Fast Quadtree/Octree adaptive meshing and re-meshing with linear mixed elements

Fabrice Jaillet, Claudio Lobos

Submitted to Engineering with Computers manuscript No.
(will be inserted by the editor)

Fast Quadtree/Octree adaptive meshing and re–meshing with linear mixed–elements

Fabrice JAILLET · Claudio LOBOS

Received: September 22, 2020 / Revised: January 29, 2021 / Accepted: February 1, 2021

Abstract In this paper, we will focus on adaptive meshing and re–meshing. We present an original approach, based on Quadtree and Octree, to construct the initial mesh and refine it using mixed–elements. We propose a fast algorithm using a determined set of Patterns to handle transitions between fine and coarse regions, and to closely approximate surface boundaries. An optimized structure for storing edges permits to efficiently manage neighbor information, dramatically reducing the balancing time. Results, both in 2D and 3D, exhibit the multiple possibilities for local refinements. Comparisons in terms of accuracy, number of elements in the resulting mesh, and computation time, clearly position our method as a competitive alternative to generate a conform adapted mesh, which can be used with standard numerical methods.

Keywords Mesh reconstruction · Adaptive refinement · Mixed–element mesh · Balanced Quadtree/Octree · Edge data structure

Article Highlights

- A fast meshing/re–meshing algorithm using deterministic patterns, suited for numerical simulation
- Mixed–elements are used to produce conform transitions and manage domain boundaries
- An optimized edge data structure for neighbor search is proposed to accelerate balancing of Quadtree and Octree meshes

F. Jaillet
Université de Lyon, IUT Lyon 1, CNRS, LIRIS, UMR5205, F-69622, Villeurbanne, France
E-mail: fabrice.jaillet@liris.cnrs.fr
ORCID: 0000-0002-7330-8116

C. Lobos
Departamento de Informática, Universidad Técnica Federico Santa María, Av. Vicuña Mackenna 3939, Zip 8940897, San Joaquín, Santiago, Chile
E-mail: clobos@inf.utfsm.cl
ORCID: 0000-0002-1793-1405

1 Introduction and context

Numerical simulation unquestionably brings answers in problems in physics, geology, engineering, medicine, and many other application fields. In many cases, the physical behavior of objects is complex, and far to be regular. Thus, the overall accuracy of the numerical solution if often lowered by the presence of singularities, interface between layers or heterogeneous materials, as well as the complexity of the interactions that may appear between objects like in hard or soft collisions, sliding, breaking or even cutting of objects. Once those regions have been identified, a current response is to locally adapt the numerical model: in changing the topology, refining the geometry, using high order elements, etc. Probably the most obvious and most popular technique is mesh adaptation. These considerations clearly show the need for fast mesh generation and further adaptive re–meshing, that underlies most of numerical approximation procedures that may be as follows:

1. Reconstruction of the geometrical input boundary of the considered structure(s).
2. Generation of the initial (coarse) discrete models, surface or volume.
3. Integration of the mechanical laws that govern the objects behavior; and simulation of the displacements and deformations.
4. Computation of error estimates, and identification of the discrete elements which have to be refined.
5. Adaptive mesh refinement, and back to step 3, until a global error criterion is satisfied.

Thus, in this article, we will focus on the meshing aspects of the simulation procedure, in an attempt to respect the trade-off between accuracy and computation time! As previously said, one key feature is adaptation of the underlying mesh. It allows concentrating the nodes in some regions regarding some criteria, which are most of the time based on geometrical approximation errors (*eg.* aspect ratio, scaled Jacobian) or simulation errors (like physically-based *a priori* or *a posteriori* error estimate [24]). High resolution elements will be constructed locally, in regions of interest (RoI) or where the numerical approximation is not regular; and coarser elements are generated in the rest of the domain. Mesh adaptation varies depending on the element type, which can be typically triangles or quadrilaterals in 2D, tetrahedra or hexahedra in 3D, or a combination of element types. These latter models, so called mixed–elements, will be particularly studied in this work.

The Quadtree [6] and the Octree [25] meshing techniques are an excellent starting point, since they are meant for mesh adaptation. We have noticed that some steps of those algorithms can be optimized to accelerate mesh generation. But most important, we are focusing on being able to consider an already generated mesh, refine it regarding simulation results, all this in an efficient way.

There exist many other meshing techniques or software, like *Triangle* [22] or *TetGen* [23], capable of refining previously generated simplex meshes; as

well as some modified standard numerical methods that are suited to non–conform Quadtree and Octree meshes. However the focus of the present work is to count with an efficient meshing algorithm capable of re–meshing, with mesh adaptation in RoI, and with a small set of mixed–elements (not general polygons or polyhedrons like possibly required by other methods) to be used with standard previously cited numerical methods.

2 State of the Art

In computer graphics, working with an arbitrary domain $\mathcal{D}$ requires a tessellation. This is commonly known as a mesh. Mesh cells, or elements, are discrete representations of the input domain, and are composed of vertices, edges and faces. Among many mesh generation methods, one well known algorithm was proposed in [6], where the data structure of Quadtree was introduced. The idea is to start with one or a small number of quadrants (typically squares) that encompasses the input domain $\mathcal{D}$. Then recursively split the quadrants into four new ones. If a new quadrant is completely outside $\mathcal{D}$, remove it from the mesh. If it is completely inside $\mathcal{D}$, stop refining it (split). Repeat these operations until a certain goal is achieved, *eg.* the error in approximating the boundary is acceptable. Fig. 1 shows an example to illustrate this algorithm (Pie and A `.poly` files from [3]).

This meshing technique was extended to 3D with the Octree [25]. Starting with some octants (typically cubes) that completely cover $\mathcal{D}$, refinement is achieved by splitting them into 8 new octants, and the remaining of the algorithm is merely identical to the 2D case.

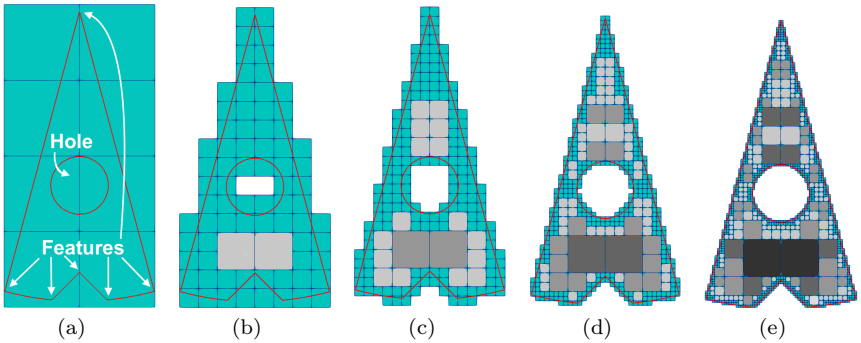


Fig. 1 The Quadtree algorithm. Note that from (a) to (b) some steps were skipped. (a)–(e) show how increasing the RL improves the approximation of the input domain described by a polyline (in red). Grey quadrants become darker as they get larger w.r.t. their neighbors

It is important to note that, at this stage, the mesh can barely be used by standard numerical methods, since the output is not conform. Concretely, if we want to use the mesh shown in Fig. 1(b), the two grey quadrilaterals would

have to be replaced by 7–node polygons to obtain a conform mesh. Only recently, some methods like [9] adapted numerical methods to work directly with hanging nodes. The above algorithm was extended to 3D in [20], however, this time, the Octree mesh must be balanced. This property (also sometimes referred as to 1–irregular) means that the difference in the Refinement Level (RL) according to its neighbors cannot be greater than one. To be clear, the RL of an octant O is the number of times the original octant was recursively split into 8 new octants to get to O. For instance in Fig. 1, (a) and (b) are balanced meshes, while all the other meshes are unbalanced. However, if the two darkest quadrants in Fig. 1(c) were refined once, this mesh would be balanced.

The first step in the balancing process consists in finding the neighbors of each quadrant. Then we check whether they are balanced w.r.t. the current quadrant. Two of the neighbors are easy to find following the Quadtree structure as they are direct siblings (*ie.* sharing the same parent root of the current quadrant), but finding the other two can be an expensive task since it should be performed using recursion over the Quadtree structure.

In the computer vision and image analysis field, this is a well studied problem. Starting with the work of [18, 19] where, following some rules over the data structure, the neighbors could be retrieved easily. Then, more recent works like [1] started to add more information to the quadrants, like its RL, so following a rule for numbering the quadrants and performing several bit computations, the neighbors could be retrieved in linear time. Further more, even faster results are achieved thanks to calculations performed at split time. Note that both Quadtree and Octree are considered in this last work [15].

It is important to highlight that all these last works solve the problem of finding the neighbors of each octant by considering the final state of a non balanced Quadtree/Octree mesh. By itself, this does not consider the problem of producing a balanced mesh. This latter problem is more complex due to possible propagation. If an unbalanced quadrant is refined, its neighbors should now be checked for balance, and if they are not, the refinement should propagate to them. Of course this procedure is very expensive and it should continue until the entire mesh is balanced. In this article, we will show that balancing can be performed much faster if we relay on updated neighbor information using an edge data structure and also if we change the paradigm of balancing at the end of the refining process. The main idea will be to regain balance immediately after its loss during the split operation.

Once the mesh is balanced, all the quadrants can be triangulated like in [25, 7] if standard numerical methods are meant to be used. The main drawback for this strategy is that the number of triangles in the final mesh will be much larger than the number of quadrants. For instance a quadrant that must cover a transition (between fine and coarse regions) may be replaced with up to 6 triangles.

An alternative to triangles is to produce an all–quadrilateral mesh [5]. Here two main issues must be tackled. The first one is how to manage the transitions. For example in 2D, if only one of the 4 quadrant edges is split (meaning it has

a hanging node), there is no template (or pattern) to manage this situation using only quadrilaterals. The only way to do this is to split the quadrant into 4 new ones and therefore continue with undesired refinement propagation. An alternative to this is to always split a quadrant into 9 new ones instead of 4. This will avoid unnecessary refinement propagation, however it will drastically increase the number of nodes in the mesh. Just for the case of one refined edge in the quadrant, the pattern that performs this transition without splitting other edges adds two nodes inside the quadrant. Therefore, it is clear that both alternatives, 4 and 9–refinement, will add internal nodes in order to maintain only quadrilaterals in the mesh. This may be avoided by using a combination of quadrilaterals and triangles to manage the transitions, leading to mixed–element meshes. The second issue of pure quadrilateral meshes is the representation of the boundary. Typically all the quadrants intersecting the boundary of $\mathcal{D}$ are removed from the mesh and then the void left behind is filled with new quadrilaterals that sometimes might have poor quality. Once more, several problematic configurations can be avoided using mixed–element meshes.

When it comes to 3D, there are several Octree based techniques that produce pure hexahedral meshes, however this necessitates to split an octant in 27 new ones [21, 26, 27, 12]. Moreover, it is important to mention that these works use a small, but incomplete, set of configurations to perform transitions between fine and coarse regions.

As an alternative, mixed–element meshes have been more developed, counting with a larger set of patterns to handle transitions [4, 16, 17]. Moreover, in a previous work [14], we were able to implement all mixed–element transition cases, avoiding any unnecessary refinement propagation (see [8] for implementation details).

Nowadays it is common to simulate a process, estimate the error and then re–mesh regarding this error, so the simulation can be more accurate at the next iteration. *Triangle* [22] and *TetGen* [23] are capable of re–meshing a pure triangular or tetrahedral mesh, however, to our knowledge, there are no Quadtree/Octree based re–meshing techniques for producing conform meshes. There are some works like [10, 11] that use balanced Quadtree and Octree meshes, but in the end, they need to modify the standard FEM so the simulation is performed over general polygons and polyhedrons, respectively.

Summarizing, the contributions of this article will be of various kinds:

- An efficient method based on Quadtree/Octree techniques for meshing a given domain $\mathcal{D}$. The use of mixed–elements will help to reduce the number of cells, with a good connectivity ratio between nodes, providing with a conform quadrilateral– or hexahedral–dominant mesh suitable to most numerical simulation methods.
- The possibility to define various Regions of Interest for adaptive meshing depending of the expected final result. Moreover, our method is also naturally well-suited for local re–meshing.

- The deterministic process leads to coherency and generality in the meshing process, while insuring to cover every configuration. The use of a limited number of *Transitions* in coarse to fine regions, and *Surface* Patterns guaranties to produce a quality mesh, handling boundaries and sharp features with accuracy.
- The proposition for an adapted structure to store neighbors information during the meshing allows to dramatically improve the efficiency in the search of neighbors in Quadtree/Octree. This could be of interest in other fields, like path planning, for example.

3 A Quadtree-based approach

Our first motivation to work with Quadtree was how easy it was to do mesh adaptation. You can rapidly increase the number of nodes and elements in a particular Region of Interest (RoI). In our method, we can combine them in many ways, depending on the final purpose of the mesh (see results in section 6 for some concrete examples). When a quadrant is affected by several RoIs, the greater RL among them is applied to it. They are as follows:

- Refine the quadrants intersecting the boundary of $\mathcal{D}$ (this is referred to as Surface refinement).
- Define a global RL for $\mathcal{D}$ (this is referred to as All refinement)
- Refine a particular region of $\mathcal{D}$ (this is referred to as Region refinement).

The target will be a mixed–element mesh for a given $\mathcal{D}$, which is described by a set $\mathcal{P}$ of closed input Polylines. Each Polyline is a collection of segments that must fulfill two constraints: it must be unfolded (no self–intersection), and the normal of the edges must be pointing outward. Thus $\mathcal{D}$ may contain holes or disconnected parts (Fig. 1(a)).

The algorithm will use $\mathcal{P}$ for two purposes: to find out if a point is inside or outside $\mathcal{D}$, and to project a point onto $\mathcal{P}$.

3.1 Splitting and balancing

In this section, we will describe our refinement algorithm. The main loop will iterate until the max RL (among defined RoIs) is reached. At each iteration, we maintain a list of *candidates* quadrants for refinement, and a list with the already *processed* quadrants. When generating a mesh from scratch, all the quadrants are *candidates* and none is *processed*.

On the one hand, if a *candidate* needs to be refined regarding at least one RoI, it will be split into four new quadrants, and each one will be a *candidate* for the next refinement iteration. On the other hand, it will be tagged as a *processed* quadrant. An already processed quadrant may only be further refined for balancing reasons.

Each time a quadrant is split, neighbors are checked for balancing using the edge data structure defined in section 3.2. If not balanced, they are added to a list. This list will be treated only when all *candidates* have been analyzed.

Now we will loop until the list of *to balance* quadrants is empty. Each time a quadrant of this list is split it may add unbalanced neighbors to the list.

Finally, when balance is achieved throughout the mesh, we will be ready to proceed with another iteration of *candidates* until the max RL is reached.

Algorithm 1: Generate a balanced Quadtree mesh

```
   Result: A balanced mesh
   List candidates: all quadrants;
   List processed: void, to_balance: void;
1  while Refinement Level is not reached do
       foreach Refinement Region (RR) do
           foreach  Quadrant (q) in candidates do
               if q do not need refinement by RR then
                   candidates.remove(q);
                   processed.add(q);
               end
           end
       end
2      foreach q in candidates do
           Refine q;
           foreach son (s) of q do
               candidates.add(s);
           end
           foreach neighbor (n) of q do
               if n is not balanced with q then
                   to_balance.add(n);
               end
           end
       end
3      while to_balance not empty do
           q = to_balance.pop();
           if q is not balanced then
               Refine q;
           end
           foreach neighbor (n) of q do
               if n is not balanced with q then
                   to_balance.add(n);
               end
           end
       end
   end
```

In Algorithm 1 step 3, it is important to check if a quadrant *to balance* was not already balanced. For instance, while balancing, a neighbor could need refinement. However it could have been already inserted in the list and processed previously in the same iteration. If not taken into consideration, this could lead to duplicate instances of the same split quadrant.

3.2 Edge data structure for balancing quadrants

The basic operation of a quadrant is to split itself into four new ones. When doing so, it might cause unbalance with respect to its neighbors. There are two strategies to re–gain balance: solve all the problems at the very end of the refinement process; or solve them while refining. As we will show later, this last strategy is much more efficient.

The main key is to store neighbor information at the edge data structure. First of all, each quadrant will have a unique *id*. The information allocated in each edge will be an array (`info`) of 3 integers: the middle node index (if any) and the index of the two quadrants that share it. By default, the mid–node is set to 0. The first node of a mesh will always be a corner, therefore, if the mid–node is 0 it means it has not been split yet into two sub–edges. The index of the two neighbor quadrants is set to the maximum value an integer can hold. This means it has no neighbors. When updating the information, we follow a simple rule to identify where to store it. For a quadrant Q, we will store its index in position 1 of the `info` array for its bottom and right edge and in position 2 for its top and left edge.

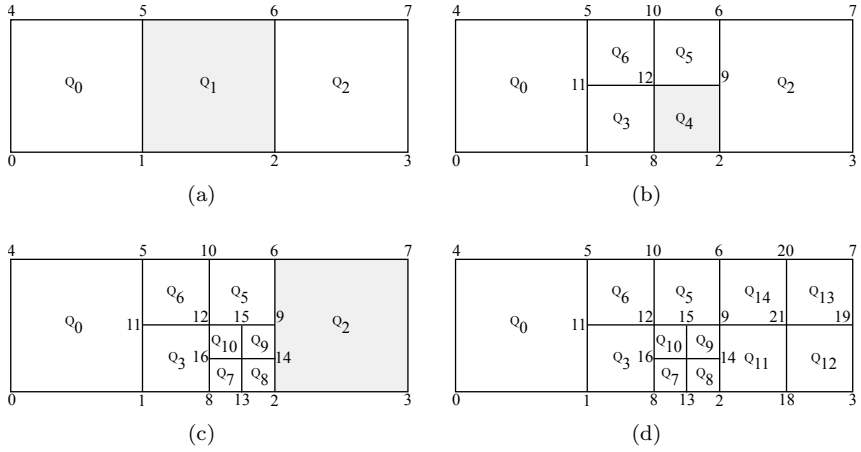


Fig. 2 Balancing a mesh while refining. The quadrant to be modified in next iteration is shown in grey. (a) initial state, (b) refining quadrant Q_1, (c) refining quadrant Q_4 produces a non balanced mesh and (d) quadrant Q_2 is refined due to balancing

Let us consider the example of Fig. 2(a) with 3 quadrants and also consider a finer mesh is wanted next to node 2. The information of edge (2,6) is the following: $[0, Q_1, Q_2]$ because it is not split yet (the first 0) and the index of neighbors are Q_1 and Q_2 following the order for the edge data structure mentioned previously. In the case on Fig. 2(b), the quadrant Q_1 at the middle was split and replaced by four new ones, namely Q_3, Q_4, Q_5 and Q_6. In the process, the edge (2,6) has been split by inserting a midpoint (9), and now we

update its information with $[9, Q_1, Q_2]$, despite Q_1 no longer exists. In fact, there is no need to update neighbor information for split edges, as this data is now carried out by the new edges. Thus, the important point is that all edges of the most refined quadrants are embedding updated information. The information of edge (2,9) is $[0, Q_4, Q_2]$, and for edge (6,9) it is $[0, Q_5, Q_2]$. At this point, we check whether neighbor quadrants are balanced. The Refinement Level (RL) for original quadrants is 0, and 1 for Q_3 to Q_6. If the RL difference between two neighbors is greater than one, then the less refined quadrant is added to a list for subsequent balance. The same updating process is of course repeated for all edges of the initial quadrant Q_1, respectively (5,6), (1,5) and (1,2).

In Fig. 2(c), Q_4 is now refined producing quadrants Q_7 to Q_{10}. At this point we are done with some edges like (8,16) with $[0, Q_3, Q_7]$ or (9,15) with $[0, Q_5, Q_9]$; however edges (2,14) with $[0, Q_8, Q_2]$ and (9,14) with $[0, Q_9, Q_2]$ present unbalance. For this reason Q_2 is added to a list for later refinement.

It is important to highlight how edges are updated. In Fig. 2(d) quadrant Q_2 was finally refined due to balancing and now edge (6,9) holds $[0, Q_5, Q_{14}]$ and edge (2,9) holds $[14, Q_4, Q_{11}]$. Q_4 is no longer existing, but sub-edges (2,14) with $[0, Q_8, Q_{11}]$ and (9,14) with $[0, Q_9, Q_{11}]$ are adequately updated, remembering that only the most refined edges need to support up-to-date information on neighbors.

4 Complete mesh generation

In the previous section, we showed how to generate a balanced set of quadrants, with optimized data structure. We will detail the rest of the process to provide a quality mixed–element mesh, fully partitioning the input domain $\mathcal{D}$ while respecting its boundaries.

4.1 Meshing algorithm

The main flow is shown in Algorithm 2, where step 1 corresponds to Algorithm 1 presented in section 3.1. As an illustration of the process, we will now detail the remaining steps through a complete example. Target $\mathcal{D}$ will be a trapezium with its respective polyline $\mathcal{P}$. Fig. 3 illustrates the refinement and balancing process for $\mathcal{D}$. The balance is lost in Fig. 3(e) and immediately recovered by our algorithm in 3(f). The process could continue by refining all the quadrants intersecting $\mathcal{P}$ and balancing them, until the desired RL is reached. This RL is application dependent, and is let to the appreciation of the user.

Transition Patterns. The second step consists in applying what we call *Transition Patterns* to all of the quadrants having at least a mid–node in an edge, so we can finally obtain a congruent mesh. These patterns cover all possible combinations (Fig. 4). Following with the trapezium example, their use is shown in Fig. 5(a), with grey triangles and white quadrilaterals.

Algorithm 2: Generation process

Result: A balanced, conform, mixed–element mesh respecting boundaries
1 Create a balanced Quadtree;
2 Apply Transition Patterns;
3 Detect Features Quadrants;
Project Close to Boundary;
Remove on Surface;
4 Shrink to Boundary;
5 Apply Surface Patterns;

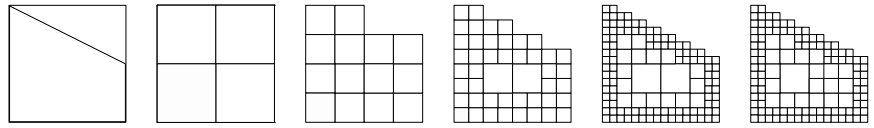


Fig. 3 Algorithm example for balancing: (a) A trapezium as input $\mathcal{P}$ and the original quadrant at $RL = 0$, (b)-(d) the quadrants intersecting the border are refined once and they are balanced, (e) boundary quadrants are refined and the balance is lost, and (f) balance is recovered for the internal quadrant at the middle-right position of the mesh.

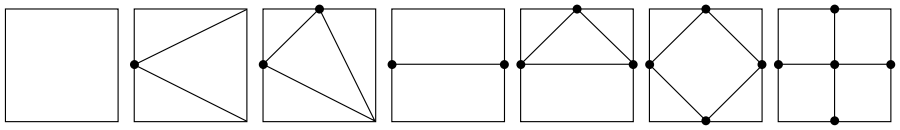


Fig. 4 Transition patterns for quadrants regarding the number of mid–nodes in their edges. Note that the last case is only useful for face subdivision in 3D.

Managing boundaries. At step 3, we start to move nodes lying on the surface of $\mathcal{P}$. First, we manage all the quadrants that "contain" a feature of $\mathcal{P}$. When the angle of two consecutive segments is not too "flat", then the common node is considered as a feature. In all our examples, we have set the interval to $]175°, 185°[$. Once a quadrant is detected as holding a feature, its closest node must match this feature. Secondly, internal nodes that are "close" to $\mathcal{P}$ are moved onto it. In our experience we obtain better results when the maximum displacement distance for a node is 30% of the shortest quadrant diagonal sharing the node. To illustrate this, Fig. 6(a) shows an example of 4 octants A, B, C, D and $\mathcal{P}$, in red. 2 features (red nodes) are covered by nodes 0 and 1 when analyzing octants A and B, respectively. Node 4 is projected onto $\mathcal{P}$ when analyzing octant C. The main goal for this is to decrease the chances of having flat elements at this stage.

Following with our trapezium example, the displacement to features is performed at its corners. This can be seen in Fig. 5(b) as well as the displacement of "close to the boundary" nodes at the trapezium top border. At the end of step 3, we remove the elements that now are completely outside $\mathcal{D}$. This can be observed for some quadrants at the diagonal in Fig. 5(c).

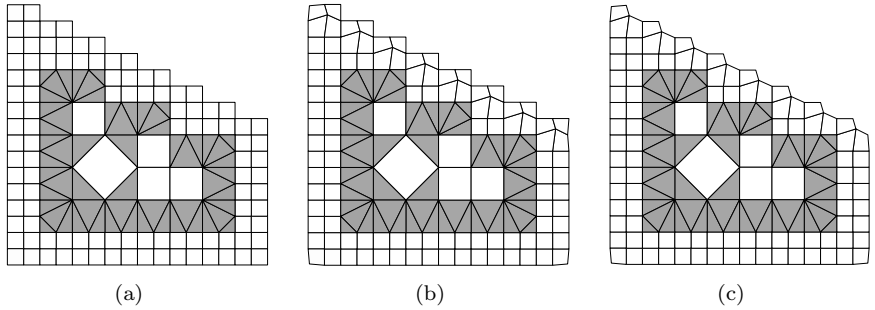


Fig. 5 Algorithm example for: (a) applying transition patterns: darker elements are triangles, (b) node displacement by proximity to the input boundary and (c) completely outside elements are removed.

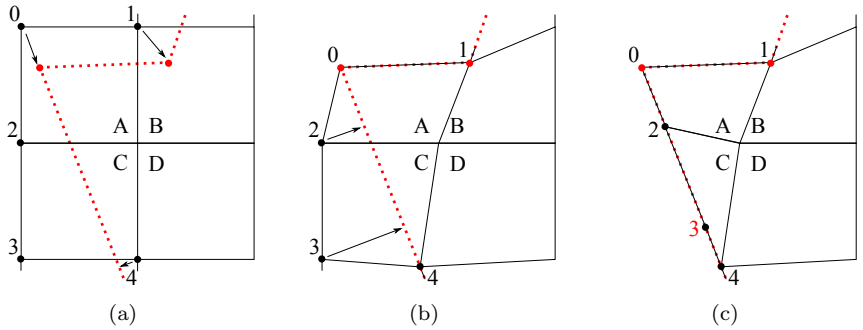


Fig. 6 Surface representation: $\mathcal{P}$ is the dashed line and red nodes are identified features. 6(a) before projection state. 6(b) nodes 0 and 1 are projected to features, while node 4 is an inside node "close to the boundary", thus it is projected. 6(c) outside nodes 2 and 3 are projected onto $\mathcal{P}$ and then by the use of surface patterns, node 3 is removed.

Shrink to boundary. Step 4 will force all the remaining nodes that are still outside $\mathcal{P}$ to project onto it. To illustrate this, Fig. 6(c) shows the projection of nodes 2 and 3 when analyzing octants A and C. The result can be seen in Fig. 7(a) for the trapezium. Even if this mesh is an accurate representation of $\mathcal{D}$, it presents poor quality quadrilaterals at the trapezium top border. For this reason, we use what we call *Surface Patterns* that will replace poor quality or invalid quadrilaterals by triangles as it can be seen in Fig. 7(b).

Surface Patterns. To determine if a quadrant must be removed, replaced by a triangle or just stay as it is, the Surface Pattern process (step 5) first considers the configuration of inside nodes and, secondly, the angles of outside nodes. Fig. 8 shows the six possible configurations of inside nodes (marked in black). Here it is important to note that nodes lying exactly on $\mathcal{P}$ (even if they were forced to be projected onto it) are considered as outside nodes.

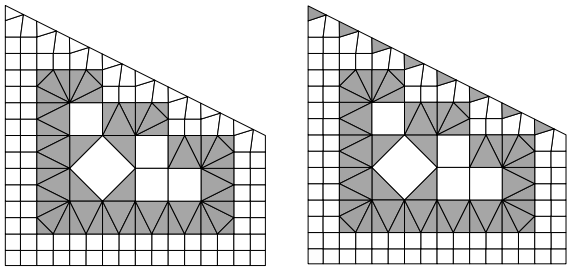


Fig. 7 Algorithm example for: (a) shrink outside nodes onto the boundary and (b) the use of *Surface Patterns* to replace some quadrilaterals with triangles to enhance quality.

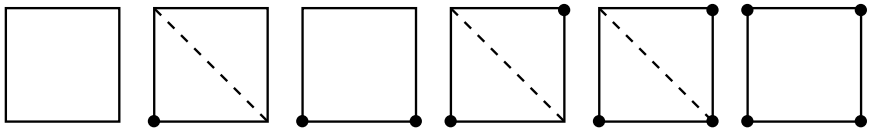


Fig. 8 Surface patterns. Black nodes are considered inside $\mathcal{D}$.

Once the pattern is figured out regarding inside nodes, we analyze the angles at the rest of the nodes. An angle $< 150°$ will make the node to be considered as an inside node. An angle $\in]175°, 185°[$ would make the quadrilateral of poor quality or even tangled (inverted quadrilateral). For this reason the quadrilateral is replaced by a triangle without considering this node. For instance, in Fig. 6(c) we see that for quadrant C there is a bad angle at node 3, therefore pattern 8(b) is used causing to replace the flatted quadrilateral by a triangle. Now, for quadrant A, we do not obtain the same result because the angle for nodes 0 and 1 is less than 150°, meaning they are labeled as inside nodes. Assuming the angle at node 2 is also $< 150°$ this will remain as a quadrilateral, otherwise node 2 will be considered outside and pattern of Fig. 8(e) will split quadrant A into two triangles. The same will occur for quadrant B regarding the angle at node 1. However for quadrant D, as the angle at node 4 is $< 150°$, it will remain as a quadrilateral.

It is possible to find even more complex situation. Let us consider the top left corner of the trapezium shown in Fig. 7(a). This quadrant has all its nodes marked as outside nodes. Moreover, the top right node of this quadrant will have an angle of 180° when projected, thus, the underlying quadrilateral is replaced by a triangle as shown in Fig. 7(b). As for the rest of quadrilaterals replaced with triangles at trapezium top border, we see that they have exactly one node inside and the two nodes directly connected to it have a good angle. However the opposite node has an angle of 180°, meaning that these quadrilaterals will be replaced by triangles as in Fig. 8(b).

4.2 Re–meshing an existing mesh

One of the main goals of this work was to count with a meshing technique well adapted for re–meshing regarding simulation errors. In other words, generate a mesh, simulate and loop until the error goes below a given threshold. Classically, the simulation is performed using a Finite Element or Finite Volume Method, but alternatives as Mass–Spring or even Mass–Tensor techniques could be used, as soon as they are able to provide with an error estimate for each element. This information will be used by the re–meshing process to add details where the solution of the simulation is not satisfying, for example where the deformation of the objects is not realistic. Locality of the mesh refinement is one of the most important properties. The goal is to avoid unnecessary refinement propagation in areas where the desired quality of the numerical solution is already reached.

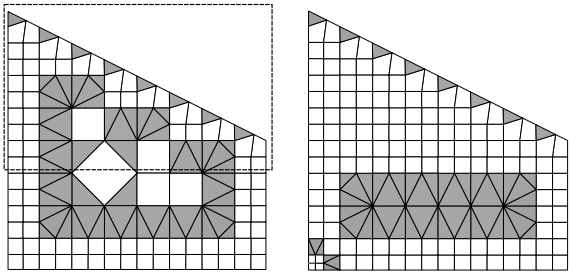
When we generate a mesh for a given domain $\mathcal{D}$, given a Polyline $\mathcal{P}$, and refine it following provided Regions of Interest (RoIs), two outputs are produced: the mesh itself in any standard format, *eg.* VTK, and also an *ad hoc* file containing all the information to re–start the process from a balanced mesh. An example of such a file can be found at Appendix A. For the trapezium example, this would be to start from Fig. 3(f). When re–meshing, one typical input is an element index list (depending on errors issued from a simulation, for example), and not a quadrant index list. Thanks to the coupled information, we are able to associate each quadrant to its corresponding sub–elements. Before refining a quadrant, its previous sub–elements are erased. Also, thanks to the edge data structure (section 3.2), it will be easy to refine the neighbors as well if balance is lost. This is presented in Algorithm 3.

Once the list is processed and the mesh is balanced, we continue with the refinement required by other RoIs following the steps in Algorithm 2. An example of this can be seen in Fig. 9, where two refinements were employed: a list of elements and a region, which is defined by another polyline intersecting $\mathcal{D}$. Only element 0 was provided in the list, located in the quadrant at bottom left corner of the trapezium. Each quadrant associated to the elements in the list will be refined only once. The region of refinement can be seen in Fig. 9(a) as a dashed line. In this case, the RL is set to 4 in the region, equal to the original RL of boundary elements. The final output where both refinement types are achieved is shown in Fig. 9(b). It is important to note that this new mesh will also automatically save its information for further re–meshing, so if we continue needing refinement after a new step of simulation we can start from this point.

Once this refining step is done, a balanced Quadtree mesh is obtained. Steps 2 to 5 of Algorithm 2 are now processed as for the initial mesh generation.

Algorithm 3: List refinement

```
Result: A balanced refined quadrant mesh
List to_balance: void;
List list: provided quadrants to be refined;
/* Refine listed quadrants */
foreach Quadrant (q) in the list do
    Refine q;
    foreach neighbor (n) of q do
        if n is not balanced with q then
            to_balance.add(n);
        end
    end
end
/* Refine unbalanced neighbors */
while to_balance not empty do
    q = to_balance.pop();
    if q is not balanced then
        Refine q;
    end
    foreach neighbor (n) of q do
        if n is not balanced with q then
            to_balance.add(n);
        end
    end
end
```



(a) (b)

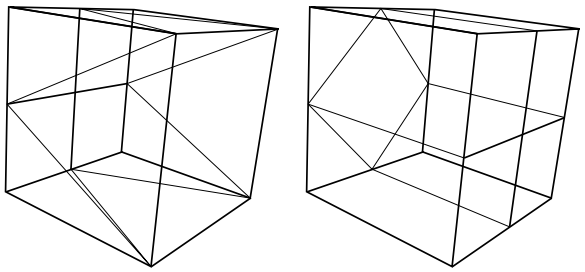
Fig. 9 Algorithm example for re–meshing: (a) starting mesh with a region for refinement (segmented line) as well as refinement of an element at the bottom left corner; and (b) the output mesh after refinement.

5 Extension to 3D

In previous section, we presented our optimized algorithm based on Quadtree subdivision to generate a 2D quality mesh. Same ideas could be extended to Octree and octants. In the following, we will emphasize the main adaptation that are required for the extension to 3D.

5.1 Meshing algorithm in 3D

The meshing algorithm in 3D is the same as Algorithm 2 in Section 3.1, however, two main considerations must be explained. First, the number of transitions patterns is much larger. Each of the 6 square faces of an octant can take one of the 7 configurations shown in Fig. 4, meaning that each one of the 12 edges can be subdivided, or not. Thus, all possible combinations have been automatically computed, and after removing equivalent cases due to rotations, there remains 325 different patterns. Each pattern was individually filled by using our 4 basic element types (tetrahedron, hexahedron, prism and pyramid), and the complete tessellation list that can be found in [8]. Fig. 10(a) shows the face patterns over an octant where the left face was split due to refinement, leading to a tessellate using 5 pyramids and 4 tetrahedra. However, there is no solution for the pattern shown in Fig. 10(b). In this case, after inserting the octant center node, we can build 6 pyramids using octant faces and this node as apex. From there it is easy to build the remaining tessellation, leading to 14 pyramids and 4 tetrahedra. There are 65 patterns needing the octant center node insertion.



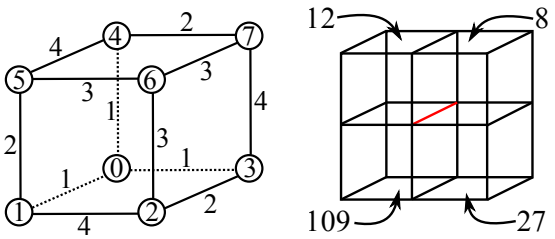
(a) Pattern #217 (b) Pattern #296

Fig. 10 Examples of 3D transition patterns (as in [8]): (a) pattern where the tessellation can be made without inserting any additional node and (b) pattern where octant center node is needed.

The second main consideration is regarding what we call *Surface patterns*, as it was shown in Fig. 8 and explained at the end of section 3.1 for 2D. This is not yet resolved in our 3D algorithm. The original version of the 3D meshing technique was introduced in [14] without the main edge structure optimization introduced in this article. It was mainly conceived for meshing anatomical structures, where sharp features are rare. Some efforts have been made as in [2] to improve the boundary handling. As in 2D, inner nodes close to the boundary are projected onto it to prevent flat elements. Despite of this, some particular sharp features in the input domain will be poorly preserved by our 3D version algorithm, which was not the case in 2D.

5.2 Edge data structure for balancing octants

The edge data structure for 3D is conceptually the same as the 2D version introduced in section 3.2. The main difference is that a 3D edge might be shared by up to 4 octants, therefore our `info` array now holds 5 integers: the mid–node index (if any) and the four eventual neighbors. Using the position in the `info` array we are able to identify where the octant is located with respect to this particular edge. Fig. 11(a) shows an octant with its node indexes. Over each of its edges appears the position at which this octant will be stored in the info array of the respective edge. For instance, if we take the edge formed by nodes (2,3), the index of this octant will be stored at position 2 of the `info` array. This can be seen in Fig. 11(b), where for the red edge, its `info` array holds index 12 in position 2.



(a) (b)

Fig. 11 Octree edge structure: (a) relative position storing for this octant regarding each of its edges and (b) an example of how octant indexes are stored for the red edge `info` = [0,8,12,109,27].

As in the 2D case, this data structure allows to quickly access neighbors for balance checking. In a eventual balance loss, neighboring octants will be added to a list for refinement at the end of the current refinement iteration.

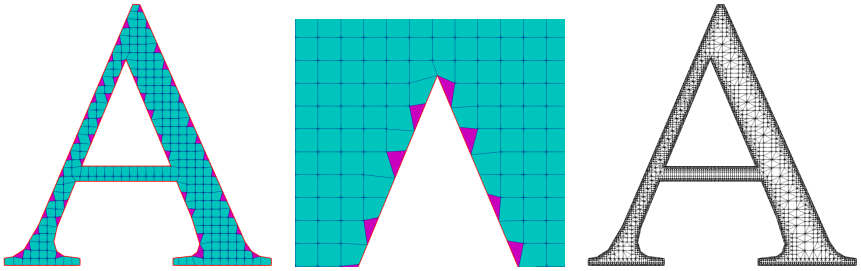
6 Results

In this section, we will present some representative results, first in 2D, and then in 3D. We will exhibit the capabilities of our method, as well as showing its efficiency in terms of computation time and number of generated elements. Some comparisons will also be performed with popular other techniques.

6.1 2D mesh generation

In Fig. 12(a), a polyline $\mathcal{P}$ representing the letter A is meshed using our algorithm at RL 5 on the whole domain $\mathcal{D}$. Result is a quadrilateral–dominant

mesh (in green–blue), where the different features and boundaries have been respected with use of *Surface Patterns* including a reduced number of triangular elements (purple). Fig. 12(b) presents a closer look, this time at *RL* 10.



(a) *RL* 5, all quadrants (b) zoom: *RL* 10, all quadrants (c) *RL* 12, on surface

Fig. 12 Meshing of the A shape: (a) uniform refinement and boundary handling, with green–blue quadrilaterals and purple triangles, (b) zoom on the top part of the inner hole for a more refined mesh, and (c) refining only the surface.

In Fig. 13, we compare our results with the ones obtained using *Triangle* [22]. As expected, for an equivalent precision, both methods produce a similar number of vertices, but mixed–meshes are containing less elements, what could be of great importance at the moment to use this kind of mesh for numerical simulation. Note that for computation on the GPU, reducing the vertex connectivity is also a great advantage. Though, this gain in size of the geometry is counter-balanced by the fact that *Triangle* is faster than our method, even if the optimized version tends to be quite competitive (Fig. 14).

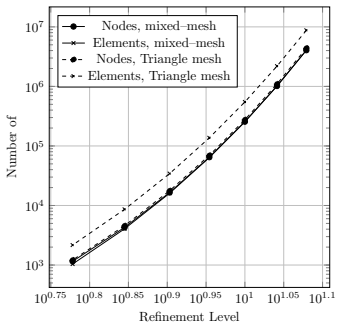


Fig. 13 Number of nodes and elements while increasing a uniform *RL* from 1 to 12 for the A example. Comparison between our method and Triangle

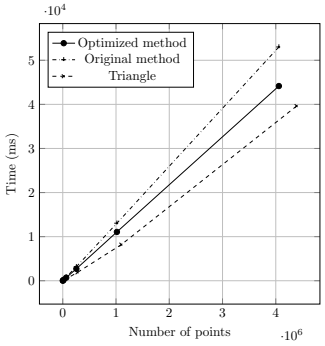


Fig. 14 Comparison times, between original and optimized versions of our algorithm and *Triangle*, for the A shape, with uniform refinement.

The next example exhibit even better the improvement in time brought by the optimized version. *RL* are varying in a [1-12] range, but are constrained

only at the boundary. This kind of tool is very useful to reduce the number of elements in inner parts. This is illustrated by Fig. 15, when inside element are refined at a level just sufficient to provide a balanced, conform mesh, keeping high resolution at the interface. For example, at RL 12 (Fig. 12(c)), the mesh contains 88102 vertices for almost 500000 cells, instead of 4 millions of vertices and 20 millions of cells for a uniform partition of $\mathcal{D}$. In Fig. 16, and Fig. 17 for a detailed view, we extracted times for each step of the algorithm. We considered the same mesh as in Fig 12(c). For both algorithms, times are similar, except for the balancing step that is dramatically decreased thanks to the optimized edge data structure, leading to a cut off time of 80%, reducing the whole process by more than 25%. Notice how the complete optimized process can be decomposed more or less equally in 4 parts: 1) the mesh refinement, 2) the creation of the balanced conform mesh, 3) the creation and saving of the *ad hoc* structure used for refinement (can be skipped if no forward refinement is required) and 4) the boundary processing.

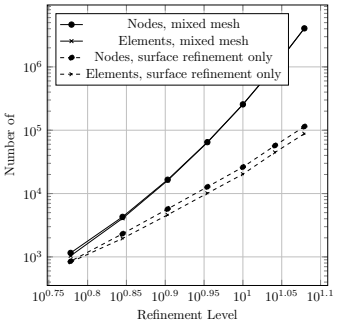


Fig. 15 Number of nodes and elements while increasing a RL from 1 to 12 for the A example. Comparison between uniform refinement, or restricted to the surface elements.

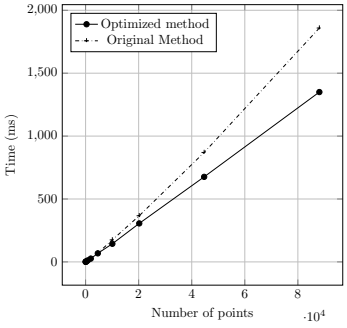


Fig. 16 Comparison times, between original and optimized versions of our algorithm, for the A shape. Refinement restricted to the surface elements.

6.2 Mesh refinement

The refinement operation could be observed on box with hole example (Fig 18), which original mesh is composed of 60 quadrants. The bottom left element has been subdivided to provide a new mesh, that will be subdivided in turn. It is worth to note how the refinement process (section 4.2) naturally preserves the balancing and quality of the elements by adding transition patterns. After 20 iterations, the final mesh contains 324 elements for 280 vertices. The refinement remains localized in the lower left area, in accordance with the desired behavior.

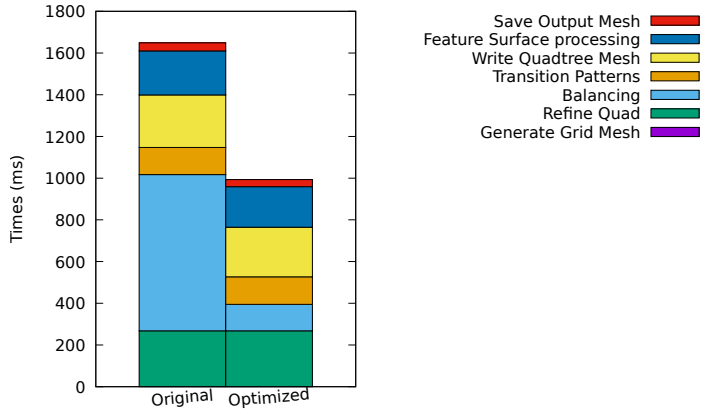
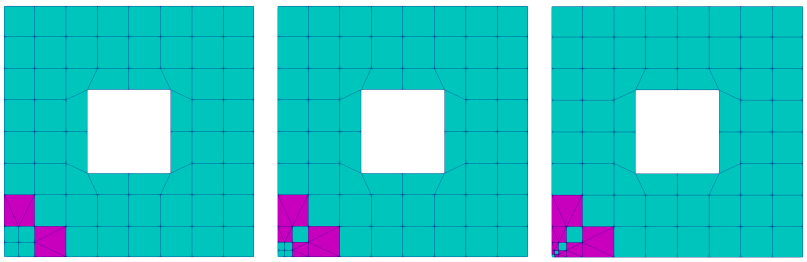
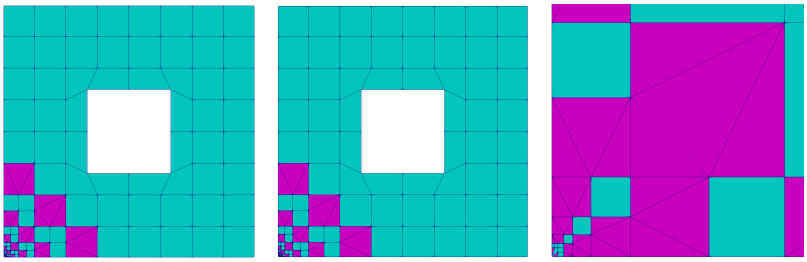


Fig. 17 Detailed times, comparison between original version of our algorithm, and new one using optimized structure for neighborhood handling, for the A example.



(a) iter. 01, 67 elements (b) iter. 02, 74 elements (c) iter. 04, 83 elements



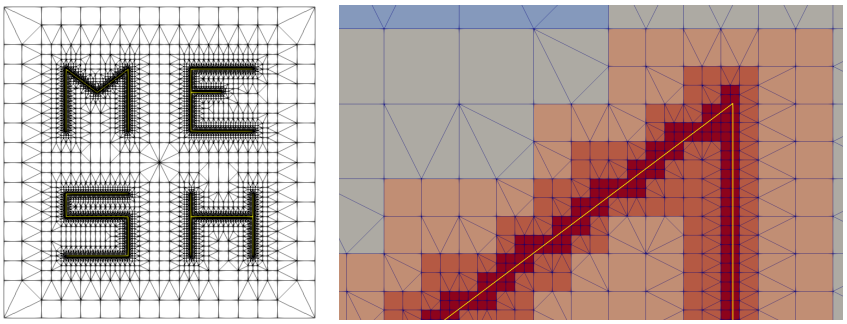
(d) iter. 06, 135 elements (e) iter. 21, 324 elements (f) iter. 21 (zoom)

Fig. 18 Recursive refinement of the bottom left element, at different *RLs* (with green–blue quadrilaterals and purple triangles)

6.3 Additional features

Another interesting feature that can easily be handled by our algorithm is the ability to define local areas that require more details. They can be described as closed boundaries to define Regions of Interest, as seen previously, or by adding constraints on the meshing process. This can be any user-specific functions or

list of inner polylines, as in Fig 19. Note that in this case, the line segments will not be integrated as boundaries, but will serve as condition to subdivide a quadrant when it intersects a given curve or segment (Fig. 19(b)). These features are very easily integrated into the whole generation process, thanks to the *Visitor* design pattern. Note how inner structure are not considered as other boundaries, thus they will not be represented by cell edges.



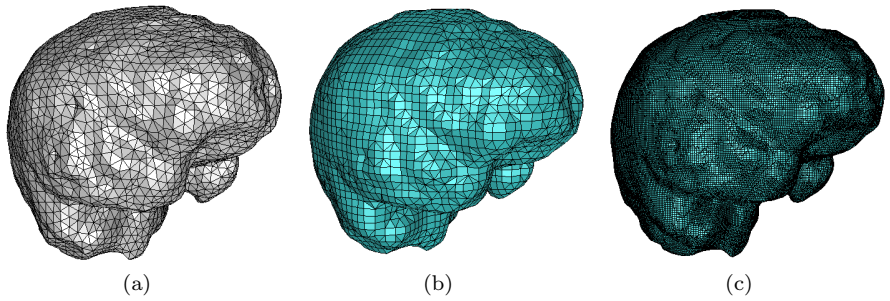
(a) (b)

Fig. 19 Refining around internal structures, here represented by the "MESH" red letters, with zoom on the upper part of the M letter, showing RLs in a color scale

6.4 3D mesh generation

In order to illustrate our 3D results, a domain representing a *brain* will be meshed with 3 techniques: the Octree with and without the edge data structure introduced in this article, and also *TetGen* [23]. Fig. 20(a) shows the input surface mesh employed in these examples. It has 3152 nodes and 6300 triangles. Fig. 20(b) shows the output mesh with a surface Refinement Level (RL) 5, meaning that all the octants intersecting the surface (input boundary) were recursively split 5 times into 8 new octants. The rest of the intern octants were left as coarse as possible. Fig. 20(c) shows a RL 7. It is important to highlight that the Octree with and without using the new edge data structure produce the same output. Also, to understand the impact of increasing the refinement at the surface, Fig. 21 shows a log–log plot regarding the number of nodes and elements in the output mesh while increasing the refinement level from 4 to 9.

Fig. 22 shows a plot where the number of nodes at each RL (from 4 to 9) are related to the time to produce the mesh whether using or not the improving edge data structure. It is easy to see how the greater the RL is, the greater the gain in time. For RL 9, the mesh contains more than 3.5 millions of nodes, the edge optimization will save almost 30% of time, from 23.8 to 15.1 minutes, to generate this fine quality mesh.



(a) (b) (c)

Fig. 20 (a) The input mesh and (b)-(c) output meshes for *RL* 5 and 7 at the surface.

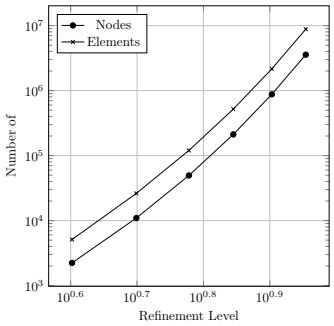


Fig. 21 Number of nodes and elements while increasing the refinement level.

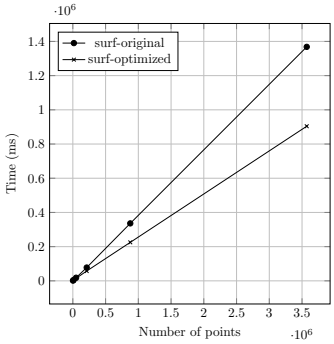


Fig. 22 Time comparison of the Original and Optimized version for Octree.

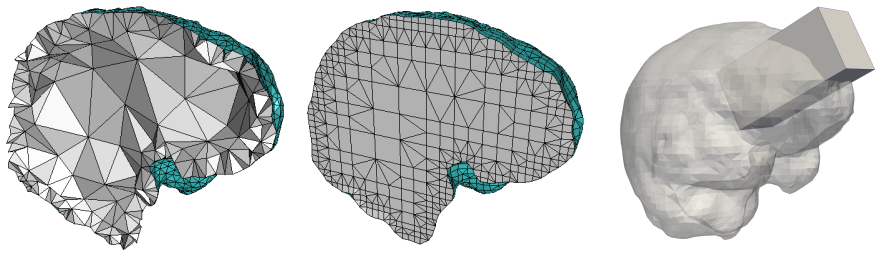
Now let us compare our meshing algorithm using the new edge data structure with *TetGen* (version 1.5.1). The surface mesh shown in Fig. 20(a) is used as input. We set the switches `-pqk`: *a quality piecewise linear complex mesh*, to generate the coarsest mesh as possible. We also asked to write the output to VTK format, which purpose was to compare similar tasks (including file writing). Fig. 23(a) shows a sagittal cut for the output mesh using *TetGen*.

For our algorithm, we set *RL* to 5 to produce a similar output, at least in representing the domain with an equivalent node quantity. Indeed, the surface of this output mesh counts with 3372 nodes, close to the 3152 nodes of the input mesh used by *TetGen*.

Now, starting from these outputs we intended to refine a given Region of Interest (RoI) that is defined by an hexahedron intersecting the input domain (Fig. 23(c)). We extracted the list of all elements having at least one node inside the RoI, and sent this list to both algorithms for refinement.

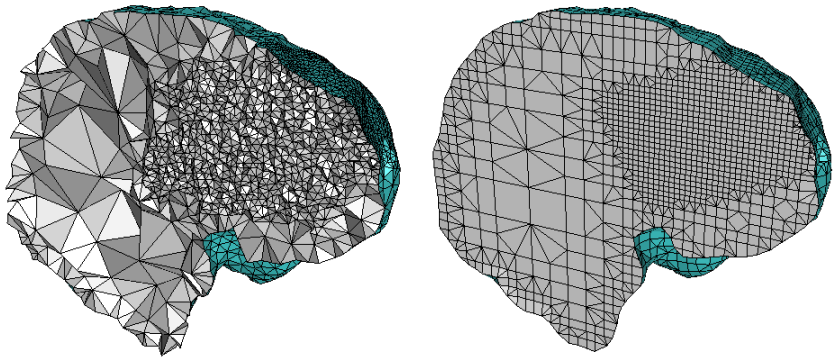
Fig. 24(a) shows the *TetGen* output, starting from the mesh shown in 23(a) and Fig. 24(b) the output for our algorithm, starting from the mesh shown in 23(b).

Table 1 resumes the statistics for both methods. In particular, it allows to compare the behavior of the algorithms when a list of elements is selected for



(a) (b) (c)

Fig. 23 (a) The coarsest mesh for *TetGen*, (b) an equivalent mesh at the surface using our technique and (c) the input mesh with a region for refinement using.



(a) (b)

Fig. 24 Output meshes after refining a given RoI (a) by *TetGen* and (b) our algorithm.

refinement. First of all, *TetGen* is much faster than our algorithm. However, when re–meshing, *TetGen* takes more than 4 times the original meshing time while we take less than the original time.

Mesh		Nodes		Elements		Time [ms]
		All	in RoI %	All	in RoI %	
TetGen	Initial	3888	3.13%	13370	5.94%	296.14
	Final	16519	39.11%	91766	48.99%	1232.66
Ours	Initial	10941	3.76%	26209	5.52%	4322.02
	Final	27220	42.63%	48199	29.54%	4167.14

Table 1 Comparison table between *TetGen* and our algorithm. Initial meshes are shown in Fig. 23; while final meshes are shown in Fig. 24. Columns % in RoI refers to the number of nodes or elements inside the Region of Interest.

When looking at the final meshes, we get 1.6 times the number of nodes of *TetGen*, but with 0.52 times the number of elements. However, if we look at the ratio of nodes inside the RoI over the total number of nodes the results seem similar.

There is one important point related to mesh quality that has not been tackled before. It is difficult to use only one metric to measure the quality of all the different element types in this work. In the case of tetrahedra and hexahedra, the problem has been long studied. However this is not true for the prism (wedge) and the pyramid. In a previous work [13], we already introduced a metric based on the Jacobian to detect invalid or bad–shaped linear elements. This is an extension of the Scaled Jacobian (J_S) defined for hexahedra to the 4 element types used in this work. It is called Element Normalized Scaled Jacobian (J_{ENS}).

The J_{ENS} ranges in [-1,1]. When negative, the element is invalid, meaning that it has a "volume inversion" somewhere. In the case of a negative value for a linear tetrahedron, it means that this tetrahedron will eventually penetrate a neighbor element. Let us say that a valid tetrahedron is the one with node a at the right of the triangle formed by nodes b, c, d. If we now move node a to the left of plane formed by b, c, d then its J_{ENS} value will be negative.

A positive J_{ENS} means the element is valid. It will be of poor quality (bad–shaped) when it is close to 0, and good as it approaches to the value 1. More details and validations can be found in [13].

In Table 2, we measured the J_{ENS} over the meshes of Table 1. If we look at our initial mesh, it presents good quality results with respect to *TetGen*. This is because the use of "surface patterns" prevents to produce invalid and poor quality elements. However, "transition patterns" are meant to manage transitions between fine and coarse regions and sometimes they fail when used at surface. It is for this reason that our Final mesh has negative and poor quality elements.

		n° of elements				
J_{ENS}		< 0	< 0.05	< 0.1	< 0.2	≥ 0.2
TetGen	Initial	0	1	58	1493	11818
	Final	0	7	182	3227	88350
Ours	Initial	0	0	0	17	26192
	Final	5	9	6	170	48009
	Repaired	0	3	6	173	48084

Table 2 Quality statistics with J_{ENS}. Initial meshes are shown in Fig. 23, while final meshes are shown in Fig. 24. Repaired mesh slightly differs from Final mesh.

Knowing that one invalid element makes the entire mesh not suitable for simulation, we decided to use our algorithm to refine once all the invalid elements. This "repaired" mesh is presented in the last row of Table 2 where we can see just a few poor quality elements remaining in the mesh. We could continue to perform this task for improving the quality, but our intention was

to add as few elements as possible to count with a valid mesh. Note that the repaired mesh has much better quality distribution than the one from *TetGen* and this was achieved with only one extra refinement iteration of those invalid elements.

7 Conclusion and perspectives

The main goals of this work were to study how to accelerate mesh generation and enable re–meshing for the Quadtree and Octree strategies when using mixed–elements. To these purposes, an edge data structure keeping neighbor information has been introduced. This, plus modifying the algorithms to recover balance immediately after refining, were the key factors for accelerating mesh generation.

Besides, the use of a well identified and limited list of Patterns at all steps, makes our process determinist, and insure to cover all cases. Thus the method is self-coherent and easy to understand; and its modular implementation makes it versatile, one can choose which steps to be run, and easy to maintain and evolving.

In the 2D case, we implemented a whole new Quadtree algorithm capable of achieving acceptable boundary representation and managing transitions between fine a coarse regions of the mesh by using mixed–elements: triangles and quadrilaterals. In the course, we noticed that a lot of time was being used to balancing the resulting mesh. The incriminated step was the neighbor search in the Quadtree structure to determine the difference of RL between 2 adjacent quadrants. So we proposed an efficient structure and adapted algorithm to optimize this search. Then we extended this data structure to 3D for a previously implemented Octree algorithm. In both cases, we were able to demonstrate that the data structure accelerates the meshing procedure, especially when the refinement level is high, or when non uniform refinement is needed. However, generation algorithms like *Triangle* and *TetGen* are still faster, but our reconstruction times remain in the same order of magnitude, while producing less elements.

When combining the data structure with the capability of re–meshing, we obtained promising results. This task involves saving *ad hoc* information in addition to the mesh itself, as explained in section 4.2. However, thanks to this, it is possible to recover a balanced mesh with its quadrants/octants and linked elements, therefore, enabling local refinement of specific regions. This process allows for a substantial gain in time. When starting from a previously generated mesh, *TetGen* spent up to 4 times the original generation time for producing a denser mesh in a particular region, while our method needed less than the original generation time for producing an equivalent result. This was shown in Table 1 for the examples of Fig. 24. We would like to highlight that the process of re–meshing can even be used to repair poor quality or invalid elements in the mesh as it was shown in section 6.

Another interesting point is that the initially constructed Quadtree/Octree structure can be kept in memory for further refinement. This is particularly interesting in a successive *a posteriori* adaptation scheme, and will avoid the need to reconstruct the previous structure before to refine it. This will save both the construction and writing time. Moreover, one can consider to exchange the mesh only once between the meshing and simulation modules at the beginning, and only sent updated elements in the subsequent iterations, thus limit the amount of transferred data. Besides, with the rapid expansion of interactive applications, *eg.* in medical simulators, embedding the numerical simulation and error estimation with the refinement module in a dynamic way would clearly be a response to be promoted.

As future work, we would like to improve the *Feature* representation in the 3D version. Unlike the 2D version that is known to cover properly all the cases, some particular configurations are still requiring work in 3D. Also, our next effort will be to avoid producing invalid elements at the surface. This may arise when performing *Transitions* and *Surface* patterns on the same cell. And, even this can be fixed by using our re–meshing strategy, we intend to automatically treat this case in order to always produce a valid mesh.

In addition, mesh simplification could be an interesting point to investigate. Note that this has not been implemented yet, but by construction, our method is well prepared for this operation, as we are keeping track of every cell split via their edges subdivision, as well as an up-to-date neighbor list, this latter being the key-point for applying the reverse operation while preserving balance and mesh conformity.

Acknowledgements Both authors have been partially supported by the Franco-Chilean ECOS–Sud Conicyt C16E05 project, and second author has been partially funded by Chilean Fondecyt-1181505 project.

A Re–meshing File Format

Whenever a mesh is generated, some additional data is stored. This file does not replace the mesh file, however it contains all the required information to re–build the balanced Quadtree or Octree state associated with that mesh. Combination of quadrants and octants is not allowed in a same file. Here is an example for the 3D case (.oct):

```
# Header
5005 19791 2241

# Vertex coordinates
+2.85692000E-01  +2.21920000E-02  +1.91885460E+01
+1.34949800E+02  +2.21920000E-02  +1.91885460E+01
...

# Edge indexes
0 22 0
1 2 39
...

# Octant & surface intersection data
```

```
8 1315 1317 1314 1306 1310 1316 1309 86 4
9 4380 4497 4499 4622 4624 4738 4740 4854 4966
8 1550 1546 1541 1547 1549 1543 442 1544 4
0
...

# Geometric Transform
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000

# Octant sub-elements
0 0 0 1 0 2 ...
```

Header. It always has 3 integers representing the number of nodes, edges and octants. Next comes 3 doubles corresponding to the coordinates of each node. They are implicitly to an index, starting from 0.

List of edges. Each edge is defined by the indexes of its two nodes. The third integer represents the index of the mid–point of the edge. If 0, this edge was not split into 2 new sub-edges. For instance in the file example the second edge (1,2) was split by index 39. Therefore, sub-edges (1,39,0) and (2,39,0) should be present later in the list.

For each quadrant or octant cell, 2 lines will be used in the file. The first line contains: the number of nodes of this cell (4 or 8), the list of indexes defining this cell, and its refinement level. For instance the first octant in the file is composed of 8 nodes, where the first node index is 1315, the last one is 86, and its RL is 4. The second line corresponds to the number of input faces of $\mathcal{P}$ this octant intersects, followed by their indexes. An octant with 0 intersections completely remains inside the domain.

Geometric Transform (optional). Can be applied to all the nodes. This transform involves rotations and translations. These values will depend on the first generated mesh. If a Region of Interest (RoI) was provided (a second surface mesh), the output mesh will be aligned with this RoI.

Numbers of sub–elements that decompose each octant. If this integer value is a 0, it means that this octant was barely intersecting the surface at the refinement stage, and therefore it was later removed when projecting inside nodes that were close to the boundary. This information is still necessary to correctly reconstruct the mesh. Of course, octants without any node inside the domain are not stored. In the example, the first 3 octants were finally removed from the mesh; the fourth octant has 1 element, therefore if element index 0 must be refined, octant 0 will be refined. The fifth octant was also removed and the next one has element indexes 1 and 2. If any of these elements were to be refined, this octant will be refined. If both element indexes are asked to be refined, the octant will be refined only once.

References

1. Aizawa K, Tanaka S (2009) A constant-time algorithm for finding neighbors in Quadtrees. IEEE Transactions on Pattern Analysis and Machine Intelligence 31(7):1178–1183, DOI https://doi.org/10.1109/TPAMI.2008.145
2. Arenas C, Lobos C (2018) Detection and representation of sharp features in Octree–based meshes using different types of elements. In: Proceedings of International Conference of the Chilean Computer Science Society, pp 1–8
3. Burkardt J (2011) `.poly` files (under LGPL). `https://people.sc.fsu.edu/~jburkardt/data/poly/poly.html`
4. Conti P, Hitschfeld N, Fichtner W (1991) Omega -an octree-based mixed element grid allocator for the simulation of complex 3-D device structures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 10(10):1231–1241
5. Ebeida M, Patney A, Owens J, Mestreau E (2011) Isotropic conforming refinement of quadrilateral and hexahedral meshes using two-refinement templates. International Journal for Numerical Methods in Engineering 88(10):974–985, DOI 10.1002/nme.3207

6. Finkel RA, Bentley JL (1974) Quad trees a data structure for retrieval on composite keys. Acta Inf 4(1):1–9, DOI 10.1007/BF00288933
7. Frey PJ, Borouchaki H (1998) Geometric surface mesh optimization. Computing and visualization in Science pp 113–121
8. González E, Lobos C (2014) A set of mixed-element transition patterns for adaptive 3D meshing. Tech. Rep. 2014/01, Departamento de Informática, UTFSM, DOI 10.13140/RG.2.1.3367.4400
9. Gravenkamp H, Eisenträger S (2017) Automatic image-based analyses using a coupled Quadtree-SBFEM/SCM approach. Computational Mechanics pp 1–26, DOI 10.1007/s00466-017-1424-1
10. Huo S, Li Y, Duan S, Han X, Liu G (2019) Novel Quadtree algorithm for adaptive analysis based on cell-based smoothed finite element method. Engineering Analysis with Boundary Elements 106:541 – 554, DOI https://doi.org/10.1016/j.enganabound.2019.06.011
11. Huo S, Liu G, Zhang J, Song C (2020) A smoothed finite element method for Octree-based polyhedral meshes with large number of hanging nodes and irregular elements. Computer Methods in Applied Mechanics and Engineering 359:112646, DOI https://doi.org/10.1016/j.cma.2019.112646
12. Ito Y, Shih A, Soni B (2009) Octree-based reasonable-quality hexahedral mesh generation using a new set of refinement templates. International Journal for Numerical Methods in Engineering 77(13):1809–1833
13. Lobos C (2015) Towards a unified measurement of quality for mixed–elements. Tech. Rep. 2015/01, Departamento de Informática, UTFSM, URL `http://www.inf.utfsm.cl/~clobos/tech.html`
14. Lobos C, González E (2015) Mixed-element octree: a meshing technique toward fast and real-time simulations in biomedical applications. International Journal for Numerical Methods in Biomedical Engineering 31(12):1–31, DOI 10.1002/cnm.2725
15. Namdari M, Hejazi S, Palhang M (2015) MCPN, octree neighbor finding during tree model construction using parental neighboring rule. 3D Research 6, DOI 10.1007/s13319-015-0060-9
16. Nicolas G, Fouquet T (2013) Adaptive mesh refinement for conformal hexahedral meshes. Finite Elem Anal Des 67:1–12, DOI 10.1016/j.finel.2012.11.008
17. Nicolas G, Fouquet T, Geniaut S, Cuvilliez S (2016) Improved adaptive mesh refinement for conformal hexahedral meshes. Advances in Engineering Software 102:14 – 28, DOI https://doi.org/10.1016/j.advengsoft.2016.07.014, URL `http://www.sciencedirect.com/science/article/pii/S0965997816301971`
18. Samet H (1982) Neighbor finding techniques for images represented by Quadtrees. Computer Graphics and Image Processing 18(1):37 – 57, DOI https://doi.org/10.1016/0146-664X(82)90098-3
19. Samet H (1989) Neighbor finding in images represented by Octrees. Computer Vision, Graphics, and Image Processing 46(3):367 – 386, DOI https://doi.org/10.1016/0734-189X(89)90038-8
20. Saputra AA, Eisenträger S, Gravenkamp H, Song C (2020) Three-dimensional image-based numerical homogenisation using Octree meshes. Computers & Structures 237:106263, DOI https://doi.org/10.1016/j.compstruc.2020.106263
21. Schneiders R (1996) Refining quadrilateral and hexahedral element meshes. In: Proceedings of the Fifth International Conference on Numerical Grid Generation in Computational Field Simulations, pp 679–688
22. Shewchuk JR (1996) Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In: Lin MC, Manocha D (eds) Applied Computational Geometry: Towards Geometric Engineering, Lecture Notes in Computer Science, vol 1148, Springer-Verlag, pp 203–222, from the First ACM Workshop on Applied Computational Geometry
23. Si H (2015) Tetgen, a delaunay-based quality tetrahedral mesh generator. ACM Trans Math Softw 41(2), DOI 10.1145/2629697
24. Verfürth R (2013) A Posteriori Error Estimation Techniques for Finite Element Methods. Numerical Mathematics and Scientific Computation, OUP Oxford
25. Yerry M, Shephard M (1983) A modified quadtree approach to finite element mesh generation. IEEE Computer Graphics and Applications 3:39–46, DOI 10.1109/MCG.

1983.262997, URL doi.ieeecomputersociety.org/10.1109/MCG.1983.262997
26. Zhang H, Chandrajit B (2006) Adaptive and quality quadrilateral/hexahedral meshing from volumetric data. Computer Methods in Applied Mechanics and Engineering 195(9 - 12):942 – 960, DOI https://doi.org/10.1016/j.finel.2007.03.001
27. Zhang H, Zhao G (2007) Adaptive hexahedral mesh generation based on local domain curvature and thickness using a modified grid-based method. Finite Elements in Analysis and Design 43(9):691 – 704, DOI https://doi.org/10.1016/j.finel.2007.03.001