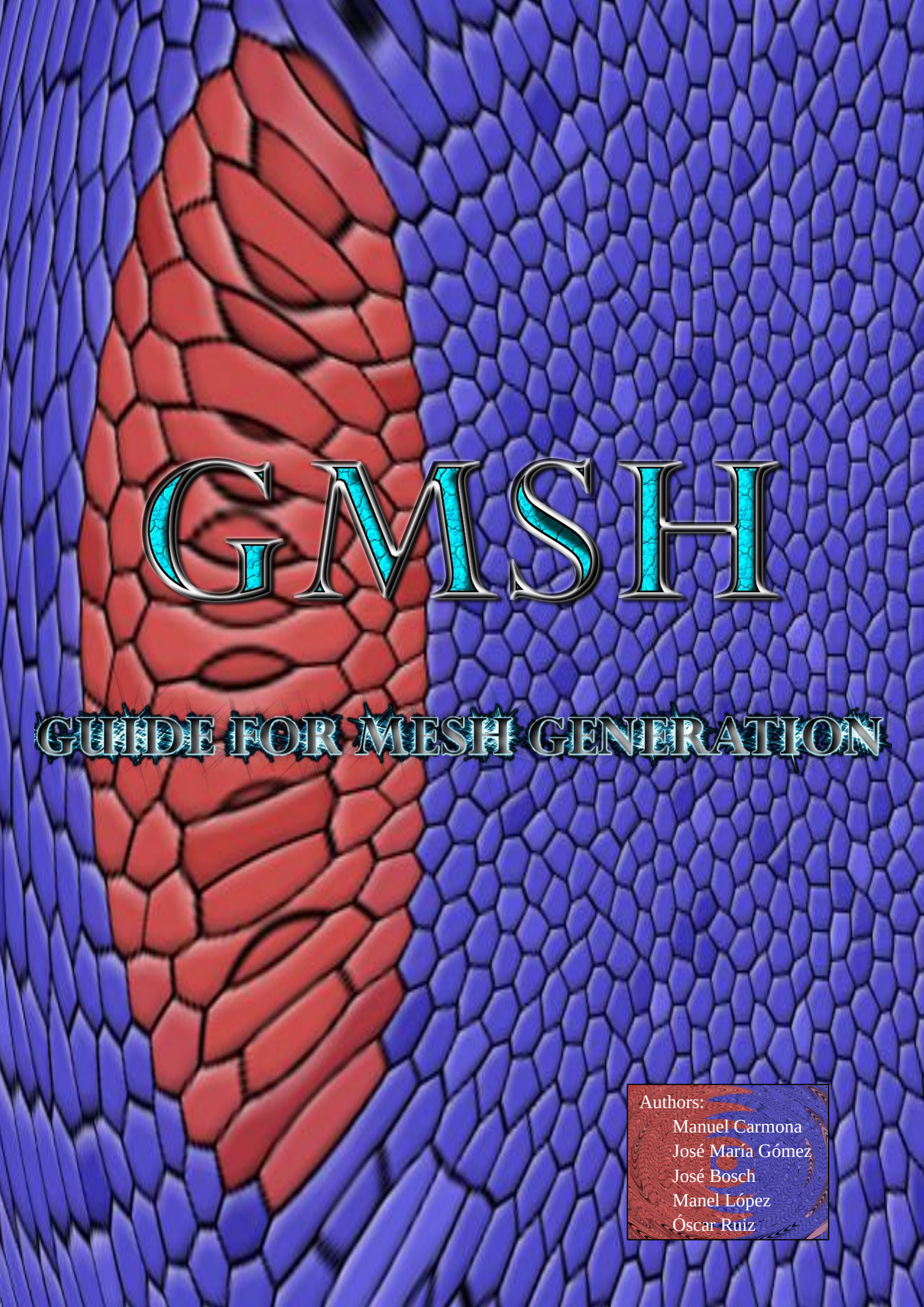

Authors:
Manuel Carmona
José María Gómez
José Bosch
Manel López
Óscar Ruiz

November/2019

Barcelona, Spain.

Table of contents

I. Introduction to GMSH

GMSH [1] is a freeware software distributed under the terms of the GNU General Public License (GPL). It is mainly a mesh generator, provided with a CAD (Computer-Aided Design) engine and a post-processor tool. Although having a graphical interface, one of its strong points is that it accepts a parametric input script for the mesh generation. As an inconvenient, it is not good when trying to define complicated geometries (for example, nowadays it is still not able to perform geometric boolean operations, which are generally quite useful). Indeed, for complex geometries, the preferred method is to use an external CAD software for generating the geometry and afterwards importing it into GMSH. Another inconvenient shows up when creating a large model because, by default, when a new entity is generated, GMSH checks the overlapping of all entities. When there is a large number of entities, these checking results in an extremely slow model generation.

As a general procedure for mesh generators (and also for GMSH), first we have to generate geometrical entities (points, lines, areas and volumes; in this order). In GMSH there are also physical groups, that are simply and basically a group of geometrical entities (points, lines, surfaces or volumes). Once the geometry is defined, we have to specify (as far as possible) how the FEM elements have to be generated. And finally, we order to mesh them (i.e., generation of the FE nodes and elements). With GMSH we can generate some simple geometries. They are currently improving this aspect, but at the time of this publication, only some limited geometry operations are possible.

FEM models can be generated graphically (through the GUI of GMSH), or by use of scripts (a file containing the instructions for the mesh generation).

GMSH scripts have the extension '*.geo*'. In these scripts, we can define all the instructions than can also be input graphically. The use of these scripts has many advantages:

- We can re-use it for other models.
- We can parameterize the model, in order to be able to change dimensions or properties without the need to start from zero.
- Portability is better. The script file occupies some kbytes, while a model can be quite large to back it up. The only disadvantage is that it has to be re-run to get the model.

In this guide, it will be explained the language and options for creating scripts for GMSH. This guide does not intend to substitute the GMSH guide provided with the software. It just intends to explain the basics of GMSH scripts generation, illustrating them with examples for a better understanding. The use of the GUI should be quite straightforward, once we know what every option means. Moreover, the GUI interface will be better developed at lab sessions.

II. Introduction to GMSH Commands

First of all, we have to take into account that some of the syntax is similar to the C language (and therefore, also to Java). This also means that, generally, commands are ended with a semicolon (';') and that expressions are case sensitive. Also, indexes (for vectors, for example) will start from 0.

Comments follow the same format than in the C language:

- // for a line comment.
- /* ... */ for a multiple-lines comment.

Regarding the types of variables, we can only find two different types: real and string. Their syntax is also similar to the C language. We can highlight some keypoints:

- *#vector[]* provides the number of elements in the vector (or list).
- To ask the user for a real value (and having also a default value), we can use the function *GetValue*. Example:

  lc = GetValue("Enter the characteristic length for the mesh: ", default_value);

- Similarly, to ask the user for a string, we can use the function *GetString*.
- For a parameter to be changed in the GUI of GMSH, we can use the section DefineNumber. Example:

  hm = DefineNumber[1e-4, Name "Parameters/hm", Min 9e-5, Max 1.1e-4, Step 5];

We can also highlight:

- The instruction *Point {Point_id}* provides the coordinates of a given point.
- *Point "*"* provides the id's of all the point entities. The same for *Line*, *Surface* and *Volume*.
- *Boundary {entities}* returns the entities that are the boundaries of *entities*. For a volume, it would return the list of surfaces that constitute its boundaries.
- For obtaining the last number+1 associated to an entity, we can use the commands: *newp* (point), *newl* (line), *newc* (curve), *news* (surface), *newv* (volume), *newll* (line loop), *newsl* (surface loop), *newreg* (any entities different than point).
- We can get the entities contained in a given volume region (defined by the coordinates of two points). Example for surfaces: *Surface In BoundingBox {xmin, ymin, zmin, xmax, ymax, zmax}*.
  We can also get the bounding box delimited by a geometrical entity. Example for a volume: *BoundingBox Volume {v1}*.

Different expressions can be given separated by commas, and grouped by '{ }'.

There are some functions for strings: *StrCat* (concatenate two strings), *Sprintf* (creates a string like 'printf' would print in the C function), *GetString* (ask for a string to the user). *~{expression}* appended to a string will add expression to the string by an underscore.

It is only possible to generate 1D vectors. Nevertheless, with the last previous command (*~{expression}*) we can add dimensions to a vector. This command can be added at the end of a string (variable in our case) more than once. Example for a matrix: vector~{row}[column]=1;

For printing in the text windows, we can use the command *Printf*, which have a similar format to C. Example: *Printf("Length: %g, Width: %g, Thickness: %g)", L,W,t);*

Operators are like in C, with '^' for exponentiation. It allows the ternary operator ': ?' (reduced if).

Built-in functions are: *Acos* (returns a value between 0 and Pi), *Asin*, *Atan* ,*Atan2*, *Ceil*, *Cos*, *Cosh*, *Exp*, *Fabs*, *Fmod*, *Floor*, *Hypot*, *Log*, *Log10*, *Modulo*, *Rand*, *Sqrt*, *Sin*, *Sinh*, *Tan*, *Tanh*.

We can define functions, but they are like macros, with no arguments. At the place where it is called, it is literally substituted by the function definition. They are defined with:

Function name
    Body of the function;
Return

These functions are called with the command Call: *Call name;*

Macros can be generated in another file and included in the current file using *Include filename;*.

':' is used to obtain a range of values with a specific increment (default to 1).

We can also build for-loops. There are two possible implementations:

- *For (expression : expression : expression)*: The last expression means the increment, and it is optional (the default value is 1).
- *For variable In {expression : expression : expression}*: In this case, *variable* gets the values of the list after *In*.

Both for-loops finish with *EndFor*.

And finally, we can also build the conditional 'if': *If (expression) ... ElseIf (expression) ... Else ... EndIf*

To start a void list: *string[] = {};*

III. Commands for geometry generation

In the last GMSH versions, it includes two different kernels for generating geometries. These two kernels are: Built-in and OpenCASCADE. The Built-in kernel is the original kernel implemented in GMSH for this purpose. The OpenCASCADE kernel was incorporated in the last years, with the advantage of supporting geometrical 'boolean' operations. Although we could use both kernels for the generation of geometries, it seems that there is not yet a good compatibility between them (not all operations in one kernel can be applied to geometries generated with the other kernel). Indeed, GMSH states that there is no translation of geometry representations from one kernel to the other.

In order to select a specific kernel, we have to use the function *SetFactory(kernel)*. The kernel parameter is a string with "Built-in" or "OpenCASCADE".

III.1. GMSH commands (with Built-in kernel)

Geometries are generated by creating first points, then lines, afterwards surfaces and, finally volumes. These are called elementary geometries. Each one has assigned an identifier (a number) at the moment they are created.

Groups of elementary geometries can be formed. They are called physical entities. They have also an identifier. Nevertheless, they cannot be modified by geometry commands. For later use in Elmer, it is important to define physical entities. They will be numbered accordingly.

For generating surfaces, first we have to generate what is called 'line loops' in GMSH. They are simply a number of lines that form a closed path (and therefore, they delimitate an area).

Let's see examples of how these entities are generated.

Points:

Example: *Point(1)={x,y,z,lc};*

Point with id 1 is created at position *x,y,z*. *lc* is an optional parameter, indicating the size of the elements near this point when these elements are generated.

Example for a group of points: *Physical Point(5)={1,2,3,4};*

Lines:

There are some different types of lines. The most basic is the straight line (Line).

Example: *Line(1)={1,2};*

A straight line with id 1 is created between points 1 and 2. A direction is also defined from point 1 to point 2 in this case. This is considered the positive direction of the line.

For creating surfaces, we need to create previously a line loop.

Example: *Line Loop(5)={1,2,-3,-4};*

In this case a line loop with id 1 is created, and it is defined by lines with ids 1, 2, 3 and 4. The line loop follows a direction that has to coincide with the indicated line directions. If one line has opposite direction, a negative value for the line has to be indicated. This also defines an orientation of the surface.

Other commands for lines are: *Circle*, *Ellipse*, *Spline*, *BSpline*, *Compound Line*, *Physical Line*.

Areas (or Surfaces):

Example: *Plane Surface(1)={2,3};*

A plane surface with id 1 is created by using the line loops 2 and 3. The first line loop defines the external boundaries of the surface. The rest of line loops define holes in this area.

For creating volumes, we have to create surface loops. They have to define a closed volume and with an appropriate orientation of the surfaces. Example: *Surface Loop(1)={1,-2,3,-4,5,-6};*

Other commands related with areas are: *Ruled Surface*, *Compound Surface*, *Physical Surface*.

Volumes:

Example: *Volume(1)={1,2};*

A volume with id 1 is created, based on surface loops 1 and 2. As in surfaces, the first surface loop defines the external boundaries of the volume, and the rest of surface loops define holes.

Other commands are: *Compound Volume*, *Physical Volume*.

Geometrical entities can also be generated from already existing entities. There are two of these processes implemented in GMSH: extrusion and transformations:

Extrusion: This is a very useful method to create geometries and, at the same time, keep a "nice" mesh.

By dragging an entity of lower level, we can generate an entity. For example, dragging a point a certain distance in a certain direction defines a line, a line defines a surface and surface defines a volume. This is valid not only for translations, but also for rotations. Moreover, we can also generate at the same time the mesh of this new generated geometry.

Example: *sout[]=Extrude{{tx,ty,tz},{rx,ry,rz},{px,py,pz},angle} {Line {1,2};};*

The last expression provides the entities that are going to be extruded; in this case, lines 1 and 2 (therefore, we are going to generate surfaces). *{tx,ty,tz}* indicates the translation vector (direction and magnitude of translation), *{rx,ry,rz}* indicates the axis of rotation, *{px,py,pz}* is a point in this axis of rotation and angle is the magnitude of rotation (in radians). If we only want a translation, we only have to take the rest of parameters out of the expression. The same reasoning is valid for only rotation.

This extrusion command returns a list (vector) of id's. The first element is the entity of the same level than the original entity generated at the end of the extrusion. The second element is the higher level entity that has been obtained by the extrusion. The other elements are the rest of entities (as a general rule, they follow the order of the elements from the original entity used to extrude it). If more than one entity is extruded at a time, the items are repeated in the same way.

Transformations: There are some operations, like scaling, that can be also applied to existing entities (or a copy of them → "*Duplicata*" command) for generating new ones. The operations that we have in GMSH are:

Translate: We just need to provide the translation vector (which defines the direction and the magnitude of translation) and the entities to be translated.

Example: *Translate {-0.01, 0, 0} { Point{1}; }*

Rotation: Like for extrusions, we have to provide a vector defining the rotation axis, a point on the rotation axis and a rotation angle (in radians).

Example: *Rotate { { 1,0,0 }, {0,0,0}, Pi/2 } {Surface {1,2,3};}*

Symmetry: We just have to provide the coefficients of the equation defining the symmetry plane ($A \cdot x + B \cdot y + C \cdot z + D = 0$) and the entities to be used.

Example: *Symmetry {1,0,0,0}{Duplicata{Surface{1};}};*

This command generates symmetry copy of the surface 1 with respect to the YZ plane, which passes through the point (0,0,0).

As a reminder, the coefficients *A*, *B* and *C* coincide with the components of a normal vector to the plane. *D* can be obtained with the coordinates of a point in the plane.

Dilatation: Dilatation (or compression) by homothetic transformation (like an image projection from a light source). It requires the position of the transformation point (equivalent to the position of the light source) and a factor (meaning how relatively far the projection is with respect to the distance from object to the light source).

Example: *Dilate { {0 ,0 ,0} , 5 }{ Duplicata { Surface {1,2}; } }*

Creates a bigger version of surfaces 1 and 2, by using the point at (0,0,0) and dilatation factor 5.

For copying also the mesh when making a duplicate of a geometry entity, we should set the option *Geometry.CopyMeshingMethod*. Nevertheless, currently this option only works for transfinite entities.

III.2. OpenCASCADE commands

The OpenCASCADE kernel allows to generate higher level entities (like lines, surfaces and volumes) directly, without previously creating points, lines or surfaces (what is called constructive solid geometry operations). Boolean operations can then be used to modify them.

Lines:

Wire creates a path made out of a collection of lines.

Example: *Wire(1) = {1,2,3};*

Surfaces:

Disk: Creates a circle (with center coordinates and radius) or an ellipse (adding an additional parameter for the y-radius).

Example for a circle: *Disk(1) = {xc,yc,zc, rad};*

Rectangle: Parameters are the coordinates for the lower-left corner, length (x) and width (y). An optional additional parameter indicates the rounding radius at corners.

Example for a rectangle with rounded corners: *Rectangle(1) = {xl,yl,zl,w,h,rround};*

Volumes:

Sphere: Parameters are the coordinates of its center and its radius. Three optional additional parameters indicates three limit angles (minimum and maximum longitude angles (from $-\pi/2$ to $\pi/2$) and latitude (from 0 to $2\cdot\pi$).

Example for a whole sphere: *Sphere(1) = {xc,yc,zc,R};*

Box: Parameters are the coordinates of the lower left corner and the box dimensions (length, width and height).

Example: *Box(1) = {x0,y0,z0,l,w,h};*

Cylinder: Parameters are the coordinates of the center of the circle at one face, vector components defining the axis direction and the radius.

Example: *Cylinder(1) = {xc,yc,zc,vx,vy,vz,R};*

Torus: Parameters are the coordinates of the center of the torus and both radius (defining the major and minor radius respectively). An optional additional parameters indicates the angle extension (from 0 to $2 \cdot \pi$).

Example: *Torus(1) = {xc,yc,zc,r1,r2,ang};*

Cone: Parameters are the coordinates of the circle center of one face, the components of the vector defining its axis (it defines also the length of the cone) and the two radius of both faces. An optional additional parameter indicates the angular extension (from 0 to $2 \cdot \pi$).

Example: *Cone(1) = { xc,yc,zc,vx,vy,vz,r1,r2,ang};*

Wedge: Parameters are the coordinates of the right-angle point and the three dimensions (length, width and thickness). An optional additional parameter defines the extension in x-direction.

Example for a simple wedge: *Wedge(1) = {xc,yc,zc,dxc,dyc,dzc};*

ThruSections: It defines a volume through a set of curve loops.

Example: ThruSections(1) = {lineloops[]};

Ruled ThruSections.

Fillet: It rounds volumes at the indicated surfaces, with defined radius values.

Example: *Fillet{vs[]}{ss[]}{rad[]}*

Chamfer: It cuts borders of volumes at the indicated lines, and by give sizes.

Example: *Chamfer{vs[]}{ls[]}{dist[]}*

Extrusions can be performed using a *Wire* defining the extrusion path.

Boolean operations:

BooleanIntersection:

Example: *v[] = BooleanIntersection {Volume{1};Delete;}{Volume{2};Delete;}*

BooleanUnion:

Example: *v[] = BooleanUnion {Volume{1};Delete;}{Volume{2};Delete;}*

BooleanDifference:

Example: *v[] = BooleanDifference {Volume{1};Delete;}{Volume{2};Delete;}*

BooleanFragments:

Example: *surfs[] = BooleanFragments{Volume{1}; Delete;}{Surface{1};Delete;};*

IV. Commands for meshing the geometry

After having defined a geometry, we should specify how the mesh has to be generated. We have different tools for this purpose in GMSH.

First of all, GMSH has three meshing algorithms: MeshAdapt, Delaunay and Frontal. In principle we do not have to worry on this, as GMSH uses a default algorithm depending on the geometry. Nevertheless, we can force GMSH to use a specific algorithm. As a general rule, MeshAdapt is good for complex surfaces, Frontal for obtaining good quality meshes and Delaunay is a fast algorithm (also appropriate for large meshes).

As it is general for meshes, it is distinguished between structured meshes and unstructured meshes. There is also the possibility of having the combination of both, called hybrid. Structured meshes are "ordered" (or "regular") meshes, while unstructured meshes are the opposite of the structured meshes. A simple example is illustrated in Figure 1.

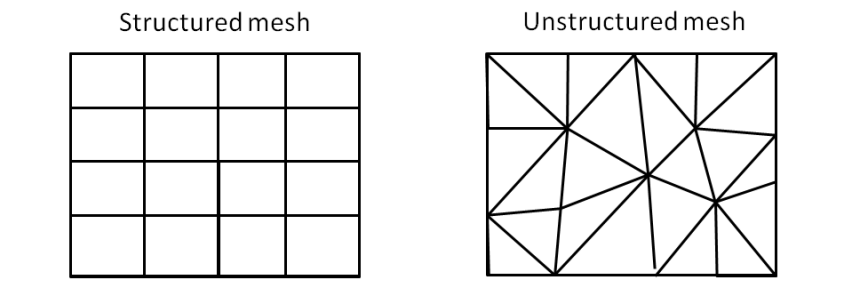

***Figure 1*. Structured (left) and unstructured (right) meshes.**

For generating structured meshes, we have basically two different methods: by extrusion of the geometry (together with the mesh) and the transfinite method (together with 'recombine').

Additionally, all meshes can be subdivided for generating all-quadrangular or all-hexahedra meshes by using the *Mesh.SubdivisionAlgorithm* option (0=none, 1=all quadrangles, 2=all hexahedra).

Therefore, mesh commands are issued for two purposes: defining the size of the elements and defining parameters for the structured mesh.

The size of the elements can be specified in three different ways:

- When creating points (as already seen before). This is the default option. In this case, the option *Mesh.CharacteristicLengthFromPoints* is set. It uses interpolation for the different points for creating the initial mesh. The characteristic element length for points can also be given with the command: *Characteristic Length {list of points} = lc*;
- If *Mesh.CharacteristicLengthFromCurvature* is set, the mesh is generated depending on the curvature of the lines.
- Specifying mesh size fields. Fields are values depending at least on position for indicating distributions of mesh densities. These fields are created with: *Field[id number]=type;* (*type* is the string defining the field and *id* is a number that we associate to this field). To modify

the different options for this field, it is used: *Field[id number].option=value;*. To specify which field will be used, there is the command: *Background Field=id number;*.

Some of these fields are:

- *Attractor*: Used to calculate at each location the minimum distance to the given points and/or lines and/or surfaces. Other fields can use this field as input.
  Options: *NodesList*, *EdgesList*, *FacesList*, *NNodesByEdge*, *FieldX*, *FieldY*, *FieldZ*.
- *Box*: Provides a value (*VIn*) inside a region (box) and another value (*VOut*) outside this region.
  Options: *VIn*, *VOut*, *XMax*, *XMin*, *YMax*, *YMin*, *ZMax*, *ZMin*.
- *Threshold*: Based on the distances calculated from other fields (*IField*) (like *Attractor*), determines a size of elements (*LcMin*) for distances under a given value (*DistMin*), another size (*LcMax*) for distance over another value (*DistMax*), and interpolated sizes in-between these distances.
  Options: *IField*, *DistMax*, *DistMin*, *LcMax*, *LcMin*, *Sigmoid*, *StopAtDistMax*.
- *MathEval*: Sizes depends on a mathematical function (*F*) that can depend on position variables (*x*, *y*, *z*), fields values (*F1*, *F2*, ...) and mathematical functions.
  Options: *F*.
- *BoundaryLayer*: In fluidics, meshes are usually generated quite fine near walls, and less fine far from walls and in a very regular way. This field provides values that are increasing as we are far from specified geometric entities. It follows the relation: $h_{wall} \cdot ratio^{\frac{dist}{hwall}}$.
  Options: *hwall_n*, *hwall_t*, *ratio*, *thickness*, *EdgesList*, *FacesList*, *NodesList*, *Quads*, *hfar*, *FanNodesList*, *FansList*, *AnisoMax*, *IntersectMetrics*.
- *Min*: Provides the minimum value of a list of fields.
  Options: *FieldsList*.
- *PostView*: Element sizes at a location will be determined by the 'solution' values given on nodes (of a certain 'background mesh') near this location.
  Options: *CropNegativeValues*, *IView*.
- Others: *Restrict*, *Max*, *Min*, *Mean*, *CenterLine*, *Gradient*, *Structured*, *Param*, *Curvature*, *Sphere*, *Cylinder*, *Frustum*, *AttractorAnisoCurve*, *MathEvalAniso*, *MinAniso*, *IntersectAniso*, *Laplacian*, *MaxEigenHessian*, *LonLat*, etc.

All three methods can be used at the same time. In this case, the minimum value (the most restrictive) of them is used.

We can highlight some of the GMSH mesh options that can be defined (some of them already mentioned and most of them are self-explicative):

- *Mesh.Algorithm*.
- *Mesh.CharacteristicLengthFromPoints*.
- *Mesh.CharacteristicLengthFromCurvature*.
- *Mesh.CharacteristicLengthExtendFromBoundary*: By default, the element sizes indicated in boundaries are then "extrapolated" to entities they define.
- *Mesh.CharacteristicLengthFactor*: Apply a factor to the characteristic lengths of the mesh.

- *Mesh.CharacteristicLengthMin*: Forces a minimum.
- *Mesh.CharacteristicLengthMax*: Forces a maximum.
- *Mesh.Optimize*.
- *Mesh.ElementOrder*.
- *Mesh.RecombineAll:* If set to 1, it forces always recombination on surfaces.
- *Mesh.ToleranceEdgeLength*.
- *Mesh.LcIntegrationPrecision*.
- *Mesh.ColorCarousel*: Specifies to what is related the color of elements. 0 for element types, 1 for elementary entities, 2 for physical entities and 3 for partitions.
- *Mesh.Color.Zero*.

For defining structured meshes, we have the following possibilities:

Extrude: It works like for geometries, but additionally we have to specify, after the entities list the command *Layers* or *Recombine*. *Layers* specify the number of elements to be generated during the translation. More than one extrusion can be specified at the same time, but in this case also a list of relative heights (total thickness 1) for each layer has to be provided. The *Recombine* command allows to generate quadrangles for 2D and prisms (or hexahedra or pyramids) in 3D.

Example: *Extrude {0,0,1} {Surface{1}; Layers{ {4,1}, {0.8,1.0} }; }*

Transfinite interpolation: This is used when a non-uniform distribution of element sizes are wanted at the boundary entities. This interpolation adjusts the element sizes depending on an function depending on the coordinates.

Example for a line: *Transfinite Line{1,-3}=10 Using Progression 2;*

In this case, lines 1 and 3 will be meshed with 10 nodes and the nodes are distributed spatially in a geometrical progression of 2 (following the direction of the line). We can also use "*Using Bump 2*" in order to refine the mesh at both end points of the line.

It is similar for surfaces and volumes, but without the right expression for progression.

Additionally, we can also perform the following operations related to meshing:

- *In Surface*: Embed points and lines in surfaces, so that the mesh of the surface will conform to the mesh of the points and/or lines.
- *Periodical*: Force the mesh of one line or surface to match the mesh on other line or surface.
- *Reverse Line*: Change the direction of a line.

IV.1. Fields in detail

Fields define a "field" providing values (in general, related to element sizes) depending on the position. The criteria for calculating these values at every position is what distinguishes one Field from another.

Many of these Fields accept as input another Field, providing a great flexibility combining them.

For testing these Fields we are going to use a simple 2D example:

```
// Avoid mesh size definition by characteristic lengths
Mesh.CharacteristicLengthFromPoints=0;
Mesh.CharacteristicLengthFromCurvature=0;
Mesh.CharacteristicLengthExtendFromBoundary=0;

L=1;
W=0.5;
r=0.5;

Point(1)={0,0,0};
Point(2)={L,0,0};
Point(3)={L,W,0};
Point(4)={0,W,0};
Point(5)={L/2,W+r,0};
Point(6)={L/2,W,0};

Line(1)={1,2};
Line(2)={2,3};
Circle(3)={3,6,5};
Circle(4)={5,6,4};
Line(5)={4,1};

Line Loop(1)={1,2,3,4,5};

Plane Surface(1)={1};
```

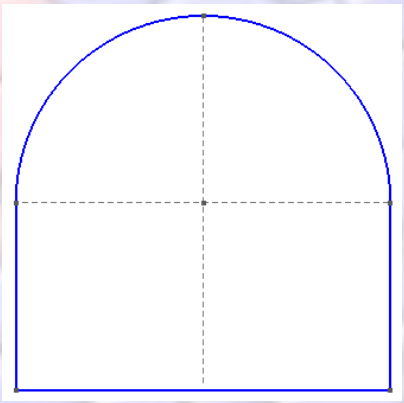

***Figure 2*. 2D geometry for testing fields.**

IV.1.1. Attractor

The calculated value is the nearest distance of the element position to the stated points, lines and/or surfaces. In the case of lines, they are subdivided in N points in order to calculate the distances.

Options:

- *NodesList*: Points.
- *EdgesList*: Lines.
- *FacesList*: Surfaces.
- *NNodesByEdge*: Number of points for the subdivision of lines.
- *FieldX*, *FieldY*, *FieldZ*: Instead of the real coordinates X, Y and Z, we can use other Fields to substitute these coordinates. These three values are the Ids of the Fields to substitute these coordinates.

If we use this Field alone, we can get into trouble if the nodes or lines are inside the geometry to be meshed (as some values will be extremely low (zero at the lines and points)). Indeed, many times this Field is used together with others for this reason.

IV.1.2. Threshold

The Threshold Field uses another Field (*IField*) (for example, Attractor) for distinguishing two regions. One of them is meshed with a characteristic length *LcMin* (when the input Field is below *DistMin*) and with another characteristic length *LcMax* (when the input Field is over *DistMax*). The region in-between *DistMin* and *DistMax* will have a progressive mesh from *LcMin* to *LcMax*.

Options: *IField*, *DistMax*, *DistMin*, *LcMax*, *LcMin*, *Sigmoid*, *StopAtDistMax*.

IV.1.3. MathEval

The element size is calculated from a mathematical expression (given as a string). In this mathematical expression we can include: element current position coordinates (*x*,*y*,*z*), other fields values (*F0*, *F1*, *F2*…) and mathematical functions. For using parameters, we can make use of the function *Sprintf* in order to generate this string.

Options:

- *F*: The function between quotation marks. Example: “0.1*y+F2*Sin(x)”.

If we know the equation of a curve, we can easily increase the mesh density around this curve. Try the following example that implements the equation x^2+(y-Sqrt(Abs(x)))^2-3 (it has also been applied displacement and scaling to x and y):

```
Field[5]=MathEval;
fct=4;
Field[5].F=Sprintf("0.01+0.01*Abs((%g*(x-1))^2+(%g*(y-1)-Sqrt(Abs(%g*(x-1))))^2-3) ",fct,fct,fct);
```

The mesh result is shown in *Figure 3*.

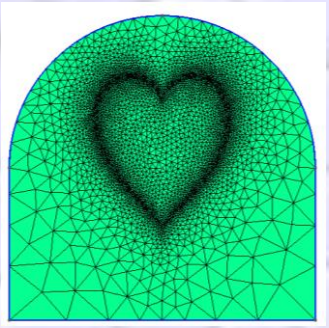

***Figure 3*. Example using MathEval field for increasing mesh density around a known curve.**

IV.1.4. Min (or Max)

The calculated value is the minimum value of the list of indicated fields (*FieldsList*).

IV.1.5. Restrict

Restricts the application of a field (*IField*) to the stated entities (*VerticesList*, *EdgesList*, *FacesList*, *RegionsList*).

IV.1.6. Structured

The calculated value is a linear interpolation between the values provided in a file in regularly distributed positions (grid; as if it were a structured volume mesh). The format of the file is given by:

```
Originx Originy Originz                                  (Position of the grid origin)
Deltax Deltay Deltaz                                     (distance between grid positions)
numberx numbery numberz                                  (number of positions in each direction)
value(0,0,0) value(0,0,1) value(0,0,2) value(0,0,3) ...  (values at the corresponding locations)
value(0,1,0) value(0,1,1) value(0,1,2) value(0,1,3) ...
value(0,2,0) value(0,2,1) value(0,2,2) value(0,2,3) ...
.....
v(1,0,0) v(1,0,1) .....
……
```

Options:

- *FileName*.
- *TextFormat*: True for an ascii file, False for a binary file.
- *OutsideValue*: Value for the region outside of the grid.

- *SetOutsideValue*: True for using the OutsideValue. If False, the last value of the field will be used.

As an example, I have used the following file (fstr.txt):

0.0 0.0 0.0
2.0 2.0 2.0
2 2 1
0.05
0.3
0.02
1.0

The field options are:

```
Field[6]=Structured;
Field[6].FileName="fstr.txt";
Field[6].TextFormat=1;
Field[6].OutsideValue=0.1;
Field[6].SetOutsideValue=1;
```

And the result is shown in *Figure 4*.

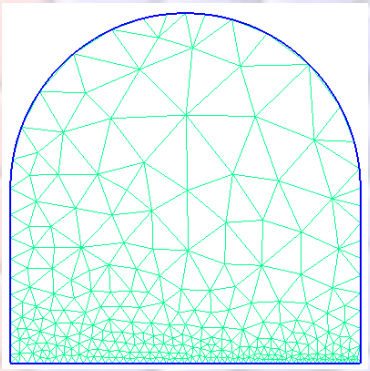

***Figure 4*. Example using the Structured field.**

IV.1.7. Box (cylinder, etc.)

Inside a region (box) the value is *VIn* and outside this region is *VOut*. The region is delimited by the options *Xmin*, *XMax*, *YMin*, *YMax*, *ZMin* and *ZMax*.

IV.1.8. BoundaryLayer

In fluidics, meshes are usually generated quite fine near walls, and less fine far from walls and in a very regular way. This field provides size values that are increasing as we are far from specified geometric entities. It follows the relation: $h_{wall} \cdot ratio^{\frac{dist}{hwall}}$.

Options:

- *EdgesList*.
- *FacesList*.
- *NodesList*.
- *hwall_n*: Parameter of the formula.
- *hwall_t*: The size of element in direction parallel to the wall.
- *ratio*: Parameter of the formula.
- *thickness*: Maximum thickness of the boundary layer.
- *hfar*: Size far from the wall.
- *Quads*: Generate recombined elements.
- *FanNodesList*.
- *FansList*.
- *AnisoMax*.
- *IntersectMetrics*.

Although not stated in the GMSH manual, we have to set the variable *BoundaryLayer Field* to the number of the field.

An example is shown in the section Example 3: Straight artery with boundary layer (page 25).

V. Examples

Here, we are going to apply what we have learned up to now to specific examples. The way in that they are developed is more directed to apply this knowledge than to develop a short script or an optimized model or mesh.

V.1. Example 1: Membrane

The first example shows one problem with many of the features that can be used in GMSH. This example is a membrane, with different regular rectangular regions. The geo file has been generated so that we can change the dimensions and the number of divisions without having to make again the file. We only have to change the vectors *xs* and *ys*:

```
//****************
//** Parameters **
//****************
Lm=GetValue("Length of the membrane: ", 1e-3);    //Membrane Length
Wm=GetValue("Width of the membrane: ", 0.5e-3);  //Membrane Width

lc=Wm/15;       //Default characteristic length

//Lists for the different rectangular positions on the membrane for direction x and y
xs[]={0,Lm/8,Lm/4,Lm/2,Lm};
ys[]={0,Wm/8,Wm/4,Wm/2,Wm};
nxs=#xs[];       //Number of elements in xs
nys=#ys[];       //Number of elements in ys


//***************
//** Geometry **
//***************
//Points for the membrane
p0=newp;        //1
p=p0;
For indys In {0:nys-1}
  For indxs In {0:nxs-1}
          Point(p)={xs[indxs],ys[indys],0,lc};
          p=newp;        //p=p+1
  EndFor
EndFor

//Horizontal lines
l0=newl;        //1
```

```
l=l0;
For indys In {0:nys-1}
  For indxs In {0:nxs-2}
        Line(l)={p0+indys*nxs+indxs,p0+indys*nxs+indxs+1};
        l=newl;         //l=l+1
  EndFor
EndFor

//Vertical lines
For indys In {0:nys-2}
  For indxs In {0:nxs-1}
        Line(l)={p0+indys*nxs+indxs,p0+(indys+1)*nxs+indxs};
        l=newl;
  EndFor
EndFor

//Line Loops
ll=newll;         //1
For indys In {0:nys-2}
  For indxs In {0:nxs-2}
        Line Loop(ll)={l0+indxs+indys*(nxs-1),l0+(nxs-1)*nys+indys*nxs+indxs+1,-
(l0+indxs+(indys+1)*(nxs-1)),-(l0+(nxs-1)*nys+indys*nxs+indxs)};
        Plane Surface(ll)={ll};
        //Recombine Surface {ll};
        ll=newll;       //ll=ll+1
  EndFor
EndFor


//**********
//** Mesh **
//**********
ss[] = Surface "*";      //All surfaces that have been generated

//Producing square elements mesh
Transfinite Surface {ss[]};
Recombine Surface {ss[]};

//Generate the volume of the membrane
Extrude {0,0,10e-6} {Surface {ss[]}; Layers {3}; Recombine;}
```

The resulting mesh for this case is shown in Figure 5 and Figure 6.

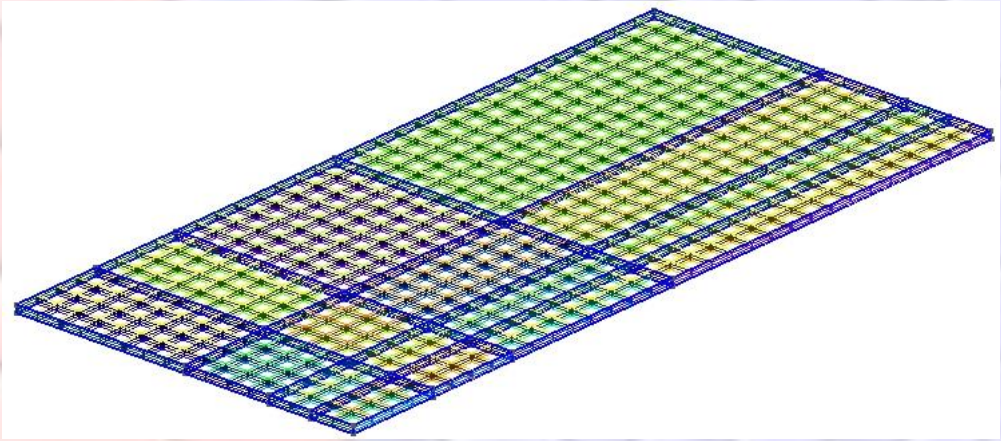

***Figure 5*. Mesh of the membrane.**

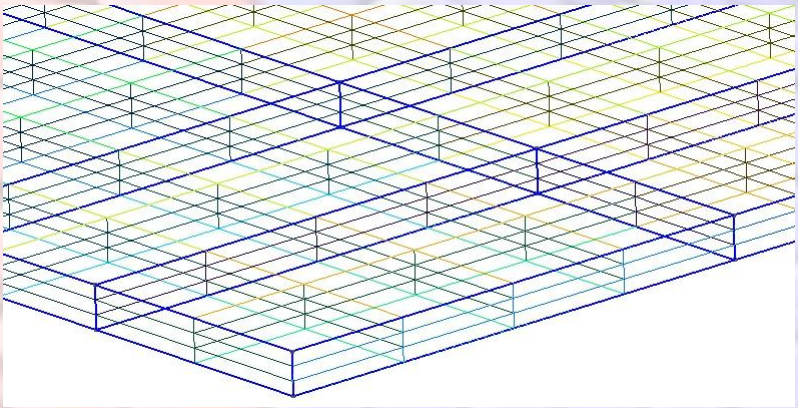

***Figure 6*. Detailed view of the membrane thickness.**

V.2. Example 2: Red Blood Cell

This example illustrates one possible option for generating a model of a 3D red blood cell (RBC). It is also generated as a parametric model.

```
//*****************
//** Parameters **
//*****************
R=10e-6;              // Radius of border of the RBC
Lcent=100e-6;         // Length of the central part of the RBC
depr=R/4;             //magnitude of the depression of central part of the RBC
th=R/5;               //Thickness of the cell

lcm=R/8;              //Characteristic length of the mesh


//***************
```

```
//** Geometry **
//***************
//First iteration for outer lines, second for inner lines
For inout In {0:1}

    np=newp;
    //Left circle
    Point(np)={0,-R+inout*th,0,lcm};
    Point(np+1)={0,0,0,lcm};
    Point(np+2)={0,R-inout*th,0,lcm};
    Point(np+3)={-(R-inout*th),0,0,lcm};
    nl=newl;
    Circle(nl)={3+(np-1),2+(np-1),4+(np-1)};
    Circle(nl+1)={4+(np-1),2+(np-1),1+(np-1)};

    //Top central line (with Spline)
    Point(np+4)={Lcent/4,R-inout*th*0.7-depr,0,lcm};
    Point(np+5)={Lcent/2,R-inout*th-depr,0,lcm};
    Spline(nl+2)={3+(np-1),5+(np-1),6+(np-1)};

    //Bottom central line
    Point(np+6)={Lcent/4,-(R-inout*th*0.7-depr),0,lcm};
    Point(np+7)={Lcent/2,-(R-inout*th-depr),0,lcm};
    Spline(nl+3)={1+(np-1),7+(np-1),8+(np-1)};

EndFor

Line(nl+5)={6,np+5};
Line(nl+6)={np+7,8};

Line Loop(1)={-2,-1,3,nl+5,-(nl+2),nl,nl+1,nl+3,nl+6,-4};

Plane Surface(1)={1};


//**********
//** Mesh **
//**********
//Number of divisions on the circular lines
Transfinite Line {1,2}=Pi*R/(2*lcm);
Transfinite Line {nl,nl+1}=Pi*R/(2*lcm);

Recombine Surface {1};

//Extrussions by 90º each
ss[]=Extrude {{0,1,0},{Lcent/2,0,0},Pi/2} {Surface {1}; Layers{10}; Recombine;};
Printf("Generated first top area = %g", ss[0]);
ss1[]=Extrude {{0,1,0},{Lcent/2,0,0},Pi/2} {Surface {ss[0]}; Layers{10}; Recombine;};
Printf("Generated second top area = %g", ss1[0]);
ss2[]=Extrude {{0,1,0},{Lcent/2,0,0},Pi/2} {Surface {ss1[0]}; Layers{10}; Recombine;};
Printf("Generated third top area = %g", ss2[0]);
```

```
ss3[]=Extrude {{0,1,0},{Lcent/2,0,0},Pi/2} {Surface {ss2[0]}; Layers{10}; Recombine;};
Printf("Generated fourth top area = %g", ss3[0]);
```

The model generated by this script is shown in Figure 7 and Figure 8.

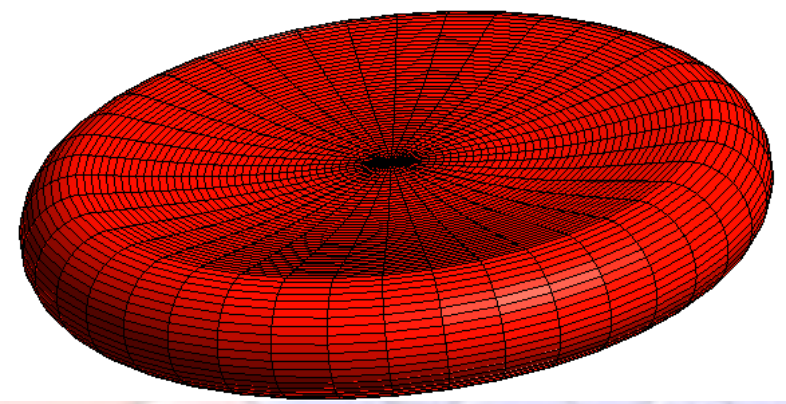

***Figure 7*. Mesh for the whole red blood cell.**

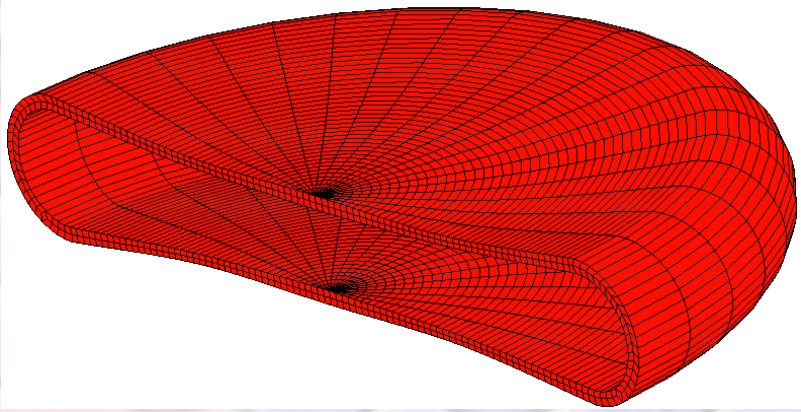

***Figure 8*. Cross section view of the red blood cell.**

V.3. Example 3: Straight artery with boundary layer

This example illustrates the use of Fields to control the mesh generation. We will use the BoundaryLayer Field in a straight artery model, supposedly to be used in a fluidic simulation. In this case, the simplest way to do it would be to use 'Using Progression' when generating lines. Nevertheless, we will do it by using Fields to illustrate its use. Nevertheless, this Field is especially "tricky" and some trial-and-error cycles have to be done. Additionally, the effects of some of its parameters are not very clear. As an example, the command *'BoundaryLayer Field*' cannot be found in the GMSH user's guide.

```
//*****************
//*** Parameters ***
//*****************
rcyli=1e-3;     //Inner radius of the cylinder
hm=0.4e-3;     //Thickness of membrane
Lcyl=10e-3;    //Length of the artery

//***************
//*** Geometry ***
//***************
//Points defining the cylinders ; To avoid 'interactions' with Fields, we do not define lc at points
For val In {1:4}
        Point(2*val-1)={rcyli*Cos((val-1)*Pi/2),rcyli*Sin((val-1)*Pi/2),0};
        Point(2*val)={(rcyli+hm)*Cos((val-1)*Pi/2),(rcyli+hm)*Sin((val-1)*Pi/2),0};
EndFor

//Center Point
np=newp;
Point(np)={0,0,0};

//Lines of the cylinders
lsci={};
lsce={};
For val In {1:3}
        Circle(2*val-1)={2*val-1, np, 2*val+1}; lsci[] +={2*val-1};
        Circle(2*val)={2*val, np, 2*val+2}; lsce[] +={2*val};
EndFor
//val is increased at the end of the loop
        Circle(2*val-1)={2*val-1, np, 1}; lsci[] +={2*val-1};
        Circle(2*val)={2*val, np, 2}; lsce[] +={2*val};

//Surfaces
Line Loop(1)={lsci[]}; Plane Surface(1)={1};
Line Loop(2)={lsce[]}; Plane Surface(2)={2,1};


//********************
//*** Mesh definition ***
//********************
//For speeding up the mesh calculations
Mesh.LcIntegrationPrecision=1e-3;
//To avoid a too high mesh density inside the cylinder
Mesh.CharacteristicLengthExtendFromBoundary = 0;
Mesh.CharacteristicLengthFromPoints=1;
```

```
Mesh.CharacteristicLengthFromCurvature=1;

//BoundaryLayer Field definition
Field[1]=BoundaryLayer;
Field[1].ratio=1.2;
Field[1].hwall_n=hm/20;
Field[1].hwall_t=hm/5;
Field[1].FacesList = {2};      //Avoids this field in Surface 2?
Field[1].EdgesList = {lsci[]};
Field[1].Quads=1;
Field[1].thickness=hm/2.5;
Field[1].hfar=hm/5;

//We define both settings as they do not work properly individually?
Background Field = 1;
BoundaryLayer Field = 1;

//Create square elements from triangular
Recombine Surface "*";

//Generate the body of the artery
surfs[]=Surface "*";
Extrude {0,0,Lcyl} {Surface{surfs[]}; Layers{15}; Recombine;}

Mesh.SurfaceEdges=1;
Mesh.SurfaceFaces=1;
```

The model generated by this script is shown in *Figure 9*.

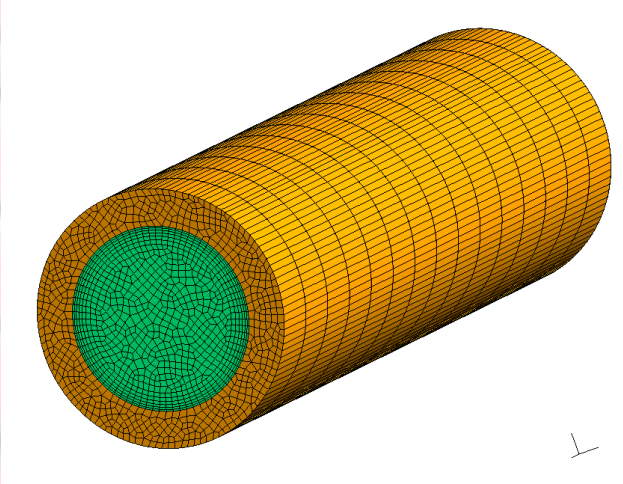

***Figure 9*. Mesh for the whole artery.**

VI. List of commands

Geometry:	Mesh:	Flux Control:	Others:
Point(id)={x,y,z,lc}; *Line(id)={points ids};* *Line Loop(id)={lines ids};* *Plane Surface(id)={line loops ids};* *Surface Loop(id)={surfaces ids};* *Volume(id)={surface loops ids};* sout[]=Extrude{{tx,ty,tz},{rx,ry,rz},{px,py,pz},angle}{entities; layers;}; Translate {vector} { entities} Rotate{{vector},{point},angle}{entities}; Symmetry{A,B,C,D}{entities}; Dilate { {point coords} , factor }{ entities } Duplicata{entities}; newp, newl, newc, news, newv, newll, newsl, newreg Point "*"	Transfinite Line{lines ids}=#nodes; Field[id number]=type; Field[id number].option=value;	Function name Body; Return Call name; For (expression:expression:expression) For var In (expression:expression:expression) EndFor If (expression) ElseIf Else EndIf	var=GetValue(string_to_show, default_value); var=GetString(string_to_show, default_value); Printf Sprintf #vec[] // /* */ ~{expr}

VII. GMSH graphics interface (GUI)

In this chapter, we will show how the GMSH GUI works. We will use the last version of GMSH at the moment where this publication was elaborated, i.e. version 2.8.3.

The first recommended action to perform is to indicate the name for the model that we will created. This will define the name of the file where the commands will be saved as we are working in the GUI creating the model. For doing this, just execute *File* → *New*.

The main window of GMSH is shown in Figure 10.

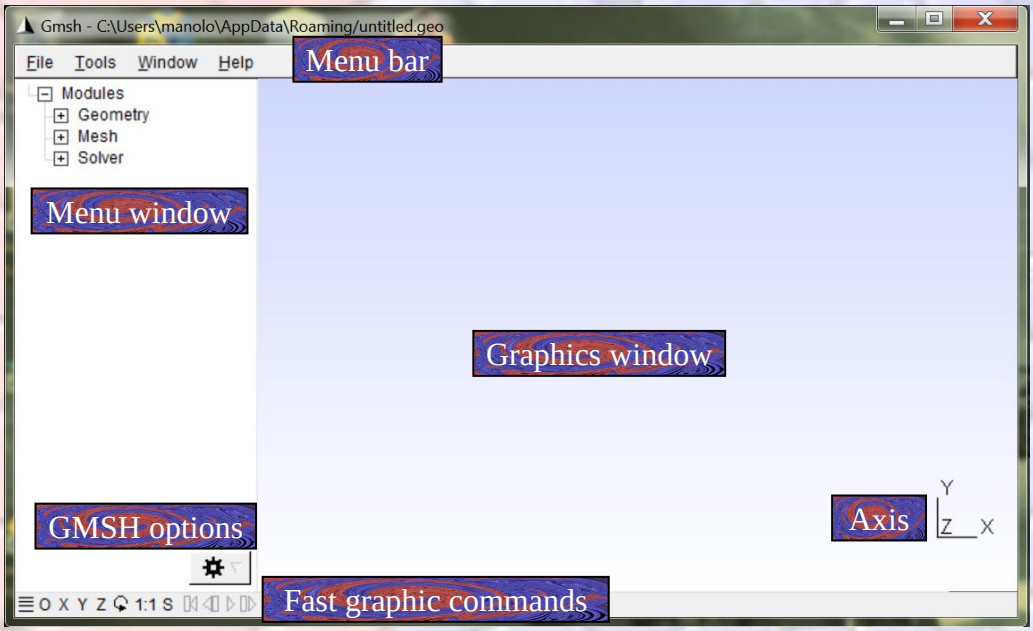

***Figure 10*. Main window of GMSH.**

The menu bar is used for opening/saving files, changing some GMSH options and setting some graphics actions. The different submenus in this bar are shown in Figure 11.

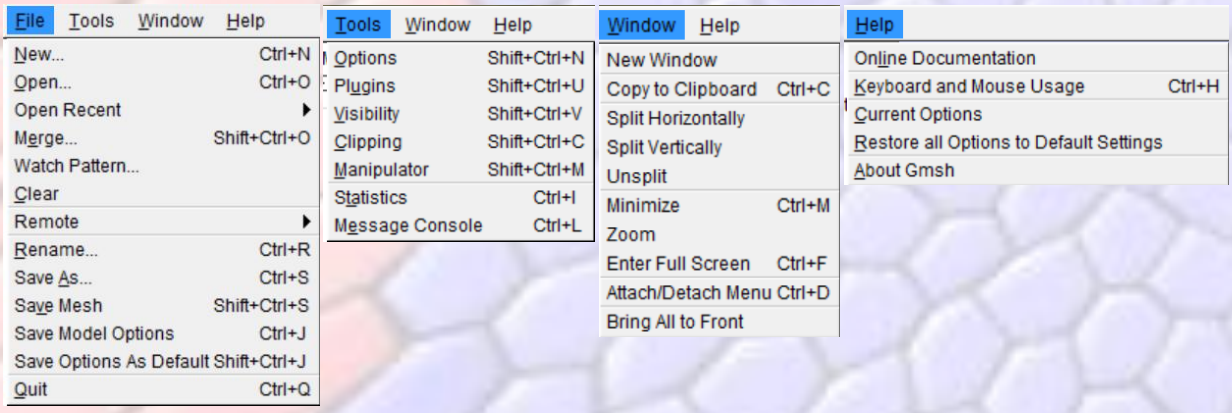

***Figure 11*. Submenus in the menu bar.**

We can highlight some of these options:

- *File* → *New*: We can use this option to delete all and start from 0 the model (by overwriting the geo file).
- *File* → *Open*: For opening/executing geo or msh files.
- *Tools* → *Options*: For setting all options of GMSH.
- *Tools* → *Visibility*: For viewing/hiding entities in the graphics window.
- *Tools* → *Clipping*: For viewing a cross-section of the geometry or the mesh. By holding down left button in options A, B, C or D of the planes section and dragging the mouse, we can change continuously the plane position or the orientation.
- *Tools* → *Manipulator*: For controlling precisely the orientation view in the graphics window and the scales (zoom) in each direction independently.

At the bottom-left position there is a drop-down menu for changing some general options of GMSH. At nearly the same position, located at the bottom bar, there is some graphics commands for setting some view directions, rotate the view, or activating the mouse selection. The *O* option allows us to set also more graphics settings (also reachable by double-left-clicking in the graphics window). Finally, by left-clicking at the free space on this bottom bar we can view/hide the message console.

The graphics window is where the model and results are shown. With the mouse, we can change continuously the orientation view: left button: rotate, right button: pan (translate), middle button: zoom with respect to the mouse position, Ctrl+left button: region zoom, Ctrl+right button: return to the default view. All mouse and keyboard options can be obtained at the menu bar *Help* →*KeyBoard and Mouse Usage*.

The menu window allows us to access all option for the model (geometry and mesh) generation. The different submenus are shown in Figure 12.

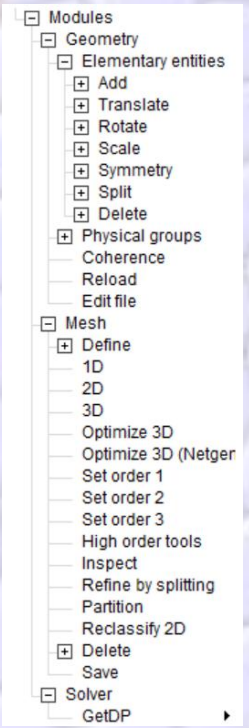

***Figure 12*. Submenus in the menu window.**

The geometry submenu is for generation of geometric entities (points, lines, surfaces and volumes) as well as physical groups. The basic option is *Add* (also for defining parameters), but we can also observe the different geometric operations that are available in GMSH (translation, rotation, etc.). Here we can also delete already created entities. After executing these commands, in general, we have the options of *q* for ending/aborting the addition of entities and 'e' for adding the defined entity, but more options can appear, depending on the specific entity and operation.

The different options in this geometry menu are shown in Figure 13.

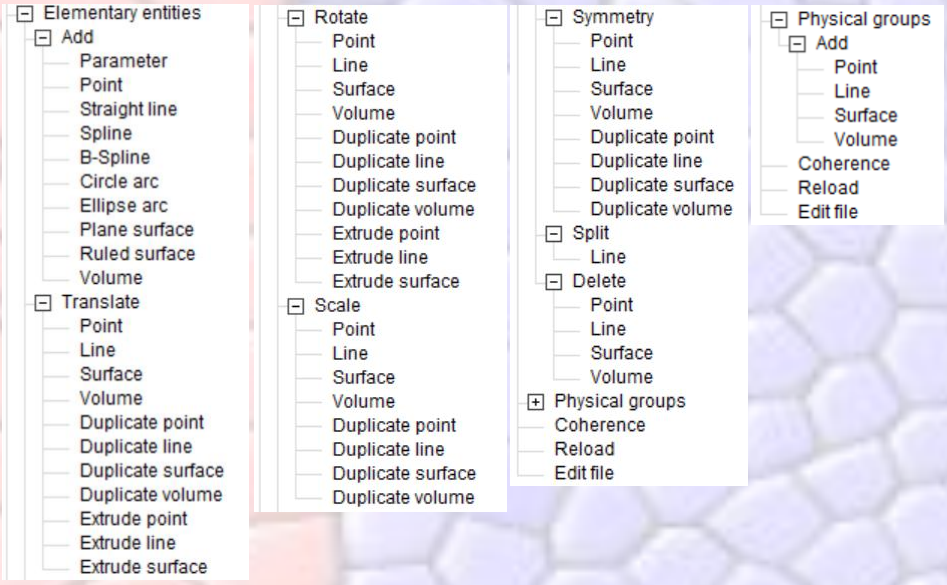

***Figure 13*. Options of the geometry submenu.**

The mesh submenu allows to control how the mesh will be generated. The different options of this submenu are shown in Figure 14.

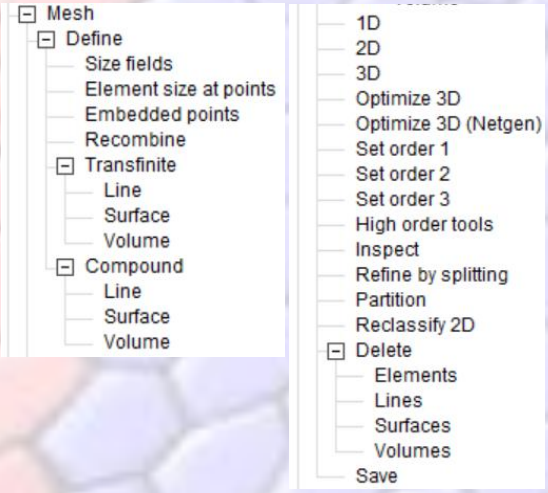

***Figure 14*. Options of the mesh submenu.**

Let's illustrate the GUI usage with one simple example. We will generate a 3D membrane, with length lm (lm=1), width wm (wm=1) and a thickness th (th=0.2). We will define the element size near points as lc (lc=0.1). First of all, we will define the parameters that we will use. We define them in *Geometry* → *Elementary entities* → *Add* → *Parameter*. We just have to click the add

button once we have defined the different fields of the parameter. The appearing windows is shown in Figure 15, where it is illustrated the definition of the lc parameter.

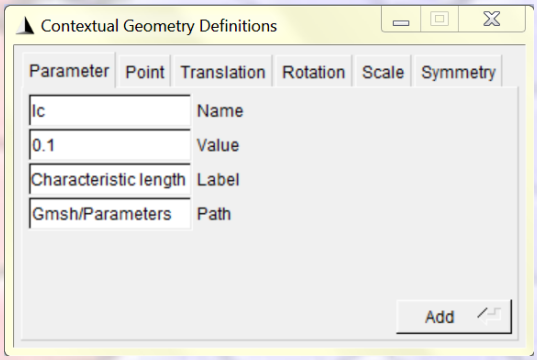

***Figure 15*. Window for creating a parameter.**

For creating the needed points (in our case, we only need to create four for creating also the four boundary lines of the membrane) we can just click on the Point sheet on the same window, or just click on *Geometry* → *Elementary entities* → *Add* → *Point*. Points can be generated by clicking on the graphics window where we want to place the point, or by writing the coordinates in the Point window, that it is shown in Figure 16. Be careful not to place the mouse on the graphics windows after you have filled the fields and before clicking the Add button because the coordinates of the mouse will overwrite the values in the coordinate fields. If you want to do it by clicking on the graphics window, use the Shift button to fix the coordinates in the windows even if we move the mouse.

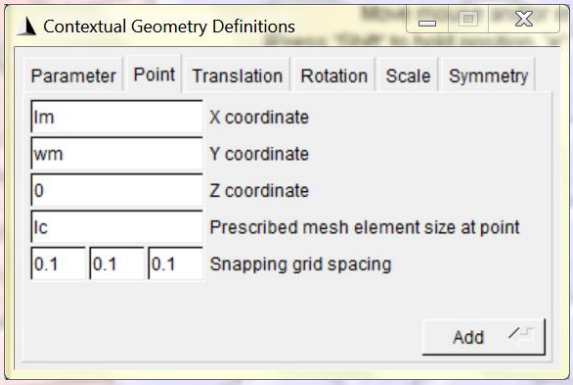

***Figure 16*. Window for creating a point.**

After creating all four points (click q to end the point creation), as illustrated in Figure 17 (left), we have to generate the lines. We have to click on *Geometry* → *Elementary entities* → *Add* → *Straight line*, and click on the two end points that will define the line. The line is created with a direction, from the first point selected to the second one. We will end up with the four lines after this process, as shown in Figure 17 (right). Click also q for finishing the creation of lines.

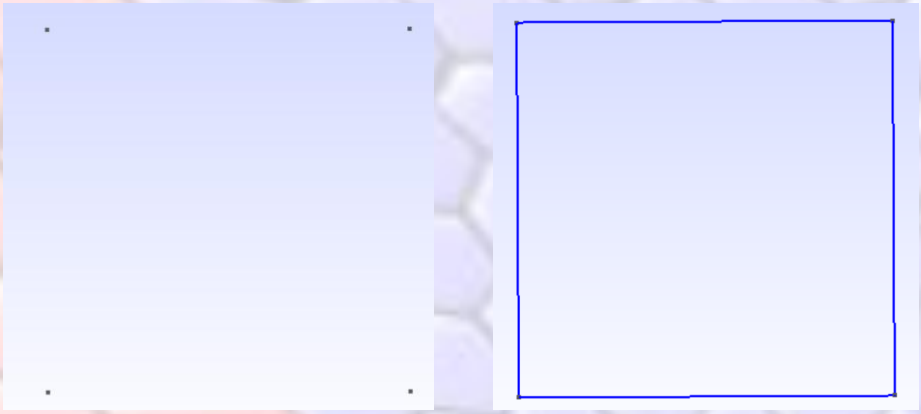

***Figure 17*. The four points (left) and the four lines (right) created for the membrane.**

Now we will create the surface. In this case, we will create a ruled surface (i.e., a surface that can be meshed using the transfinite interpolation). Therefore, we have to click on *Geometry* → *Elementary entities* → *Add* → *Ruled Surface*. In this case, we just have to click on one line, and GMSH will find the rest (there is no other possibilities for creating a surface in our case). We have to press e to finish the selection of lines for the surface (or q for aborting the line selection if we would have not selected a right set of lines), and afterwards press q for definitely creating the surface. The surface appears as shown in Figure 18. The dashed lines indicate that the surface has been created.

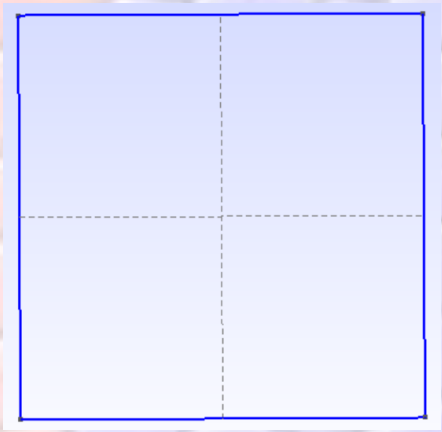

***Figure 18*. Created ruled surface of the membrane.**

Before creating the volume, we will mesh the surface. In this way, when we create the volume (by translation of this already created surface), we will also create simultaneously the volume mesh. For this, we have to indicate two things:

- We want to use transfinite interpolation for creating a regular mesh. This is done by clicking on *Mesh* → *Define* → *Transfinite* → *Surface* and selecting the surface (clicking on the dashed lines). (we can select all related entities in an area by keeping pressed the Ctrl key, while defining an area with the mouse).
- We want to generate square elements and not triangular. This is done by clicking on *Mesh* → *Define* → *Recombine* and selecting the same surface.

In order to check how the surface elements will be, we can generate the mesh by clicking on *Mesh → 2D*. The results should be like shown in *Figure 19* with a mesh of 10x10 square elements. Clear them with the bar menu option *File → Clear*.

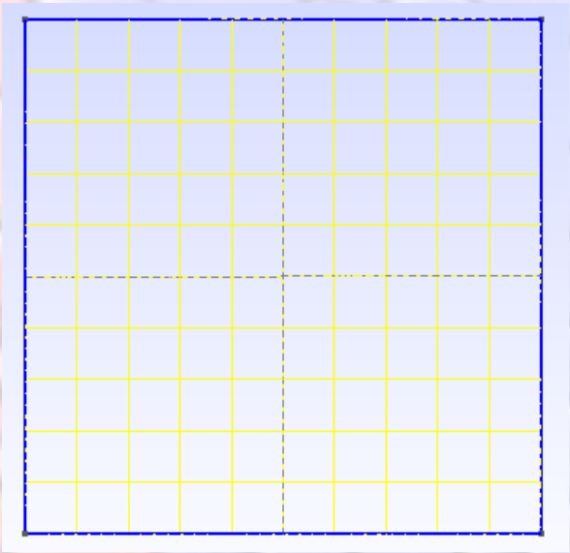

***Figure 19*. Meshed surface.**

Now, let's create the volume. We will use the extrude option *Geometry → Elementary entities → Translate → Extrude surface*. We have to enter the value th value in the Z Component field. Unfortunately, not all options are available in the GUI. For specifying the number of elements in the extruded Z direction and indicating that we want cube-shaped elements, we will have to edit the geo file and include *Layers{3}; Recombine;* inside the last field of the extrude command. The instruction would be something like: *Extrude {0, 0, th} {Surface{5}; Layers{3}; Recombine;}*. Save the file and run this file making *File → Open*.

By generating the 3D mesh, clicking on *Mesh → 3D*, we should obtain something similar to what is shown in Figure 20.

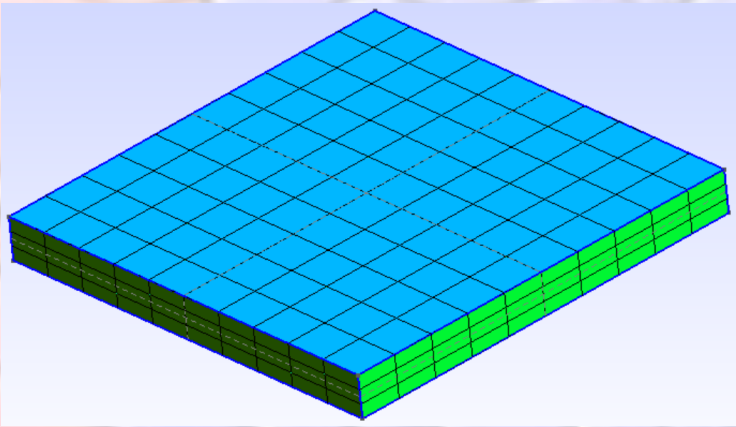

***Figure 20*. Meshed volume.**

VIII. Useful macros

The aim of this section is to provide useful macros for the generation of models in GMSH. We will use some simple rules: the use of '_' in order to define the new created variables in the macro (all or most of them will be deleted at the end of the function), the needed inputs and the provided outputs will be indicated in the macro header.

It has to be said, that currently these macros seem not to be useful because it has been added a command for obtaining geometric entities within a bounding box. Example:

_surfspos[] = Surface In BoundingBox {-1e-5,-1,-1,1e-5,1,1};

The index order seems to be: xmin,ymin,zmin,xma,ymax,zmax. The current manual (4.0.7), nevertheless, seems to indicate that the order should be xmin,xmax,ymin,ymax,zmin,zmax.

VIII.1. Point at a position

```
//Macro for finding a point (_pfin) at given coordinates (xpc,ypc,zpc) within a deviation (dev)
//Optional: Restricted to a list of points (plist). If it exists, it is deleted at the end of the function.
//If _pfin is -1, it means that no point was found at this position.
Function Pointcoord
        //needed inputs: xpc, ypc, zpc, dev(deviation)
        _valdel1=Exists(plist);
        _ps[]=Point "*";
        _nps=#_ps[];
        _pfin=-1;       //_pfin will contain the last point found with these coordinates
        For i In {0:_nps-1}
                _coords[]=Point{_ps[i]};
                If (_coords[0]<xpc+dev && _coords[0]>xpc-dev)
                        If (_coords[1]<ypc+dev && _coords[1]>ypc-dev)
                                If (_valdel1)
                                        If (Find(_ps[i],plist[]))
                                                _pfin=_ps[i];
                                                Printf("Point found: %g",_pfin);
                                                i=_nps-1;
                                        EndIf
                                Else
                                        _pfin=_ps[i];
                                        Printf("Point found: %g",_pfin);
                                        i=_nps-1;
                                EndIf
                        EndIf
                EndIf
```

```
        EndFor
        If (_pfin==-1)
                Printf("No point found");
        EndIf
        Delete _ps; Delete _nps; Delete _coords;
        If (_valdel1)
                Delete plist;
        EndIf
        Delete _valdel1;
Return
```

VIII.2. Lines at a position

```
//Macro for finding the lines completely within one positions (vpos) in a direction (dir; "x", "y" or "z" values) within a deviation (dev)
//Optional: restricting orientation of the lines (orient[]; If it does not exists, no orientation is considered; orient[] is deleted if exists)
//Optional: Search restricted to a list of lines (llist)
//The vector _linespos[] contains all these lines.
//Possible extensions: limiting the lines also to a given list
Function Linesatpos
        //needed inputs: vpos, dir (direction), dev (deviation), orient (orientation; optional), llist (restrict list; optional)
        If (!StrCmp(dir,"x"))
                _indd=0;
        ElseIf (!StrCmp(dir,"y"))
                _indd=1;
        Else
                _indd=2;
        EndIf
        _valdel=Exists(orient);
        _valdel1=Exists(llist);
        _linelist[]=Line "*";
        _nlines=#_linelist[];
        _linespos[]={};
        _pospold[]={};
        _conf=0;
        For _lact In {0:_nlines-1}
                _ps[]=Boundary {Line{_linelist[_lact]};};
                _nps=#_ps[];
                For _pact In {0:_nps-1}
                        _posp[]=Point{_ps[_pact]};
                        _valp=_posp[_indd];
```

```
                If ((_valp<(vpos-dev)) || (_valp>(vpos+dev)))
                        _conf=1;
                        _pact=_nps-1; //Equivalent to break
                EndIf
                If (_pact==0)
                        _pospold[]={_posp[]};
                EndIf
        EndFor
        If (_conf==0)
                If (!_valdel)
                        If (_valdel1)
                                If (Find(_linelist[_lact],llist[]))
                                        _linespos[] += {_linelist[_lact]};
                                EndIf
                        Else
                                _linespos[] += {_linelist[_lact]};
                        EndIf
                Else
                        _vagrdir=0; _modprod=0; _modprod1=0;
                        For io In {0:2}
                                _vagrdir += (_posp[io]-_pospold[io])*orient[io];
                                _modprod += (_posp[io]-_pospold[io])^2;
                                _modprod1 += orient[io]^2;
                        EndFor
                        _modprod=Sqrt(_modprod);_modprod1=Sqrt(_modprod1);
                        _vagrdir=_vagrdir/(_modprod*_modprod1);
                        If (Fabs(_vagrdir)>0.95)
                                If (_valdel1)
                                        If (Find(_linelist[_lact],llist[]))
                                                _linespos[] += {_linelist[_lact]};
                                        EndIf
                                Else
                                        _linespos[] += {_linelist[_lact]};
                                EndIf
                        EndIf
                EndIf
                _conf=0;
EndFor
Printf("number of lines: %g",#_linespos[]);
For _vis In {0:(#_linespos[]-1)}
        Printf("s_%g: %g",_vis,_linespos[_vis]);
EndFor
Delete _indd; Delete _linelist; Delete _nlines; Delete _lact; Delete _pact;
Delete _ps; Delete _nps; Delete _posp; Delete _pospold;
```

```
        Delete _valp; Delete _conf; Delete _vis;
        If (_valdel)
                Delete orient; Delete _vagrdir; Delete _modprod; Delete _modprod1;
        EndIf
        If (_valdel1)
                Delete llist;
        EndIf
        Delete _valdel; Delete _valdel1;
Return
```

VIII.3. Lines between two positions

With just a few modifications with respect to the previous code, we can also obtain the surfaces between two positions in one direction.

```
//Macro for finding the lines completely within two positions (vpos1 and vpos2; vpos1<vpos2) in a direction (dir; "x", "y" or "z" values) within a deviation (dev)
//Optional: restricting orientation of the lines (orient[]; If it does not exists, no orientation is considered; orient[] is deleted if exists)
//Optional: Search restricted to a list of lines (llist)
//The vector _linespos[] contains all these lines.
Function Linesat2pos
        //needed inputs: vpos1 (position1), vpos2 (position2 (>position1)), dir (direction), dev (deviation), orient (orientation)
        If (!StrCmp(dir,"x"))
                _indd=0;
        ElseIf (!StrCmp(dir,"y"))
                _indd=1;
        Else
                _indd=2;
        EndIf
        _valdel=Exists(orient);
        _valdel1=Exists(llist);
        _linelist[]=Line "*";
        _nlines=#_linelist[];
        _linespos[]={};
        _pospold[]={};
        _conf=0;
        For _lact In {0:_nlines-1}
                _ps[]=Boundary {Line{_linelist[_lact]};};
                _nps=#_ps[];
                For _pact In {0:_nps-1}
                        _posp[]=Point{_ps[_pact]};
                        _valp=_posp[_indd];
```

```
                If ((_valp<(vpos1-dev)) || (_valp>(vpos2+dev)))
                        _conf=1;
                        _pact=_nps-1; //Equivalent to break
                EndIf
                If (_pact==0)
                        _pospold[]={_posp[]};
                EndIf
        EndFor
        If (_conf==0)
                If (!_valdel)
                        If (_valdel1)
                                If (Find(_linelist[_lact],llist[]))
                                        _linespos[] += {_linelist[_lact]};
                                EndIf
                        Else
                                _linespos[] += {_linelist[_lact]};
                        EndIf
                Else
                        _vagrdir=0; _modprod=0; _modprod1=0;
                        For io In {0:2}
                                _vagrdir += (_posp[io]-_pospold[io])*orient[io];
                                _modprod += (_posp[io]-_pospold[io])^2;
                                _modprod1 += orient[io]^2;
                        EndFor
                        _modprod=Sqrt(_modprod);_modprod1=Sqrt(_modprod1);
                        _vagrdir=_vagrdir/(_modprod*_modprod1);
                        If (Fabs(_vagrdir)>0.95)
                                If (_valdel1)
                                        If (Find(_linelist[_lact],llist[]))
                                                _linespos[] += {_linelist[_lact]};
                                        EndIf
                                Else
                                        _linespos[] += {_linelist[_lact]};
                                EndIf
                        EndIf
                EndIf
        EndIf
        _conf=0;
EndFor
Printf("number of lines: %g",#_linespos[]);
For _vis In {0:(#_linespos[]-1)}
        Printf("s_%g: %g",_vis,_linespos[_vis]);
EndFor
Delete _indd; Delete _linelist; Delete _nlines; Delete _lact; Delete _pact;
Delete _ps; Delete _nps; Delete _posp; Delete _pospold;
```

```
        Delete _valp; Delete _conf; Delete _vis;
        If (_valdel)
                Delete orient; Delete _vagrdir; Delete _modprod; Delete _modprod1;
        EndIf
        If (_valdel1)
                Delete llist;
        EndIf
        Delete _valdel; Delete _valdel1;
Return
```

VIII.4. Surfaces at a position

```
//Macro for finding the surfaces completely within one positions (vpos) in a direction (dir; "x", "y" or "z" values) within a deviation (dev)
//Optional: Search restricted to a list of surfaces (slist)
//The vector _surfspos[] contains all these surfaces.
//Possible extensions: limiting the lines also to a given list
Function Surfsatpos
        //needed inputs: vpos, dir (direction), dev (deviation), slist (restricted list; optional)
        If (!StrCmp(dir,"x"))
                _indd=0;
        ElseIf (!StrCmp(dir,"y"))
                _indd=1;
        Else
                _indd=2;
        EndIf
        _valdel1=Exists(slist);
        _surflist[]=Surface "*";
        _nsurfs=#_surflist[];
        _surfspos[]={};
        For _sact In {0:_nsurfs-1}
                _conf=0;
                _linelist[]=Boundary {Surface{_surflist[_sact]};};
                _nlines=#_linelist[];
                For _lact In {0:_nlines-1}
                        _ps[]=Boundary {Line{_linelist[_lact]};};
                        _nps=#_ps[];
                        For _pact In {0:_nps-1}
                                _posp[]=Point{_ps[_pact]};
                                _valp=_posp[_indd];
                                If ((_valp<(vpos-dev)) || (_valp>(vpos+dev)))
                                        _conf=1;
```

```
                                _pact=_nps-1; //Equivalent to break
                        EndIf
                EndFor
        EndFor
        If (_conf==0)
                If (_valdel1)
                        If (Find(_surflist[_sact],slist[]))
                                _surfspos[] += {_surflist[_sact]};
                        EndIf
                Else
                        _surfspos[] += {_surflist[_sact]};
                EndIf
        EndIf
    EndFor
    Printf("Number of surfaces: %g",#_surfspos[]);
    For _vis In {0:(#_surfspos[]-1)}
        Printf("s_%g: %g",_vis,_surfspos[_vis]);
    EndFor
    Delete _indd; Delete _surflist; Delete _nsurfs;
    Delete _linelist; Delete _nlines; Delete _sact; Delete _lact; Delete _pact;
    Delete _ps; Delete _nps; Delete _posp;
    Delete _valp; Delete _conf; Delete _vis;
    Delete _valdel1;
Return
```

VIII.5. Surfaces between two positions

With just a few modifications with respect to the previous code, we can also obtain the surfaces between two positions in one direction.

```
//Macro for finding the surfaces completely within two positions (vpos1 and vpos2; vpos1<vpos2) in a direction (dir; "x", "y" or "z" values) within a deviation (dev)
//Optional: Search restricted to a list of surfaces (slist)
//The vector _surfspos[] contains all these lines.
Function Surfsat2pos
    //needed inputs: vpos1 (position1), vpos2 (position2 (>position1)), dir (direction), dev (deviation), slist (restricted list; optional)
    If (!StrCmp(dir,"x"))
        _indd=0;
    ElseIf (!StrCmp(dir,"y"))
        _indd=1;
    Else
        _indd=2;
```

```
        EndIf
        _valdel1=Exists(slist);
        _surflist[]=Surface "*";
        _nsurfs=#_surflist[];
        _surfspos[]={};
        For _sact In {0:_nsurfs-1}
                _conf=0;
                _linelist[]=Boundary {Surface{_surflist[_sact]};};
                _nlines=#_linelist[];
                For _lact In {0:_nlines-1}
                        _ps[]=Boundary {Line{_linelist[_lact]};};
                        _nps=#_ps[];
                        For _pact In {0:_nps-1}
                                _posp[]=Point{_ps[_pact]};
                                _valp=_posp[_indd];
                                If ((_valp<(vpos1-dev)) || (_valp>(vpos2+dev)))
                                        _conf=1;
                                        _pact=_nps-1; //Equivalent to break
                                EndIf
                        EndFor
                EndFor
                If (_conf==0)
                        If (_valdel1)
                                If (Find(_surflist[_sact],slist[]))
                                        _surfspos[] += {_surflist[_sact]};
                                EndIf
                        Else
                                _surfspos[] += {_surflist[_sact]};
                        EndIf
                EndIf
        EndFor
        Printf("Number of surfaces: %g",#_surfspos[]);
        For _vis In {0:(#_surfspos[]-1)}
                Printf("s_%g: %g",_vis,_surfspos[_vis]);
        EndFor
        Delete _indd; Delete _surflist; Delete _nsurfs;
        Delete _linelist; Delete _nlines; Delete _sact; Delete _lact; Delete _pact;
        Delete _ps; Delete _nps; Delete _posp;
        Delete _valp; Delete _conf; Delete _vis;
        Delete _valdel1;
Return
```

Acronyms:

2D: Two-dimensional

3D: Three-dimensional

CAD: Computer Aided Design

FEM: Finite Element Method

GPL: General Public License.

RBC: Red Blood Cell

[1] http://geuz.org/gmsh/