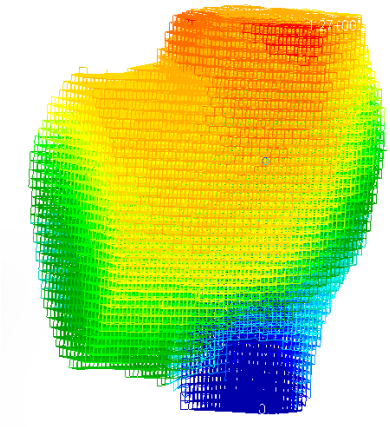
User guide



Software 3DMetMec:
Metamaterial slicing engine able to fulfill the mechanical properties and spatial adaptation required by a 3D design.

- - - - - - -

Group for Advanced Modeling and Simulation of Nonlinear Solids (GAMOSINOS)

Polytechnical University of Madrid

Authors

Elemer San Miguel Niddam, Luis Saucedo Mora, Miguel Ángel Sanz Gómez, Francisco Javier Montans Leal

2021

Contents

List of Figures

List of Tables

Listings

1 First Steps

In this chapter, the reader can find a description of the first steps taken in this project. It provides a brief analysis of the initial scope of the project as well as an introduction to some peculiarities of The Julia Programming Language.

1.1 Problem Description

In a first approximation, the desired algorithm should:

1. Be able to determine whether a given point of $\mathbb{R}^3$ falls inside a solid, given via `stl`.

2. Define a 3D grid, and return which points of said grid fall inside the given solid, using the function described in 1.

3. Use the points obtained in the previous step to create cells and form the metamaterial.

Having completed 1, it is trivial to figure out how to implement 2 and 3. Now a question arises: How can whether a point falls inside a solid or not be determined?.

1.2 First Simplifications

In this case, the idea was to solve a similar problem in a smaller dimension. Detecting points inside a solid is equivalent to detecting points inside a section of said solid. So, in order to check whether a point is inside a solid or not, one may cut the solid with a plane that contains that point, and if the point lies inside the section, it is also inside the solid.

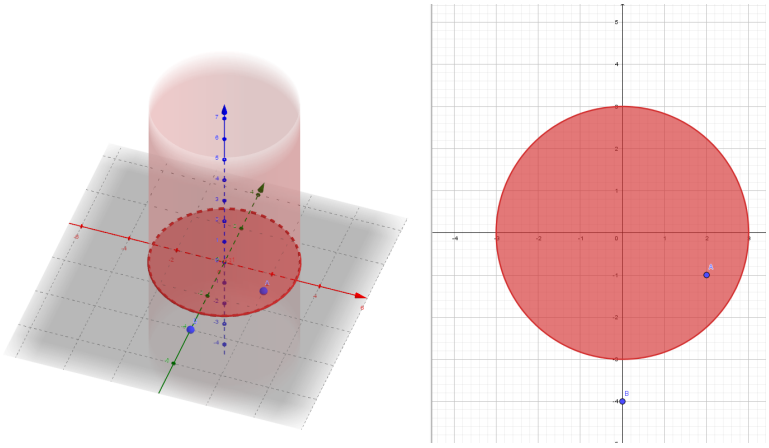


Figure 1.1: Demonstration of the possibility of detecting inside points using sections.

Thus, it is safe to say the problem is greatly simplified. Nevertheless, an algorithm able to get any section from a solid is now needed, and so is an algorithm to find whether a point lies inside said section.

Since the solid geometry is imported via `stl`, it will be made up of triangles, and it is clear that any of the solid's sections can be represented as the intersection between each of the solid's triangles and the plane.

This means that, by finding the intersection between the plane and each triangle, we can build a curve (or several of them) made of segments, that separates the inside of the solid from the outside.

As can be seen in the following figure, the idea is to obtain a curve (or curves), given by an ordered list of points that starts and ends in the same element.

If the geometry is non-convex, it might occur that a section is described by more than one curve, in that case, we want to be able to detect and save all of them.

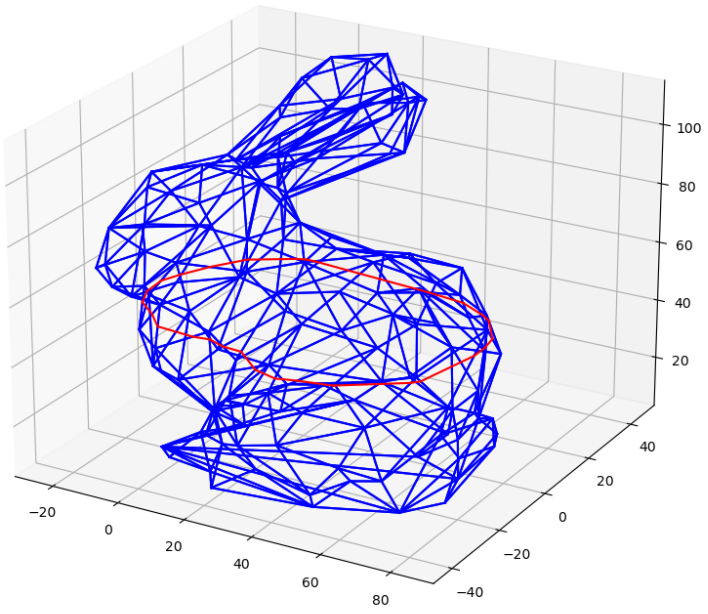


Figure 1.2: Desired result of the section-finding algorithm (red).

Finally, a ray casting algorithm[7] must be implemented to determine if a point is inside each of the curves. After interpreting the result, whether said point lies inside the solid or not will be determined.

A grid of points can now be created. Repeating this procedure and keeping only those that are inside the solid allows using them as nodes and, by joining them with rods, a metamaterial is constructed cell by cell.

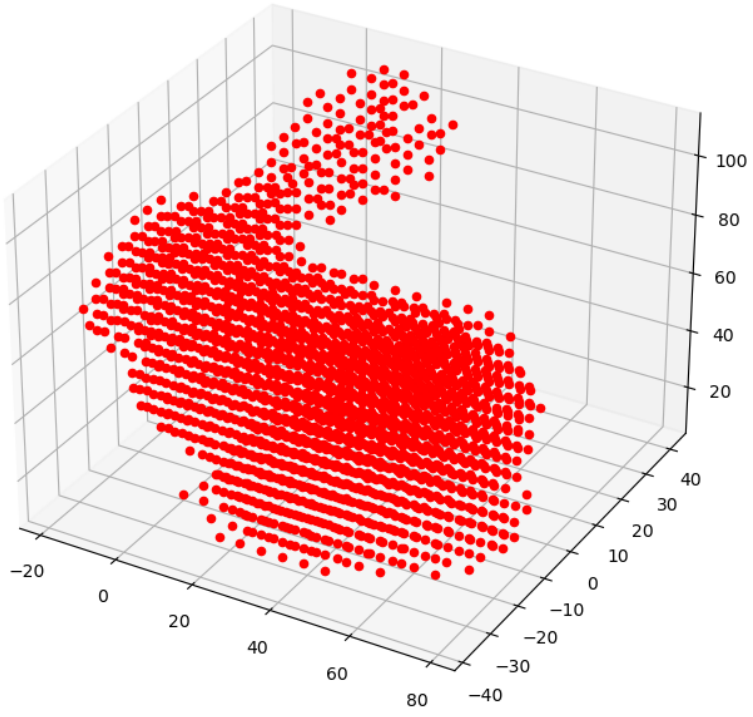


Figure 1.3: Cloud of nodes to be used in the cell making.

1.3 The Julia Environment

Most of this project has been developed using *The Julia Programming Language*. It has proven to be a suitable alternative to proprietary software environments such as *MATLAB*, with the advantage of being free and open source.

One of the main differences with *MATLAB* is the matrix concatenation and manipulation syntax, but in the end, any imaginable operation can be performed.

The other main difference is the variable's scope, which makes variables defined outside a loop uneditable at first hand, while being accessible.

If someone were to modify a variable that was declared outside a given loop, an undefined variable exception will occur.

To fix this, the loop may be executed inside a function, where the scope of variables is more intuitive, or, alternatively, the variable can be declared as global inside the loop.

Below are a few examples demonstrating this behaviour.

```
#THIS CODE WORKS: RETURNS 1111111111
var = 1
for i=1:10
    print(var)
end

#THIS CODE THROWS AN UndefVarError
var = 1
for i=1:10
    var=var+1 #Julia fails to modify "var"
end
print(var)

#THIS CODE WORKS: RETURNS 11
var = 1
for i=1:10
    global var
    var=var+1
end
print(var)
```

Listing 1.1: Julia scope of variables example

Other rather confusing difference between *Julia* and *MATLAB* is the usage of packages. Julia comes with close to no features when installed, and the user is responsible for finding a package that suits their needs.

This can seem quite normal, as at the end of the day it is Python-alike, but, when it comes plotting libraries, huge complications arise.

First, there is no standard way to handle plots, so each package implements them on its own way, making it harder to find proper help and documentation.

Then there is also the fact that the package has to compile each time a plot is executed, making the code run slower than expected.

Finally, the visualisation isn't great, as the program struggles when rotating in a 3D plot.

In this project, the plotting is handled with the `PyPlot` package, which is a `matplotlib` variant that behaves the same way *MATLAB* does. Additionally, the `DelimitedFiles` package is used to open and read `.csv` files.

2
Algorithmics

Contents

In this chapter the reader can find a full description of the algorithms used to implement the detection of points inside solids imported via `stl`.

2.1 Plane-Triangle Intersection

As previously stated, the first step is finding the intersection between a plane and several triangles, which can be performed one by one.

This first algorithm will take as inputs the coefficients of the equation of a plane $\pi : Ax + By + Cz = D$ as well as the Cartesian coordinates of the vertices of a triangle. It must then output the intersection between both.

To determine the intersection between a triangle and a plane, we can begin by checking whether they actually intersect or not.

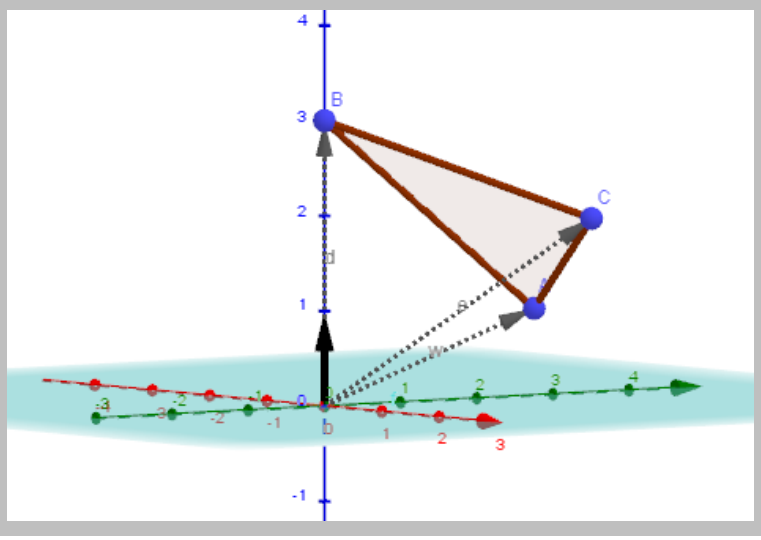
This is as easy as finding if all of the vertices are in the same semispace from the two that the plane limits. If that were the case, the intersection between the triangle and the plane is empty.

To do this, we can use any point of the plane and the vector normal to the plane. The following examples show this theory in use:

Example 2.1.1 (Determining whether a triangle and plane intersect.)
We begin with the plane $\pi : z = 0$ *and triangle* Δ *with vertices* $P_1 = (1, 2, 1)$, $P_2 = (0, 0, 3)$ *and* $P_3 = (2, 2, 2)$.

A point inside the plain is $P_\pi = (0, 0, 0)$ *and the plane's normal vector is* $\vec{n} = (0, 0, 1)$:
$(P_1 - P_\pi) \cdot \vec{n} = 1$
$(P_2 - P_\pi) \cdot \vec{n} = 3$
$(P_3 - P_\pi) \cdot \vec{n} = 2$

Since $sg(1) = sg(3) = sg(2)$, *all of the vertices are in the same semispace, and therefore, the plane and triangle do not cut.*

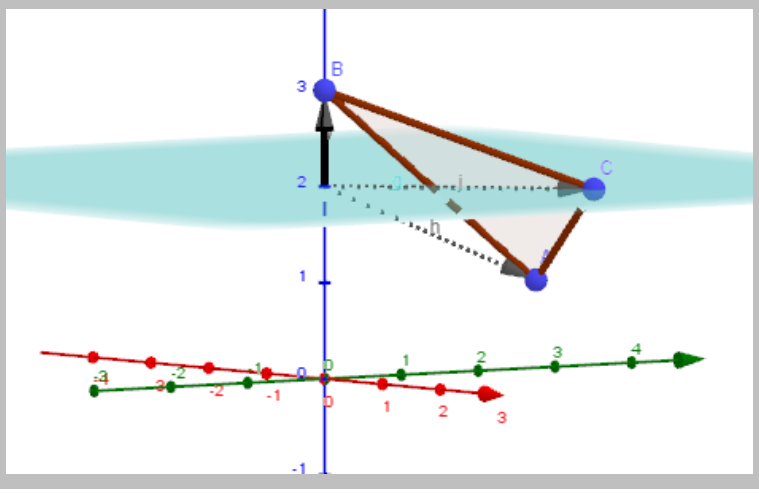


Example 2.1.2 (Determining whether a triangle and plane intersect.)
We begin with the plane $\pi : z = 2$ *and triangle* Δ *with vertices* $P_1 = (1, 2, 1)$, $P_2 = (0, 0, 3)$ *and* $P_3 = (2, 2, 2)$.

A point inside the plain is $P_\pi = (0, 0, 2)$ *and the plane's normal vector is* $\vec{n} = (0, 0, 1)$:
$(P_1 - P_\pi) \cdot \vec{n} = -1$
$(P_2 - P_\pi) \cdot \vec{n} = 0$
$(P_3 - P_\pi) \cdot \vec{n} = 1$

Since $sg(1) \neq sg(-1)$, *there are vertices in different semispaces, and therefore, the plane and triangle cut.*

Moreover, since one of them returns a value of zero, it is contained in the plane.



The previous examples suggest that the algorithm used is actually calculating the distance between each vertex and the plane, with sign. And so it is. This is why if a vertex is contained in the plane, the returned value is zero.

Having this in mind, it is turn to implement an algorithm that can retrieve the intersection between the plane and the triangle.

If it were a segment, only the two extremes would be needed, which wuold, of course, be contained in the sides of the triangle.

Using the previous algorithm, the problem can be divided in different cases, depending on the sign of the distances (different signs imply different semispaces):

- If no vertex is inside the plane:
  - If all of them have the same sign: null intersection.
  - If one of them has a different sign: segment with each extreme in one of the sides that share said vertex.
- If one vertex is contained in the plane:
  - If each of the others lie in different semispaces: segment with the first vertex as one extreme, being the other inside the opposite side.
  - If both of the others lie in the same semispace: only the first vertex.

- If two of the points are in the plane: their common side is the intersection.
- If the three vertices are in the plane: the whole triangle is contained.

It is advisable for the reader to check that all the cases can be classified this way. Although this section develops an algorithm that studies every case, the final code focuses on those that return a segment.

This is, not only because they are more common, but also due to the water-tightness of the imported solid, which ensures that the cases that return a point are unnecessary. The last case above can be dangerous though.

Having the previous classification in mind, the only thing left is to be able to retrieve the intersection between a side and the plane, as that is how the segment extremes can be found when they are not contained in the plane.

The parametric equation for a line segment is as follows:

$$P(\lambda) = P_1 + (P_2 - P_1)\lambda$$
$$\lambda \in [0, 1]$$

Being P a generic point inside the segment and P_1 and P_2 the extremes.

If we force this equation and the plane's equation at the same time, we can find a λ that satisfies both, and, therefore, the intersection between the side and the plane.

$$P(\lambda)_x = P_{1x} + (P_{2x} - P_{1x})\lambda$$
$$P(\lambda)_y = P_{1y} + (P_{2y} - P_{1y})\lambda$$
$$P(\lambda)_z = P_{1z} + (P_{2z} - P_{1z})\lambda$$
$$A\,P(\lambda)_x + B\,P(\lambda)_y + C\,P(\lambda)_z = D$$

There is no need to limit the values of λ since, as long as each point of the side is in a different semispace, that condition will be fulfilled.

The last equation determines the λ of the solution, which can be used to obtain the point's coordinates:

$$\lambda_s = \frac{D - A\,P_{1x} - B\,P_{1y} - C\,P_{1z}}{A\,(P_{2x} - P_{1x}) + B\,(P_{2y} - P_{1y}) + C\,(P_{2z} - P_{1z})}$$
$$P_s = P(\lambda_s) = P_1 + (P_2 - P_1)\lambda_s$$

The code that resulted from applying all of this can be found here.

2.2 Curve Generator

Using the previously described algorithm allows for the generation of the section curves.

A curve is defined as an ordered collection of points. In this case, the desired curves are closed, which means they have the same starting and ending point.

By obtaining the intersection of every triangle from the mesh and the plane, a collection of segments is obtained (other outputs can normally be ignored).

The first step then is to obtain every triangle from the solid's mesh, which is easily achieved via an ASCII `stl` file. In *Julia*, the `readdlm` function from the `DelimitedFiles` package is used.

The following algorithm will be able to find all the curves that define the section, but the solution may not be unique.

For example, a self-intersecting curve can be expressed as two different curves, but this should not affect the end result: the definition of the inside of the solid.

1. The algorithm starts by picking a starting segment.
    - One of its extremes will be the starting point.
    - The other is the second point.
2. Now the algorithm must find a segment that includes the second point.
3. After this, a third point is defined, and a segment including it must be found.
4. This process repeats until the point to be found is the same as the starting point.

The algorithm finishes when the start and end points are the same, which means a closed curve was found.

If there are still segments left, the algorithm can be applied again to find another curve.

The code that resulted from applying all of this can be found here.

The following figure shows the result of applying this algorithm:

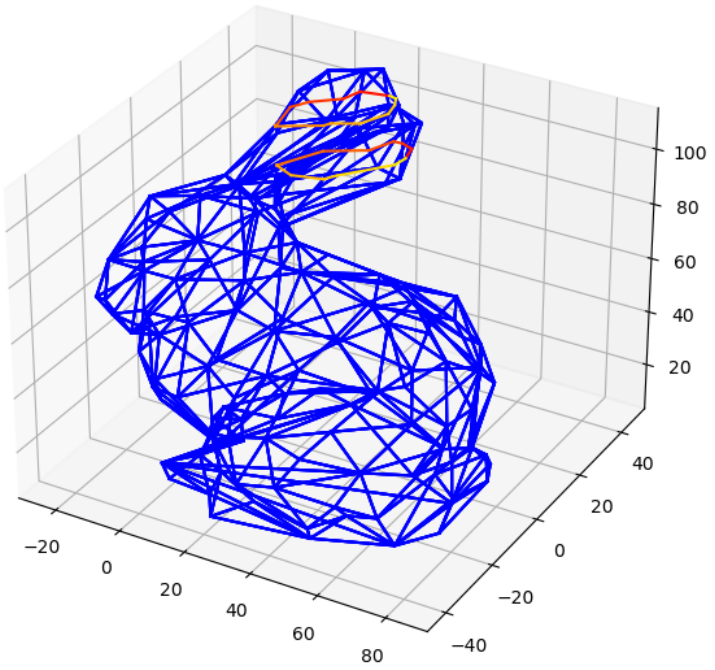


Figure 2.1: Desired result of the curve-gathering algorithm (red to yellow).

Note that the color gradient demonstrates that the points inside the curve are correctly ordered.

Moreover, in this case, two curves are found, showcasing that functionality of the algorithm.

The next step is, using this newly generated curves, and a point in space, to determine whether said point lies inside the section of the solid, and therefore the solid itself.

As stated before, a ray casting algorithm will be implemented.

2.3 Ray Casting

The ray casting algorithm is one of the simplest ways to detect if a point is inside or outside a curve, and the simplest algorithm for that purpose in the case of polygons.

It relies on the fact that, if the given closed curve is finite, a ray (one half of a line) casted from any point in space will eventually reach the outside of said curve and never enter again.

This means that, if we trace back every cut with the curve, and since every cut changes from inside to outside or vice versa, the parity of the number of cuts determines whether the point is inside or outside the curve.

For example, if the number of cuts is 5, we can trace from the infinite and the cuts are as follow: o-i-o-i-o-i. This means the point is actually inside the curve.

To sum up, if the cuts between a ray and a curve are even, the point from which the ray is casted lies outside the curve. If it were uneven, said point would be inside the curve.

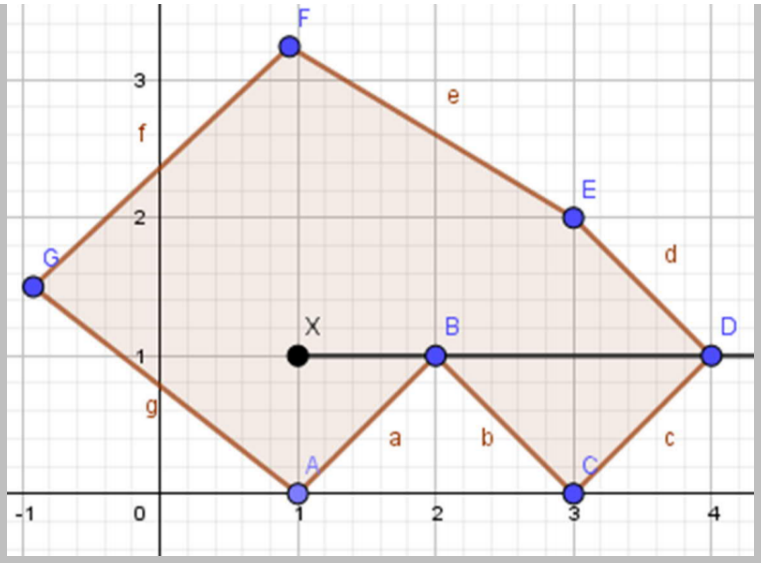
However, a problem can arise if the ray is tangent to the curve in one point, because then a cut happens, but the ray stays inside or outside. This can be fixed by looking at the derivative of the curve in the intersection point.

But, in the case of polygons, being "tangent" could mean the ray passes through a vertex, but passing through a vertex does not always imply this "tangency".

Example 2.3.1 (Determining whether a point is inside a polygon.)
The following polygon, point and ray are used (see figure).

As we can see, the ray technically cuts the polygon twice, both times through a vertex, but only one should be counted (D). This is because the cut in B is actually a "tangency".

This can be detected by checking if the previous point (A) and the next one (C) are in the same semi-space defined by the ray, which they are.
In the case of the cut in D, each of the following and previous points are i none semi-space.

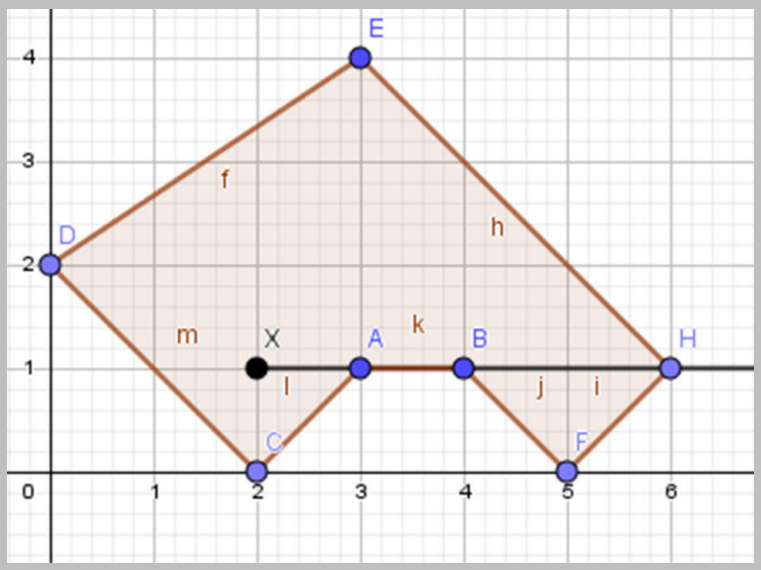


The number of cuts is, then, 1.

Example 2.3.2 (Determining whether a point is inside a polygon.)
The following polygon, point and ray are used (see figure).

This case is similar to the previous one, but now the tangency extends to a whole segment.

For this case, the idea is to study the point that goes before it (C), and the one that goes after (F), since they are in the same semispace, this can be treated as a tangency and not a cut.



The number of cuts is, then, also 1.

The previous examples help understand problematic cases induced by "tangency".

Now it is time to develop the algorithm:

1. Count all the cuts with segments that do not occur in vertices nor contain the whole segment (see this figure):
   - If both extremes are in the semi-space defined by the line perpendicular to the ray that does not contain the ray, there is no cut.
   - If at least one of the extremes is in said semi-space:
     - If each of them lies in a different semispace from those defined by the line that contains the ray, there is a cut.
2. For the rest of the cuts:
   - If a vertex's previous and next points are not in the ray.
     (a) If each of them is in one semi-space, there is a cut.
   - If a vertex's previous or next points are in the ray.
     (a) Obtain the next and previous points that are not in the ray.
     (b) If each of them is in one semi-space, there is a cut.

The following figure demonstrates the possible cases for a ray-segment intersection, excluding those occurring at one of the extremes.

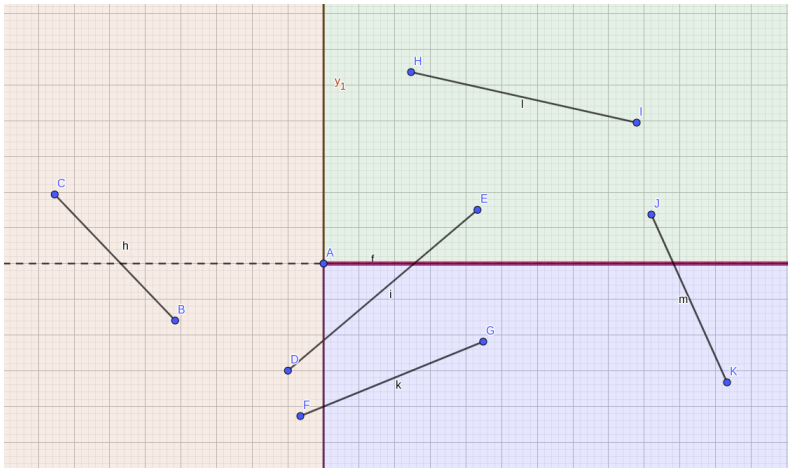


Figure 2.2: Possible ray-segment intersection cases.

The ray (red) origins at A, and extends to the right. As can be noted, only those segments with one extreme outside the red zone can be cut. Moreover, each of the extremes has to lie in a different semi-space from those defined by the line containing the ray (dashed).

Having all of this into consideration allows for an implementation of the ray casting algorithm. By retrieving the number of cuts a ray, sourced at a point, and a polygon have, it is determined whether or not said point lies inside the polygon.

The implementation in *Julia* of this algorithm can be found here.

3

Metamaterial Slicing Engine

Contents

The previous chapter described the algorithms used to determine whether a point is inside a solid or not.

In this chapter, those algorithms will be repeatedly used to create a cloud of points that describe the solid, which can then be connected with each other to form the metamaterial structure.

Moreover, an attempt to choose the mechanical properties of each cell is done, with the intention to match those determined in the generative design.

Finally, with the structure completely described, it will be time to export the data with the proper syntax, to be interpreted by different programs (*OpenSCAD*, *Nastran*, *Paraview*, etc.)

This chapter is, therefore, named after the program that it describes.

3.1 Common program

This section describes the initial steps of the program, which do not depend on the desired export.

It begins by importing the 3D solid geometry, importing the table of rigidity samples and defining a grid of points that contains this solid. This is done ensuring the distances between points are approximately equal in every axis.

Then, the common program mainly consists of two steps:

1. Determining which points from the grid are inside the solid (the set of these points will be called the cloud of nodes from now on).

2. Determining the geometrical parameters of each cell, by forcing the desired mechanical properties.

These two steps will result in two 3D arrays:

1. `cloud`: represents all the points in the grid. If a point's value is set to 1, said point is part of the cloud of nodes.

2. `cloud_mech`: defines a radius for each point of the grid. This radius is determined forcing the desired rigidity in that point.

To create the `cloud` array, and fulfill step 1, the `IsInside` function, composed of the algorithms described in Chapter 3, is used. By executing this function for every point inside the grid, all of those that are inside the solid are found.

To create the `cloud_mech` array, and fulfill step 2, the rigidity values are obtained in each point, by interpolating those imported via the rigidity table. After this, an approximate equation[4] is used to determine the radius that would give the desired rigidity. Finally, this radius is saved inside `cloud_mech`.

3.1.1 Obtaining radius from desired rigidity

First, the metamaterial density is obtained from the desired rigidity using the following approximate equation:

$$\left(\frac{\rho}{\rho_s}\right)^2 \approx \frac{E}{E_s} \text{ [4]}$$

Where E is the resulting/desired rigidity, E_s is the solid material's rigidity, ρ is the metamaterial density and ρ_s is the solid material's density.

It is obvious that this expression does not account for anisotropy, which will undoubtedly be present in the metamaterial.

To fix this, the cell orientation should be chosen to take advantage of this anisotropy. This should ensure the rigidity is at least the one obtained with the approximate formula.

Thus, the idea is to vary ρ in each node to produce the desired E. To achieve this, the edge's radius is modified using the following relationship for cubic cells:

$$\frac{\rho}{\rho_s} = 3\pi\left(\frac{r}{l}\right)^2 - 8\sqrt{2}\left(\frac{r}{l}\right)^3 \text{ [6]}$$

Being r the radius of the edges and l their length.

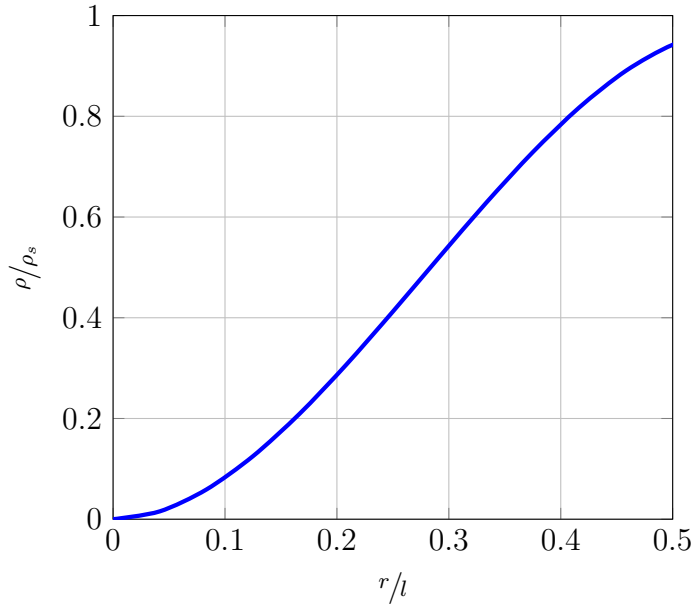
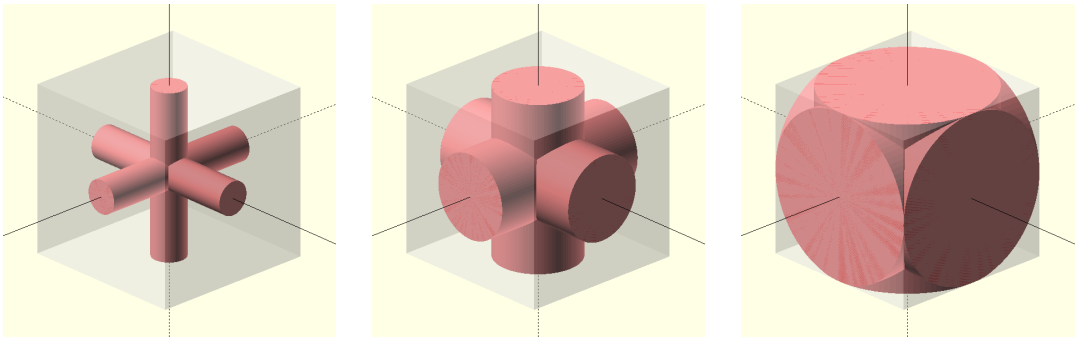


Figure 3.1: Relationship between radius and density.

To illustrate this equation, the following examples (rendered in OpenSCAD) are presented:



(a) $r/l = 0.1$ **(b)** $r/l = 0.25$ **(c)** $r/l = 0.5$

Figure 3.2: Examples of the relationship between radius and density.

As the reader may have noticed, the program needs to have the radius as a function of the density, and not otherwise. This means the previous expression must be inverted, which can be done approximately using polynomials:

$$\frac{r}{l} \approx 0.25 + 0.3859 \cdot \left(\frac{\rho}{\rho_s} - 0.412\right) - 0.05401 \cdot \left(\frac{\rho}{\rho_s} - 0.412\right)^2 + 0.26612 \cdot \left(\frac{\rho}{\rho_s} - 0.412\right)^3$$

The polynomial used includes the next two terms of the Taylor Series as well. The comparison between this approximation and the original equation can be visualized here:

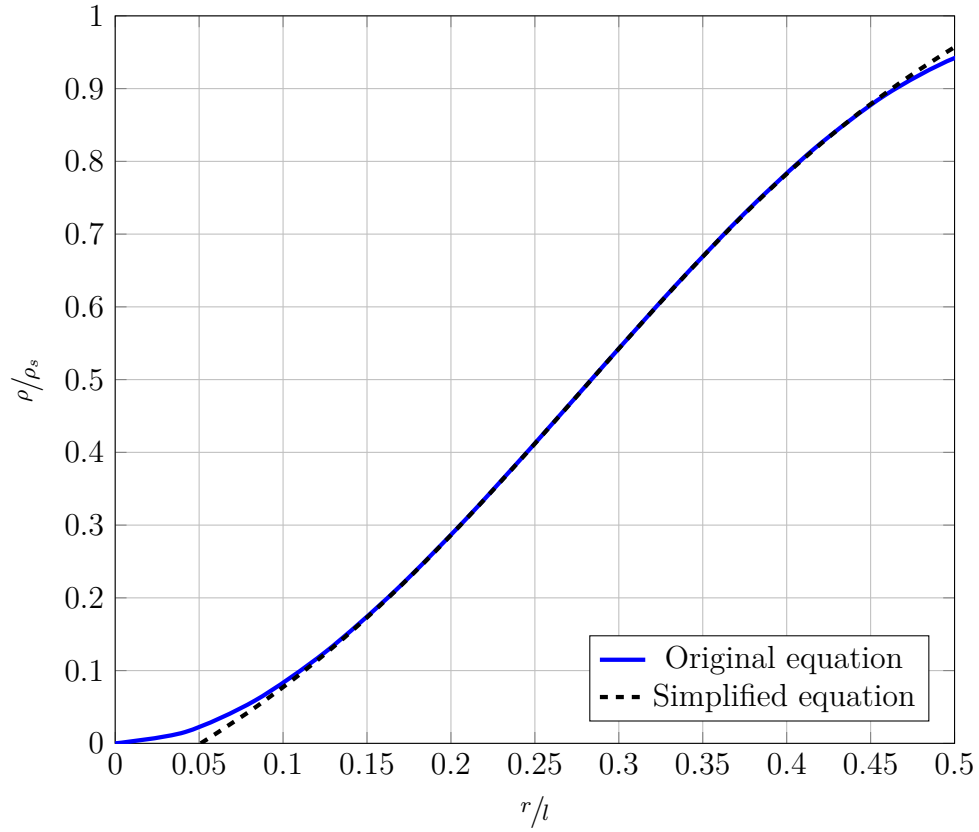


Figure 3.3: Simplified relationship between radius and density.

As can be seen, the simplified version fits almost perfectly in the middle range for r/l, only to deviate substantially for values under 0.1 and over 0.49. Values in the extremes are generally to be avoided, as the material is not really expected to behave as a metamaterial with too thin or too thick cell edges.

This means the previous approximation will be adequate for the most common cases, where r/l indeed falls inside $[0.1, 0.49]$.

A different approach to find the inverse function is to iterate the following expression:

$$x_n = \sqrt{\frac{y}{3\pi - 8\sqrt{2}x_{n-1}}}$$

Being $x = r/l$ and $y = \rho/\rho_s$. Using $x_0 = 0.25$ returns the following graph:

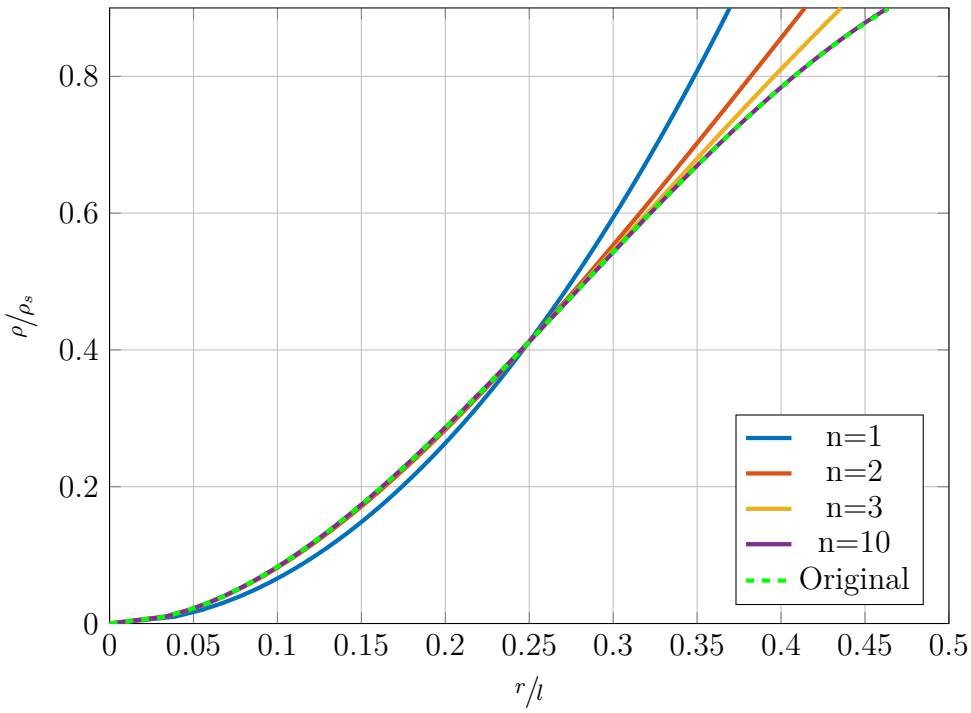


Figure 3.4: Iterated inverse relationship between radius and density.

As it can be seen, 10 iterations are enough to yield results close enough to the original function in its whole domain.

As of *August 20, 2021*, the *MSE* uses the polynomial approximation to obtain every radius.

3.2 Output formats

The purpose of the *MSE* is to transform a solid geometry into a metamaterial. This, of course, results in a series of output files, each corresponding to a different feature.

- 3D geometry to be printed: .scad
- FEM analysis: .bdf and .fem
- Property analysis: .vtu

Each of these files are to be opened using a different commercially available program.

In this section, each of those programs will be introduced, as well as an insight into said output files' structure.

3.2.1 OpenSCAD (.scad)

OpenSCAD is a free as in price, open-source, Computer-Assisted-Design piece of software, that allows script-based geometry generation. This means the 3D geometry is rendered from a text file, with `.scad` extension.

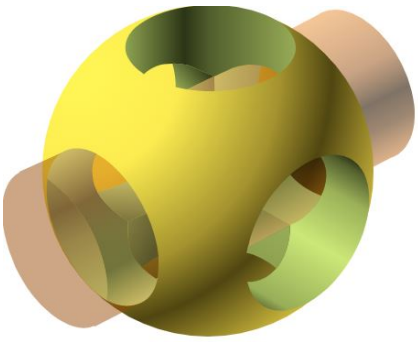


Figure 3.5: OpenSCAD logo.

This offers a huge advantage, as it allows for easy scripted geometry generation. Any program can generate 3D geometry easily by generating a .scad file, and then running it with *OpenSCAD*.

This is the premise to use *OpenSCAD*.

Regarding its features, OpenSCAD includes:

- Basic 3D geometric primitives: cube, sphere, cylinder, etc.
- Basic boolean operations: intersection, difference and union.
- Basic 3D geometric operations: scale, rotate, translate, hull, etc.
- Scripting: function definition, variables, looping, etc.

This features make really it easy to define a function that draws a cylinder from a point in space to another, with a given radius.

This function is then repeatedly used to form the metamaterial, changing the radius as necessary. The following example does exactly that:

```
1  $fn=6;
2  marg=0.98;
3  //draw segment between 2 specified points
4  module rod(p1,p2,r){
5      translate((p1+p2)/2)
6          rotate([-acos((p2[2]-p1[2]) / norm(p1-p2)),0,
7          -atan2(p2[0]-p1[0],p2[1]-p1[1])])
8              cylinder(r1=r, r2=r, h=norm(p1-p2), center=true);
9  }
10 //geometry definition
11 translate([-41,-101,28])
12     sphere(r=0.202);
13 rod([-41,-101,28],[-41,-101,31],marg*0.202);
14 rod([-41,-101,28],[-41,-98,28],marg*0.202);
15 rod([-41,-101,28],[-38,-101,28],marg*0.202);
```

Listing 3.1: OpenSCAD example.

Every `.scad` file exported from *MSE* reads the same, up to line 11, where geometry definition begins.

Two parameters are defined at the very beginning of the file, `$fn` defines the resolution of the primitives, and `marg` defines a multiplier to make edges slightly smaller than nodes, so they fit more naturally.

Low values of `$fn` are used to plot different primitives. For example, a cylinder with a value of $\$fn = 3$ will produce a triangular prism. In the case of *MSE*, the low values of `$fn` are due to memory limitations, and in order to reduce rendering times.

Rendering times with this method get large very easily, being in the order of days for node counts of around 70,000 (40x40x40).

Regarding the results, watertightness is ensured, and the geometry does not present any artifact. The main issue would be the lack of precision in the shape of spheres and cylinders, but, after 3D printing, it becomes unnoticiable.

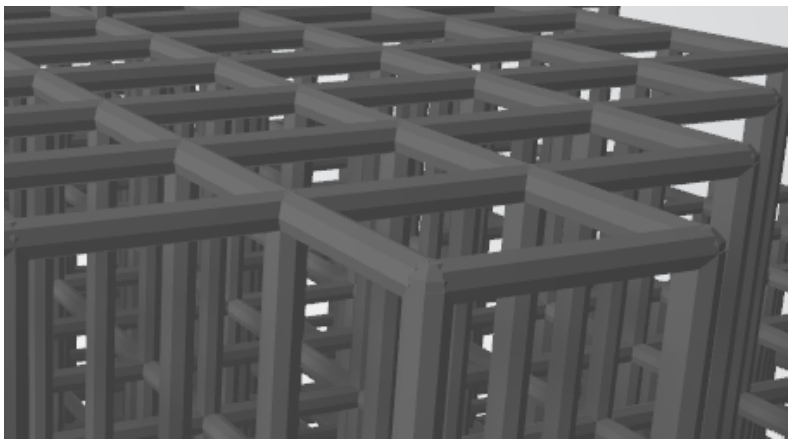


Figure 3.6: OpenSCAD quality example.

Initially, code snippets were implemented allowing for the generation of other geometry files, including one for *FreeCAD*. This solved the quality issue, resulting in much sharper renders.

This upgrade was at the expense of a higher rendering time, as well as reliability, since the program had a tendency to halt and/or crash.

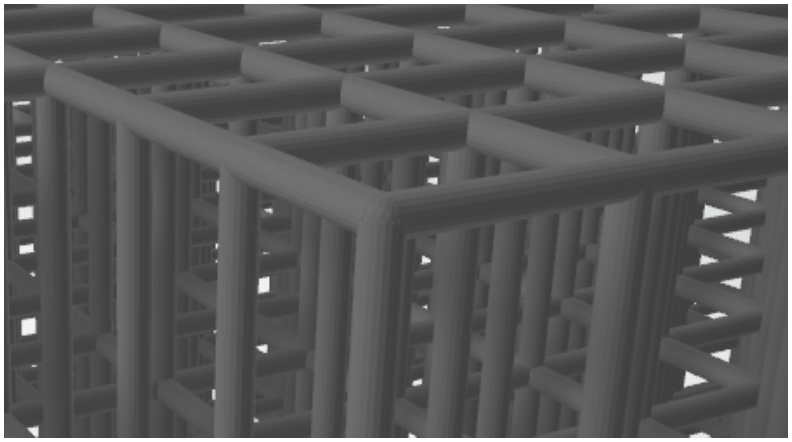


Figure 3.7: FreeCAD quality example.

3.2.2 Patran-Nastran (.bdf)

The *Patran-Nastran* suite, developed by `MSC Software`, is composed of a FEM model generator (*Patran*), a FEM solver (*Nastran*) and a results analyzer (*Patran*).

Figure 3.8: MSC Software logo.

Just as with *OpenSCAD*, one of Nastran's most important features (present in almost every FEM solver), is the possibility to import models via text file (`.bdf` extension), allowing for the model generation inside the *MSE* program.

In this project, the following configuration[5] is used:

- SOL 101: Linear-steady-elastic solution (small deformation).
- CBAR elements with PBARL configuration of type BAR.
- Only SPC1 used (no MPC).
- FORCE applied in the nodes.

All of this allows for a preliminary study under normal working conditions (no plasticity or damage).

Regarding the structure of the `.bdf` file, the following example may help:

```
ID,WIRES,SIZ40
SOL,101
TIME,5
CEND
TITLE=GENERATIVE METAMATERIAL
SUBTITLE=CASE 1
LOAD=10
SPC=11
DISP=ALL
STRESS=ALL
STRAIN=ALL
ELFORCE=ALL
SPCFORCE=ALL
```

```
PARAM,BAILOUT,-1
BEGIN BULK
MDLPRM,HDF5,0

$ NODES COME HERE

GRID,1,,0.0,0.0,0.0
GRID,2,,0.0,10.0,0.0
GRID,3,,10.0,10.0,0.0
GRID,4,,10.0,0.0,0.0

$ ELEMENTS COME HERE

CBAR,1,11,1,2,1.0,0.0,0.0
CBAR,2,12,2,3,1.0,1.0,0.0
CBAR,3,13,3,4,1.0,0.0,0.0
CBAR,4,12,4,1,1.0,1.0,0.0

$ GEO/MECH PROPS COME HERE

PBARL,11,21,,BAR
,0.75,0.75,,
PBARL,12,21,,BAR
,0.8,0.8,,
PBARL,13,21,,BAR
,0.84,0.84,,

MAT1,21,21e4,,0.3

$ BC AND LOADS COME HERE

FORCE,10,3,,15.0,0.0,0.0,-1.0

SPC1,11,123456,1
SPC1,11,123456,2
```

Listing 3.2: BDF example.

The first 16 lines are almost the same in every file generated by *MSE*, as they are in charge of setting common parameters such as the solution to use and the desired output information[5].

Next comes the `BULK DATA` section, where the FEM nodes (`GRID`) and elements (`CBAR`) are defined.

After that, the geometric (`PBARL`) and mechanical (`MAT1`) properties of the FEM elements are listed.

Finally, the boundary conditions (`SPC`) and the loads `FORCE` are defined.

The previous example generates the following model:

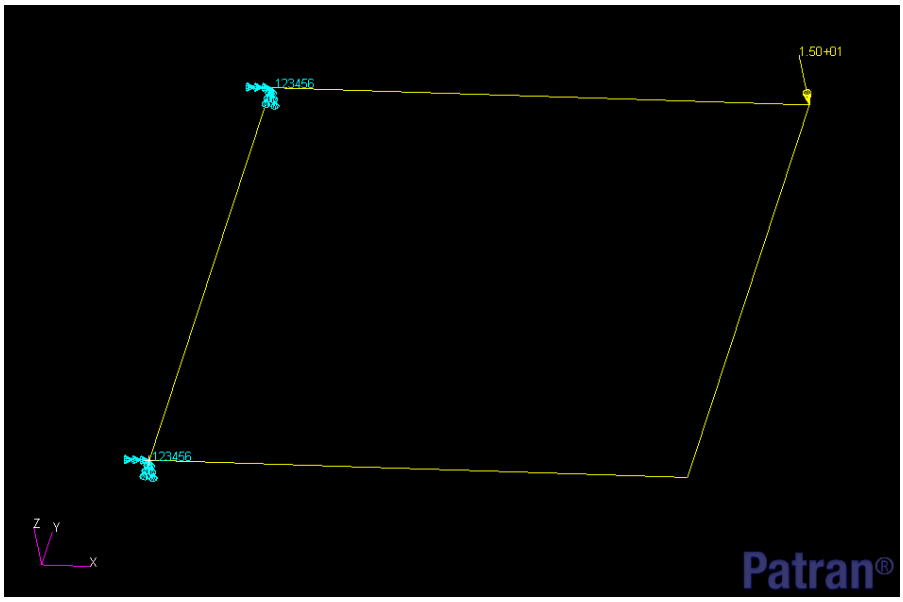


Figure 3.9: Patran model example.

Which, solved by *Nastran*, gives the following result:

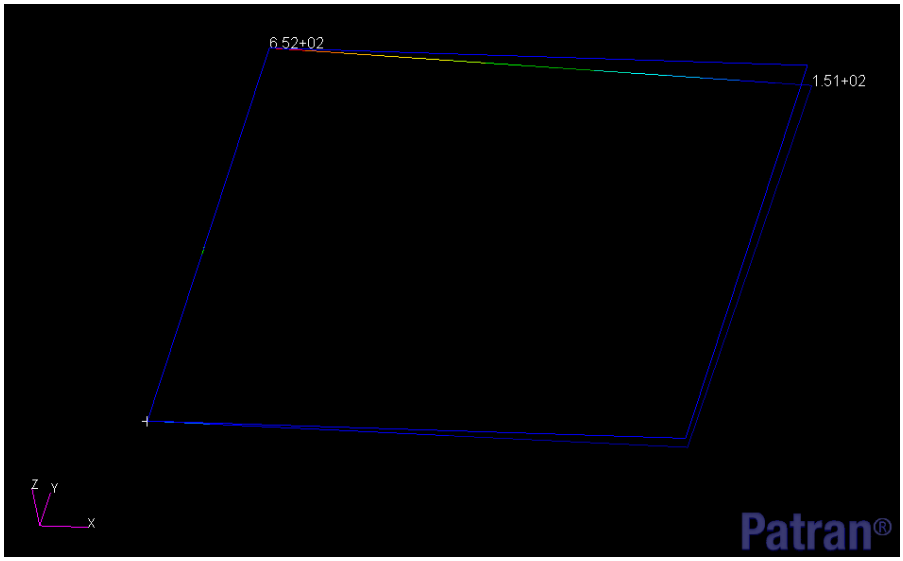


Figure 3.10: Patran model example solved.

All the intricacies of the `BDF` format may be consulted in Nastran's *Quick Reference Guide*[5], which was the main source for its implementation in *MSE*.

3.2.3 ParaView (.vtu)

ParaView is an open-source platform intended for the visualization and analysis of data, most commonly linked to a given geometry.

This makes it perfect for the study of FEM data, since the programs included with commercial solvers usually lack the power and flexibility required.

Figure 3.11: ParaView logo.

ParaView, just like the rest of programs here discussed, has the possibility of importing models via text files (`.vtu`), which again is something *MSE* takes advantage of.

ParaView also includes endless visualization options, that substantially help in the task of visualizing data.

In the case of *MSE*, the `.vtu` file generated contains information about the element's radius and desired Young Modulus.

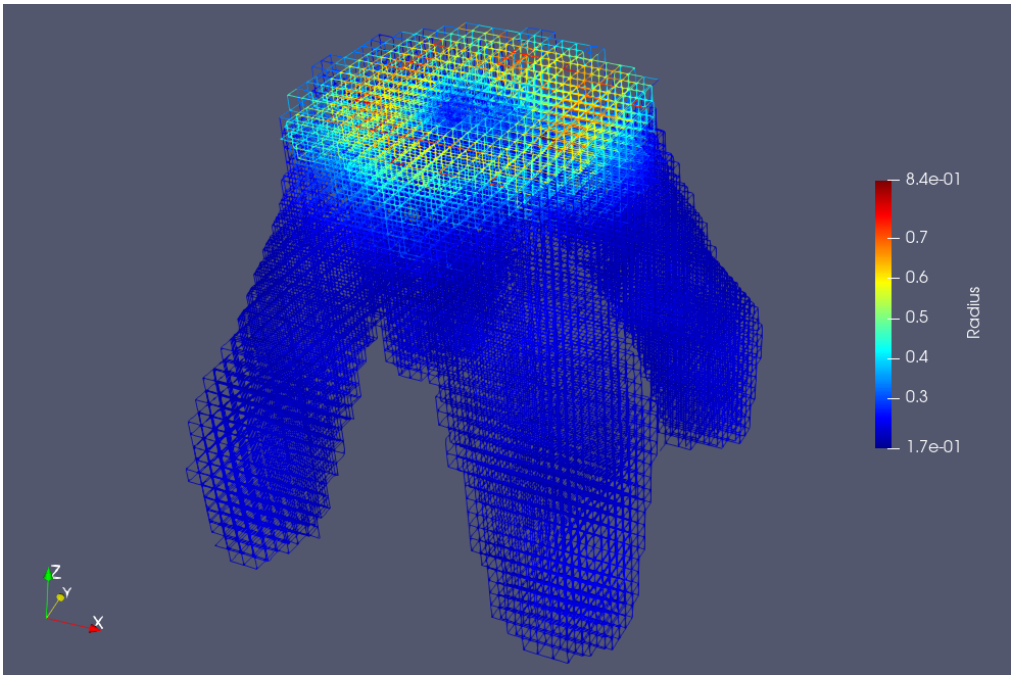


Figure 3.12: ParaView radius plot.

Regarding the VTU format, the following can be said:[3]

- The ASCII VTU format is based in XML.
- Tags are used to define the geometry and properties:
  - Points are defined in `<Points>`.
  - Their connectivity (elements) appears in `<Cells>`.
  - Element types are also set inside `<Cells>`.
  - Each of the properties is defined inside `<CellData>`.
- Each of the previous properties is contained in a `<DataArray>`, allowing for several entries of each type (multiple `<DataArray>` inside `<Points>`, `<Cells>` and `<CellData>`.).

```
<?xml version="1.0"?>
  <VTKFile type="UnstructuredGrid">
    <UnstructuredGrid>
      <Piece NumberOfPoints="4"  NumberOfCells="4">
        <Points>
          <DataArray type="Float64" NumberOfComponents="3" format="
   ascii">
0.0     0.0     0.0
0.0     10.0    0.0
10.0    10.0    0.0
10.0    0.0     0.0
          </DataArray>
        </Points>
        <Cells>
            <DataArray type="Int32" Name="connectivity" format="
   ascii">
1 2
2 3
3 4
4 1
          </DataArray>
          <DataArray type="Int32" Name="offsets" format="ascii">
2
4
6
8
```

```
25          </DataArray>
26          <DataArray type="UInt8" Name="types" format="ascii">
27 3
28 3
29 3
30 3
31          </DataArray>
32        </Cells>
33        <CellData>
34          <DataArray type="Float64" Name="Radius" NumberOfComponents
    ="1" format="ascii">
35 0.1
36 0.2
37 0.3
38 0.4
39          </DataArray>
40          <DataArray type="Float64" Name="Young Modulus"
    NumberOfComponents="1" format="ascii">
41 100
42 115
43 150
44 200
45          </DataArray>
46        </CellData>
47      </Piece>
48    </UnstructuredGrid>
49  </VTKFile>
```

Listing 3.3: VTU example.

Which in *ParaView* is rendered as shown in the figure below.

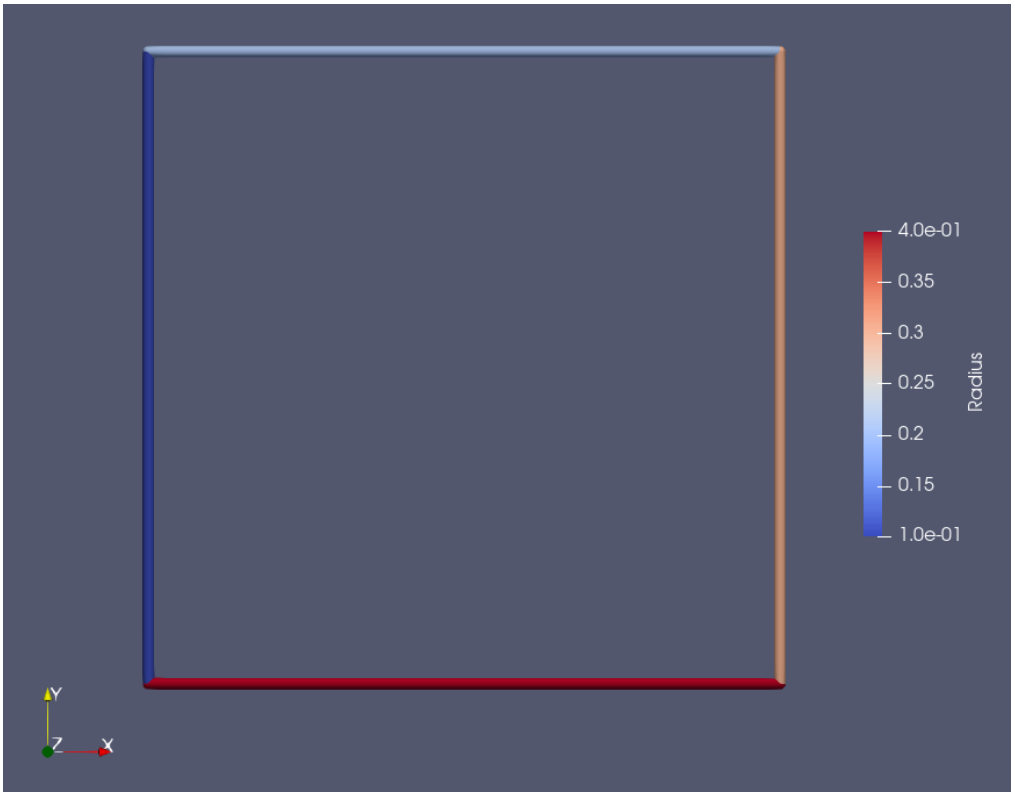


Figure 3.13: ParaView VTU example plot.

3.2.4 Others

In this subsection, other, less relevant, output formats are discussed.

FreeCAD

As explained before, *FreeCAD* was initially used to render the 3D stl model of the metamaterial, but had several disadvantages that left *OpenSCAD* as a better alternative.

FreeCAD is an open-source computer-aided design suite, composed of several modules able to share information. It allows for parametric design (adding layers of complexity with each module), as well as preliminary testing via FEM, using the `CalculiX` solver (also open-source).

Figure 3.14: FreeCAD logo.

The metamaterial geometry was generated via a python macro for *FreeCAD* (`.FCMacro`) that first all figures to a model first, to then select them all and export the result.

Overall, *FreeCAD* is a very interesting alternative to a considerable number of proprietary software platforms, ranging from design to validating and production.

This is why we advise using and collaborating with tools like these, since that way, knowledge and technology are made accessible to everyone.

MeshLab

Many times during this project, a `.stl` file had to be modified to fit the memory/computation requirements.

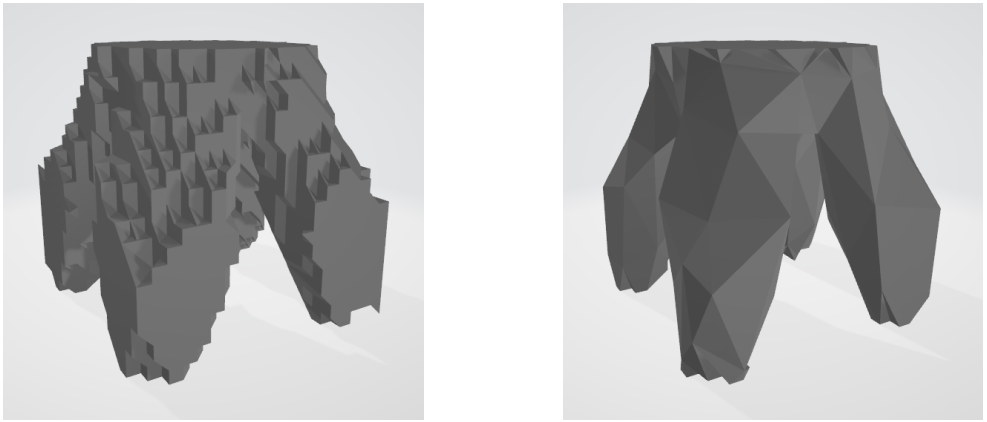
The input `.stl` in *MSE* should have less than 1000 triangles for the program to run in decent time spans. This means, many times the triangle count has to be considerably reduced. Moreover, OpenSCAD has a tendency to export `.stl` files with more triangles than strictly needed in the case of metamaterial, so a mesh simplification is also needed.

MeshLab[1] is an open-source system able to analyze, process and edit triangular meshes. It contains many features, such as filters designed to modify and fix the geometry.

Figure 3.15: MeshLab logo.

One of its advantages is the option to select in what triangles a filter should be applied, which allows to maintain some of the geometry intact.

For the simplification of meshes, the *Simplification: Quadric Edge Collapse Decimation* filter was used.



(a) Before decimation. **(b)** After decimation.

Figure 3.16: MeshLab Quadric Edge Collapse Decimation comparison.

3.3 MSE Workflow

This section provides a brief insight into the usage of the *Metamaterial Slicing Engine*.

3.3.1 Preprocessing

Given *MSE*'s limitations, it is necessary to conduct a series of changes to the input data for the program to run adequately.

First, as stated before, the geometric model has to be simplified to the order of < 1000 triangles, for *MSE* to run at a decent speed.

For this, MeshLab is used, and the result is saved as *"geo_data.stl"*.

It is very important to know that *MSE* only accepts `ASCII` encoded `.stl` files. Moreover, the vertex coordinates need to be in decimal notation, and the indentation should use a single space.

```
solid "OpenSCAD"
 facet normal -0.944150984287 -0.329513221979 0
  outer loop
   vertex -44.2175102234 -28.7303390503 17.0843906403
   vertex -40 -25 4.99999952316
   vertex -40 -30 4.99999952316
  endloop
 endfacet
endsolid "OpenSCAD"
```

Listing 3.4: STL example.

Now, regarding the desired mechanical properties, the generative design used exports a table with all kind of properties in each point, out of which *MSE* only uses the coordinates and Young Modulus. This means it is mandatory to first modify the `.csv` file to meet said requirements.

An example of the structure for the *"mech_data.csv"* can be found in the next page. It is important to use the character ";" as the separator.

	A	B	C	D
1	Points:0	Points:1	Points:2	Young Modulus
2	-45	-15	40	414448
3	-40	-10	35	529045
4	-45	-10	40	526132
5	-40	-10	40	362887
6	-45	-10	45	305014
7	-45	-5	45	503212

Table 3.1: CSV mech_data example.

If one wanted to fix the value of the young modulus for the whole geometry, a `CSV` file with the previous structure but with a single entry (random-finite point coordinates and desired Young Modulus) besides titles should be introduced.

3.3.2 Runtime

The following example shows the result of running *MSE* with a given geometry and mechanical properties:

```
  METAMATERIAL STL SLICER ENGINE
------------- v 0.1 -------------

--------- DEFINE PARAMS ---------
Define grid size, higher means smaller cells (int):
25

Define n of PBARL sizes, higher means smoother cells (int):
Default (15)

Output file format (BDF or SCAD or VTU or ALL):
Default (ALL)

Cell type (c or d or any combination):
cd

Define Young Modulus (int):
210000

Define boundary conditions case (0=None):
2

----------- EXECUTION -----------
Grid is: 24x24x27
Creating node cloud...
Getting mechanical properties in each node...
Generating "MSE_24x24x27_15-c.bdf" file...
Generating "MSE_24x24x27_15-d.bdf" file...
Generating "MSE_24x24x27_15-c.scad" file...
Generating "MSE_24x24x27_15-d.scad" file...
Generating "MSE_24x24x27_15-c.vtu" file...
Generating "MSE_24x24x27_15-d.vtu" file...
```

Listing 3.5: MSE runtime example

As it can be seen, the user is prompted with 5 questions, to define 5 different runtime parameters. For the first, third and fifth, the user typed a value, and, for the rest, just pressed enter to use the default values.

Next, the grid size is printed, showcasing how the grid size parameter is used. Instead of dividing each direction (x, y and z) in the same number of cells, each of them is divided to fit the geometry's size in that direction.

This means that, in our case, the solid is as wide as long, and slightly taller. Therefore the grid size is $24x24x27$ which results in a total of 15,552 nodes, a number quite close to the 15,625 expected from a $25x25x25$ grid, but with the advantage of producing almost cubic cells.

After this, the node cloud is generated, and the mechanical/geometric properties for each node are obtained. During these two processes, a percentage is periodically updated, to show the progress in real time. This, of course, can not be seen after the program has completed.

Note that boundary condition cases are hardcoded into *MSE*, meaning the only way to define new cases is by modifying the existing code, or by selecting case 0 and defining the boundary conditions inside *Patran*, or whichever `FEM` model editor is preferred.

3.3.3 Postprocessing

The files generated by *MSE* are all ready to use, but it is important to note that in the case of `.scad` files the geometry still needs to be rendered to obtain a `.stl` file.

Regarding the generated `.stl` file, it sometimes can be a good idea to apply a decimation filter to the geometry so as to make it easier to handle.

Finally, the `.stl` file can be easily imported into any 3D-Printing slicer to generate a ready to print file. Given the anisotropy of the structure, it is highly recommended to orient the piece before slicing.

4
Cases Studied

Contents

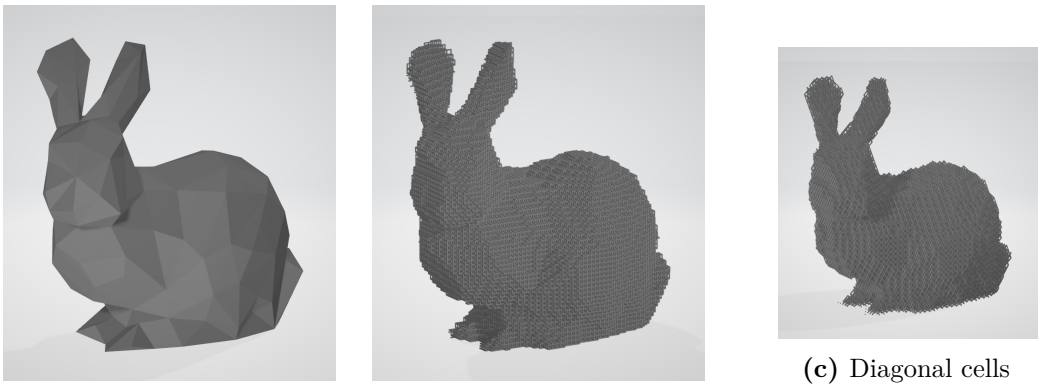
4.1 Stanford Bunny

The Stanford Bunny is widely recognised as one of the most used meshes to troubleshoot and test geometry related software. In this case, a low-poly version was repeatedly used to test the early phases of the program.

One may find several figures portraying the bunny in this document, mainly in Chapter 2 and Chapter 3.

Moreover, before implementing the mechanical mimetic engine, several versions of the bunny were turned into metamaterial and rendered as `.stl`. This tests were

able to confirm the method and to discover its computational cost.



(a) Original mesh **(b)** Cubic cells **(c)** Diagonal cells

Figure 4.1: Stanford Bunny examples.

Some of this meshes were 3D-Printed for demonstration purposes:

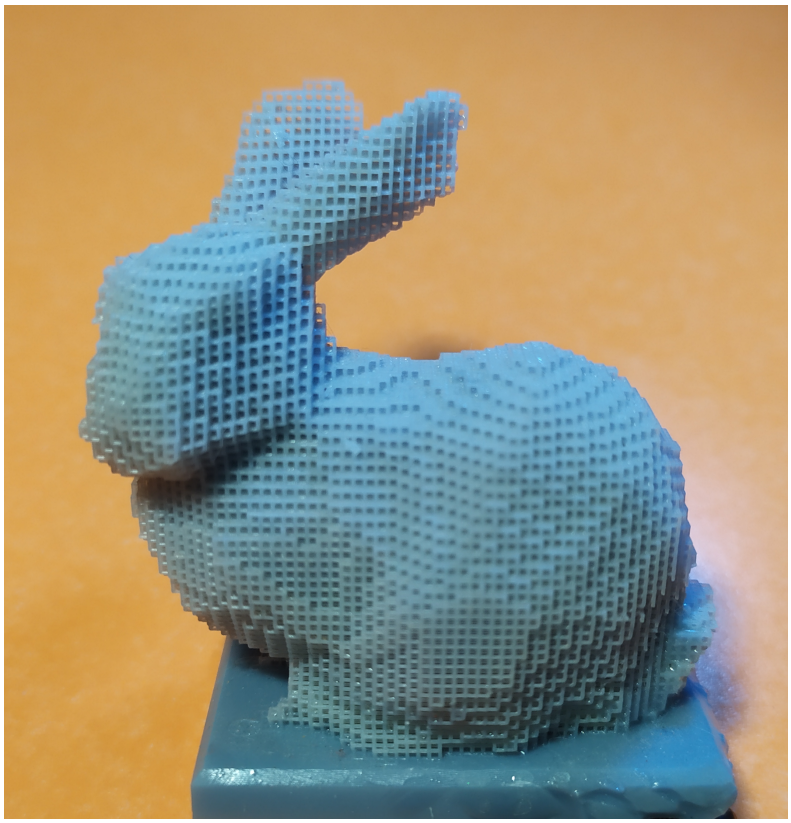


Figure 4.2: 3D-Printed Stanford Bunny.

This one was printed using `SLA` resin in an *Anycubic Photon Mono X* printer.

4.2 Cylinders

After validating the metamaterial engine, it was time to attempt implementing the mimetic engine, which is in charge of defining the geometric properties in every cell, in search of obtaining the desired mechanical properties in each point.

The first tested generative design combines two different loads with some boundary conditions, resulting in the following geometry:

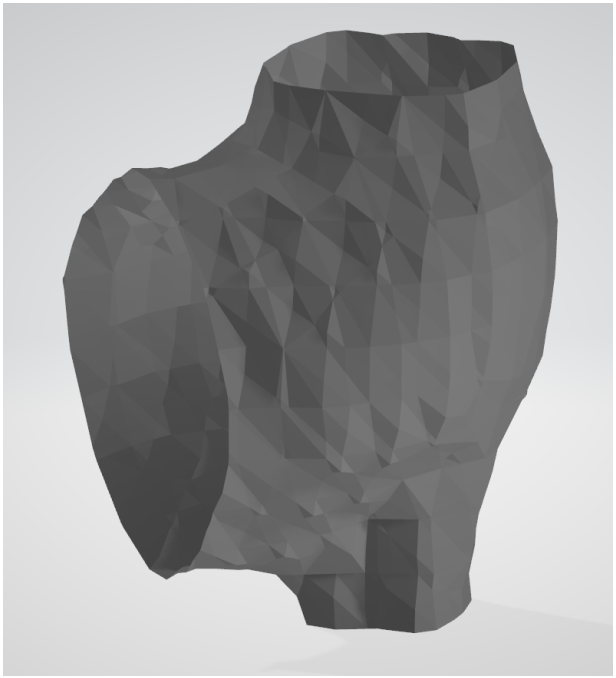


Figure 4.3: Cylinders original mesh.

It was then fixed to ensure watertightness, as well as decimated to reduce the triangle count:

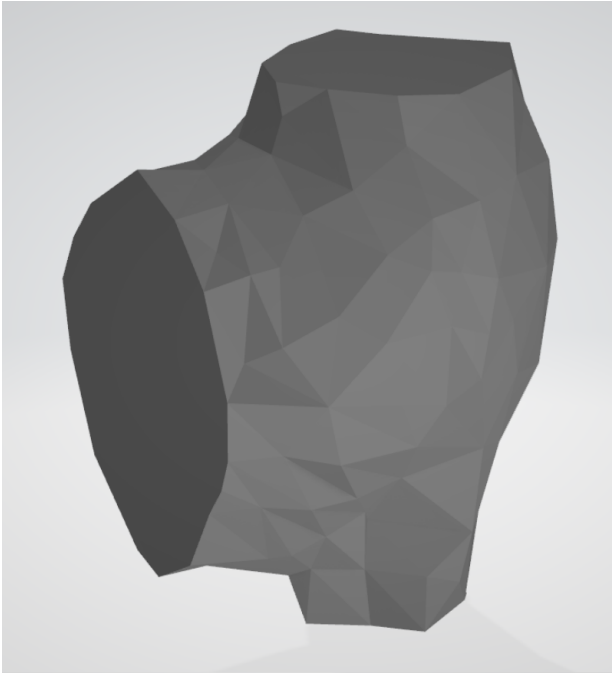


Figure 4.4: Cylinders fixed mesh.

Next, a mimetic design was rendered from the `.scad` generated via *MSE*:

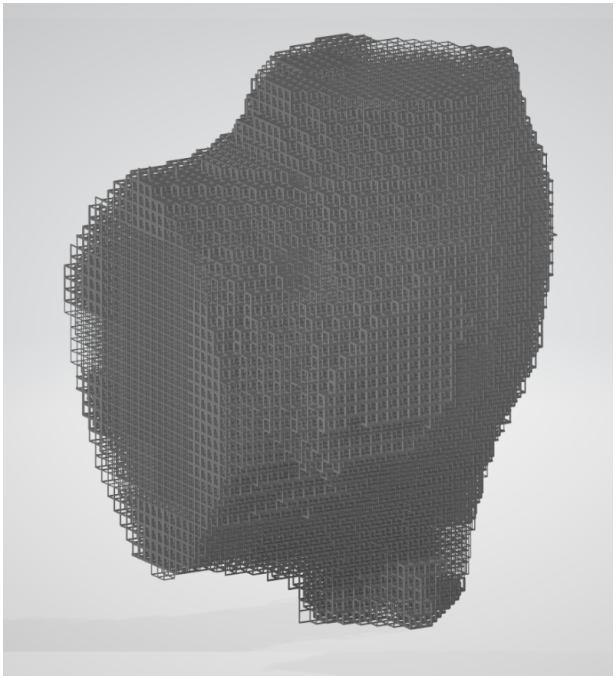


Figure 4.5: Cylinders metamaterial mesh.

Finally, the `.bdf` file was run in *Nastran*, which rendered the following result:

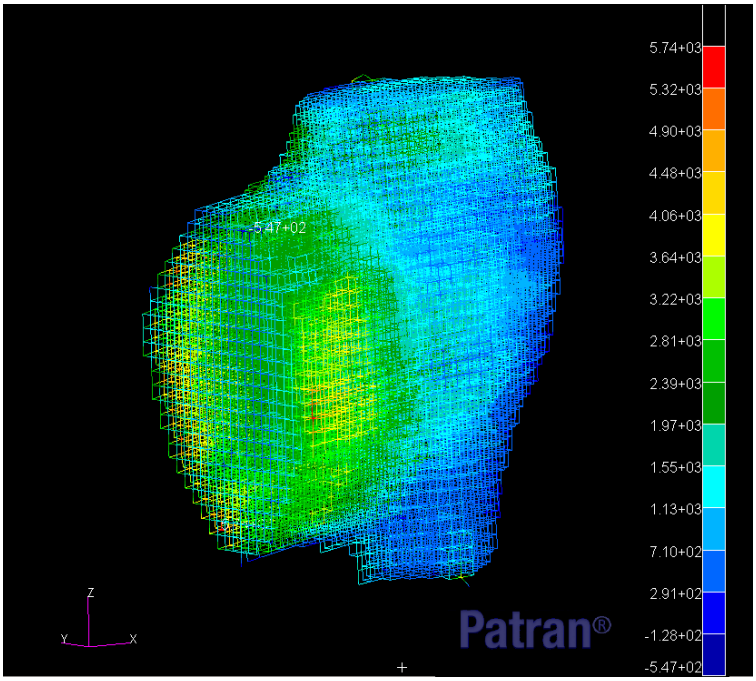


Figure 4.6: Cylinders metamaterial FEM analysis.

4.3 Stool

The next generative designs (Stool, Bone, Hollow Bone and Tube) were generously handed by my colleague *Miguel González Marco*[2], who happens to be working in the generative design area. Although these designs were generated using traction loads, a more realistic use case for them is under compression forces.

This design, meant to mimic that of a chair, suffered a similar transformation to the previous one:

(a) Original mesh **(b)** Decimated mesh **(c)** Metamaterial mesh

Figure 4.7: Stool mesh evolution.

As can be seen in the next figure, the upper part of the metamaterial design shows a ring of stiffness. This ring represents the need to transmit the load to the four legs of the stool, reinforcing the idea that the method here described is valid.

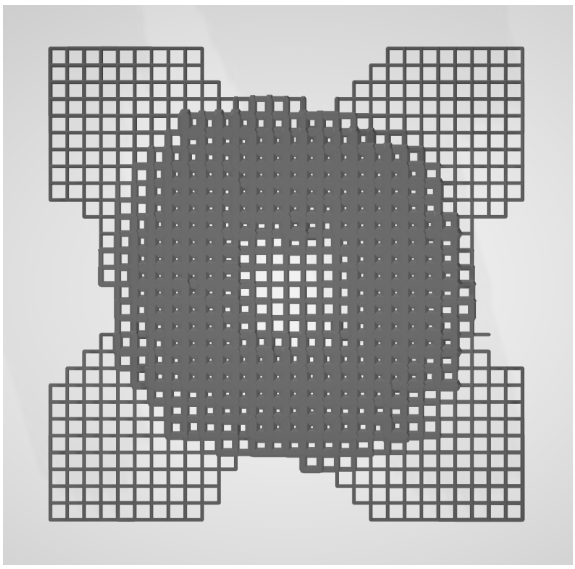


Figure 4.8: Stool metamaterial mesh.

Furthermore, the model was imported to *Patran-Nastran* and the following result was obtained:

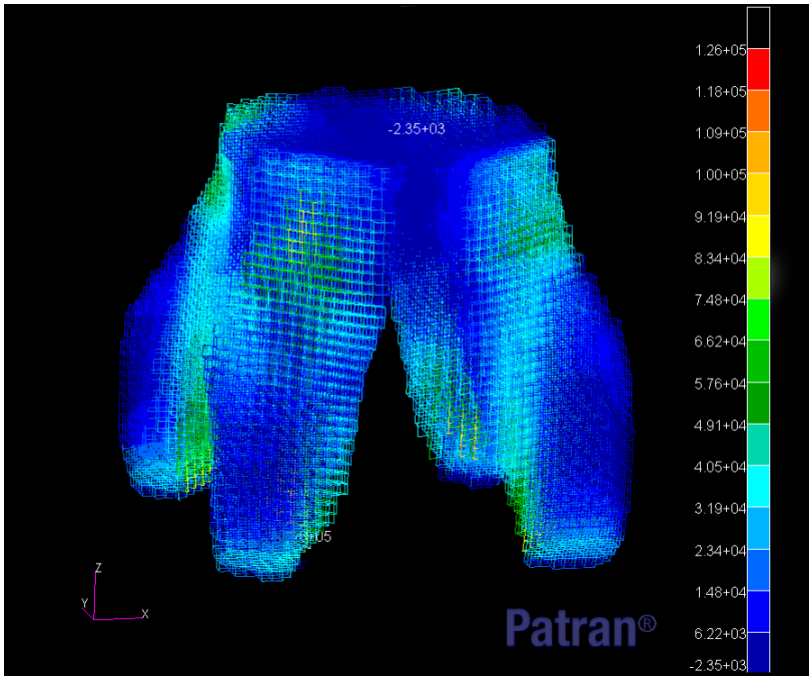


Figure 4.9: Stool metamaterial FEM analysis.

4.4 Bone

Just as with the previous examples, this design suffered the following transformation:

(a) Original mesh **(b)** Decimated mesh **(c)** Metamaterial mesh

Figure 4.10: Bone mesh evolution.

In this case, the generative design ended with a Young Modulus' table with

considerable stiffness gradients. This resulted in the final mesh having appreciable rigid blobs.

4.5 Hollow Bone

In this case, the design features a hole, making it completely necessary to have a strong inner point detection algorithm. This proves *MSE* detects holes as part of the exterior of the geometry.

(a) Original mesh **(b)** Decimated mesh **(c)** Metamaterial mesh

Figure 4.11: Hollow Bone mesh evolution.

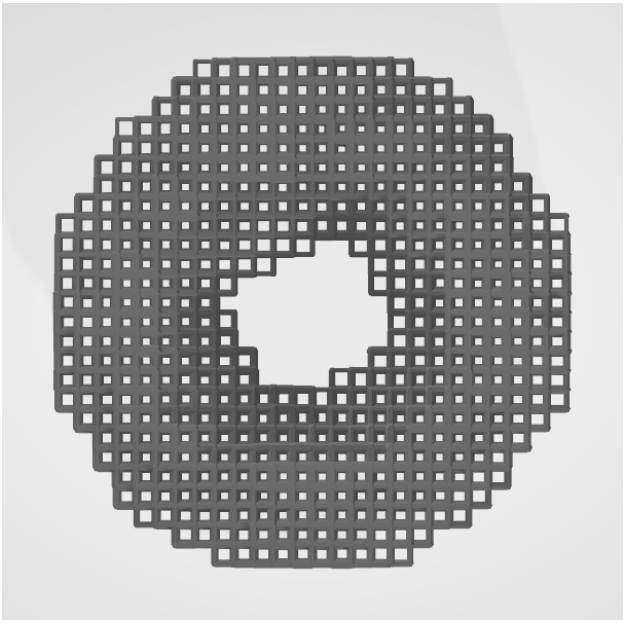
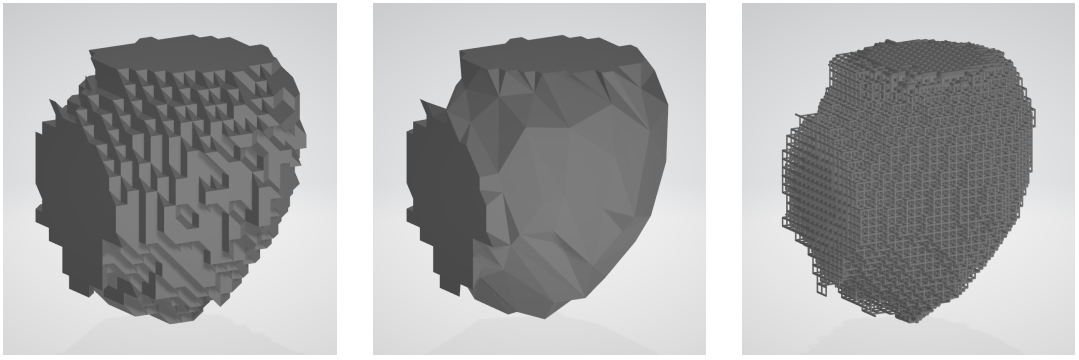


Figure 4.12: Hollow Bone top view.

4.6 Tube

Finally, the last case is designed to support the load of a pipe elbow:



(a) Original mesh **(b)** Decimated mesh **(c)** Metamaterial mesh

Figure 4.13: Tube mesh evolution.

The next figure shows the generative design stiffness criterion, material is placed near the boundary conditions, where it is most needed to transfer loads:

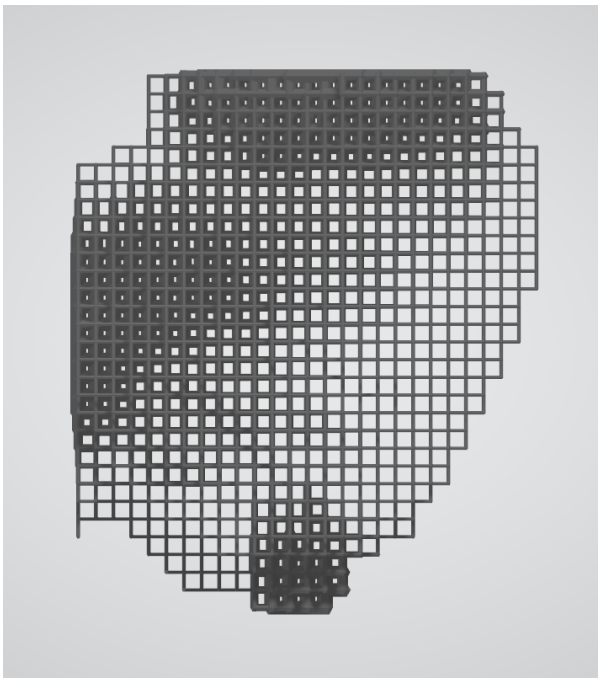


Figure 4.14: Tube detail.

Lastly, a FEM model was generated and tested in *Patran-Nastran*:

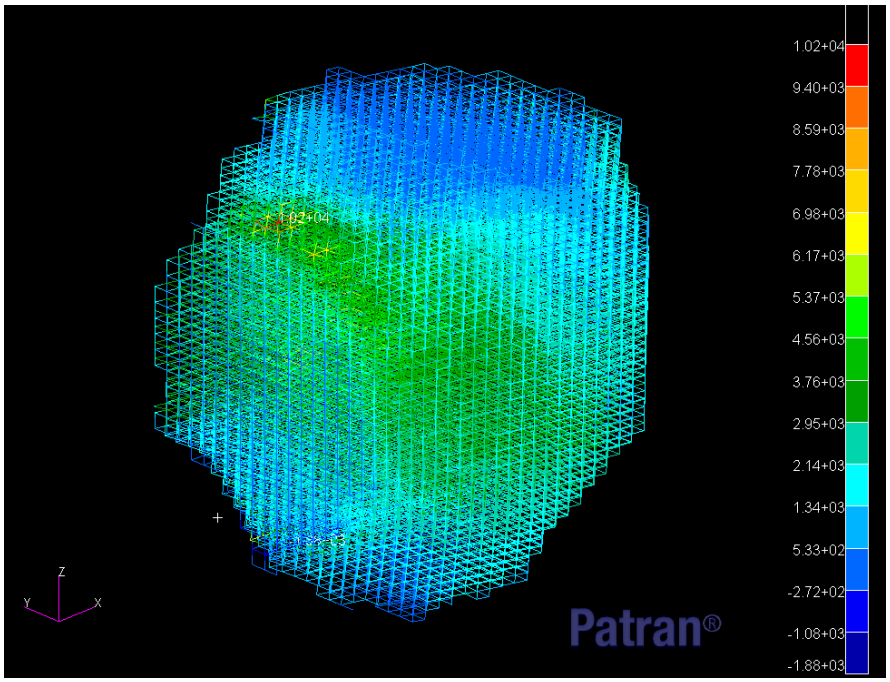


Figure 4.15: Tube metamaterial FEM analysis.

Appendices

A
Code Samples

A.1 Plane-Triangle Intersection Algorithm

```
function GetPoints(plane::Vector,p1::Vector,p2::Vector,p3::Vector)
    #usage: plane vector Ax+By+Cz=D -> [A,B,C,D];
    #triangle p1,p2,p3 by its R3 points;
    A = plane[1]
    B = plane[2]
    C = plane[3]
    D = plane[4]

    n = plane[1:3]

    if A!=0
        pp = [D/A,0,0]
    elseif B!=0
        pp = [0,D/B,0]
    elseif C!=0
        pp = [0,0,D/C]
    else
        return [0,0,"FATAL ERROR: UNDEFINED PLANE"]
    end

    v1 = p1-pp
    v2 = p2-pp
    v3 = p3-pp

    s1 = sign(sum(v1.*n))
    s2 = sign(sum(v2.*n))
```

```
    s3 = sign(sum(v3.*n))

    #we analyze the vertices that are inside the plane si=0
    if s1^2+s2^2+s3^2 == 0
        return [0,0,"TRIANGLE in PLANE"]
    elseif s1^2+s2^2 == 0
        return [p1,p2,"p1 and p2 in PLANE"]
    elseif s3^2+s2^2 == 0
        return [p2,p3,"p2 and p3 in PLANE"]
    elseif s1^2+s3^2 == 0
        return [p1,p3,"p1 and p3 in PLANE"]
    elseif (s1 == 0) & (s2==s3)
        return [p1,0,"p1 in PLANE"]
    elseif (s2 == 0) & (s1==s3)
        return [p2,0,"p2 in PLANE"]
    elseif (s3 == 0) & (s1==s2)
        return [p3,0,"p3 in PLANE"]
    elseif s1==s2==s3
        return [0,0,"TRIANGLE ^ PLANE=0"]
    else
    #we try to find what vertex is alone in one side, since the
segment between the other two doesn't intersect the plane

        if s1==0 s1=1 end       #if exists, we treat the vertex in
plane as if in subesp 1
        if s2==0 s2=1 end
        if s2==0 s3=1 end

        # news = [s1,s2,s3] + ((([s1,s2,s3].^2).^0.5-[1,1,1]).^2)
.^0.5      #we ignore vertices in the plane and treat them as in
 supesp 1
        #     s1 = news[1]
        #     s2 = news[2]      #THIS METHOD PROVED TO BE SLOWER
        #     s3 = news[3]

        ss = sign(s1+s2+s3)
        if s1 != ss     #we generalize every case, being pa the
alone point
            pa = p1
            pb = p2
            pc = p3
        elseif s2 != ss
            pa = p2
            pb = p1
            pc = p3
```

```
        else
            pa = p3
            pb = p2
            pc = p1
        end

        lambda1 = D-A*pa[1]-B*pa[2]-C*pa[3]
        lambda1 = lambda1/(A*(pb[1]-pa[1]) +
         B*(pb[2]-pa[2]) + C*(pb[3]-pa[3]))

        lambda2 = D-A*pa[1]-B*pa[2]-C*pa[3]
        lambda2 = lambda2/(A*(pc[1]-pa[1]) +
        B*(pc[2]-pa[2]) + C*(pc[3]-pa[3]))

        return [pa+lambda1*(pb-pa),pa+lambda2*(pc-pa),"SUCCESS"]

    end

end
```

Listing A.1: GetPoints function

A.2 Curve Generator Algorithm

The following code relies in the function `GetPoints`, which determines the intersection between a triangle and a plane. This function is defined here.

```
DATA = readdlm("Bunny-LowPoly_ascii.stl", ' ', '\n')  ##Import
   geometry
siz=size(DATA)

global vertex_DATA = [0 0 0]
global inter_DATA = []

for i = 3:7:siz[1]    ##get intersection with all triangles

    v1 = DATA[i+1,5:7]
    v2 = DATA[i+2,5:7]
    v3 = DATA[i+3,5:7]

  x = [v1[1],v2[1],v3[1],v1[1]]
  y = [v1[2],v2[2],v3[2],v1[2]]
  z = [v1[3],v2[3],v3[3],v1[3]]

  plot3D(x,y,z, "b")
```

```
18
19      global vertex_DATA = [vertex_DATA; transpose(v1); transpose(v2);
        transpose(v3)]
20
21      CORTE = GetPoints([0.0,0.0,1.0,50],v1,v2,v3)
22      if CORTE[1]!=0
23          if CORTE[2]!=0
24              push!(inter_DATA, [CORTE[1],CORTE[2]])  ##save segments
25          end
26      end
27  end
28
29  curves = []
30
31  while length(inter_DATA)>0  ## while there are curves to find
32
33      ##sort a curve
34
35      global curves
36
37      global sorted_DATA = [inter_DATA[1][1],inter_DATA[1][2]]
38      deleteat!(inter_DATA, 1)
39      global find = sorted_DATA[2]
40
41      while sum((sorted_DATA[1] .- find).^2)>0.0001 ##otherwise point
       is starting point and curve was found
42
43          global find
44          global inter_DATA
45          global sorted_DATA
46
47          for i=1:length(inter_DATA)
48
49              global find
50              global inter_DATA
51              global sorted_DATA
52
53              if sum((inter_DATA[i][1] .- find).^2)<=0.0001 ##detect
       point (direct == can not be applied due to rounding errors)
54                  push!(sorted_DATA, inter_DATA[i][2])
55                  find = inter_DATA[i][2]
56                  deleteat!(inter_DATA, i)
57                  break
58              end
59
```

```
            if sum((inter_DATA[i][2] .- find).^2)<=0.0001
                push!(sorted_DATA, inter_DATA[i][1])
                global find = inter_DATA[i][1]
                deleteat!(inter_DATA, i)
                break
            end

        end

    end

    push!(curves, sorted_DATA)    ##add curve

end
```

Listing A.2: Code to gather all curves

A.3 Ray Casting Algorithm

```

function IsInside(curve::Vector,p::Vector)

    ## get cuts with points
    pcuts = 0
    # get intersections
    inter = zeros(1,length(curve))

    for i = 1:length(curve)
        if curve[i][2]==p[2]
            if curve[i][1]>=p[1]
                global inter[i] = 1
            end
        end
    end

    #we loop through the points to see where inter changes from 0 to
    1 and vice versa
    for i = 1:length(inter)

        if inter[i] == 0

            if inter[(i)%length(inter)+1] == 1  #next element
                if inter[(i-2+length(inter))%length(inter)+1] == 1
   #prev element
                    inter[i] = 4    #finish and start here
```

```
25                else
26                    inter[i] = 2    #start here
27                end
28            else
29                if inter[(i-2+length(inter))%length(inter)+1] == 1
    #prev element
30                    inter[i] = 3    #finish here
31                end
32            end
33
34        end
35
36     end
37
38     inside = false  #we start outside of a group of 1s
39     start = -1
40     for i = 1:length(inter)
41
42         if inside
43             if (inter[i] == 3)   #if reached end
44
45                 y1 = min(curve[start][2],curve[i][2])
46                 y2 = max(curve[start][2],curve[i][2])
47
48                 if (y1<p[2]) && (y2>p[2])   #if each one in one
    region
49                     pcuts+=1
50                 end
51
52                 inside = false
53             end
54
55             if (inter[i] == 4)   #if reached end and start
56
57                 y1 = min(curve[start][2],curve[i][2])
58                 y2 = max(curve[start][2],curve[i][2])
59
60                 if (y1<p[2]) && (y2>p[2])   #if each one in one
    region
61                     pcuts+=1
62                 end
63
64                 start = i
65                 inside = true
66             end
```

```
67
68          else
69              if (inter[i] == 2) || (inter[i] == 4)
70                  start = i
71                  inside = true
72              end
73          end
74
75      end
76
77      if inside   #if still inside, find last ending
78          for i = 1:start
79              if (inter[i] == 3) || (inter[i] == 4)   #if reached end
80
81                  y1 = min(curve[start][2],curve[i][2])
82                  y2 = max(curve[start][2],curve[i][2])
83
84                  if (y1<p[2]) && (y2>p[2])   #if each one in one
    region
85                      pcuts+=1
86                  end
87
88                  inside = false
89                  break
90              end
91          end
92      end
93
94      ##check cuts with segments
95      scuts = 0
96
97      for i = 1:length(curve)
98
99          v1 = curve[i]
100         v2 = curve[(i)%length(curve)+1]
101
102         if !(max(v1[1],v2[1])<=p[1]) && ((max(v1[2],v2[2])>p[2])&&(
    min(v1[2],v2[2])<p[2]))
103
104             if ((v1[1]+(v2[1]-v1[1])/(v2[2]-v1[2])*(p[2]-v1[2]))>p
    [1])
105
106                 scuts +=1
107
108             end
```

```
109
110         end
111
112     end
113
114     return [pcuts, scuts]
115
116 end
```

Listing A.3: Code to determine if a point is inside a polygon

B
Diagrams

Contents

B.1 3DMetMec Flow Chart

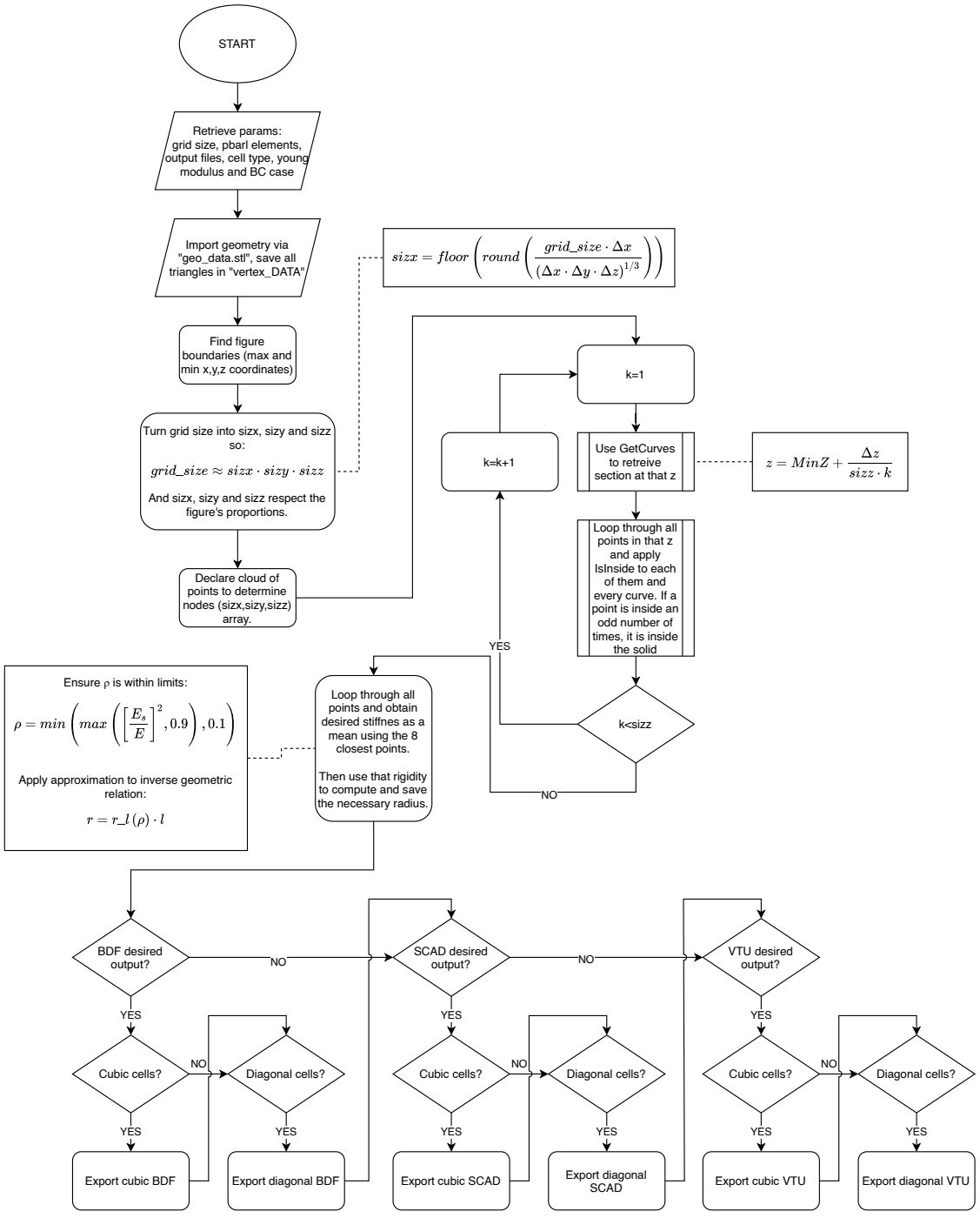


Figure B.1: 3DMetMec program flow chart.

B.2 Function's Flow Charts

B.2.1 GetPoints flow chart

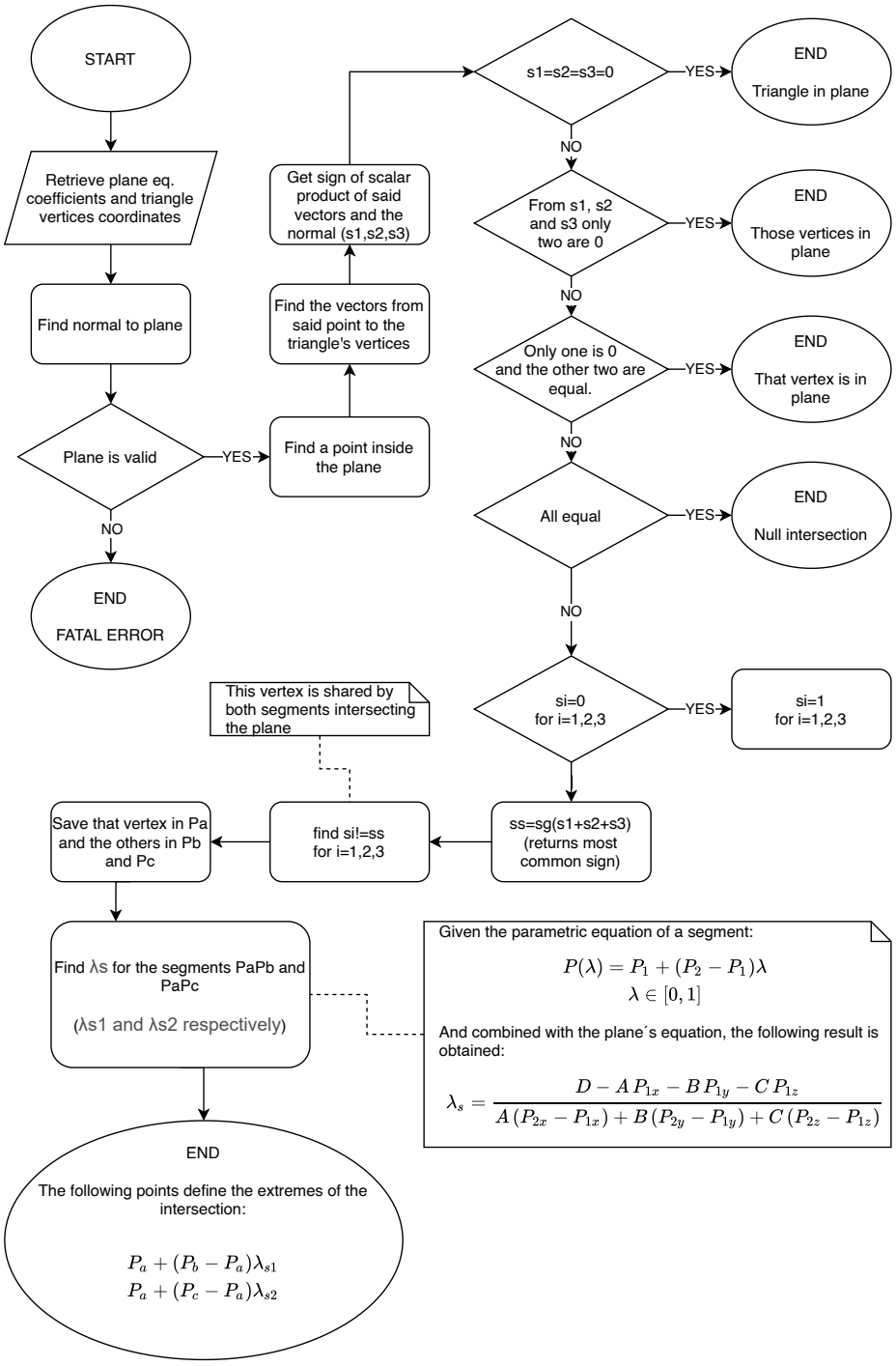


Figure B.2: GetPoints function flow chart.

B.2.2 Curves generator flow chart

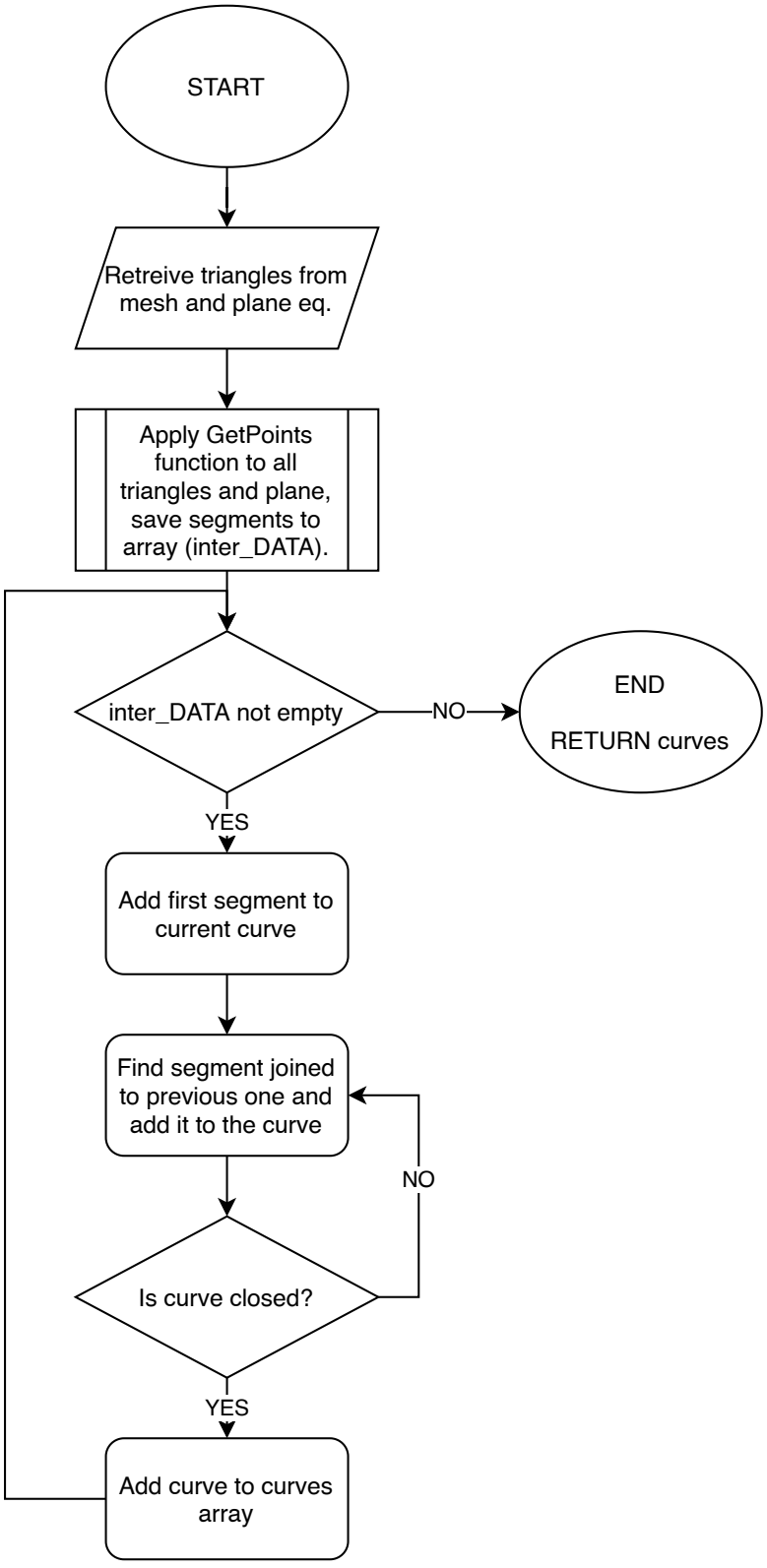


Figure B.3: Curves generator flow chart.

B.2.3 IsInside flow chart

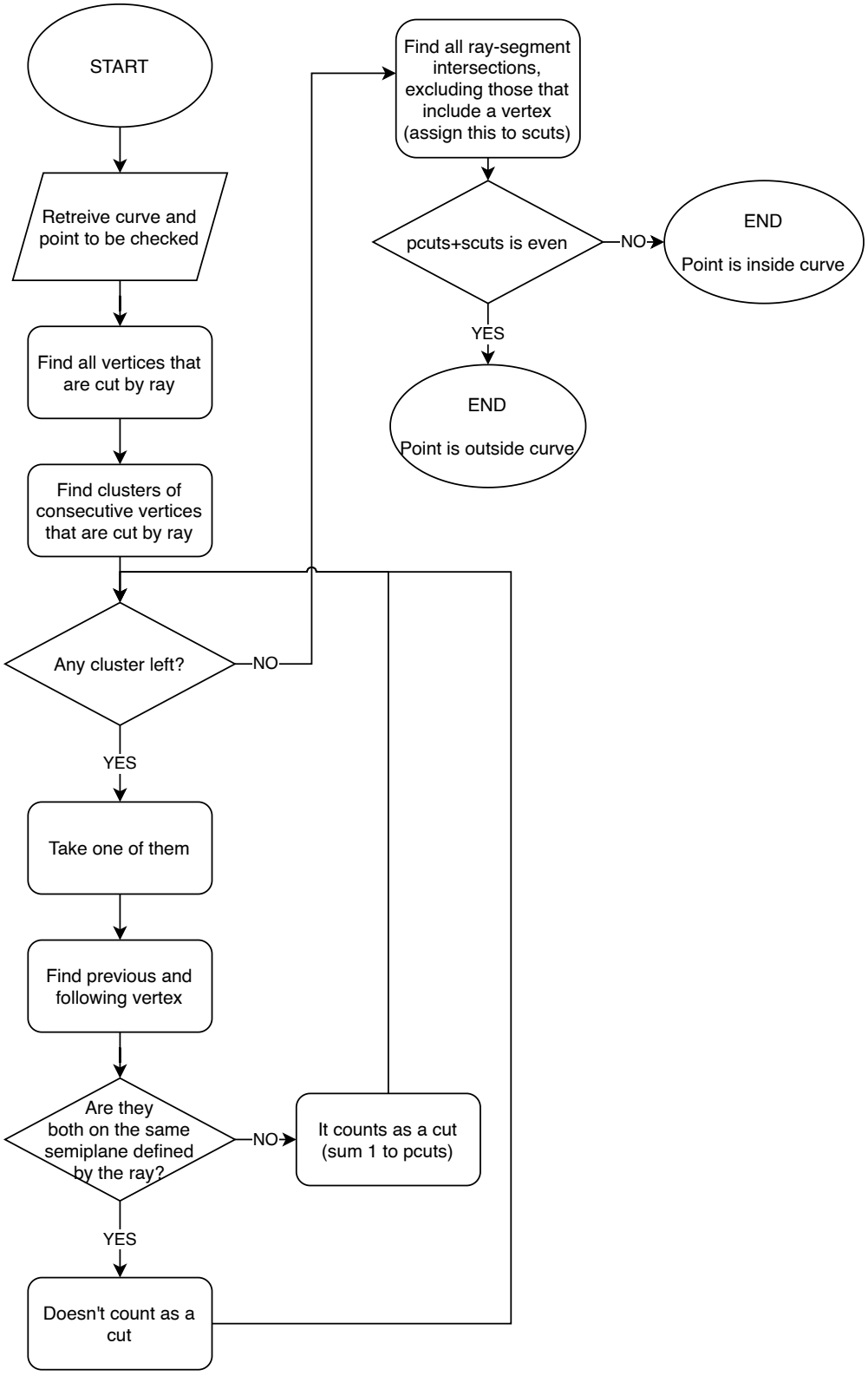


Figure B.4: IsInside function flow chart.

C
Parallel Computing

Contents

C.1 Introduction

Parallel computing is a way of programming that focuses in the simultaneous completion of independent tasks.

For instance, one may have a set of data, and at some given moment needs to apply the same function to all of them, as long as the evaluation of said function for a point is independent from the results of the others, parallel computing is easily achievable.

In the case of *MSE*, parallel computing could be implemented for the evaluation of the `IsInside` function. As mentioned before, this function determines whether or not a point is inside a curve, and is evaluated thousands of times during

runtime. It is important to note that this function evaluations are completely independent from one another.

If they were not, a serial computation would be the only way to go.

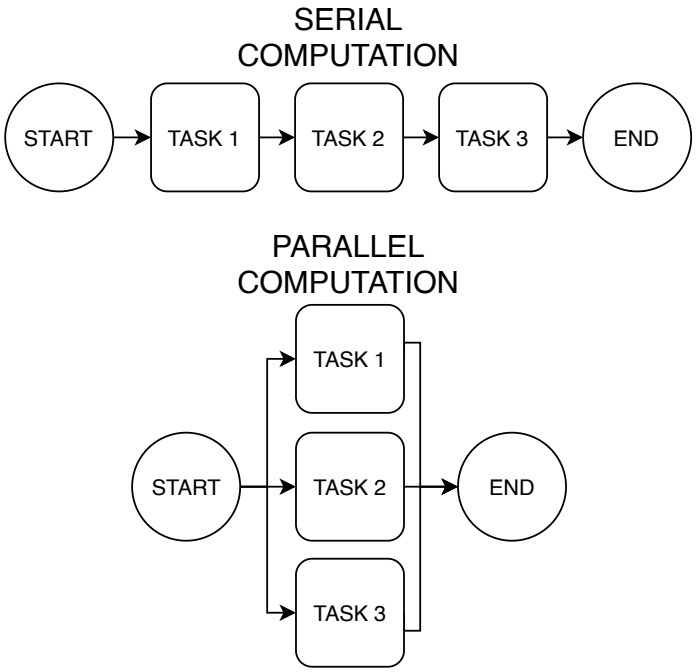


Figure C.1: Parallel vs. serial computation.

Parallel computing is most commonly known for being the tool for rendering high fidelity graphics in real time, so its implementation mostly occurs in the graphics card.

Graphic cards, in contrast with CPUs, are characterized by having many cores (in the order of thousands vs. the tens) that have a very limited instruction set and run at relatively slow speeds (~1GHz vs ~4GHz).

This means GPUs are optimal for running repetitive independent jobs that do not require high level instructions.

This is the case for most matrix operations, which can easily define any geometric manipulation.

C.2 CUDA.jl

Julia has, amongst other packages, one dedicated to parallel computing using *Nvidia* GPUs, called `CUDA.jl`. The name comes from the `Cuda Cores`, which are the ones featured in *Nvidia* graphic cards.

It is quite simple to use, as it only really requires using `CuArrays`, which are special arrays handled by `CUDA.jl` that force the GPU to compute every operation done to them.

Bibliography

[1] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In Vittorio Scarano, Rosario De Chiara, and Ugo Erra, editors, *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008.

[2] Miguel Gonzalez Marco. A damage based framework to improve the strength of strain driven generative designs, 2021.

[3] Kitware Inc. Vtk file formats. `https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf`. [Online; accessed 22-Aug-2021].

[4] Jae-Hwang Lee, Jonathan P Singer, and Edwin L Thomas. Micro-/nanostructured mechanical metamaterials. *Advanced materials*, 24(36):4782–4810, 2012.

[5] MSC Nastran et al. Quick reference guide. *MSC. SOFTWARE*, 1, 2018.

[6] Paul Bourke. Intersecting cylinders. `http://paulbourke.net/geometry/cylinders/`, 2016. [Online; accessed 20-Aug-2021].

[7] Wikipedia contributors. Point in polygon — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Point_in_polygon&oldid=1015369730#Ray_casting_algorithm`, 2021. [Online; accessed 19-May-2021].