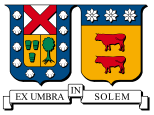
Documentation
Quality 2D Mixed–Element Mesh Generation and Refinement

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

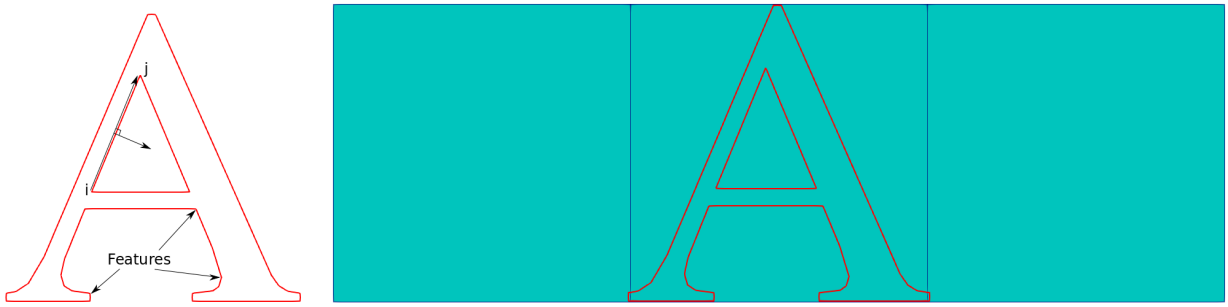
This document will show you how to obtain mixed–elements meshes with the provided code. Two main alternatives are explained here. The first will show you how to use the code as a standalone program. The second will show you how to bundle your own code with the mixed-element mesh generator. In both cases, 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.

Contents

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. More examples in the data directory.

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

`/* Generate quadtree: */`

1 **foreach** *Quadrant* **do**
  repeat
2 Subdivide Quadrant;
    foreach *new Sub-Quadrant* **do**
      if *Intersects Input or Is Inside* **then**
3 Insert Sub-Quadrant
      end
    end
  until *desired Refinement Level*;
end
4 Create balanced quadtree;
5 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 $\mathcal{S}$.
The different option for refinement could be:

- refine **all** Quadrants (`-a N`). The spitting operation will be performed over each Quadrant enclosing $\mathcal{S}$. N will define the refinement level rl for this operation. For instance, `-a 2` will split all the initial Quadrants twice, replacing them by potentially 16 new Quadrants. In reality, this is achieved level by level: first subdividing and replacing the Quadrants in 4 new ones (Algorithm 1 step 2), keeping only those that lay inside or that intersect $\mathcal{S}$ (step 3, using again a *visitor*). Then, only those ones are candidates to the second level of subdivision, repeating the steps of splitting and checking of intersection on sub-Quadrants (Fig. 2).
- refine **surface** Quadrant `-s N`. The splitting operation will be performed over each Quadrant that intersects a section of $\mathcal{S}$. For instance, `-s 2` will split all the initial Quadrants into 4 new ones and then, only the Quadrants that still intersect $\mathcal{S}$ will be split in 4 more, still keeping only intersecting ones (Fig. 3)
- (not tested yet in 2D!) refine by **block** `-b` followed by the name of file where some block regions are specified. A block is defined by 2 points: $(min_x, min_y, min_z), ((max_x, max_y, max_z)$ and a refinement level rl to be applied over the Quadrants that intersect this block. This is a text file where first we specify the number of blocks and then we detail each one of them. An example with 2 regions is now

provided:

```
n_regions 2
0 0 0
10 10 0
5
0 0 0
20 20 0
4
```

In this 2D case, min_z and max_z are mandatory for compatibility reasons with the 3D format, but will not be used. Thus, the portion of $\mathcal{S}$ intersecting the quadrangle $(0,0) \rightarrow (10,10)$ will be refined to level 5 and the complement inside the block $(0,0) \rightarrow (20,20)$ that intersects $\mathcal{S}$ will be refined to level 4.

- (not tested yet in 2D!) refine Quadrants laying in a specific **region** `-r`. Its followed by 2 arguments: the name of file where another input surface is specified in `mdl` format and a *rl*. All the Quadrants inside $\mathcal{S}$ and the specified region will be refined to *rl*.

Finally, note that large meshes can be produced with a small value for N. For instance, if the starting point is only one cube and we set `N=10`, the code may produce up to $4^{10} = 1,048,576$ elements.
Of course, all these switches might be combined (as in Fig. 3) to produce flexible tools and comply with the user requirements. If a Quadrant belongs to more than one refinement region (no matter the type of it), it will be refined to maximum level among those regions.

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

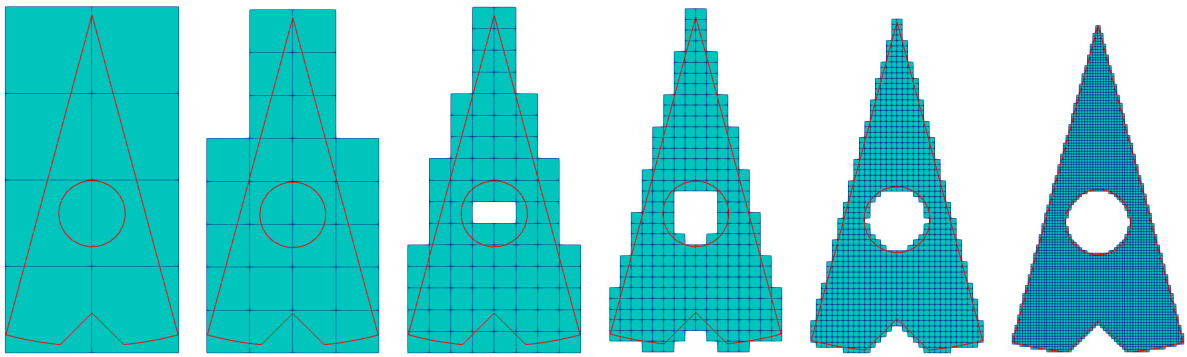


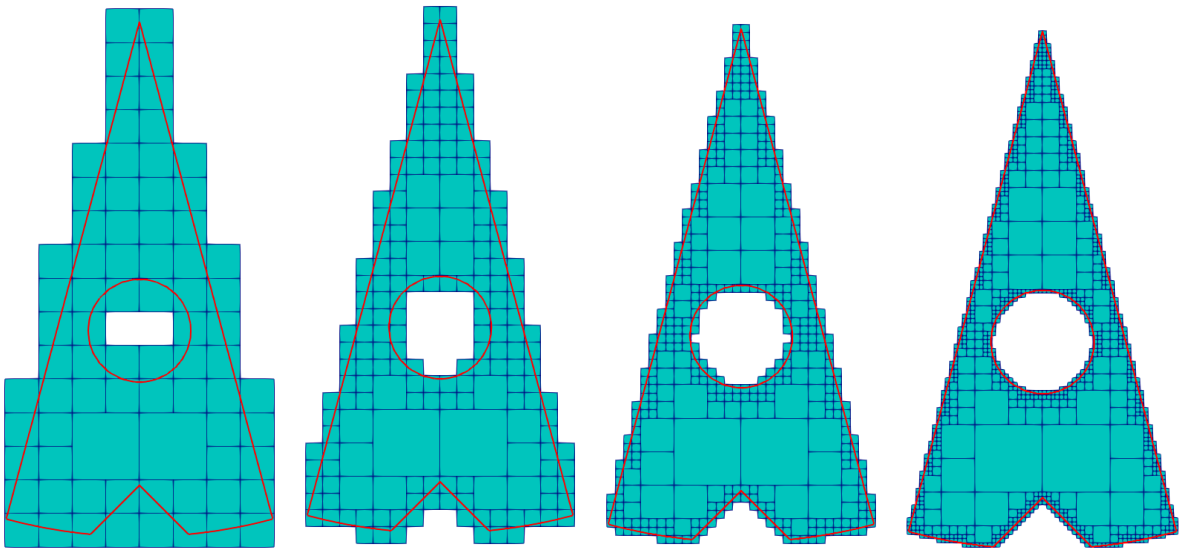
Figure 2: Effect of the refinement level applied on the complete structure (`-a N` switch, with `N=1..6`)

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 (step 4). 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...

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



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

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

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 (step 5).
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.

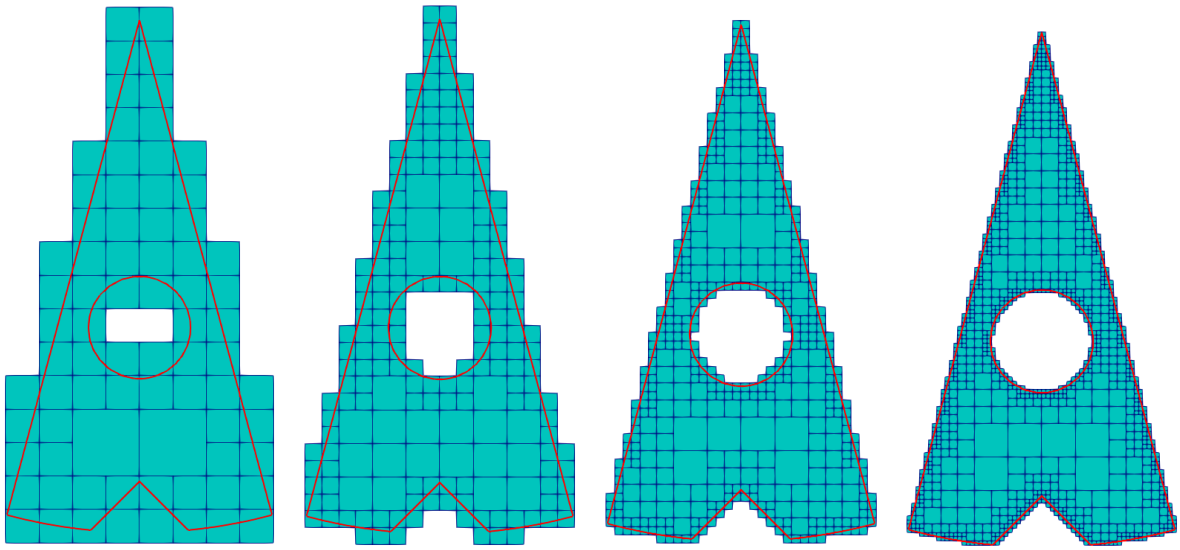


Figure 4: After the balancing step of previously refined mesh (sec. 2.2). Note that in the first case, nothing is to be done.

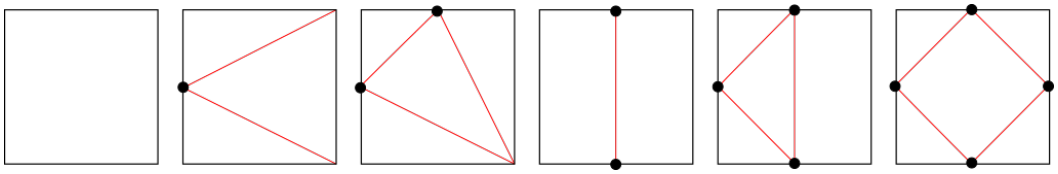
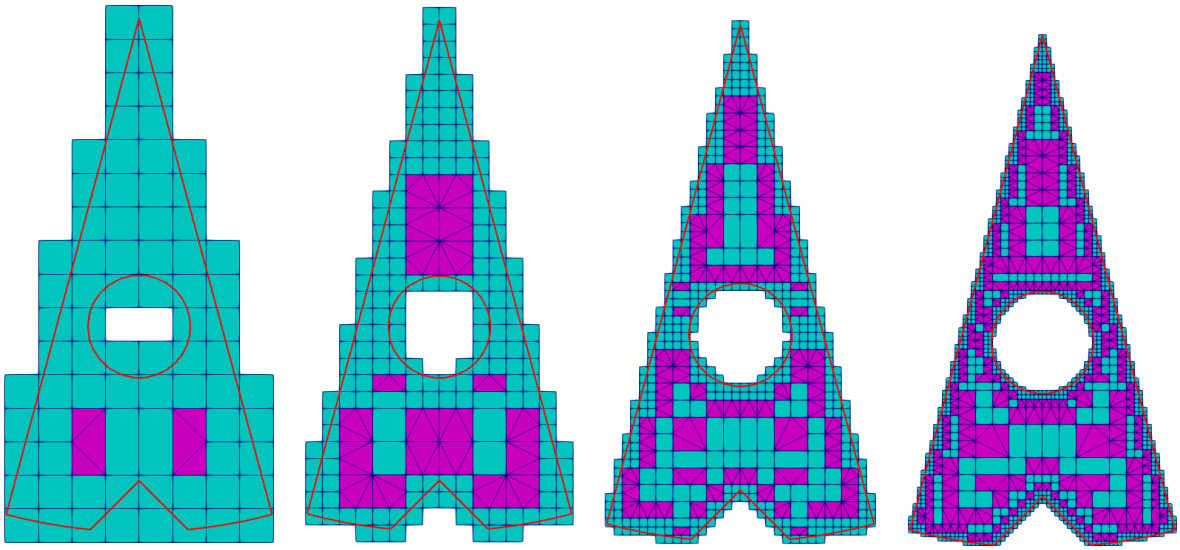


Figure 5: All possible configurations for transition patterns in 2D. Black dots represent an edge subdivision, while red lines draw the internal edges of the Quadrant sub-elements.



(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:` `*/`

0 Handle Boundary;

`/* Generate quadtree:` `*/`

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

  repeat

2 Subdivide Quadrant;

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

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

3 Insert Sub-Quadrant

  end

  end

  until *desired Refinement Level*;

end

4 Create balanced quadtree;

5 Apply Transition Patterns;

`/* Input surface fitting:` `*/`

6 Detect Features in Input;

7 Project Intern Nodes Close to Boundary;

8 Remove Outside Quadrant;

9 Shrink Extern Nodes to Boundary;

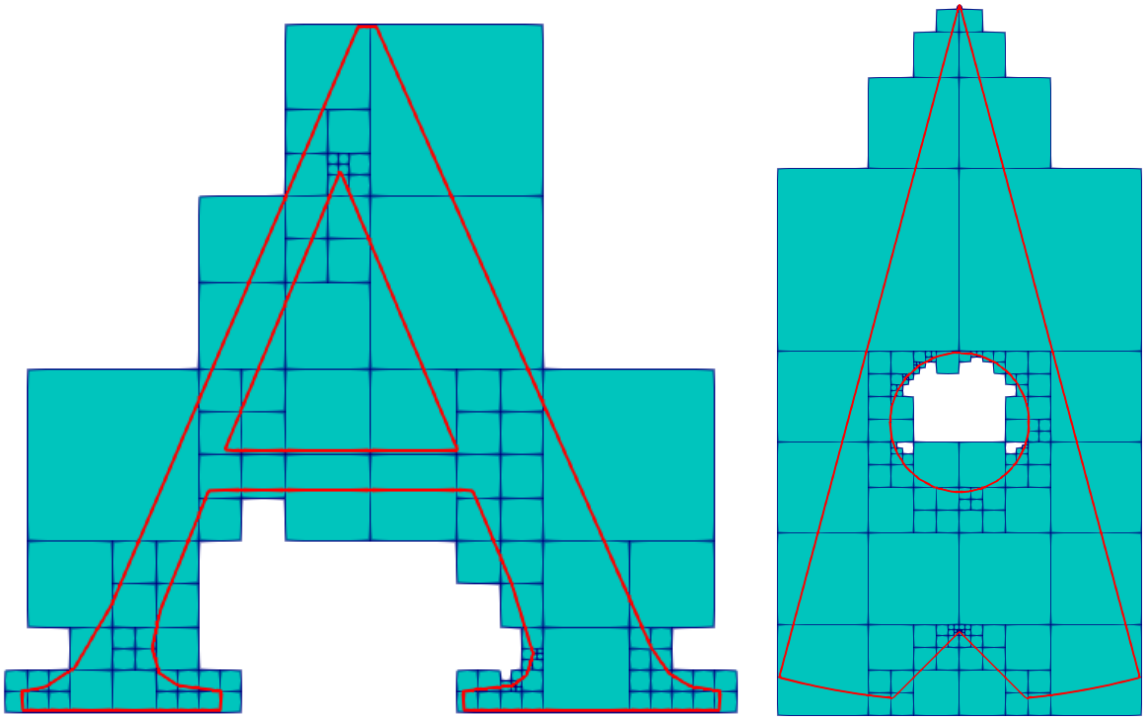
10 Apply Surface Patterns;

3.1 Boundary and sharp features handling

preprocessing

1. // test1: if nb Features >= 2 // first condition is optimization if NbFeatures has been already computed
2. // test2: if $distFProjectionFeature > distMax$ for each node // done: compute distMax before...
3. // test3: if the number of intersection of the Polyline and Quadrant edge >= 3

The result for the A and Pie Polyline is shown on Fig. 7, where it can be appreciated that subdivision only occurred where required by the Input Polyline.



(a) (b)

Figure 7: After preprocessing for Boundary handling, leading to a better representation of the Input Polyline.

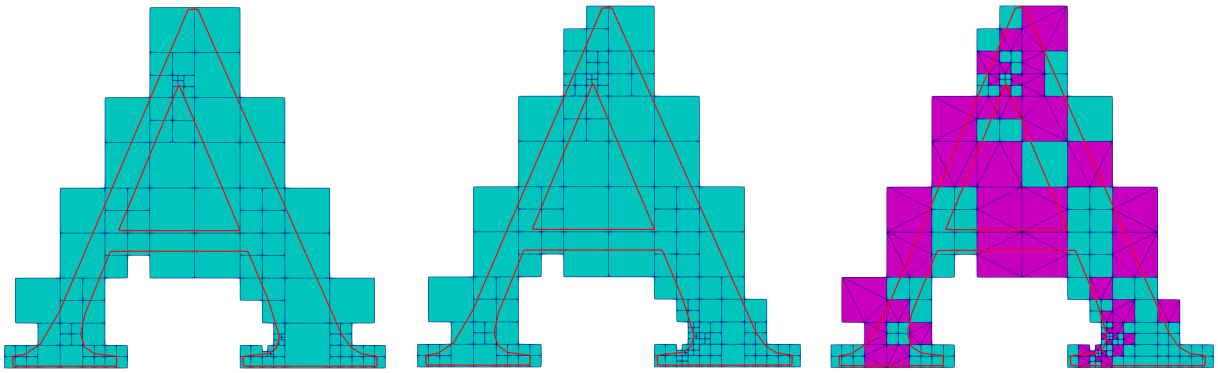
setting, or not, a refinement level for Boundary handling This preprocessing will now serve as input for Algorithm 1 or 2.

3.2 Generating first quadrants over boundary prepared mesh

This is exactly the same as in Algorithm 1. The only difference is the initial Quadrant list, that as been prepared as described in section 3.1

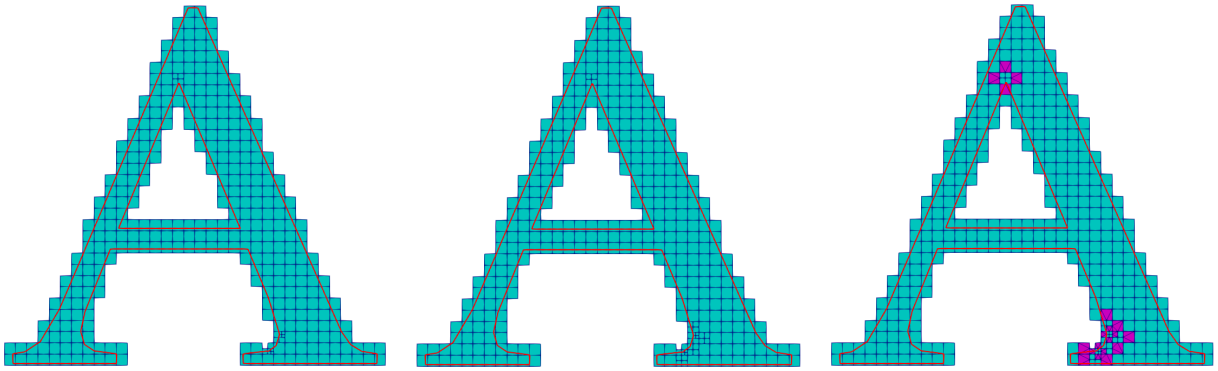
3.3 Handling Input surface

describe sub-algorithms for each step, in particular the Surface Patterns that are the main contribution of the method and most delicate step...



(a) (b) (c)

Figure 8: After refinement level 3, balancing and transition steps (see Algorithm 1).



(a) (b) (c)

Figure 9: After refinement level 5, balancing and transition steps (see Algorithm 1).

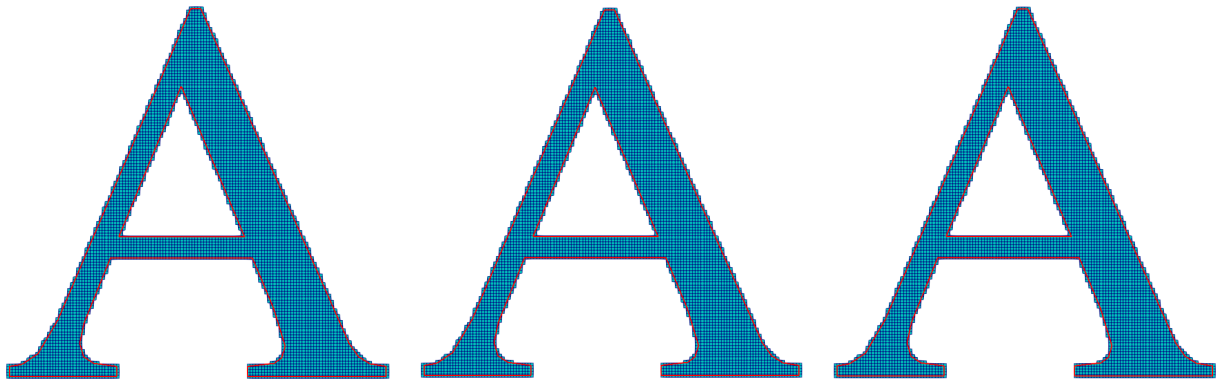
4 Remeshing

todo description, main point is the recycling of previously generated Quad Mesh, and Quad identification for the element list to be refined.

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 Inside* **then**
      Insert Sub-Quadrant
    end
  end
end
Goto *Algorithm 2, step 4*

Some results are shown on Fig. **??**. On each image, the bottom left element is refined. Notice how mesh is balanced and transition are created to produce a conformal mesh.



(a) (b) (c)

Figure 10: After refinement level 7, balancing and transition steps (see Algorithm 1).

identify which Quad contains the element, information from file
subdivide Quad + goto subsection balanced
(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 MixedQuadTree repository. So, once you're logged in on github:

- go to https://github.com/jaillet/MixedQuadTree/
- click on the 'Fork' button
- 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 Getting the program, and Compiling

```
$ git clone /https://github.com/jaillet/MixedQuadTree.git
$ cd MixedQuadTree/
$ git checkout branch develop (this will create a new branch)
$ mkdir build ; cd build
$ cmake ../src -D_CMAKE_BUILD_TYPE=Debug (or Release) // Uppercase is important
```

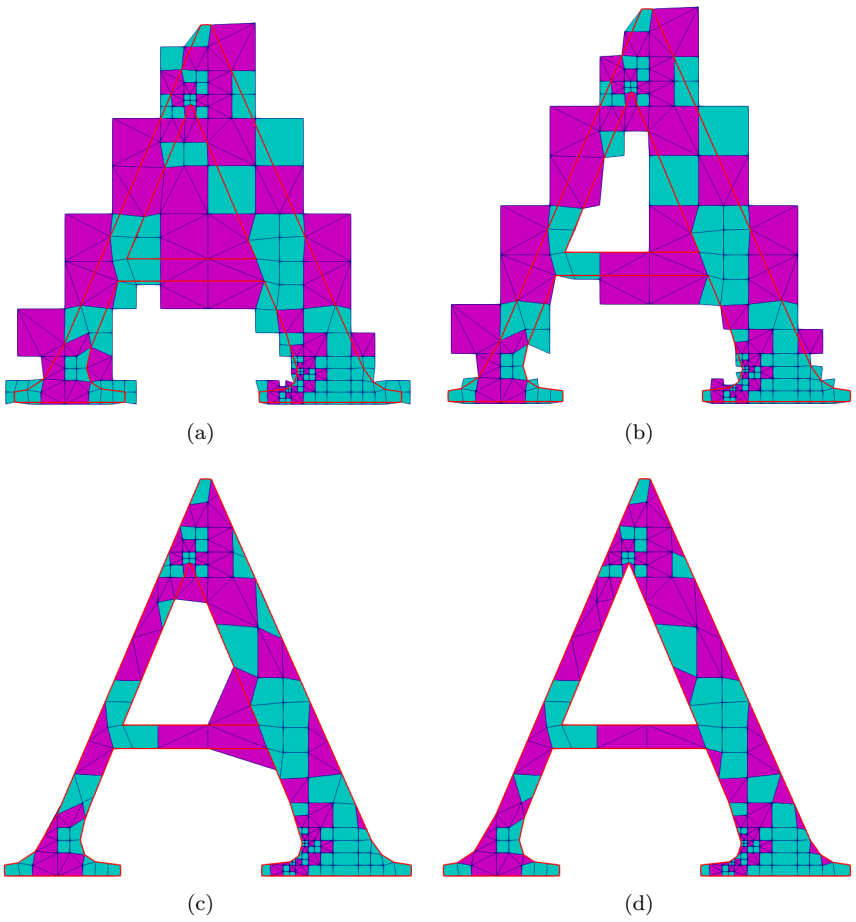


Figure 11: Algorithm 2, level 3. After projecting interior points close to boundary (step 7), removing exterior quadrants (step 8), projection external nodes to boundary (step 9) and finally applying Surface Patterns (step 10).

```
$ cmake .
$ make -j#nproc
```

5.4 Git usage

At this point, you can edit and commit files on you branch using the git workflow.
In order to easily access to the initial MixedQuadTree repository and stay updated with the other developers advances, you can set it as a remote repository, called 'upstreamMixedQuadTree':

```
$ git remote add upstreamMixedQuadTree git@github.com:MixedQuadTree/MixedQuadTree.git
```

Git remotes are great because they allows to have multiple pull/push repositories. You can see the list of your remote repositories under a shell:

```
$ git remote
$ git remote -v
```

At this point, you should have two remote repositories:

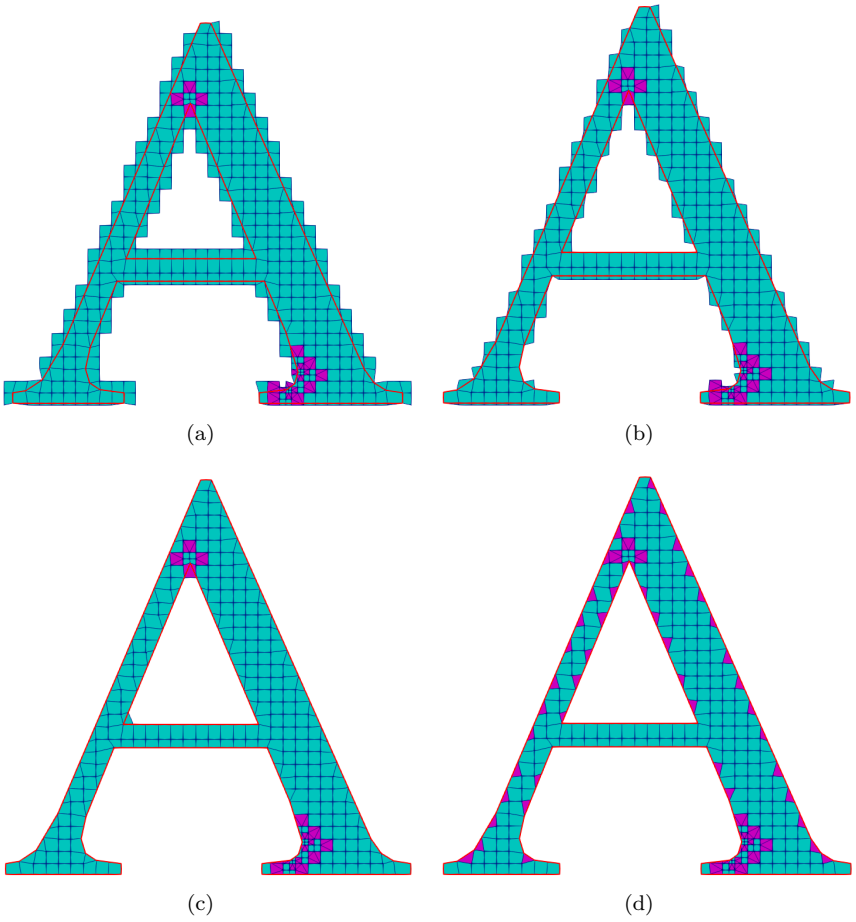


Figure 12: Algorithm 2, level 5. After projecting interior points close to boundary (step 7), removing exterior quadrants (step 8), projection external nodes to boundary (step 9) and finally applying Surface Patterns (step 10).

- origin, your own fork of the initial MixedQuadTree repository
- upstreamMixedQuadTree, the remote repository you have just added containing the initial MixedQuadTree repository

You can see the state of a remote repository called 'remotename' under a shell with the command 'git remote show remotename':

```
$ git remote show origin
$ git remote show upstreamMixedQuadTree (you can use the tab-key for auto-completion)
```

In the normal workflow, you can pull from the main MixedQuadTree repository (upstreamMixedQuadTree)...

```
$ git pull upstreamMixedQuadTree master
```

IMPORTANT NOTE ⇒ but only push to your own fork of MixedQuadTree (origin) !!!

```
$ git push origin master
```

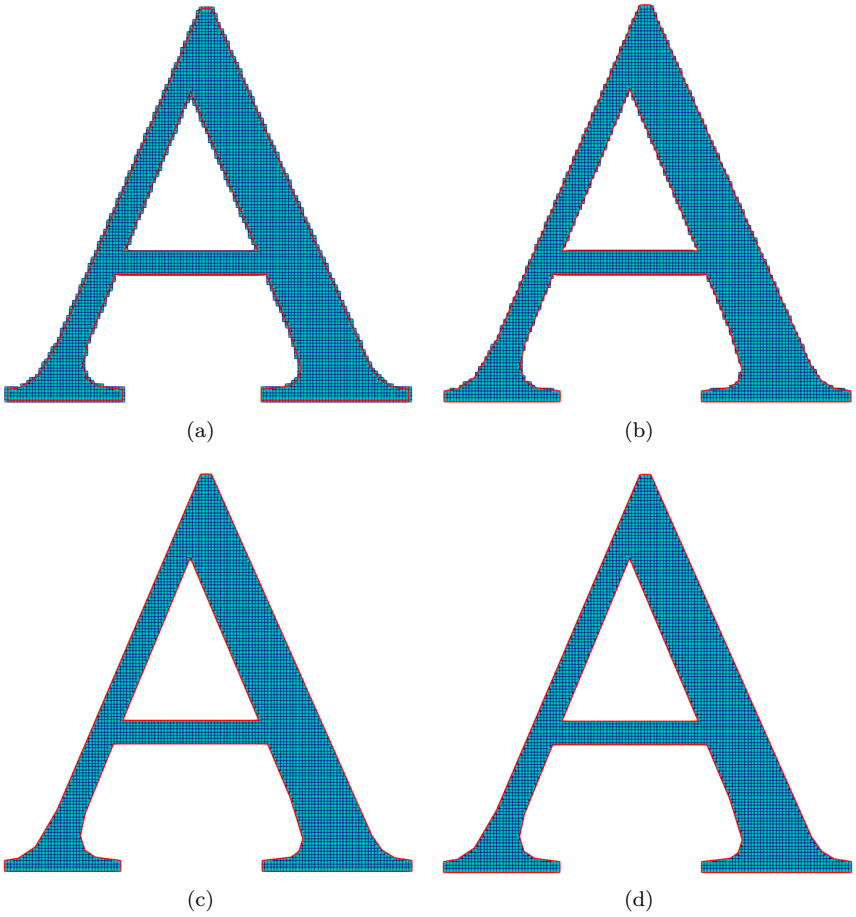


Figure 13: Algorithm 2, level 7. After projecting interior points close to boundary (step 7), removing exterior quadrants (step 8), projection external nodes to boundary (step 9) and finally applying Surface Patterns (step 10).

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 your branch in *origin*), write your comments and click on *'Send Pull Request'*.

6 Standalone program use

6.1 Produce a conformal high-quality mixed–elements mesh

To produce a mesh you must execute in a terminal the following:

```
$ ../mesher_roi [-option | parameters]
```

If you do not indicate any option nor parameters, the program will list all the available options for you. Options for input/output:

```
usage: ./mesher [-p] input.poly [-u] output
                [-s] ref_level [-a] ref_level [-b] file.reg
```

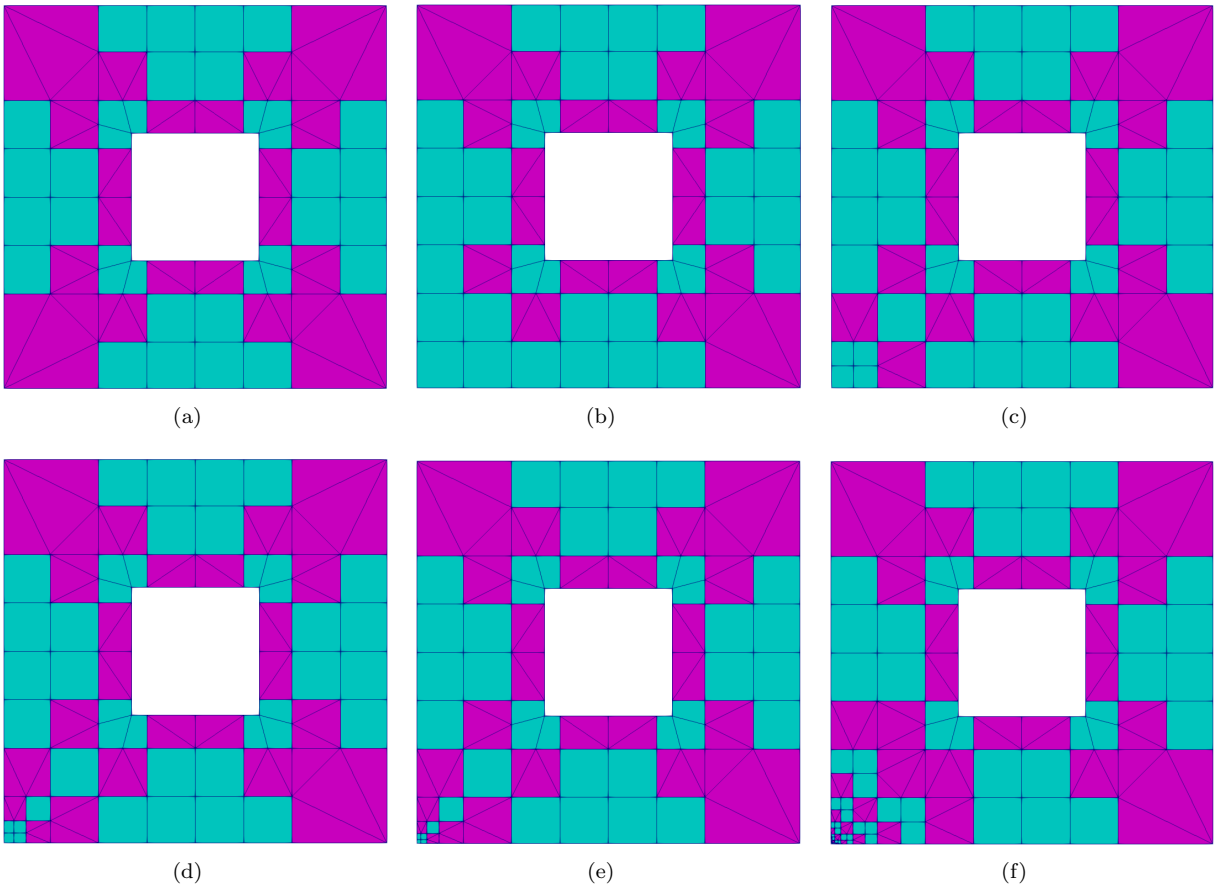
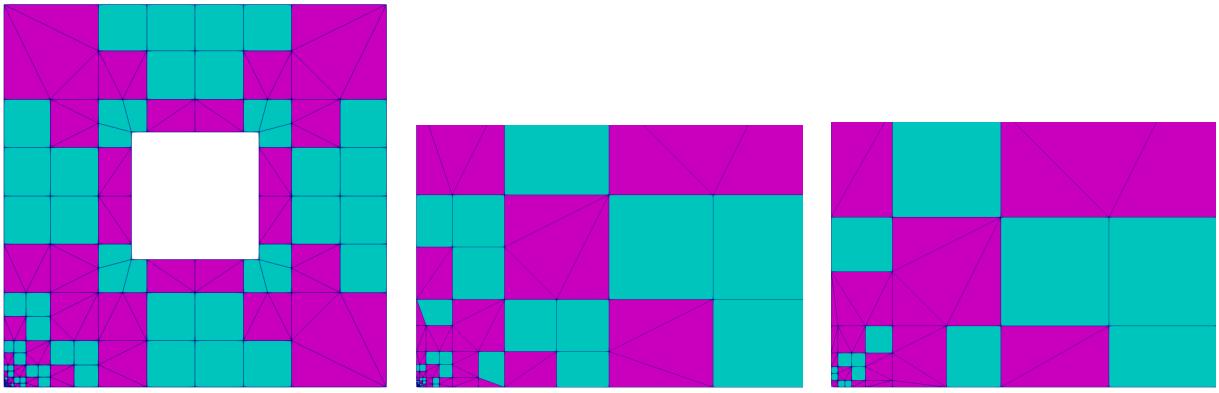


Figure 14: Algorithm 3. With *Box* example, the bottom left element is refined once at each iteration of the algorithm. (a-c) iteration 1 to 5, (d) iteration 8.

```
                [-r] input_surface rl [-option 1] .. [-option n]
where:
  one of the parameters must be an input surface mesh in
  mdl or off format. If output name is not provided it
  will be saved in input_name.m3d. Options:
    -d input surface as .mdl file (alternative to -p).
    -s Refine Quadrants intersecting the input surface.
       Parameter ref_level is the refinement level
    -a Refine all elements in the input domain.
       Parameter ref_level is the refinement level
    -b Refine block regions provided in file file.reg
    -r Refine surface region. Will refine all the elements
       in the provided input_surface at level rl
    -q if supported (only VTK by now), write quality attributes to output file.
    -g save output mesh in GetFem format (gmf)
    -v save output mesh + input in VTK ASCII format (vtk)
    -m save output mesh in M3D ASCII format (m3d)
    -x save output mesh in GMSH ASCII format (gmsh)
    -i save output mesh in MVM ASCII format (mvm)
    -o save output mesh in OFF ASCII format (off)
```



(a) (b) (c)

Figure 15: Algorithm 3. Same *Box* example as Fig. 14, iteration 20. (b) Zoomed once and (b) twice in the bottom left corner.

- `-p`: with the name of an input file (`.poly` extension) where the surface Polyline is specified. In this version of the code, only one input can be specified.
- `-d` input surface as .mdl file (alternative to -p).
- `-u`: next to it comes the name of the output file. The extension (file format) will be automatically added and it can be changed with one (or several) of the next options `-g, -v, -m, -x, -i, -o`. (check if default for name and format!)
- `-q`: add decoration to the VTK file (must be used along with `-v`). Useful for visualization or debugging purpose

Now, to provide the Refinement Level (*rl*) several options can be used (`-a, -s, -b, -r`), see section 2.2 for more details. we can now execute the following to test the meshing process:

```
$ ../mesher_roi -p a.poly -a 2 -s 5 -u testa2s5 -v -q
```

6.2 Advanced mesh refinement

When generating a mesh with the code explained in section 6.1, a quadrant mesh file will be automatically written. Using the *octant* file (extension `.oct`, for compatibility reasons with 3D to be changed??) elements, quadrants and even regions of the generated mesh can be refined. Currently, only quadrants can be subdivided in 4 new quadrants (squares), however, if a particular element must be refined, the *octant* file allows to detect to which quadrant it belongs and refine it. All the refinement options of section 6.1 can be used, and one extra option to list particular elements of the mesh can be employed. we can now execute:

```
$ ../mesher\_roi -p a.poly -c testa2s5.oct -a 4 -l list.txt -u testa2s5ref -v
```

generating the result shown in Figure **??**.
As you noticed, in this example we start from a Quadrant mesh already generated `out.oct`, so the element alignment is conserved as in previous example. We then specify that all the quadrants in the mesh must, at least, count with a refinement level 4 (`-a 4`).
Then we can provide, in a simple text file listing `list.txt`, specific element indexes we want to refine. One element per line. Please note that is *plain text* file only (be sure not to be using *rtf* files which add more information). All the elements will be refined exactly one extra time, therefore it is not necessary to specify a target refinement level. Finally this will generate two outputs: `testa2s5ref.vtk` and `testa2s5ref.oct`. The first is because we specified option `-v` and the last because we will always generate the `.oct` file in case further refinement is required.

7 Using the code from another program

todo for 2D, compile external lib and function call + data structure for information exchange between the mesher and the external program.....

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.