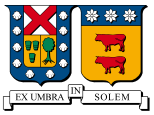
Documentation

2D Mixed–element mesh generator

Version 2018.11.23 roi–type

Claudio Lobos
clobos@inf.utfsm.cl
Departmento de Informática – utfsm – Chile.
Fabrice Jaillet
fabrice.jaillet@univ-lyon1.fr
Université de Lyon, IUT Lyon 1 – LIRIS CNRS UMR-5205 – France.



UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

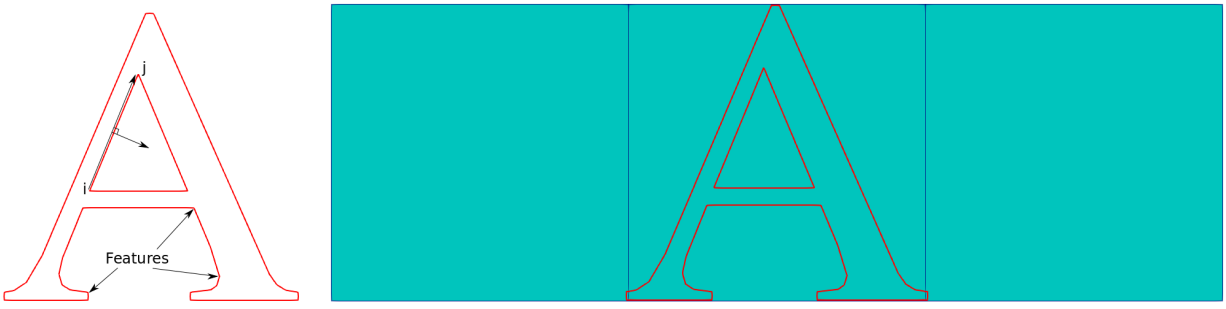
Abstract

This document will show you how to obtain mixed–elements meshes with the provided code. Two main alternatives are explained here. The first will show you how to use the code as a standalone program. The second will show you how to bundle your own code with the mixed-element mesh generator. In both cases, mixed–element are employed to manage transitions between fine and coarse regions and at the boundary of the domain. All the remaining regions will be meshed with structured regular quadrangles.

1 Data Structures and initialisation

Quadrant, QuadEdge, Polyline Final resulting Mesh

This mesh generator will allow you to create a mixed–element mesh starting from a boundary 2D mesh (Polyline) composed of edges (Fig. 1(a)). Let this input surface mesh be $\mathcal{S}$. Two constraints must be fulfilled by $\mathcal{S}$: it must be unfolded (no self–intersection), and the normal of the edges must be pointing outside.
The algorithm will use $\mathcal{S}$ for two purposes: to find out if a point is inside or outside the domain and to project a point onto $\mathcal{S}$. Therefore, if you have two different meshes representing the exact same domain, you should use the one with less edges. The algorithm will compute faster the output mesh.
The first step of the algorithm is to compute the Bounding box (Bbox) of $\mathcal{S}$. This Bbox will not necessarily be a perfect square. Therefore, an algorithm will be employed to automatically compute a set of squares containing $\mathcal{S}$ (Fig. 1(b)).



(a) (b)

Figure 1: (a) An example of complex polyline with holes and sharp features. This corresponds to the Input to be meshed (source: a.poly) . (b) The embedding (poorly defined...) initial quadrants structure.

1.1 Polyline as Input boundary

A **Polyline** is composed of (Fig. 1(a)):

- a vector of **3DPoint**, with coordinates, that are all lying inside the **BoundingBox** of the Polyline,
- and their associated angle. If the angle is not lying in the range $[MinAngle = 175, MaxAngle = 185]$, then it's considered as a sharp point, and called a **Feature** in the remaining.
- a vector of **PolyEdge**, each one defined by the indexes of the extremity' nodes, and a normal. By convention, the normal is not normed and is pointing out the Input. For an oriented edge (i, j), its right side will be outside and its left side inside. Thus, the Input must be defined by closed sets of connecting edges with counter-clockwise orientation. This way, holes may be simply created.

The format for Input file might be either `.poly` as described on the Triangle_1.6 website; find an example here: unit_square.poly. It could also be defined as `.mdl` file, see unit_square.mdl:

1.2 Quadrant as support of surface mesh

The other important structure is the **Quadrant**:

- is composed of a *pointindex*, vector of 4 corner indices in a vector of **MeshPoint**: Note here that MeshPoint is embedding a previously seen **3DPoint**, as well as some information describing its state. More details below.
- When the Quadrant is subdivided into non quadrant elements (in Transitions for example), it contains a vector of *sub_elements*, each one described by a vector of indexes as presented right above.
- It also contains a list of intersecting **PolyEdges**, and intersecting **Features**. These informations are useful to speed up the inside/outside discrimination for the Quadrant or to better handle the boundary and surface elements, for example.
- and some additional data, for processing purpose that will be described later.

Along with the Quadrant, comes the **QuadEdge**:

- defined by 3 indices, the 2 extremities and potential *midpoint* when this edge is split.

And finally, the **MeshPoint**:

- that is embedding a 3DPoint, as coordinates of the Quadrant's corners, referenced as indexes.
- MeshPoint are conversely connected to Quadrant by an index map of *elements*, referencing all the Quadrants having this point as corner.
- it also is the support for some crucial processing information on it's *state*, merged in a single byte: the point is `inside`, has been `projected`, is representing a `feature`, has been previously `checked`, etc...

These three structures will be used together along with the Polyline by the **Mesher**, that will construct them gradually:

- a vector of **MeshPoint**,
- a vector of **Quadrant**,
- a set of **QuadEdge**. Note that this kind of structure is costly (quasi 30% of the whole computation time), as insertion is done in a sorted container, but it guarantees uniqueness of the element, and reasonably fast access. By the way, it might not be exactly adapted to parallelism...
- It also contains a list of **Refinement Regions**, which usage will be presented later on.

2 Main steps of the method

This mesh generator is based on the Quad-tree technique introduced in [1]. This technique recursively split a Quadrant in a finite number of equivalent sons. For instance if the Quadrant is a square, to refine it one level means that it will be replaced by 4 new Quadrants. All of them will be sons of the replaced Quadrant in the Quad-tree structure. By allowing the use of quadrants with cut corners, this modeling technique overcomes some of the drawbacks of standard Quad-tree encoding for finite element mesh generation [2]. In our case, Quadrants will be continuously split until a maximum provided level is reached. Quadrants lying completely outside $\mathcal{S}$ will be removed.

todo: mixed-elements mesh generation, describe here the main steps of the method

2.1 A quadtree-based method

show different steps of the method:

Algorithm 1: Generation process

Result: A mixed-elements mesh

1 **foreach** *Quadrant* **do**
    repeat
2 Subdivide Quadrant;
        foreach *new Sub-Quadrant* **do**
            if *Intersects Input or Is Inside* **then**
                Insert Sub-Quadrant
            end
        end
    until *desired Refinement Level*;
end
3 Create balanced quadtree;
4 Apply Transition Patterns;

1. subdivision, quad + ROI + off-domain quad removing
2. balanced : the resulting mesh is not balanced if one of the edges of a Quadrant is subdivided twice. In this case, subdivide the Quadrant as in Algorithm 1 step 2. And repeat until the mesh is balanced.
3. transition patterns (OMP)

todo: describe visitors

2.2 Refinement level: all, surface and/or region

Now it is possible to introduce the most important parameter of the method: the Refinement Level (rl). As previously said, the method is based on Quad-tree, and this user parameter will set the level of subdivision required to reconstruct the input $\mathcal{S}$.
todo: describe region refinement format and usage The different option for refinement could be:

- refine all Quadrants (`-a`). The splitting operation will be performed over each Quadrant enclosing $\mathcal{S}$. For instance, `-a 2`
- refine Quadrant **surface**,
- refine Quadrant laying in a specific **region**, determined by

The **Visitor Pattern** has been implemented to circulate (*visit*) the Quadrants and determine whether they should be split or not (*accept*).
todo: describe visitors

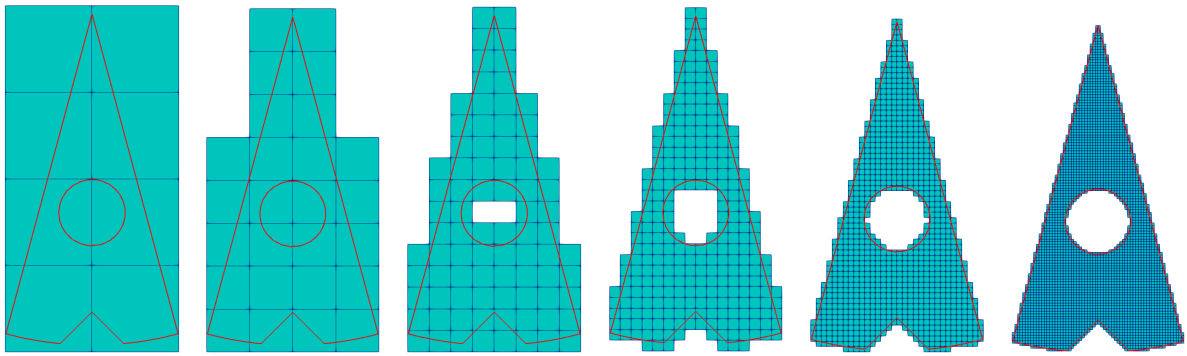


Figure 2: Effect of the refinement level applied on the complete structure (`-a` switch), (1 to 6)

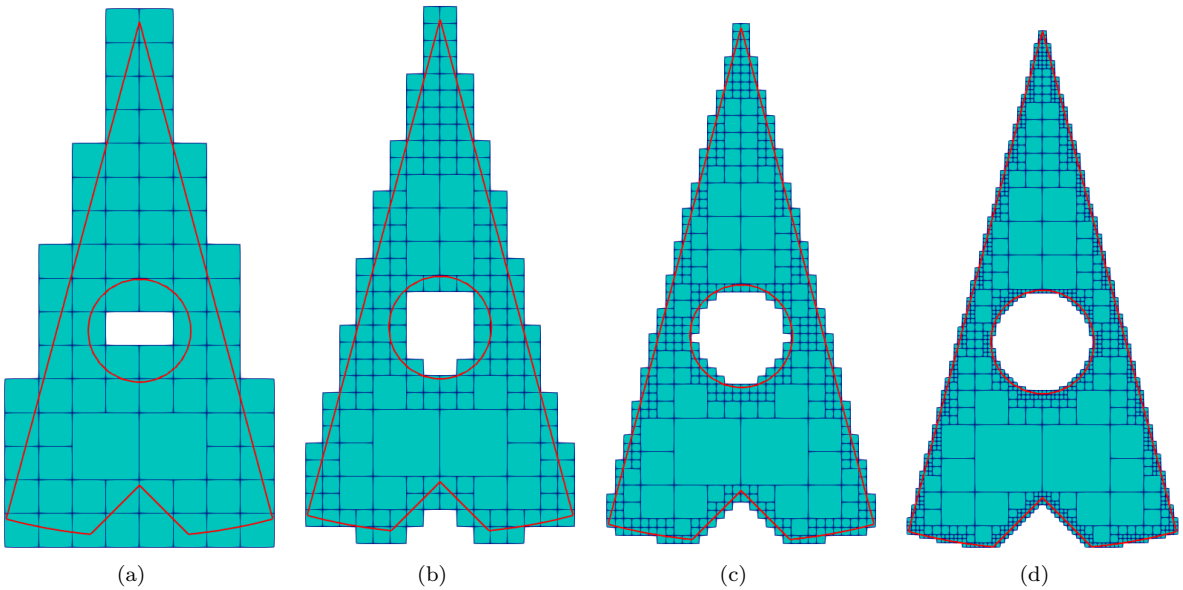


Figure 3: Effect of combined refinement levels, with level 2 applied on the complete structure (`-a` switch), and incremental refinement (3 to 6) only applied on surface quadrants (`-s` switch).

2.3 Balancing the mesh

A Quad mesh is said *balanced* when there is at most a difference of 1 level of subdivision between any two adjacent elements. In practice, this is determined by checking whether a Quadrant contains at least one QuadEdge that is subdivided twice as in Algorithm 1 step 2. Thus, for each Quadrant, every QuadEdge is checked. If one of them is subdivided, then the 2 sub-edges must be checked. Note that the *search* and *access* operations on QuadEdges are accelerated by the ordered set container. But whether this data structure is adapted to parallel computing is a good question...

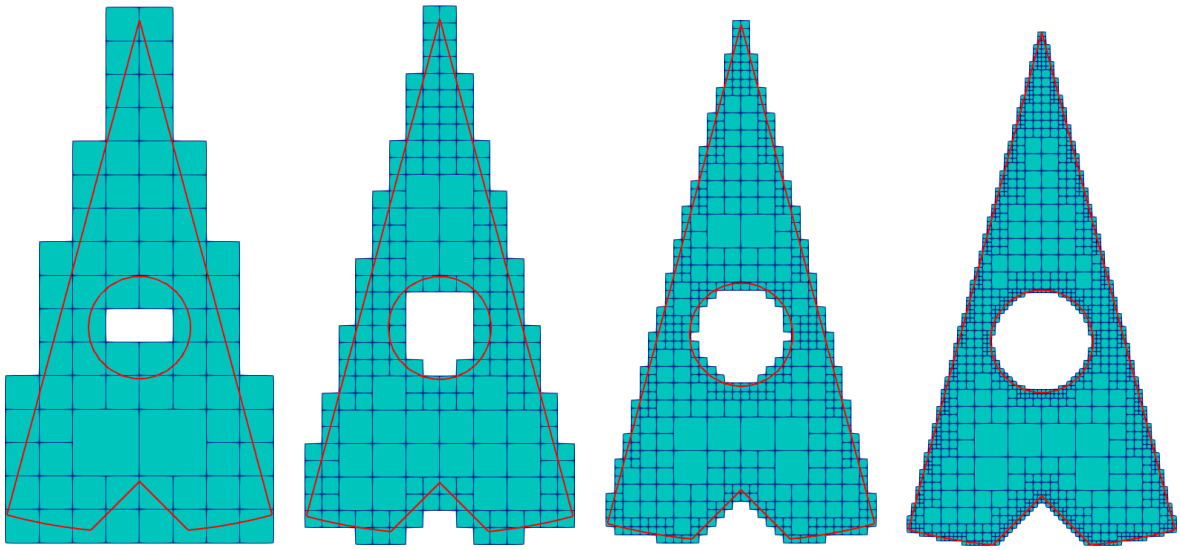


Figure 4: After the balancing step of previously refined mesh (sec. 2.2). Note that in the first case, nothing is to be done.

2.4 Transition patterns

Now the mesh is balanced, we need to handle transitions between Quadrants, to produce a *conformal mesh*, when the elements exactly share nodes and edges. This technique is known to save time during the computation, and also to avoid node interpolation errors. If 2 adjacent Quadrants have the same refinement level, then there is nothing to do. Otherwise, the less refined Quadrant must be treated. Again, in practice, this is determined by checking whether a Quadrant contains at least one QuadEdge that is subdivided.
For the Transition, the chosen technique was to apply a Pattern corresponding to the number and relative position of the refined QuadEdges. In 2D, there are only 6 cases to consider (Fig. 5). Note that:

- newly created internal edges (red lines in Fig. 5) are not of type QuadEdge. In that sense, they could not be subdivided again, and do not participate in the remaining of the subdivision process. They are indirectly stored as sub-elements, ie. a list of connected nodes (indexes, to be more precise),
- all patterns are producing good shaped sub-elements,
- and that the last one could have been simplified into 4 sub-squares, but that will require one additional middle vertex. This choice could be reconsidered if the number of elements matters more than number of points, or to improve quality, or to better handle Quadrants with Features.

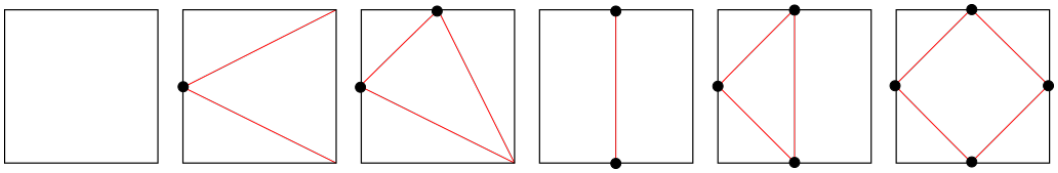
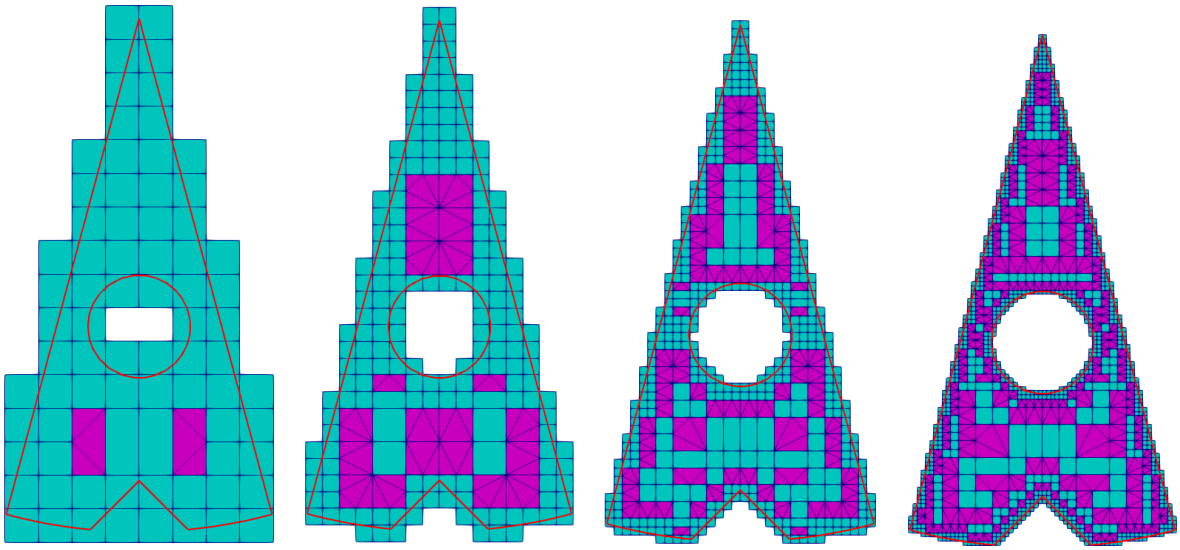


Figure 5: All possible configurations for transition patterns in 2D. Black dots represent an edge subdivision, while red lines draw the internal edges of the Quadrant sub-elements.



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

Figure 6: After the transition step. Note that the more difference between levels, the more transition patterns are used...

3 Fitting quadrants to Input Surface

Algorithm 2: Generation process and Input surface fitting

Result: A mixed-elements mesh

`/* Preprocess: handle boundary */`

`/* Generate quadtree */`

1 **foreach** *Quadrant* **do**

  repeat

2 Subdivide Quadrant;

  foreach *new Sub-Quadrant* **do**

  if *Intersects Input or Is Inside* **then**

  Insert Sub-Quadrant

  end

  end

  until *desired Refinement Level*;

end

3 Create balanced quadtree;

4 Apply Transition Patterns;

`/* Input surface fitting */`

5 Detect Features in Input;

Project Close to Boundary;

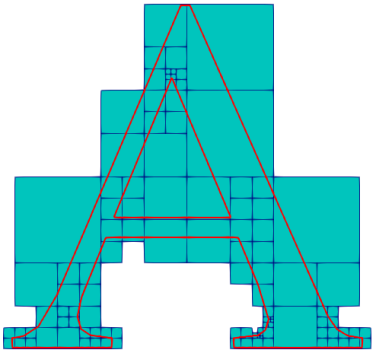
Remove on Input Surface;

Shrink to Boundary;

6 Apply Surface Patterns;

3.1 Boundary and sharp features handling

preprocess

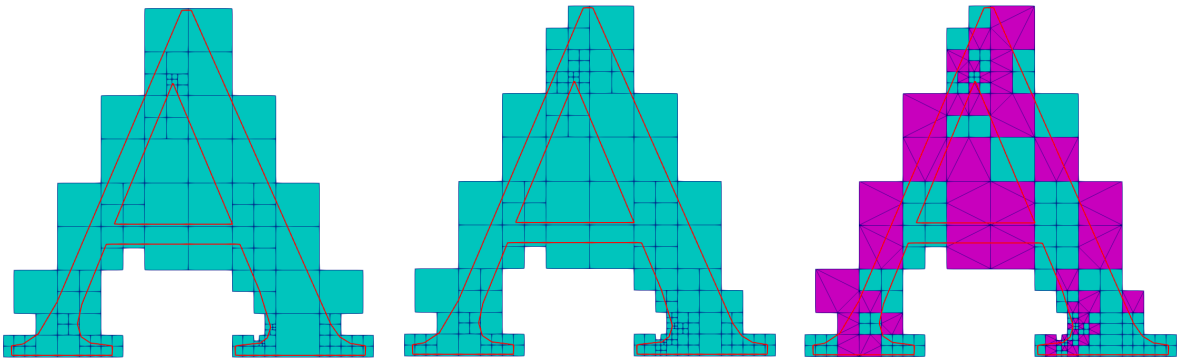


(a)

Figure 7: After boundary handling.

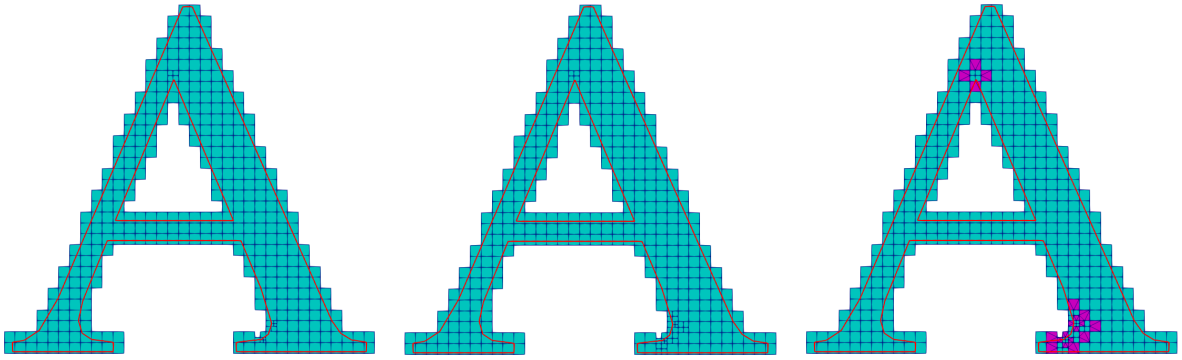
The first step to
// test1: if nb Features >= 2 // first condition is optimization if NbFeatures has been already computed // test2: if $distFProjectionFeature > distMax$ for each node // done: compute distMax before... // test3: if the number of intersection of the Polyline and Quadrant edge >= 3
setting, or not, a refinement level for Boundary handling

3.2 Generating first quadrants over boundary prepared mesh



(a) (b) (c)

Figure 8: After refinement, balancing and transition steps, level 3.



(a) (b) (c)

Figure 9: After refinement, balancing and transition steps, level 5.

(a) (b) (c)

Figure 10: After refinement, balancing and transition steps, level 7.

3.3 Handling Input surface

4 Remeshing

Algorithm 3: Refinement process

Result: A refined mixed-elements mesh
foreach *Element to refine* **do**
  Identify containing Quadrant;
  Subdivide Quadrant;
  foreach *Sub-quadrant* **do**
    if *Intersect input or Is In* **then**
      Insert Sub-Quadrant
    end
  end
end
Goto *Algorithm 1, step 3*

identify which Quad contains the element, information from file
subdivide Quad + goto subsection balanced

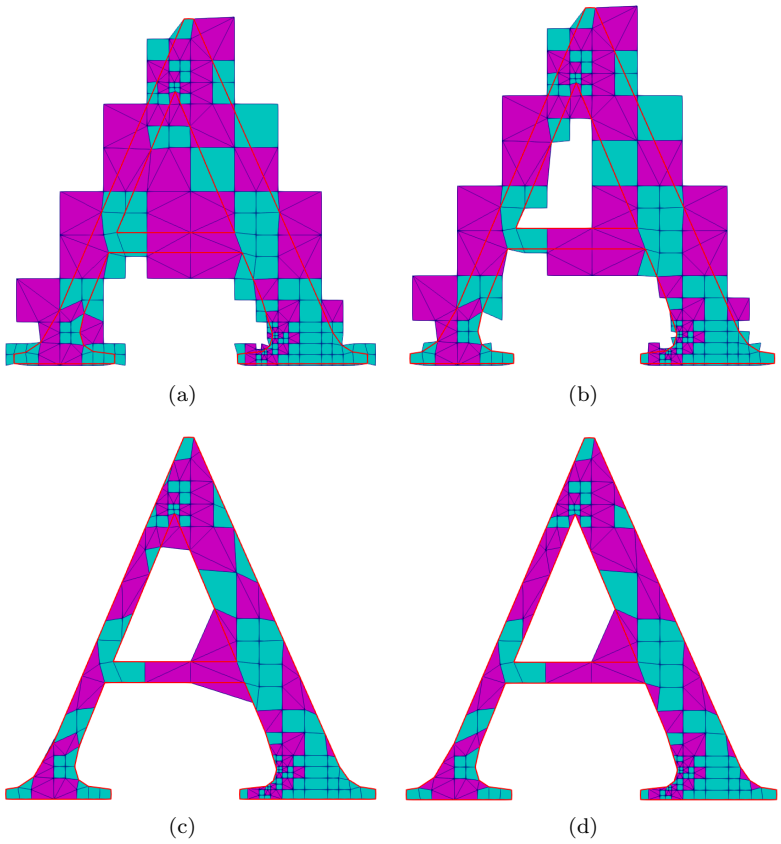


Figure 11: After projecting interior points close to boundary, , level 3.

(local remeshing? neighbourhood??)

5 Installing

In order to install this application you will need a Unix–based system, a `c++` compiler and `cmake`. You should do the following:

5.1 Create an account on github.com

- go to github.com, then 'Signup and Pricing' ⇒ 'Free for open source' ⇒ 'Create a free account'
- choose Username/Email/Password and it's done
- you can also update your account settings (website, avatar...)

5.2 Git Fork MixedQuadTree

Let's fork the initial MEPP repository. So, once you're logged in on github

- go to https://github.com/jaillet/MixedQuadTree/
- click on the 'Fork' button

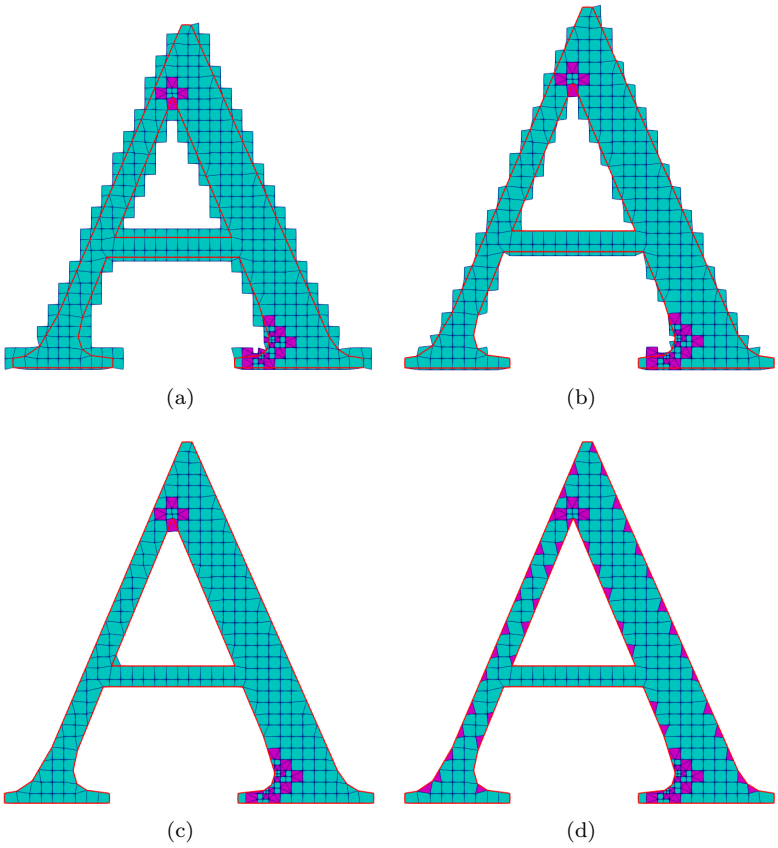


Figure 12: After projecting interior points close to boundary, level 5.

- once the fork is finished, you'll have your own copy of MixedQuadTree with the following path: https://github.com/**yourUsername**/MixedQuadTree
- at this URL, you're able to browse the sources, see the commit log, report issues...

5.3 Compiling

```
$ git clone /https://github.com/jaillet/MixedQuadTree.git
$ cd MixedQuadTree/
$ git checkout branch develop
$ mkdir build ; cd build
$ cmake ../src -D_CMAKE_BUILD_TYPE=Debug (or Release) // Uppercase is important
$ cmake .
$ make -j#nproc
```

This has been tested on Linux and Mac without any known error nor warning.

5.4 Git usage

At this point, you can edit and commit files using the git workflow
but only push to your own fork of MixedQuadTree (origin) !!!

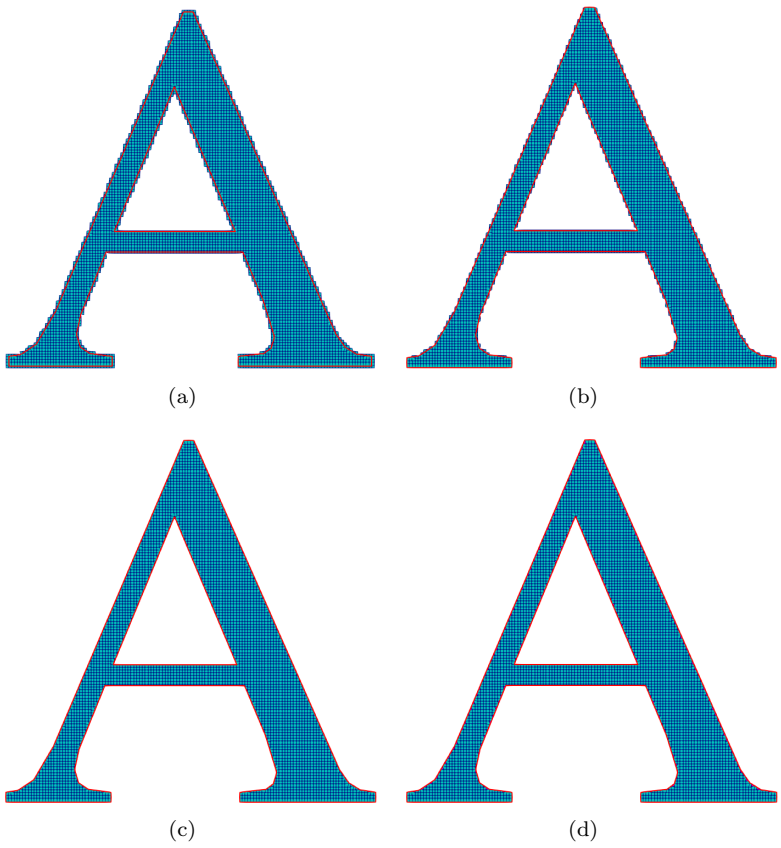


Figure 13: After projecting interior points close to boundary, level 7.

The merge between the original repository is done by ANOTHER DEVELOPER after a **'pull request'**. To make a 'pull request', go to https://github.com/yourUsername/MixedQuadTree, click on 'Pull Request'. Then, choose the two branches you would like to merge (master in *jaillet* and yourbranch in origin), write your comments and click on 'Send Pull Request'.

Acknowledgement

Several researchers have contributed with ideas, guidance, and general discussion to this project. Special thanks to Nancy Hitschfeld, Yohan Payan, Marek Bucki, Vincent Luboz.
Also several great students have contributed in coding and integrating with other software. Many thanks to Eugenio González, Sebastián Durán, Esteban Daines, Cristopher Arenas and Sebastián Tobar. This list will continue to grow because we are currently implementing new features that will add great value to this meshing tool and their will be available in a future version.
This work was financed by grant: FONDECYT de Iniciación 11121601 and ECOS-CONICYT C11-E01 and C16-E05.

References

[1] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Inf.*, 4(1):1–9, March 1974.

[2] M. Yerry and M. Shephard. A modified quadtree approach to finite element mesh generation. *IEEE Computer Graphics and Applications*, 3:39–46, 01 1983.