0	0	0	47	167	401	287184	287799
liver_7_2	0.008232	0	10	27	40	117	271	499960	500425
	0.031518	0	0	27	37	113	278	499941	500396
	-0.979086	25	8	7	65	190	309	499774	500378
liver_7_3	0.019535	0	12	45	46	60	233	257762	258158
	0.031518	0	0	43	40	56	225	257743	258107
	-0.934351	10	4	4	48	98	242	257663	258069
liver_7_4	0.008232	0	7	12	19	123	281	375444	375886
	0.033574	0	0	12	19	121	290	375443	375885
	-0.979086	15	4	3	36	153	301	375338	375850
liver_7_5	-0.519822	21	2	12	99	204	420	406565	407323
	0.031121	0	0	9	92	207	424	406480	407212
	0.056925	0	0	0	92	193	428	406428	407141

[a] Inverted Elements: [-1,0[

[b] Invalid Elements: [0,0.03[

[c] Questionable Interval 1: [0.03,0.05[

[d] Questionable Interval 2: [0.05,0.1[

[e] Questionable Interval 3: [0.1,0.15[

[f] Questionable Interval 4: [0.15,0.2[

[g] Good Quality: [0.2,1[

[h] Total Elements

Table 6.6: Number of elements per J_{ENS} interval for the cortex instances with RL 5 at the ROI.

Instance	Mesh J_{ENS}	Inv[a]	Inv[b]	Que[c]	Que[d]	Que[e]	Que[f]	Good[g]	Total[h]
cortex_5_0	-0.009743	1	8	8	14	35	31	12798	12895
	0.037720	0	0	7	12	35	34	12803	12891
	0.053261	0	0	0	7	35	42	12791	12875
cortex_5_1	-0.000639	1	7	2	9	18	29	11939	12005
	0.041439	0	0	2	9	18	24	11930	11983
	0.058775	0	0	0	9	18	25	11936	11988
cortex_5_2	0.002585	0	3	2	13	22	36	8253	8329
	0.030672	0	0	4	13	21	36	8253	8327
	0.059482	0	0	0	13	22	38	8217	8290
cortex_5_3	-0.999649	3	0	1	11	44	86	11757	11902
	0.041507	0	0	2	12	47	79	11763	11903
	0.000000	0	2	2	22	48	84	11728	11886
cortex_5_4	-0.222353	2	2	1	9	34	63	18402	18513
	0.042440	0	0	1	7	32	64	18402	18506
	0.056449	0	0	0	7	31	67	18390	18495
cortex_5_5	-0.100899	2	2	0	8	30	55	9769	9866
	0.072093	0	0	0	7	33	57	9774	9871
	0.072093	0	0	0	7	33	57	9774	9871
cortex_5_6	-0.491747	9	0	0	2	47	71	9506	9635
	-0.981794	11	1	3	19	55	79	9431	9599
	-0.981794	11	1	3	19	55	79	9431	9599

[a] Inverted Elements: [-1,0[

[b] Invalid Elements: [0,0.03[

[c] Questionable Interval 1: [0.03,0.05[

[d] Questionable Interval 2: [0.05,0.1[

[e] Questionable Interval 3: [0.1,0.15[

[f] Questionable Interval 4: [0.15,0.2[

[g] Good Quality: [0.2,1[

[h] Total Elements

Table 6.7: Number of elements per J_{ENS} interval for the cortex instances with RL 6 at the ROI.

Instance	Mesh J_{ENS}	Inv[a]	Inv[b]	Que[c]	Que[d]	Que[e]	Que[f]	Good[g]	Total[h]
cortex_6_0	0.003271	0	7	8	33	48	67	75830	75993
	0.031988	0	0	6	32	47	72	75820	75977
	0.051806	0	0	0	30	48	75	75821	75974
cortex_6_1	-0.448111	3	7	7	22	59	98	63171	63367
	0.037047	0	0	5	19	57	104	63165	63350
	-0.012889	1	0	0	19	64	109	63165	63358
cortex_6_2	-0.085751	2	5	7	21	31	78	40773	40917
	0.036888	0	0	6	20	38	86	40770	40920
	-0.270037	7	4	5	29	49	109	40713	40916
cortex_6_3	0.005869	0	9	3	13	58	136	62248	62467
	0.034045	0	0	2	14	59	142	62234	62451
	0.058320	0	0	0	13	57	141	62235	62446
cortex_6_4	-0.241382	7	2	1	17	47	111	118084	118269
	0.065137	0	0	0	12	42	113	118094	118261
	0.065137	0	0	0	12	42	113	118094	118261
cortex_6_5	-0.066082	2	5	2	6	51	102	56395	56563
	0.038331	0	0	2	5	52	104	56401	56564
	0.044116	0	0	1	4	52	103	56410	56570
cortex_6_6	-0.619333	3	1	4	5	42	77	47886	48018
	0.032520	0	0	4	5	50	76	47870	48005
	0.075723	0	0	0	4	49	75	47856	47984

[a] Inverted Elements: [-1,0[

[b] Invalid Elements: [0,0.03[

[c] Questionable Interval 1: [0.03,0.05[

[d] Questionable Interval 2: [0.05,0.1[

[e] Questionable Interval 3: [0.1,0.15[

[f] Questionable Interval 4: [0.15,0.2[

[g] Good Quality: [0.2,1[

[h] Total Elements

Table 6.8: Number of elements per J_{ENS} interval for the cortex instances with RL 7 at the ROI.

Instance	Mesh J_{ENS}	Inv[a]	Inv[b]	Que[c]	Que[d]	Que[e]	Que[f]	Good[g]	Total[h]
cortex_7_0	-0.235302	1	21	23	80	108	220	473642	474095
	0.030904	0	0	20	79	104	226	473639	474068
	-0.801307	10	2	7	85	142	285	473541	474072
cortex_7_1	-0.481033	4	15	19	58	103	260	388947	389406
	0.030665	0	0	14	60	103	274	388947	389398
	0.050069	0	0	0	59	100	285	388903	389347
cortex_7_2	-0.030257	2	8	11	52	83	209	209426	209791
	0.033142	0	0	11	48	79	206	209379	209723
	0.051032	0	0	0	47	77	205	209382	209711
cortex_7_3	-0.316319	3	12	6	22	136	303	349860	350342
	0.037141	0	0	5	21	132	306	349840	350304
	0.051535	0	0	0	20	130	303	349830	350283
cortex_7_4	0.009385	0	10	11	36	152	315	769613	770137
	0.030015	0	0	8	35	147	316	769619	770125
	0.051006	0	0	0	36	148	317	769618	770119
cortex_7_5	0.000001	0	3	2	8	90	230	293036	293369
	0.038597	0	0	2	7	90	234	293043	293376
	0.076452	0	0	0	7	91	225	293038	293361
cortex_7_6	-0.726812	2	5	7	13	72	198	242480	242777
	0.034500	0	0	6	14	75	191	242452	242738
	0.052033	0	0	0	18	74	198	242427	242717

[a] Inverted Elements: [-1,0[

[b] Invalid Elements: [0,0.03[

[c] Questionable Interval 1: [0.03,0.05[

[d] Questionable Interval 2: [0.05,0.1[

[e] Questionable Interval 3: [0.1,0.15[

[f] Questionable Interval 4: [0.15,0.2[

[g] Good Quality: [0.2,1[

[h] Total Elements

6.2.3. Lowest J_{ENS} and AR

Table 6.9: Mesh J_{ENS}, Mesh AR y AR per type of element for the liver instances with RL 5 at the ROI.

Instance	J_{ENS} [a]	AR [b]	Hex [c]	Tet [d]	Pyr [e]	Wed [f]
liver_5_0	0.020270	0.025864	0.353553	0.025864	0.257546	0.292679
	0.042937	0.096493	0.353553	0.096493	0.257546	0.292679
	0.080986	0.191630	0.353553	0.191630	0.309753	0.292679
liver_5_1	-0.027474	0.010108	0.353553	0.010108	0.223002	0.305625
	0.035620	0.051116	0.353553	0.051116	0.223002	0.305625
	0.050222	0.063615	0.353553	0.063615	0.289330	0.305625
liver_5_2	0.000118	0.000165	0.353553	0.000165	0.210317	0.320257
	0.032577	0.070934	0.353553	0.070934	0.210317	0.320257
	0.056075	0.081244	0.353553	0.081244	0.265623	0.307789
liver_5_3	0.019386	0.048812	0.353553	0.048812	0.210317	0.320257
	0.032577	0.070934	0.353553	0.070934	0.210317	0.320257
	0.056075	0.081244	0.353553	0.081244	0.265623	0.320257
liver_5_4	0.000118	0.000165	0.353553	0.000165	0.276675	0.320257
	0.032577	0.070934	0.353553	0.070934	0.278696	0.320257
	0.091241	0.134768	0.353553	0.134768	0.278696	0.307789
liver_5_5	-0.041692	0.001224	0.353553	0.001224	0.227585	0.282525
	0.059450	0.108585	0.353553	0.108585	0.227585	0.301815
	0.059450	0.108585	0.353553	0.108585	0.227585	0.301815

[a] Lowest J_{ENS} in the Mesh
[b] Lowest AR in the Mesh
[c] Lowest AR between Hexahedra
[d] Lowest AR between Tetrahedra
[e] Lowest AR between Pyramids
[f] Lowest AR between Wedges

Table 6.10: Mesh J_{ENS}, Mesh *AR* y *AR* per type of element for the liver instances with RL 6 at the ROI.

Instance	J_{ENS} [a]	AR [b]	Hex [c]	Tet [d]	Pyr [e]	Wed [f]
liver_6_0	-0.008273	0.003107	0.491449	0.003107	0.136774	0.315640
	0.036668	0.046078	0.491449	0.046078	0.136774	0.315640
	0.051381	0.097903	0.491449	0.097903	0.228360	0.259496
liver_6_1	-0.107687	0.000015	0.451257	0.000015	0.175740	0.317470
	0.031186	0.039991	0.451257	0.039991	0.175740	0.317470
	0.064924	0.119643	0.451257	0.119643	0.175740	0.317470
liver_6_2	0.024276	0.053402	0.461681	0.053402	0.194853	0.315043
	0.031571	0.067890	0.461681	0.067890	0.194853	0.315043
	0.050636	0.107315	0.461681	0.107315	0.194853	0.315043
liver_6_3	-0.658588	0.006886	0.461681	0.006886	0.194853	0.309684
	0.031571	0.067890	0.461681	0.067890	0.194853	0.309684
	0.050636	0.107315	0.461681	0.107315	0.194853	0.302851
liver_6_4	0.035969	0.072249	0.491449	0.072249	0.295857	0.315043
	0.035969	0.072249	0.491449	0.072249	0.295857	0.315043
	0.055803	0.072249	0.491449	0.072249	0.295857	0.315043
liver_6_5	-0.484636	0.010190	0.353553	0.010190	0.205078	0.312488
	0.057798	0.074480	0.353553	0.074480	0.205078	0.312488
	0.057798	0.074480	0.353553	0.074480	0.205078	0.312488

[a] Lowest J_{ENS} in the Mesh
[b] Lowest *AR* in the Mesh
[c] Lowest *AR* between Hexahedra
[d] Lowest *AR* between Tetrahedra
[e] Lowest *AR* between Pyramids
[f] Lowest *AR* between Wedges

Table 6.11: Mesh J_{ENS}, Mesh *AR* y *AR* per type of element for the liver instances with RL 7 at the ROI.

Instance	J_{ENS} [a]	AR [b]	Hex [c]	Tet [d]	Pyr [e]	Wed [f]
liver_7_0	-0.006435	0.002100	0.497318	0.002100	0.137836	0.284869
	0.033603	0.040153	0.497318	0.040153	0.137836	0.284869
	0.051874	0.092002	0.497318	0.092002	0.230478	0.258125
liver_7_1	-0.045873	0.002708	0.471596	0.002708	0.222046	0.296777
	0.032360	0.038973	0.471596	0.038973	0.222046	0.296777
	0.052098	0.066886	0.471596	0.066886	0.222046	0.296777
liver_7_2	0.008232	0.010497	0.353553	0.010497	0.154693	0.309781
	0.031518	0.054004	0.353553	0.054004	0.154693	0.309781
	-0.979086	0.000619	0.322641	0.000619	0.178765	0.136950
liver_7_3	0.019535	0.042329	0.463868	0.042329	0.173084	0.309781
	0.031518	0.054004	0.463868	0.054004	0.173084	0.309781
	-0.934351	0.001335	0.322641	0.001335	0.183626	0.136950
liver_7_4	0.008232	0.010497	0.353553	0.010497	0.154693	0.312522
	0.033574	0.046011	0.353553	0.046011	0.154693	0.312522
	-0.979086	0.000619	0.353553	0.000619	0.178765	0.152083
liver_7_5	-0.519822	0.000286	0.353553	0.000286	0.210402	0.298330
	0.031121	0.056545	0.353553	0.056545	0.210402	0.301969
	0.056925	0.073660	0.353553	0.073660	0.210402	0.301969

[a] Lowest J_{ENS} in the Mesh
[b] Lowest *AR* in the Mesh
[c] Lowest *AR* between Hexahedra
[d] Lowest *AR* between Tetrahedra
[e] Lowest *AR* between Pyramids
[f] Lowest *AR* between Wedges

Table 6.12: Mesh J_{ENS}, Mesh *AR* y *AR* per type of element for the cortex instances with RL 5 at the ROI.

Instance	J_{ENS} [a]	AR [b]	Hex [c]	Tet [d]	Pyr [e]	Wed [f]
cortex_5_0	-0.009743	0.001485	0.353553	0.001485	0.149989	0.323838
	0.037720	0.087566	0.353553	0.087566	0.225299	0.324094
	0.053261	0.140746	0.353553	0.140746	0.253287	0.309168
cortex_5_1	-0.000639	0.000218	0.522315	0.000218	0.232392	0.294145
	0.041439	0.108655	0.522315	0.108655	0.232392	0.294145
	0.058775	0.141132	0.522315	0.141132	0.238242	0.294145
cortex_5_2	0.002585	0.005723	0.536592	0.005723	0.205097	0.323838
	0.030672	0.042106	0.536592	0.042106	0.240023	0.323838
	0.059482	0.081617	0.536592	0.081617	0.240023	0.323838
cortex_5_3	-0.999649	0.101051	0.353553	0.101051	0.233464	0.316153
	0.041507	0.101051	0.353553	0.101051	0.259568	0.316153
	0.000000	0.078139	0.353553	0.078139	0.174202	0.189164
cortex_5_4	-0.222353	0.003869	0.353553	0.003869	0.226341	0.311163
	0.042440	0.131622	0.353553	0.131622	0.226341	0.300113
	0.056449	0.076700	0.353553	0.076700	0.274342	0.300113
cortex_5_5	-0.100899	0.001395	0.487193	0.001395	0.299254	0.316153
	0.072093	0.067101	0.487193	0.067101	0.299254	0.316153
	0.072093	0.067101	0.487193	0.067101	0.299254	0.316153
cortex_5_6	-0.491747	0.010296	0.353553	0.010296	0.326315	0.323838
	-0.981794	0.004275	0.151413	0.004275	0.162811	0.172722
	-0.981794	0.004275	0.151413	0.004275	0.162811	0.172722

[a] Lowest J_{ENS} in the Mesh
[b] Lowest *AR* in the Mesh
[c] Lowest *AR* between Hexahedra
[d] Lowest *AR* between Tetrahedra
[e] Lowest *AR* between Pyramids
[f] Lowest *AR* between Wedges

Table 6.13: Mesh J_{ENS}, Mesh AR y AR per type of element for the cortex instances with RL 6 at the ROI.

Instance	J_{ENS} [a]	AR [b]	Hex [c]	Tet [d]	Pyr [e]	Wed [f]
cortex_6_0	0.003271	0.007474	0.353553	0.007474	0.102974	0.290617
	0.031988	0.065498	0.353553	0.065498	0.123552	0.290617
	0.051806	0.076512	0.353553	0.076512	0.194971	0.290617
cortex_6_1	-0.448111	0.012529	0.471063	0.012529	0.182439	0.317385
	0.037047	0.044228	0.471063	0.044228	0.182439	0.317385
	-0.012889	0.005552	0.471063	0.005552	0.237022	0.212824
cortex_6_2	-0.085751	0.002414	0.495702	0.002414	0.204381	0.314790
	0.036888	0.055750	0.495702	0.055750	0.242985	0.314790
	-0.270037	0.000718	0.360571	0.000718	0.133009	0.204553
cortex_6_3	0.005869	0.009525	0.485197	0.009525	0.193825	0.298347
	0.034045	0.044694	0.483181	0.044694	0.313496	0.298347
	0.058320	0.078801	0.483181	0.078801	0.313496	0.298347
cortex_6_4	-0.241382	0.006345	0.461448	0.006345	0.190631	0.293232
	0.065137	0.077851	0.461448	0.077851	0.299610	0.293232
	0.065137	0.077851	0.461448	0.077851	0.299610	0.293232
cortex_6_5	-0.066082	0.009932	0.353553	0.009932	0.270002	0.318995
	0.038331	0.041366	0.353553	0.041366	0.313496	0.318995
	0.044116	0.062024	0.353553	0.062024	0.313496	0.318995
cortex_6_6	-0.619333	0.007568	0.353553	0.007568	0.210859	0.319161
	0.032520	0.047833	0.353553	0.047833	0.210859	0.319161
	0.075723	0.089537	0.353553	0.089537	0.312103	0.319161

[a] Lowest J_{ENS} in the Mesh
[b] Lowest AR in the Mesh
[c] Lowest AR between Hexahedra
[d] Lowest AR between Tetrahedra
[e] Lowest AR between Pyramids
[f] Lowest AR between Wedges

Table 6.14: Mesh J_{ENS}, Mesh *AR* y *AR* per type of element for the cortex instances with RL 7 at the ROI.

Instance	J_{ENS} [a]	AR [b]	Hex [c]	Tet [d]	Pyr [e]	Wed [f]
cortex_7_0	-0.235302	0.001698	0.428678	0.001698	0.136693	0.284616
	0.030904	0.056098	0.428678	0.056098	0.136693	0.284616
	-0.801307	0.034727	0.376691	0.034727	0.181537	0.241339
cortex_7_1	-0.481033	0.002471	0.458545	0.002471	0.090796	0.298347
	0.030665	0.044293	0.458545	0.044293	0.165944	0.246730
	0.050069	0.077843	0.458545	0.077843	0.183330	0.246730
cortex_7_2	-0.030257	0.009419	0.452223	0.009419	0.163098	0.298347
	0.033142	0.071654	0.452223	0.071654	0.171826	0.298347
	0.051032	0.080065	0.452223	0.080065	0.228446	0.298347
cortex_7_3	-0.316319	0.001324	0.461448	0.001324	0.187203	0.307995
	0.037141	0.044725	0.461448	0.044725	0.282294	0.307995
	0.051535	0.069860	0.461448	0.069860	0.282294	0.307995
cortex_7_4	0.009385	0.016720	0.471063	0.016720	0.186296	0.298347
	0.030015	0.047257	0.471063	0.047257	0.233686	0.298347
	0.051006	0.080728	0.471063	0.080728	0.272333	0.298347
cortex_7_5	0.000001	0.000003	0.353553	0.000003	0.109902	0.298347
	0.038597	0.053094	0.353553	0.053094	0.227568	0.298347
	0.076452	0.166019	0.353553	0.166019	0.227568	0.298347
cortex_7_6	-0.726812	0.011137	0.353553	0.011137	0.183286	0.283776
	0.034500	0.044851	0.353553	0.044851	0.183286	0.283776
	0.052033	0.065723	0.277454	0.065723	0.241087	0.283776

[a] Lowest J_{ENS} in the Mesh
[b] Lowest *AR* in the Mesh
[c] Lowest *AR* between Hexahedra
[d] Lowest *AR* between Tetrahedra
[e] Lowest *AR* between Pyramids
[f] Lowest *AR* between Wedges

6.2.4. Number of Elements in *AR* Intervals

Table 6.15: Number of elements per *AR* interval for the liver instances with RL 5 at the ROI.

Instance	Mesh *AR*	[0,0.2]	]0.2,0.4]	]0.4,0.6]	]0.6,0.8]	]0.8,1]
liver_5_0	0.025864	5	198	2385	3035	790
	0.096493	3	192	2385	3039	789
	0.191630	2	191	2378	3037	787
liver_5_1	0.010108	21	328	3084	3920	780
	0.051116	19	322	3075	3924	777
	0.063615	13	319	3072	3909	771
liver_5_2	0.000165	21	373	3778	7153	875
	0.070934	20	381	3769	7149	876
	0.081244	14	388	3770	7140	873
liver_5_3	0.048812	19	217	3139	4047	792
	0.070934	16	223	3139	4054	791
	0.081244	11	227	3144	4051	787
liver_5_4	0.000165	15	337	3197	5178	815
	0.070934	14	345	3188	5174	816
	0.134768	9	352	3187	5162	814
liver_5_5	0.001224	18	368	3735	5707	873
	0.108585	13	367	3725	5704	875
	0.108585	13	367	3725	5704	875

Table 6.16: Number of elements per *AR* interval for the liver instances with RL 6 at the ROI.

Instance	Mesh *AR*	[0,0.2]	]0.2,0.4]	]0.4,0.6]	]0.6,0.8]	]0.8,1]
liver_6_0	0.003107	27	1044	13063	19136	4016
	0.046078	23	1038	13065	19151	4014
	0.097903	12	1050	13050	19150	4008
liver_6_1	0.000015	30	1230	15999	26624	4204
	0.039991	22	1217	15999	26639	4206
	0.119643	20	1221	15993	26643	4206
liver_6_2	0.053402	26	1485	18113	54027	4328
	0.067890	25	1491	18111	54037	4328
	0.107315	15	1504	18129	54028	4324
liver_6_3	0.006886	30	876	16263	25664	4395
	0.067890	25	876	16272	25673	4387
	0.107315	11	892	16286	25667	4375
liver_6_4	0.072249	21	1369	16201	38265	4216
	0.072249	21	1369	16201	38265	4216
	0.072249	19	1372	16204	38259	4214
liver_6_5	0.010190	42	1601	18632	42623	4377
	0.074480	28	1634	18565	42612	4400
	0.074480	28	1634	18565	42612	4400

Table 6.17: Number of elements per *AR* interval for the liver instances with RL 7 at the ROI.

Instance	Mesh *AR*	[0,0.2]	]0.2,0.4]	]0.4,0.6]	]0.6,0.8]	]0.8,1]
liver_7_0	0.002100	47	4150	57172	124169	18222
	0.040153	32	4134	57152	124184	18210
	0.092002	18	4152	57127	124185	18193
liver_7_1	0.002708	64	4783	71550	192231	19259
	0.038973	44	4761	71510	192259	19257
	0.066886	37	4754	71498	192254	19256
liver_7_2	0.010497	63	5796	78851	396316	19399
	0.054004	56	5790	78839	396331	19380
	0.000619	68	5960	78871	396140	19339
liver_7_3	0.042329	76	3653	71011	163574	19844
	0.054004	67	3632	71004	163583	19821
	0.001335	48	3774	71023	163450	19774
liver_7_4	0.010497	51	5307	71257	280338	18933
	0.046011	46	5313	71251	280350	18925
	0.000619	46	5381	71222	280281	18920
liver_7_5	0.000286	92	6010	81722	298975	20524
	0.056545	60	6056	81598	298921	20577
	0.073660	57	6035	81561	298932	20556

Table 6.18: Number of elements per *AR* interval for the cortex instances with RL 5 at the ROI.

Instance	Mesh *AR*	[0,0.2]	]0.2,0.4]	]0.4,0.6]	]0.6,0.8]	]0.8,1]
cortex_5_0	0.001485	23	358	4743	6611	1160
	0.087566	11	367	4761	6594	1158
	0.140746	5	379	4760	6582	1149
cortex_5_1	0.000218	16	343	4347	6080	1219
	0.108655	7	346	4338	6081	1211
	0.141132	6	351	4341	6083	1207
cortex_5_2	0.005723	10	286	3425	3562	1046
	0.042106	10	284	3435	3557	1041
	0.081617	9	289	3422	3538	1032
cortex_5_3	0.101051	13	345	4394	6001	1149
	0.101051	12	354	4389	5998	1150
	0.078139	25	364	4355	5989	1153
cortex_5_4	0.003869	8	423	5403	11444	1235
	0.131622	4	428	5397	11441	1236
	0.076700	6	425	5402	11432	1230
cortex_5_5	0.001395	12	234	4091	4253	1276
	0.067101	9	239	4089	4259	1275
	0.067101	9	239	4089	4259	1275
cortex_5_6	0.010296	13	285	3944	4273	1120
	0.004275	23	343	3880	4214	1139
	0.004275	23	343	3880	4214	1139

Table 6.19: Number of elements per *AR* interval for the cortex instances with RL 6 at the ROI.

Instance	Mesh *AR*	[0,0.2]	]0.2,0.4]	]0.4,0.6]	]0.6,0.8]	]0.8,1]
cortex_6_0	0.007474	38	1304	22016	47174	5461
	0.065498	27	1313	22026	47158	5453
	0.076512	22	1318	22027	47159	5448
cortex_6_1	0.012529	26	1254	18715	38142	5230
	0.044228	17	1269	18706	38138	5220
	0.005552	12	1289	18699	38136	5222
cortex_6_2	0.002414	19	1092	15600	19438	4768
	0.055750	15	1108	15607	19435	4755
	0.000718	33	1162	15583	19396	4742
cortex_6_3	0.009525	30	1149	21130	34452	5706
	0.044694	21	1161	21134	34435	5700
	0.078801	17	1163	21124	34437	5705
cortex_6_4	0.006345	23	1674	24383	86343	5846
	0.077851	11	1678	24377	86355	5840
	0.077851	11	1678	24377	86355	5840
cortex_6_5	0.009932	21	937	20673	29103	5829
	0.041366	14	947	20666	29111	5826
	0.062024	13	949	20663	29118	5827
cortex_6_6	0.007568	14	863	19682	21491	5968
	0.047833	14	872	19680	21475	5964
	0.089537	11	869	19683	21458	5963

Table 6.20: Number of elements per *AR* interval for the cortex instances with RL 7 at the ROI.

Instance	Mesh *AR*	[0,0.2]	]0.2,0.4]	]0.4,0.6]	]0.6,0.8]	]0.8,1]
cortex_7_0	0.001698	85	5318	94532	350055	24105
	0.056098	61	5343	94537	350034	24093
	0.034727	49	5487	94588	349881	24067
cortex_7_1	0.002471	74	5044	81201	280555	22532
	0.044293	55	5073	81191	280545	22534
	0.077843	39	5085	81183	280515	22525
cortex_7_2	0.009419	58	4231	66869	117723	20910
	0.071654	43	4211	66854	117720	20895
	0.080065	32	4218	66861	117721	20879
cortex_7_3	0.001324	60	4443	89402	231980	24457
	0.044725	43	4437	89382	231991	24451
	0.069860	37	4430	89366	232001	24449
cortex_7_4	0.016720	59	6563	104013	634423	25079
	0.047257	45	6559	104017	634426	25078
	0.080728	39	6566	104004	634442	25068
cortex_7_5	0.000003	14	3742	88134	176412	25067
	0.053094	11	3751	88139	176410	25065
	0.166019	8	3756	88128	176400	25069
cortex_7_6	0.011137	27	3620	83681	130623	24826
	0.044851	24	3610	83661	130619	24824
	0.065723	20	3624	83644	130606	24823

Table 6.21: Number of elements in AR's lower intervals for the liver instances with RL 5 at the ROI.

Instance	[0,0.04]	[0.04,0.08]	[0.08,0.12]	[0.12,0.16]	[0.16,0.2]
liver_5_0	1	0	2	0	2
	0	0	1	0	2
	0	0	0	0	2
liver_5_1	1	3	3	5	9
	0	3	3	4	9
	0	1	3	3	6
liver_5_2	2	1	5	6	7
	0	1	4	7	8
	0	0	1	7	6
liver_5_3	0	4	4	4	7
	0	1	4	4	7
	0	0	1	4	6
liver_5_4	2	1	3	3	6
	0	1	2	4	7
	0	0	0	4	5
liver_5_5	2	0	3	6	7
	0	0	2	4	7
	0	0	2	4	7

Table 6.22: Number of elements in AR's lower intervals for the liver instances with RL 6 at the ROI.

Instance	[0,0.04]	[0.04,0.08]	[0.08,0.12]	[0.12,0.16]	[0.16,0.2]
liver_6_0	2	3	10	4	8
	0	1	10	3	9
	0	0	2	2	8
liver_6_1	4	2	2	9	13
	1	0	2	7	12
	0	0	1	7	12
liver_6_2	0	3	8	6	9
	0	2	8	6	9
	0	0	2	5	8
liver_6_3	2	6	12	4	6
	0	3	12	4	6
	0	0	3	3	5
liver_6_4	0	1	5	5	10
	0	1	5	5	10
	0	1	3	5	10
liver_6_5	3	2	3	13	21
	0	1	0	12	15
	0	1	0	12	15

Table 6.23: Number of elements in AR's lower intervals for the liver instances with RL 7 at the ROI.

Instance	[0,0.04]	[0.04,0.08]	[0.08,0.12]	[0.12,0.16]	[0.16,0.2]
liver_7_0	3	7	12	11	14
	0	2	11	8	11
	0	0	2	5	11
liver_7_1	17	4	10	14	19
	1	2	11	18	12
	0	1	8	17	11
liver_7_2	3	13	13	16	18
	0	10	11	16	19
	5	2	6	18	37
liver_7_3	0	20	25	17	14
	0	15	22	17	13
	2	2	5	16	23
liver_7_4	3	10	11	11	16
	0	7	11	11	17
	3	1	7	12	23
liver_7_5	12	13	10	27	30
	0	4	7	18	31
	0	2	7	18	30

Table 6.24: Number of elements in AR's lower intervals for the cortex instances with RL 5 at the ROI.

Instance	[0,0.04]	[0.04,0.08]	[0.08,0.12]	[0.12,0.16]	[0.16,0.2]
cortex_5_0	6	3	3	6	5
	0	0	3	5	3
	0	0	0	2	3
cortex_5_1	6	2	1	3	4
	0	0	1	2	4
	0	0	0	2	4
cortex_5_2	1	0	1	5	3
	0	2	1	5	2
	0	0	2	4	3
cortex_5_3	0	0	3	5	5
	0	0	3	5	4
	0	1	2	11	11
cortex_5_4	2	0	0	4	2
	0	0	0	3	1
	0	1	1	3	1
cortex_5_5	3	3	1	4	1
	0	2	1	4	2
	0	2	1	4	2
cortex_5_6	1	0	3	3	6
	2	0	6	5	10
	2	0	6	5	10

Table 6.25: Number of elements in AR's lower intervals for the cortex instances with RL 6 at the ROI.

Instance	[0,0.04]	[0.04,0.08]	[0.08,0.12]	[0.12,0.16]	[0.16,0.2]
cortex_6_0	4	3	5	13	13
	0	2	3	12	10
	0	2	1	10	9
cortex_6_1	3	3	5	9	6
	0	1	4	7	5
	1	0	1	6	4
cortex_6_2	3	2	5	7	2
	0	1	4	8	2
	2	4	3	11	13
cortex_6_3	8	3	4	6	9
	0	3	3	7	8
	0	1	3	6	7
cortex_6_4	4	2	2	3	12
	0	1	0	2	8
	0	1	0	2	8
cortex_6_5	6	3	2	5	5
	0	2	2	3	7
	0	1	2	3	7
cortex_6_6	2	2	2	4	4
	0	2	2	5	5
	0	0	1	5	5

Table 6.26: Number of elements in AR's lower intervals for the cortex instances with RL 7 at the ROI.

Instance	[0,0.04]	[0.04,0.08]	[0.08,0.12]	[0.12,0.16]	[0.16,0.2]
cortex_7_0	6	22	17	21	19
	0	6	17	21	17
	1	2	6	22	18
cortex_7_1	7	11	17	20	19
	0	1	15	18	21
	0	1	6	17	15
cortex_7_2	6	2	12	19	19
	0	1	10	17	15
	0	0	5	16	11
cortex_7_3	9	7	11	12	21
	0	6	10	11	16
	0	4	8	11	14
cortex_7_4	5	9	7	13	25
	0	5	7	12	21
	0	0	9	10	20
cortex_7_5	2	1	2	0	9
	0	1	1	0	9
	0	0	0	0	8
cortex_7_6	5	4	6	6	6
	0	3	7	6	8
	0	1	5	6	8

6.2.5. Time in Proposed Algorithms Versions 1 and 2

Table 6.27: Time and Time per iteration comparison between proposed algorithm's versions 1 and 2 for the liver instances.

Instance	Time[a]	Time[b]	It[bd]	T/It[be]	Time[c]	It[cd]	T/It[ce]	Imp[cf]
liver_5_0	0.739	1.445	2	0.706	0.972	2	0.233	67.0%
liver_5_1	0.926	1.884	2	0.958	1.193	2	0.267	72.1%
liver_5_2	1.116	2.231	2	1.115	1.495	2	0.379	66.0%
liver_5_3	0.935	1.854	2	0.919	1.201	2	0.266	71.1%
liver_5_4	0.933	1.873	2	0.940	1.252	2	0.319	66.1%
liver_5_5	1.127	2.208	2	1.081	1.484	2	0.357	67.0%
liver_6_0	3.728	11.833	3	4.053	5.871	3	1.072	73.6%
liver_6_1	4.310	8.607	2	4.297	5.756	2	1.446	66.3%
liver_6_2	6.118	18.266	3	6.074	10.544	3	2.213	63.6%
liver_6_3	4.296	12.937	3	4.321	6.775	3	1.240	71.3%
liver_6_4	5.194	5.051	1		5.154	1		
liver_6_5	5.580	16.694	3	5.557	9.373	3	1.897	65.9%
liver_7_0	16.248	50.066	3	16.909	28.654	3	6.203	63.3%
liver_7_1	23.708	71.974	3	24.133	38.229	3	7.261	69.9%
liver_7_2	32.832	100.269	3	33.719	60.527	3	13.848	58.9%
liver_7_3	18.841	58.694	3	19.927	32.085	3	6.622	66.8%
liver_7_4	29.059	59.420	2	30.361	39.321	2	10.262	66.2%
liver_7_5	27.754	87.327	3	29.787	51.336	3	11.791	60.4%

[a] Original Algorithm
[b] Proposed Algorithm V1
[c] Proposed Algorithm V2
[d] Number of Iterations
[e] Time per Iteration
[f] Time per Iteration improvement

Table 6.28: Time and Time per iteration comparison between proposed algorithm's versions 1 and 2 for the cortex instances.

Instancia	Tiempo[a]	Tiempo[b]	It[bd]	T/It[be]	Tiempo[c]	It[cd]	T/It[ce]	Imp[cf]
cortex_5_0	2.312	4.415	2	2.103	2.917	2	0.605	71.2%
cortex_5_1	2.111	4.031	2	1.920	2.698	2	0.587	69.4%
cortex_5_2	1.776	3.387	2	1.611	2.247	2	0.471	70.8%
cortex_5_3	2.060	3.993	2	1.933	2.595	2	0.535	72.3%
cortex_5_4	2.819	5.492	2	2.673	3.637	2	0.818	69.4%
cortex_5_5	1.813	3.506	2	1.693	2.275	2	0.462	72.7%
cortex_5_6	1.800	8.702	5	1.726	3.637	5	0.459	73.4%
cortex_6_0	9.090	17.803	2	8.713	11.509	2	2.419	72.2%
cortex_6_1	7.600	22.303	3	7.352	11.694	3	2.047	72.2%
cortex_6_2	5.853	17.051	3	5.599	8.660	3	1.404	74.9%
cortex_6_3	6.736	13.336	2	6.600	8.600	2	1.864	71.8%
cortex_6_4	12.101	35.728	3	11.814	19.211	3	3.555	69.9%
cortex_6_5	6.873	13.430	2	6.557	8.503	2	1.630	75.1%
cortex_6_6	6.377	18.798	3	6.211	9.164	3	1.394	77.6%
cortex_7_0	45.342	89.020	2	43.678	58.917	2	13.575	68.9%
cortex_7_1	40.016	118.788	3	39.386	60.850	3	10.417	73.6%
cortex_7_2	24.771	48.355	2	23.584	30.766	2	5.995	74.6%
cortex_7_3	35.539	106.010	3	35.236	53.201	3	8.831	74.9%
cortex_7_4	60.464	183.344	3	61.440	102.460	3	20.998	65.8%
cortex_7_5	31.360	62.191	2	30.831	38.641	2	7.281	76.4%
cortex_7_6	28.186	83.139	3	27.477	40.078	3	5.946	78.4%

[a] Original Algorithm
[b] Proposed Algorithm V1
[c] Proposed Algorithm V2
[d] Number of Iterations
[e] Time per Iteration
[f] Time per Iteration improvement

Bibliography

[1] C. Lobos and J. Sepúlveda, "Patrones de transición para una malla generada por la técnica octree y simulación mediante elementos finitos," vol. 10, no. 1, pp. 167–176, 2012.

[2] E. González, *Diseño de patrones de transición entre zonas refinadas y gruesas de una malla de volumen de elementos mixtos*. Universidad Técnica Federico Santa María, Chile, 2013.

[3] E. González and C. Lobos, "A set of mixed-element transition patterns for adaptive 3d meshing," Tech. Rep. 2014/01, Departamento de Informática, UTFSM, October 2014.

[4] C. Lobos, "A set of mixed-elements patterns for domain boundary approximation in hexahedral meshes," *Studies in health technology and informatics*, vol. 184, pp. 268–272, 2013. Proceedings of MMVR20.

[5] T. G. R. Eymard and R. Herbin, "Finite volume methods," *Handbook of Numerical Analysis*, vol. VI, 2000. North-Holland, Amsterdam.

[6] C. Lobos and N. Hitschfeld, "3d noffset mixed-element mesh generator approach," in *In Proceedings of the 14th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, pp. 47–52, 2006.

[7] K. Bathe, *Finite Element Procedures in Engineering Analysis*. Prentice Hall Inc, New Jersey, 1982.

[8] R. Schneiders, R. Schindler, and F. Weiler, "Octree–based generation of hexahedral element meshes," in *In Proceedings of 5th International Meshing Roundtable*, pp. 205–215, 1996.

[9] Y. Zhang and C. Bajaj, "Adaptive and quality quadrilateral/hexahedral meshing from volumetric data," *Computer Methods in Applied Mechanics and Engineering*, vol. 195, pp. 942–960, February 2006.

[10] Y. Ito, A. Shih, and B. Soni, "Octree-based reasonable-quality hexahedral mesh generation using a new set of refinement templates," *International Journal for Numerical Methods in Engineering*, vol. 77, pp. 1809–1833, March 2009.

[11] N. Hitschfeld, "Generation of 3d mixed element meshes using a flexible refinement approach," *Engineering with Computers*, vol. 21, pp. 101–114, 2005.

[12] G. Nicolas and T. Fouquet, "Adaptive mesh refinement for conformal hexahedral meshes," *Finite Elements in Analysis and Design*, vol. 67, no. 0, pp. 1–12, 2013.

[13] D. Contreras and N. Hitschfeld-Kahler, "Generation of polyhedral delaunay meshes," *Procedia Engineering*, vol. 82, pp. 291 – 300, 2014. 23rd International Meshing Roundtable (IMR23).

[14] C. Lobos and E. González, "Mixed-element octree: a meshing technique toward fast and real-time simulations in biomedical applications," *International Journal for Numerical Methods in Biomedical Engineering*, vol. 31, no. 12, pp. 1–31, 2015.

[15] V. Parthasarathy, C. Graichen, and A. Hathaway, "A comparison of tetrahedron quality measures," *Finite Elements in Analysis and Design*, vol. 15, pp. 255–261, 1993.

[16] J. Shepherd and C. Johnson, "Hexahedral mesh generation for biomedical models in scirun," *Engineering with Computers*, vol. 25, pp. 97–114, 2009.

[17] S. Kelly, *Element shape testing. In: Ansys Theory Reference*, ch. 13. Ansys Inc., 1998.

[18] C. Lobos, "Towards a unified measurement of quality for mixed–elements," Tech. Rep. 2015/01, Departamento de Informática, UTFSM, April 2015.

[19] E. Daines Ostria, *Mejoramiento de la calidad de elementos mixtos en patrones de trancición para la técnica octree*. Universidad Técnica Federico Santa María, Chile, 2015.

[20] L. Freitag, "On combining laplacian and optimization-based mesh smoothing techniques," *Trends in Unstructured Mesh Generation*, vol. 220, pp. 37–44, Jun 1997.

[21] M. Bucki, C. Lobos, Y. Payan, and N. Hitschfeld, "Jacobian-based repair method for finite element meshes after registration," *Engineering with Computers*, vol. 27, no. 3, pp. 285–297, 2011.