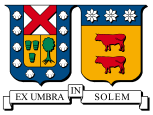
DOCUMENTATION
Mixed–element mesh generator

Version 2017.7.21 ROI–type

Claudio Lobos
clobos@inf.utfsm.cl
Departmento de Informática – UTFSM – Chile.



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 rest will be covered by regular hexahedra.

1 Installing

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

```
$ unzip mesher.zip
$ cmake src/
$ make
```

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

2 Standalone program use

This mesh generator will allow you to create a mixed–element mesh starting from a surface mesh composed of triangles. Let this input surface mesh be Ω. Two constraints must be fulfilled by Ω: it must be unfolded (no self–intersection), and the normal of the triangles must be pointing outside.
The algorithm will use Ω for two purposes: to find out if a point is inside or outside the domain and to project a point onto Ω. Therefore, if you have two different meshes representing the exact same domain, you should use the one with less triangles. The algorithm will compute faster the output volume mesh.
The first step of the algorithm is to compute the Bounding box (Bbox) of Ω. This Bbox will not necessarily be a perfect hexahedron (cube). Therefore, an algorithm will be employed to automatically compute a set of cubes containing Ω. For instance, if Ω is a baseball bat, the starting point will be 3 or 4 cubes, one next to the other.
Now it is possible to introduce the other important input: the Refinement Level (rl). This mesh generator is based on the Octree technique [9, 8]. This technique recursively split an Octant in a finite number of equivalent sons. For instance if the Octant is a cube, to refine it one level means that it will be replaced by 8 new Octants. All of them will be sons of the replaced Octant in the Octree structure. Note that in some works this splitting operation produces 27 new cubes each time an Octant is refined [7, 10, 3]. In our case, Octants will continue to refine until a maximum provided level is reached. Octants lying completely outside Ω will be removed.
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:

- `-d`: next to it comes the name of an input file (`mdl` extension) where the surface mesh is specified. In this version of the code only one input surface can be specified.
- `-o`: another option to specify an input: `off` format file.
- `-u`: next to it comes the name of the output file. The extension (file format) will be automatically added: `mvm` (Mixed Volumen Mesh).
- `-g`: save the ouput mesh in GETFEM format: `gmf` extenstion (Getfem Mesh Format). Note you can still specify an output file name with option `-u`, *e.g.*: `-u myoutput -g` will produce output file: `myoutput.gmf`.
- `-v`: equivalent to option `-g`, but saves in VTK – ASCII format.

Now, to provide the Refinement Level (rl) several options can be used:

- `-a`: next to it comes the number of times (N) the splitting operation will be performed over the initial octants enclosing Ω. For instance if we type: `-a 2` this will split all the intial octants into eight new ones and each one of them in eight more.
- `-s`: next to it comes the number of times (N) the splitting operation will be performed over each octant that intersects a section of Ω. For instance if we type: `-s 2` this will split all the intial octants into eight new ones and then, only the octants that still intersect Ω will be split in eight more.
- `-b`: next to it comes 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 rl to be applyed over the octants 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 10
5

0 0 0
20 20 20
4
```

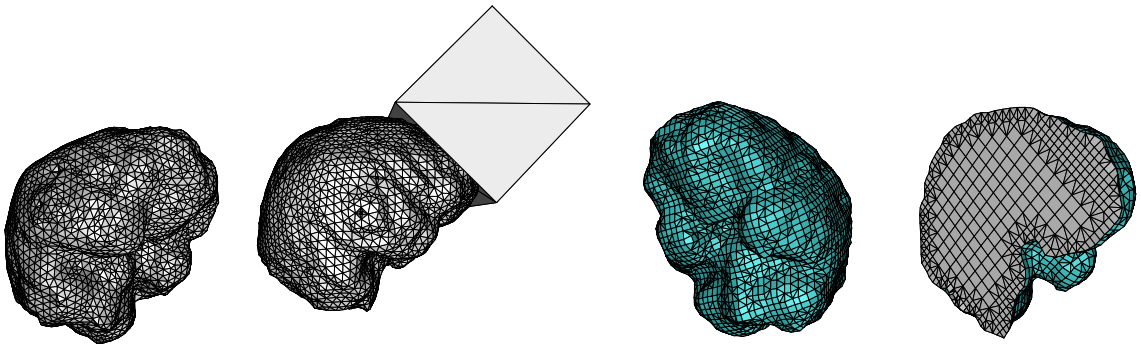
In this case the portion of Ω intersecting the hexahedron $(0,0,0) \rightarrow (10,10,10)$ will be refined to level 5 and the complement inside the block $(0,0,0) \rightarrow (20,20,20)$ that intersects Ω will be refined to level 4.

Important Note: if an octant belongs within more than one refinement region (no matter the type of it), it will be refined to maximum level among those regions.

- `-r`: next to it comes 2 arguments; the name of file where another input surface is specified in `mdl` format and a *rl*. All the octants inside Ω and the specified region will be refined to *rl*. Note that this geometry can be any closed triangle mesh, for instance a tetrahedron. Also note that mesh generation time will be directly influenced by the number of triangle faces the input meshes have. In order to accelerate the process you should use as less triangles as possible that correctly define the input domains and regions. Let us say that the mesh called `myregion.mdl` intersects a portion of Ω, then the instruction: `-r myregion.mdl 5` will refine to level 5 all the octants in that region. The last point regarding this option is automatic rotation. The code will compute the "dominant normals" of the input region and will try to align them with referential axes. This transformation will also be applied to Ω. Once the mesh is generated, the inverse transformation will be applied to the output so the domain is well represented.

Finally, note that "big" meshes can be produced with a small value for N. For instance, if the starting point is only one cube and we set $N = 9$, the code may produce up to $8^9 = 134,217,728$ elements.
Let illustrate all this with the followig example: `./mesher_roi -o cortex.off -r reg.mdl 5 -s 5 -a 4 -u out -g`. Meaning that Ω is defined in input file `cortex.off` in `off` format, then another surface that intersects a portion of Ω is defined in file `reg.mdl` and the octants in that region must be refined to level

5. At the same time, every octant intersecting the surface must also be refined to level 5 and all the rest of the octants to level 4. Finally output must be written to file `out.gmf` in GetFem mesh format. The inputs and outputs of this execution can be seen in Figure 1.



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

Figure 1: An example with different refinement levels: (a) input domain Ω, (b) Ω and region of refinement specified by another surface mesh, (c) the surface of the output mesh and (d) the same output with a saggital cut where the alignement in the top front section of the brain is perfect.

3 Using the code from another program

In order to generate a volume mesh, we only need a list of nodes, a list of triangular faces and the refinement level. This section explains how to interact with the code if you want to program your own application. For example, this is useful if you have a surface mesh that is not in the proper file format, or you need to write the output in another file format.
File `Main.cpp` is in charge of input reading, executing the mesh generator with proper parameters and write the output. In order to specify an input surface mesh, we need to create a `TriMesh` object. This object is composed of two vectors: one with the points and the other with triangles. The Points are specified through instances of class Point3D as follow:

```
1. vector<Point3D> points; //the vector of points
2. points.reserve(1);      //re-size the vector if you know the quantity of nodes
3. double x=0, y=0, z=0;   //define the coordinates of a point
4. Point3D p(x,y,z);       //create a point
5. points.push_back(p);   //add the point to the vector
```

Now for the faces and create a surface mesh: `TriMesh` object with both vectors (points and faces).

```
1. vector<vector<unsigned int> > faces; //the vector of faces
2. faces.reserve(1);                    //re-size the vector
3. vector<unsigned int> a_face(3,0);    //define a face with 3 nodes
4. a_face[1] = 1; a_face[2] = 2;        //update references
5. faces.push_back(a_face);             //add the point to the vector
6. TriMesh tm1(points,faces); //create a surface mesh
```

Provide the maximum refinement level among all the options requiered to generate the output mesh. This value enables important optimization in time.

```
1. list<RefinementRegion *> all_regions;
2. RefinementRegion *rr = new RefinementAllRegion(4);
```

```
3. all_regions.push_back(rr);
4. Mesher mesher;
5. unsigned int max_ref_level = 4;
6. FEMesh output = mesher.generateMesh(tm1,ref_level,out_name,all_regions);
7. vector<Point3D> output_pts = output.getPoints();
8. vector<vector<unsigned int> > output_elem = output.getElements();
9. cout <<''X: ''<<output_pts[0][0]<<'' Y: ''<<output_pts[0][1]<<''X: ''<<output_pts[0][2]<<'\n';
```

Now you can handle the points of the output mesh as shown at line 9 of the previous code: `output_pts[n]` will give you access to point `n` and then you can access its coordinates with another vector accesss. For instance, coordinate `z` of point `n` is at `output_pts[n][2]`. As for the elements, they are in the vector. Each component of the vector is a new element where the index of the nodes defining the element are in another vector. For instance the second node of the first element is at `output_elem[0][1]`. The notation (order of the nodes index) for each tipe of element is defined in section 4.
Please see class `Main.cpp` in the source files to known how to define the other types of regions.

4 Element node conventions and mvm extension

The Mixed Volume Mesh format allows to easily describe a mesh composed of different element types. It will be extended later, but for the moment it can manage the four basic element types generated by the meshing technique.

```
MIXED
6 2

-1  1 -1
-1  1  1
 1  1  1
 1  1 -1
 0  3  0
 3  1  0

P 0 1 2 3 4
T 2 3 4 5
```

This example of the `mvm` format shows the sintaxis. First goes the header "MIXED", then the number of nodes and the number of elements in the mesh. After that there is the list of coordinates X, Y and Z for each node. The first node has index 0 and so on until the last one with index: number of nodes -1 (in the example the last has index 11). And finally there is the list of elements. The description of an element starts with a string. The four basic employs: `H` for the hexahedron, `R` for the prism, `P` for the pyramid and `T` for the tetrahedron.

Important Note: some elements employ more than one character to specify its type. For instance, `P6` corresponds to a pentagonal base pyramid (having in total 6 nodes).

The four basic element types considered in this work are shown in Figure 2. For instance if we define a prism (wedge) with the following indexes: 11 20 19 0 8 4, it means that one triangular face t, is defined by indexes: 11, 20 and 19. Moreover, nodes 0, 8 and 4 will have a positive distance w.r.t. to t, when its normal is computed counterclockwise.

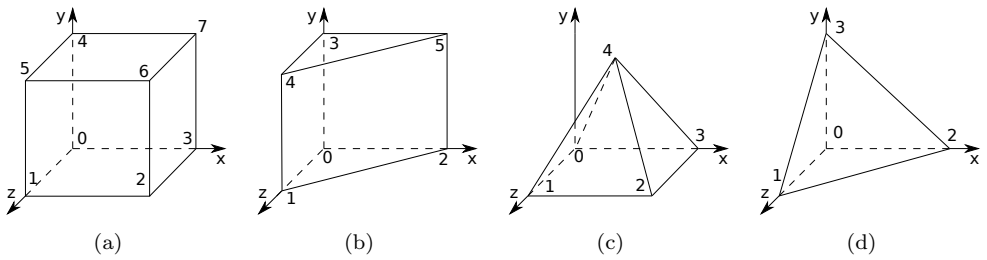


Figure 2: Basic elements: (a) hexahedron, (b) prism (wedge), (c) pyramid and (d) tetrahedron.

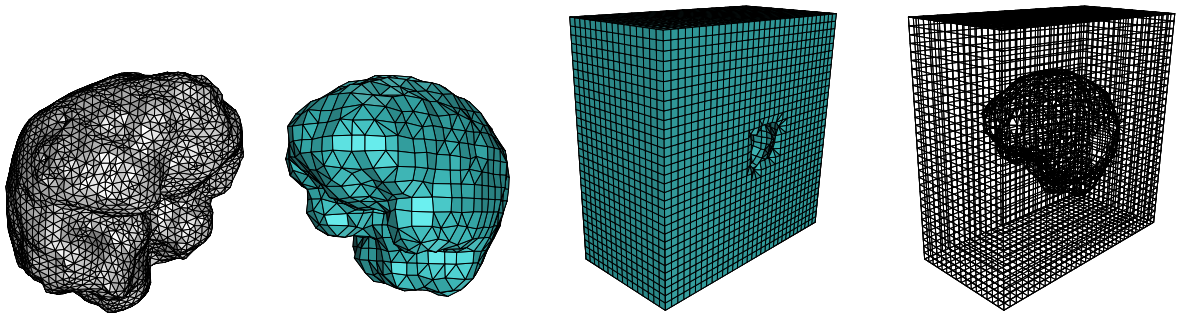
The vector `output_elem` (defined in setction 3) will contain the entire set of elements. Each component of this vector specifies a unique element with another vector.

5 Troubleshooting and FAQ

This section try to explain common errors and how to solve them.

5.1 Inversed normal orientation at intput

This is one of the most common errors. If the input has inverted normal orientation, i.e., normals pointing towards inside the domain, the result will be an hexahedron, and inside of it, the domain will be represented as a void space.



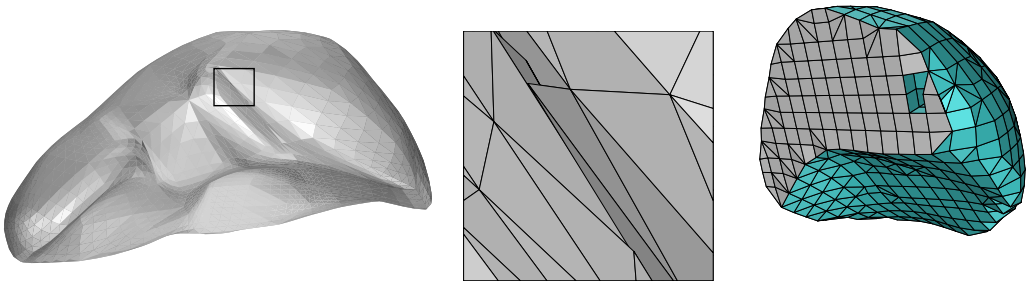
(a) (b) (c) (d)

Figure 3: Inverted normals example.

If we take as input Figure 3(a), we would expect to have as result something like Figure 3(b). However, if we obtain something like Figure 3(c) (or Figure 3(d) which is the wireframe version of it), it means that the input has inverted normals. The solution is to invert each triangle of the input. For instance if we consider triangle $0-1-2$ (defined by node index), we should replace it with triangle $0-2-1$.

5.2 Self intersection of the input

When the input has some inverted triangles, or it self–intersects, it is very likely that the test employed for knowing if we are inside or outside will fail. Even Though the code will succeed in producing a mesh, it will probably count with holes left by "intern outside" octants. An example of this is shown in Figure 4.

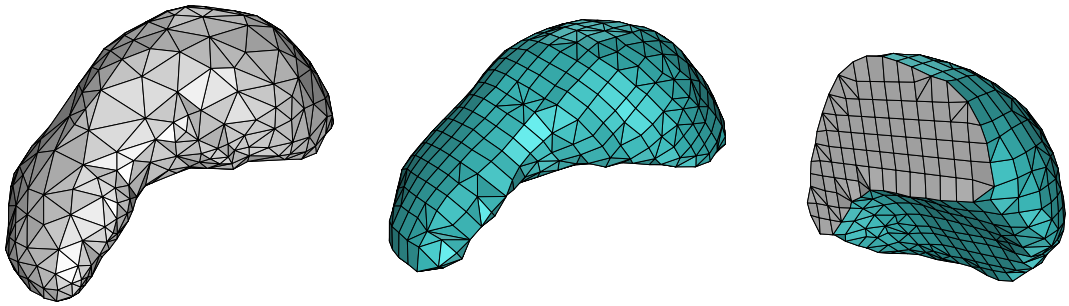


(a) (b) (c)

Figure 4: Input self intersection example.

Figure 4(a) shows an input describing a liver. If we zoom to the region shown in a square at Figure 4(a), we will see Figure 4(b) where some triangles intersect. The output of the algorithm can be seen in Figure 4(c). In this picture we see a sagittal cut of the output where some elements are missing.

There are several options to overcome these type of issues. One is described in this work [6], where with the use of CGAL, the domain is re-meshed. However this may increase error of approximation: if we consider the input mesh to be an approximation, using CGAL would result in an approximation of the approximation.



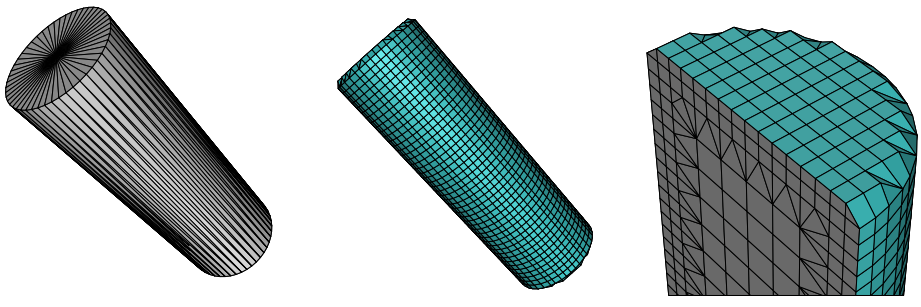
(a) (b) (c)

Figure 5: Input self intersection: solved with input change.

Figure 5(a) shows the output of CGAL using as input the liver of Figure 4(a). This new output does not count with inverted triangles nor self-intersecting regions. Figure 5(b) shows the volume mesh generated using the mesh of Figure 5(a) as input. Finally, Figure shows the same sagittal cut of the output where some elements were missing in Figure 4(c), only this time they are not.

5.3 Poor border representation of sharp features.

The focus of this work is to be able to produce adaptive hex–dominant meshes. So far, mixed–elements are employed to manage transitions between fine and coarse regions and at the boundary of the domain in order to better represent curved domains as explained in [4].
Therefore, this implementation is well prepared for generating meshes when the curves of the domain are smooth. If the domain has sharp features, then this implementation will fail in representing them. An example of this can be seen in Figure 6(a), which describes a simple cylinder. The output mesh is shown in Figure 6(b) where all surface elements were refined to level 4. Figure 6(c) shows a saggital cut and zooms to the top of the output cylinder. It is clear the lack of representation at domain's edges: an angle of 90° or 270° between neighbor faces of the input domain.



(a) (b) (c)

Figure 6: Problem of sharp features representation.

Unfortunately, there is no solution for this issue in the current state of the code. We are already working in

considering sharp features for future versions of the algorithm.

5.4 How can I get a mesh of 4000 nodes?

The algorithm is not prepared to produce a mesh of x nodes. The problem can be established as: the difference of node quantity between refinement level n and $n+1$ is too large. How can I gain more control over node quantity if I want something in the middle of 2 refinement levels?
There is no direct solution to this problem, however, one practical solution is to artificially increase the size of domain's Bounding Box (Bbox). Let us say that input geometry is defined between nodes: $\begin{bmatrix}\pm a & \pm a & \pm a\end{bmatrix}$ and it count with m nodes. The way to artificially increase its Bbox is by modifying the input file to have $m+1$ nodes. This new node will be used by no element, therefore, it should be the last node of the list and it coordinates should be, for instance, $\begin{bmatrix}a+b & a+b & a+b\end{bmatrix}$. With this, the size of the Bbox should no longer be $(2a)^3$, but $(2a+b)^3$. If we now restart the program we should obtain more coarse elements for the same level of refinement.
Once more, there is no proper solution. You will have to try with different sizes of b to get the right node density for your simulation problem.

6 About implementation and embedded algorithms

This implementation is based on the Octree technique and it employs the 3D *Cohen–Sutherland* clipping algorithm [2, p. 113]. This allows fast computation of cube–triangle intersection, which is one of the most expensive functions needed by the Octree technique.
When no intersection is detected there are two options: the Octant is completely inside or completely outside the domain. For this we use the Signed Distance algorithm introduced here [1] and it was implemented by Vincent Luboz. Now we consider only one node of the Octant and the set of faces intersected by its parent. If the node is outside, then the entire Octant is removed, otherwise is no longer analyzed for intersections and it is directly split whenever needed. This is an important optimization.

7 Citing this work

Please cite this work with [5]:

```
@ARTICLE{Lobos2015a,
  author = {C. Lobos and E. Gonzlez},
  title = {Mixed-element Octree: a meshing technique toward fast and real-time simulations in biomedical
   applications},
  journal = {International Journal for Numerical Methods in Biomedical Engineering},
  volume = {31},
  number = {12},
  year = {2015},
  pages = {1--31},
  doi = {10.1002/cnm.2725},
  timestamp = {2016.01.06}
}
```


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 and Fabrice Jaillet.
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.

References

[1] J.A Baerentzen and H. Aanaes. Signed distance computation using the angle weighted pseudonormal. *Visualization and Computer Graphics, IEEE Transactions on*, 11(3):243–253, May 2005.

[2] J. D. Foley. *Computer graphics: principles and practice*. Addison–Wesley Professional, 1996.

[3] Y. Ito, A. Shih, and B. Soni. Octree-based reasonable-quality hexahedral mesh generation using a new set of refinement templates. *International Journal for Numerical Methods in Engineering*, 77(13):1809–1833, March 2009.

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

[5] C. Lobos and E. Gonzlez. 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, 2015.

[6] C. Lobos and R. Rojas-Moraleda. From segmented medical images to surface and volume meshes, using existing tools and algorithms. In *Proceedings of ADMOS*, pages 436–447, 2013.

[7] R. Schneiders. Refining quadrilateral and hexahedral element meshes. In *Proceedings of the Fifth International Conference on Numerical Grid Generation in Computational Field Simulations*, pages 679–688, April 1996.

[8] M. Shephard and M. Georges. Automatic three-dimensional mesh generation by the finite octree technique. *International Journal for Numerical Methods in Engineering*, 32:709–749, 1991.

[9] M. A. Yerry and M. S. Shephard. Automatic three–dimensional mesh generation by the modified–octree technique. *International Journal for Numerical Methods in Engineering*, 20(11):1965–1990, 1984.

[10] Y. Zhang and C. Bajaj. Adaptive and quality quadrilateral/hexahedral meshing from volumetric data. *Computer Methods in Applied Mechanics and Engineering*, 195(9–12):942–960, February 2006.