POLYTECHNIC OF TURIN

Master's degree
in Mathematical Engineering

Master's Degree Thesis

Generation of conforming meshes in complex geological formations starting from point clouds describing internal interfaces

Supervisors
prof. Stefano Berrone
Supervisor signature

. . . . . .

Candidate
Diego Destino
Candidate signature

. . . . . .

Academic Year 2020-2021

Summary

Several new numerical methods are applied by the candidate on input point clouds of layers of earth in order to create a proper description of the space containing them by using tetrahedra.

Acknowledgements

I warmly say thanks to my professor and advisor Stefano Berrone who followed me through this journey and accepted every issue occurring by helping me every time without hesitating and to professor Sandra Pieraccini who joined me during the analysis. I also say thanks to Alessandro D'Auria and Federico Tesser who helped me to do a lot of things with my computer I did not even know I could. And last but not the least I want to say thanks to all the people who were close to me in each step of my journey in university, from my mother, father and family who were there from the beginning to give me advise and to sustain me economically to my new Turin relatives composed by my girlfriend who really held my hand during every step I made and never left me alone teaching me a new way of living and be happy and her family who became a little my second one.

Contents

List of Figures

Part I

First part

Chapter 1

General introduction

1.1 General principles

There are several operations that in order to create a proper approximation of a layer of geological nature are computed by hand. This makes the process of analysis of soil very long and it would be good if there was a method in order to automatize the process of definition of surface data into a triangular mesh. On the other hand, if we suppose that

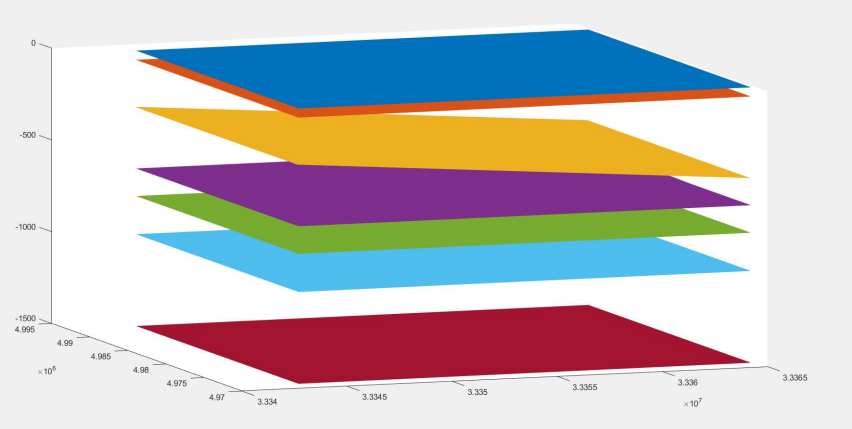

Figure 1.1. Examples of layers represented as surfaces.

a layer is described by a set of data points of which we know x, y and z coordinates, and we think of it as a surface with a precise boundary, we can create a surface which approximates our input data.

To be more precise, here are all the hypotheses that we make:

1. We know all the coordinates of 4 precise boundary point lines (even if not straight lines). Specifically, we have found 4 consecutive sets of points of the surface that

together form the entire boundary of it and we have ordered such points in a way that we can distinguish the sets precisely.

2. For each set we have found a parametrization in [0,1] of the curve of points , for example:
$$c_i(u) = (x_{i,u}, y_{i,u}, z_{i,u}) \quad \forall u \in [0,1]$$
Where c_i represents the i^{th} curve $\forall i \in \{0,1,2,3\}$.

3. We have made so that non consecutive boundary curves are parametrized in the same sense. In a geometrical way we can see the image below. In an analytical way we can imagine that the average tangent vector of the curve gives a positive scalar product with the other one.

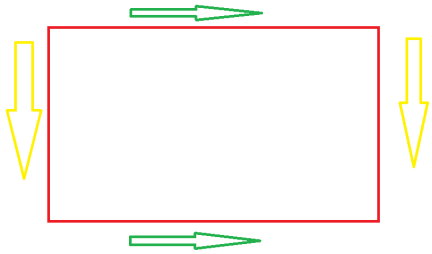

Figure 1.2.

1.2 Tranfinite interpolation formula

If we have 4 boundary lines which are curves parametrized in [0,1] , such that opposite curves are parametrized in the same sense, we can use the **Transfinite Interpolation** formula in order to create a surface contained in those 4 boundary lines.

$$S(u,v) := (1-v)c_1(u)+vc_3(u)+(1-u)c_2(u)+uc_4(v)-[(1-u)(1-v)P_{1,2}+uvP_{3,4}+u(1-v)P_{1,4}+(1-u)vP_{3,2}]$$
$$\forall (u,v) \in [0,1]\text{x}[0,1]$$

Where $c_i(\alpha)$ represents the i^{th} curve evalued in parameter α, $P_{i,j}$ represents the intersection point between curves i and j, and c_1 is opposite to c_3 such as c_2 and c_4. We can notice that if we know the regularity in terms of derivability of the curves $C^{der_i} \ \forall i \in \{0,1,2,3\}$, since the formula is C^∞ in c_i it preserves the same derivability as the minimimum of the boundary derivabilities.

As seen above, the formula is simple but in order to be used we need to satisfy all the hypotheses.

1.3 Gradient Method

After we have found the transfinite interpolation formula , if we consider an input point $P_i = (x_i, y_i, z_i)$ we can define his projection $Proj(i)$ onto the trasfinite surface as the point that minimizes the squared euclidian distance from the point:

$$Proj(i) := S(argmin_{(u,v)in[0,1]^2}(||S(u,v) - P_i||_2^2))$$

In order to find such point we need to minimize a functional which in general is non-linear in (u, v). It is a constrained optimization because the parametric coordinates are forced to live in a square [0,1]x[0,1].

If we want to optimize a non-linear optimization problem, a numerical method often used is the gradient method. If we have a functional $S(u, v) \in C^1[0,1]^2$, we can define its **gradient** as a vector $Grad_S(u, v) = (Grad_{S,u}(u, v), Grad_{S,v}(u, v)) = (\partial_u S(u, v), \partial_v S(u, v))$ as follows:

$$Grad_{S,u}(u,v) = \lim_{h \to 0} \frac{S(u+h,v) - S(u,v)}{h} \qquad Grad_{S,v}(u,v) = \lim_{h \to 0} \frac{S(u,v+h) - S(u,v)}{h}$$

It is impossible to compute the limit numerically and if we do not have an explicit analytic formula for it, we can use an approximation which is called **finite differences approximation**. Simply, we choose an h small enough to be accurate, and we compute, for example $\frac{S(u+h,v)-S(u,v)}{h}$.

In this way we get an approximation of $Grad_{S,u}(u, v)$ and the same way for v. It is obviously a concern the precision of such approximation, and it is important to see that if h is too large the gradient will not be accurate, and on the other hand, if h is too small the gradient could result in a NaN value, meaning that the computer, due to numerical cancellations issues, did not find a numerical result.

The value of h is a tuning parameter of the approximation.

Now the objective is to create a numerical method which follows a proper descent direction starting from an initial guess (u_0, v_0) and by choosing the right step at each iteration until convergence. Also, it has got to respect the constraints on (u, v).
After we have computed the gradient, we can notice that its vector gives the direction of steepest increase of the functional in the point (u, v). Since we want to minimize such functional, we choose $-Grad_S(u, v)$ as the direction to follow in the next step of the algorithm. In this way we have the steepest descent direction. We choose it as the descent direction to follow

It is also important to understand how far we want to go at each iteration, so we want to find a step $\alpha > 0$ such that, if we call (u_{old}, v_{old}) our previous coordinates we want $(u_{new}, v_{new}) := (u_{old}, v_{old}) + \alpha * (-Grad_S(u_{old}, v_{old}))$ to satisfy certain properties.

In this thesis, we decided to apply a line search with **backtracking strategy**, meaning that if we choose an initial parameter $\alpha_0 > 0$ and we have a reduction parameter $\rho \in (0,1)$ we reduce α_0 as far as we reach the so called **Armijo conditions**. If we call our functional to minimize $F(u,v) := ||S(u,v) - P_i||_2^2$ we write as follows:

1. $\alpha = \alpha_0$
2. if $F((u_{old}, v_{old}) + \alpha(-Grad_S(u_{old}, v_{old})) \leq F(u_{old}, v_{old}) + c_1 \alpha Grad_S(u,v)'(-Grad_S(u,v))$ then exit and choose α as the last one. Otherwise $\alpha = \alpha\rho$ and repeat second step.

Armijo control garantuees that we are descending a reasonable amount of distance with respect to the previous one.
If second condition is never satisfied, we reach a maximum number of iteration $iter_{arm}$ chosen apriori.

By continuing to compute each time the gradient and by following backtracking strategy eventually we reach a point in which:

$$|F(u_{new}, v_{new}) - F(u_{old}, v_{old})| \leq tol_f |F(u_{old}, v_{old})| \quad , \quad ||(u_{new}, v_{new}) - (u_{old}, v_{old})||_2^2 \leq tol_p$$

Where tol_f and tol_p are chosen parameters. If the condition is never reached, we reach a fixed maximum number of iterations $NMAX$. This would be in a nutshell the numerical method if we did not have the constraints on u and v.

In order to respect such constraints, we just make some adjustments.

1. When we compute the approximation of the gradient, we pay attention to the value of h such that $u + h \in [0,1]$ and $v + h \in [0,1]$. In order to do so, we choose the sign of h according to u and v being greater or lower than 0.5
2. When we compute α with backtracking strategy we add a control. If at a certain point, with given α and gradient we obtain (u_{new}, v_{new}) outside the box we make a double check:
   (a) if $u_{new} > 1$ we choose α such that $u_{new} = 1$. Viceversa for $u < 0$ and we store the value of α in a new variable α_u.
   (b) Same for v and we store the value in a variable α_v
   (c) we take the minimum between α_u and α_v and we choose it as new α

   Then, with new value of α, we proceed with armijo controls.

An adjustment of this method comes from the fact that in a lot of cases we would like to use a value of h such that the approximation is as close as possible to the original Gradient value but we do not occur into numerical cancelation problems, which depends on the coordinates on which we are computing the gradient. A very often used step in literature for the gradient approximation is this:

$$h = ||(u_{old}, v_{old})||_2 \sqrt{\epsilon}$$

Where ϵ is the machine precision (usually $\epsilon = 10^{-16}$. But this is possible only if u and v are far from 0. Indeed, the most natural solution to this problem is to change the domain of our parametric coordinates from $[0,1]^2$ to a domain far from 0 that we call $[a,b]^2$. The transformation to the new coordinates $\tilde{u}$ and $\tilde{v}$ is the following:

$$\tilde{u} = u(b-a) + a \quad \tilde{v} = v(b-a) + a$$

And I set a and b to 100 and 300.

1.3.1 At the boundary of the surface

Let us take a generic point in the set of and let us investigate its projection on the plane. The situation under investigation is when the projection is a point which is close to the boundary. Let us assume that its actual optimal u is 0 and v is any value. Since the next step will not change u and we are looking for the minimum step between u and v coordinates, we conclude that we must converge and stay on the same point.

This can be wrong because maybe if we changed only the v coordinate in that situation by fixing u we would move along the boundary and find a new optimal point which could cause the method to start again and converge to a true minimum.

To avoid to be stuck in such boundary points, if iwe get to u or v on the boundary (not both of them), we take the coordinate which is not on the boundary and move along that line by applying a monodimensional step of a gradient computed only on its perspective. If the method goes elsewhere, we start again by having forgotten that we had converged before.

We truly converge if even with that added step we do not move from the actual point (it means that u and v have reached their convergence).

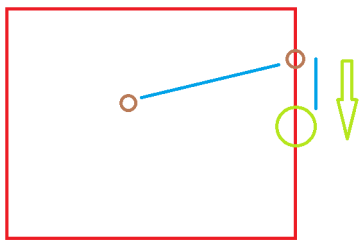

Figure 1.3. Example of the situation described above. The brown points represent the iterations of Gradient method. As we can see, when we end up on the boundary we want to make sure to reach the green point.

This is finally the constrained gradient method used in this thesis in order to solve the problem:

$$\min_{(u,v)\in[0,1]\times[0,1]} ||S(u,v) - P_i||_2^2$$

Where we remind that P_i is the input data point in exam and $S(u,v)$ is the transfinite surface point corresponding to (u,v).

After we have found the projection of each input data point we choose a portion of such points taken in descending order of their distance to their projection, and we store the coordinates (u,v) corresponding to such projections. Then we take a chosen number of boundary points created by transfinite surface formula, and togheter with the previous points we obtain a triangular mesh on the square $[0,1]^2$ by using **Triangle Mesh Generator** (Jonathan Richard Shewchuk). Then we consider the tridimensional data of such triangles and we obtain a tethraedal mesh of all the layers and faults manipulated as seen above by using **Tetgen** (Hang Si).

The advantage of this mesh is that we took very few points (hoping that everything works fine) from each surface and the rest of it will be described by its transfinite surface interpolation formula. With all of this we can then work on optimized meshes which do not contain a non-necessary amount of objects in portions of space where it is not required. This is the main idea of the work I present as a candidate. Every step of this work will be discussed in betail below.

1.4 Kd-Tree search for M Nearest Neighbors

If we want to find the M closest points to a fixed point P we can use Kd-Tree search for M Nearest Neighbors algorithm.

Let us talk about what a **Kd tree** is.

1.4.1 Kd-Tree

A Kd-Tree is a binary tree formed by analyzing a dataset which separates its space into different portions delimited by halfplanes perpendicular to axes. In particular, given a point dataset in 3D space, a 3d-Tree is a tree created as follows:

1. Let us choose our first split coordinate. Let us say x.
2. Let us take the median point with respect to split coordinate value (the point which has the value of the median).

3. Let us consider two half spaces generated by the points which have split coordinate less than the median and the point which have split coordinate greater than the median. The new generated node is the point found before. It will generate two child nodes.

4. Each of the two sets gets split again by using a different split coordinate following the rule which changes at each iteration and assigning the found points to the child nodes.

The way the coordinates which generate the rule vary is simply a subsequent way (x,y,z,x,y,z,ecc.).

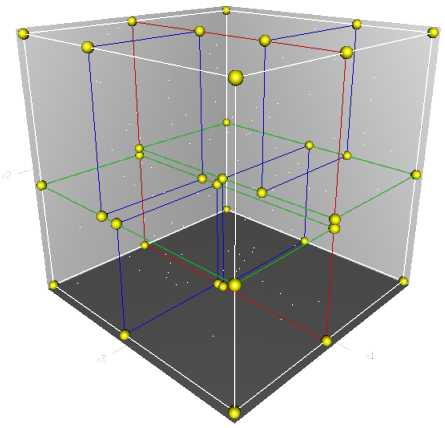

Figure 1.4. Example of 3d-Tree. Image taken from https://en.wikipedia.org/wiki/K-d$_t$*ree*

1.4.2 M nearest search

To find the closest point to our point P after having created the 3d-Tree of a 3D set of points, we must follow the algorithm:

1. Visit the tree following the splitting rules of each node of the tree.

2. After having reached the right leaf save it as the current best point and unfold the tree back up.

3. If a node is closer to P save it as the current best.

4. If the hyperplane not considered during ascending to a node intersects the hypersphere with center P and with radius equal to best distance, do not consider any of the descendings of the node and keep ascending the tree. Else if there is an intersection descend the tree following the splitting rules and check if the distance from the leaf node is smaller than the current best one.

5. for each visited node do as above until the tree top has been reached.

To generalize in order to find the M nearest neighbours of the point P you apply M times the same algorithm discarding the previously found neighbours each time.

1.5 Ransac algorithm

Let us have a set of data points. Let us define **model** as any structure (which can be parametrical or not) for which if we have a portion of such points, we can find its optimal one in a deterministic way and we can compute the distance between an arbitrary point and the model.

Ransac algorithm is a nondeterminstic algorithm (it gives different outputs with the same input repeated several times), which fits a model to a set of data that does not suffer from noise (few points which present a different distribution of coordinates compared to the rest of the points). It can be sum up like this:

1. Take a random set of points from data and call them possible inliers.

2. fit a model to such points. See if the other points are distant from the model less than a chosen threshold. Each time it happens, add such points to the possible inliers.

3. Fit the model to new set of inliers and test it on each point by computing a measure of error. If the error is better than the current best save the model.

4. Do it again a fixed number of times

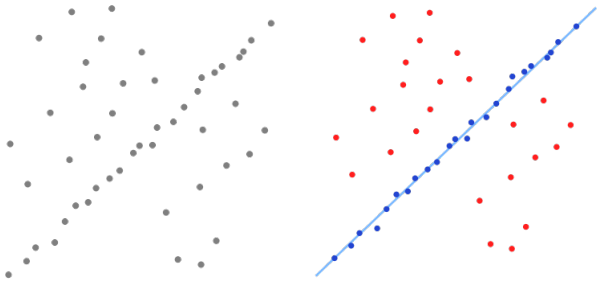

Figure 1.5. On the left, a line with many outliers. In blue, the line estimated by ransac algorithm which ignores the large number of ouliers. Images taken from https://it.wikipedia.org/wiki/RANSAC.

1.6 Principal Component Analysis

Principal component analysis (PCA) is a method used in order to find the right set direction axes along which a certain dataset presents the greates variability in terms of coordinates. Let us describe better what we mean by that.

1.6.1 Definitions

Let us assume that we have a dataset formed by several observations each of them presenting several features. To be more precise each dataset table S of N records and n features can be thought as a series of N realizations of a random multidimensional variable $X = (X_1, X_2, .., X_n)$.
Let us define several values:

1. The **sample mean** of N samples $x_1, .., x_N$ of a r.v. is defined as $\mu := \frac{1}{N}\sum_{i=1}^{N} x_i$.

2. The **sample variance** of N samples of a r.v. is defined as $\sigma := \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2$.

3. The **sample covariance** of two sets of N samples of two different random variables x_1 and x_2 is defined as: $Cov(x_1, x_2) := \frac{1}{N-1}\sum_{i=1}^{N}(x_{1_i} - \mu_1)(x_{2_i} - \mu_2)$.

4. The **sample correlation** of two sets of N samples of two different random variables x_1 and x_2 is defined as: $Corr(x_1, x_2) := \frac{Cov(x_1,x_2)}{\sigma_1 * \sigma 2}$

5. Let S be a matrix representing the dataset as row vectors of samples of multidimensional random variables:
$$S = \begin{bmatrix} x_{1_1} & x_{1_2}... & x_{1_n} \\ ... & ... & .... \\ x_{N_1} & x_{N_2}... & x_{N_n} \end{bmatrix}$$
We call $\bar{S} := \begin{bmatrix} x_{1_1} - \mu_1 & x_{1_2} - \mu_2... & x_{1_n} - \mu_n \\ ... & ... & .... \\ x_{N_1} - \mu_1 & x_{N_2} - \mu_2... & x_{N_n} - \mu_n \end{bmatrix}$
and $\hat{S} := \begin{bmatrix} \frac{x_{1_1}-\mu_1}{\sigma_1} & \frac{x_{1_2}-\mu_2}{\sigma_2}... & \frac{x_{1_n}-\mu_n}{\sigma_n} \\ ... & ... & .... \\ \frac{x_{N_1}-\mu_1}{\sigma_1} & \frac{x_{N_2}-\mu_2}{\sigma_2}... & \frac{x_{N_n}-\mu_n}{\sigma_n} \end{bmatrix}$ which are the **re-centered** matrix dataset and the **z-normalized** matrix dataset.

6. We define the sample covariance matrix $V(S) = \frac{1}{N-1}\bar{S}^T\bar{S}$ and the correlation matrix $C(S) = \frac{1}{N}\hat{S}^T\hat{S}$. We have $V(S)_{i,j} = Cov(x_i, x_j)$ and $C(S)_{i,j} = Corr(x_i, x_j)$.

1.6.2 Main Idea

If we take the correlation matrix of a given dataset $V(S)$, we can prove that it is a diagonalizable matrix with orthogonal matrix representing its eigenvectors. Moreover it has real eigenvalues greater or equal than zero. If we call P the matrix with columns as the eigenvectors $v_1, v_2...v_n$ of $V(S)$ written in the canonical basis, if we consider an observation $X = (x_1, .., x_n)$ of the dataset we can project it onto the space described by the eigenvectors of $V(S)$ and find its coordinates with respect to such basis $Y = (y_1, ...y_n)$. We have:

$$Y = P^T X \quad X = PY$$

The interesting thing is that if we take only the first m entries of Y, we have the representation of X into the subspace generated by the first m eigenvectors of $V(S)$.

Now, if we order the eigenvectors with respect to descending order to the respective eigenvalues, we can show that each set of m eigenvectors taken from the first is the best subspace of dimension m which explains the variance of our dataset! Also, the variance explained by them is proportional to the percentage of eigenvalue sum with respect to the sum of all of them.

Thus we can take the proper amount of principal components, i.e. eigenvectors, in order to explain the dataset with a lower dimensionality. This is the main idea behind PCA.

Chapter 2

Boundary points

Given a 3D dataset: $P_i = (x_i, y_i, z_i) \ \ \forall i = 1 : dim$ where dim is the number of points in the dataset, the first objective is to find all the boundary points of the dataset. First of all let us understand what we mean by **boundary point**:

K-Neighborhood: Taken a point P_i, its **K-Neighborhood** is the set of the K points which are closer to P_i in terms of euclidean distance.

Approximating plane: Given a set of points, their **approximating plane** is the plane which describes their distribution in space in a chosen sense, by a chosen algorithm. Some algorithm, such as the one that will be used later, also define a list of inliers, which are the actual points that are described by the plane, and a set of outliers, that are discarded due to lack of fitness to the plane.

If we construct the approximating plane of the K-neighborhood of a point P_i , we can consider only the inliers. If then we construct an orthonormal base $\{e_1, e_2\}$ of that plane with center P_i, given a neighbor $Q_j \ j = 1 : dimInl_i$ (where $dimInl_i$ is the number of inliers around the point) we can consider the vector $v := P_i - Q_j$ and the angle θ_j formed by v and e_1 by computing his cosinum and sinum with the following formula:

$$cos(\theta_j) =< e_1, v > \quad sin(\theta_j) =< e_2, v > \quad tg(\theta_j) = \frac{sin(\theta_j)}{cos(\theta_j)}$$

where $<,>$ is the scalar product.
We can do that for all the inliers Q_j by obtaining the set $\Theta_i := \{\theta_j, \ j = 1 : numInl_i\}$.

Angular Gap: For a Point P_i, if we construct the approximating plane ,only consider the inliers of that plane and compute Θ_i after having chosen an orthonormal basis for the plane, we can order the angles θ_j in ascending order of tangent value. After done that we can compute $\delta_{j,j+1} := tg(\theta_{j+1} - tg(\theta_j)$ for each $j = 1 : numInl_i$ and also $\delta_{numInl_i,1} := tg(2\pi - \theta_1) - tg(\theta_{numInl_i})$. We say that the point P_i has an **angular gap** if there is a $\delta_{i,j}$ such that it is greater than the threshold t.

boundary point: We say that a point P_i in a dataset is a **boundary point** if, chosen a threshold t, it has an angular gap.
So, the objective is to find all the boundary points of the dataset. The main idea for this way of finding boundary points can be found in the article Zhang [2016].

2.1 Finding boundary points

In order to do so, I implemented a c++ program and used a library which is called **Point cloud library** (Rusu and Cousins [2011]).

In order to find the K-neighbordhood of a point P_i, I decided to use the $Kd-Tree$ algorithm, which was less expensive than a naive computation of all the distances between all the points and P_i.
In order to fit the approximating plane I decided to use *RansacAlgorithm* to find the ouliers and inliers.
The algorithms above are all given by the library **PCL** (Rusu and Cousins [2011]).

In order to compute an orthonormal basis $\{e_1, e_2\}$ for the plane I considered two inlier points Q_1 and Q_2 and I computed the normal to the plane by taking the cross product between the vectors $v_1 := P_i - Q_1$ and $v_2 := P_i - Q_2$, and then normalized obtaining $n := \frac{v_1 \mathrm{x} v_2}{||v_1 \mathrm{x} v_2||_2}$. I set $e_1 := \frac{v_1}{||v_1||_2}$ and then I simply define e_2 as follows:

$$e_2 := n \mathrm{x} e_1$$

After that, everything goes as said above and boundary points are finally found. They are not ordered in a particular way.

Sometimes it could happen that interior points are found even if not wanted. It is the case in which a point is on the edge of a curvature area of the surface, i.e. a portion of surface in which there is a steep change in the direction of the normal to the surface. To solve that problem, simply we make another run of the above algorithm but this time we consider as input the boundary points found so far and the neighborhood is every point each time. In this way, since the interior points that have been wrongly found before now are surrounded by the true wanted points, they will not present an angular gap and will be discarded leading to the final set.

For layers 8 and 9, which are very special in terms of shape, another final elimination of internal points has been made by projecting each point to the plane xy.

2.2 Creating boundary pieces

After having found all the boundary points, we have to merge them in a proper way such that we obtain the 4 boundary points. Before that, there is an intermediate phase, which is the creation of **boundary pieces**.

Let us now describe this procedure:

1. We set a flag to 0 for each of the points to remember that they have yet to be checked. We start from the beginning of boundary points until the end.

2. We take a boundary point P_i. If its flag is 1, we skip it and repeat this step.

3. We consider a K-Neighborhood of such point Q_j. We fit a line through it neighborhood by using ransac algorithm and we consider only the inliers of such line. We compute the tangent vector of such line and we call it $tang_i$.

4. We cycle through all the boundary pieces found so far $piece_u$. If none are found so far, we simply create a new boundary piece $piece_i$ and we merge the inliers and store its tangent vector after having set to 1 the flag of all the considered points.

5. If there is at least one piece we consider the tangent vector of this boundary piece $tang_u$ and a random point of it, let us say P_u.

6. We compute the cross product between $tang_i$ and $tang_u$ in order to see if they are parallel vectors (the cross product is close to 0). If they are not parallel, we skip this boundary piece.

7. If they are parallel under a certain tolerance, we consider the vector that links our point P_i to P_u and we call it w. We compute the cross product between w and $tang_i$ to see if the vectors are parallel under a certain tolerance. If it is not so we skip the boundary piece.

8. If it is so, we have that the piece $piece_u$ can be merged with the set of inliers of the line fitted around P_i. We update the piece and store its tangent vector. The point P_i and all the inliers' flags are set to 1 (considered).

After this, we end up with several boundary pieces that could not be the 4 that we need but in the most cases could still be split into small ones. We then need to further merge the pieces.

Chapter 3

Boundary edges

3.1 Merging boundary pieces

We want to make sure that now we end up with 4 boundary sets of points that form our curves for transfinite interpolation surface.

Let us describe the procedure used to merge such boundary pieces:

1. We set a flag to 0 for each piece which means they have not been checked yet. We cycle from the first boundary piece until the end.

2. If the boundary piece's flag is 1 we skip it. Otherwise we call it $piece_i$ and we cycle through the following boundary pieces until the end (from $i+1$ to end).

3. If the piece's flag is 1 we skip it. Otherwise we call it $piece_j$ and we take two random points from the pieces P_i and P_j and the tangent vectors $tang_i$ and $tang_j$.

4. We consider two angles $\theta_{1i,j}$ and $\theta_{2i,j}$. The first one is the angle formed by the vector $P_i - P_j$ and $tang_i$, i.e. $sin(\theta_{1i,j}) = \frac{(P_i - P_j) \times tang_i}{||tang_i|| ||\theta_{1i,j}||}$. The second is the angle formed by the two tangent vectors, i.e. $sin(\theta_{2i,j}) = \frac{tang_j \times tang_i}{||tang_i|| ||tang_j||}$. We compute the sinum of such angles and we if they are both below a certain tolerance we store $piece_j$ in a list inherent to $piece_i$ and keep their values. Otherwise we skip the piece and go ahead to another j.

5. By considering the pieces taken in ascending order of $sin(\theta_1)$, we cycle through them and compute the angle between the tangent vectors of the actual piece and the tangent vector of the previous one. If we reach a gap, which means that the angle is too big, it means that eventually we reached another boundary curve and we stop. Each time a new sinum is computed and is not too big the piece is added to the actual piece. When the process stops the flags of all the pieces analyzed are set to 1 and if there is one of them which forms a gap a new boundary is created.

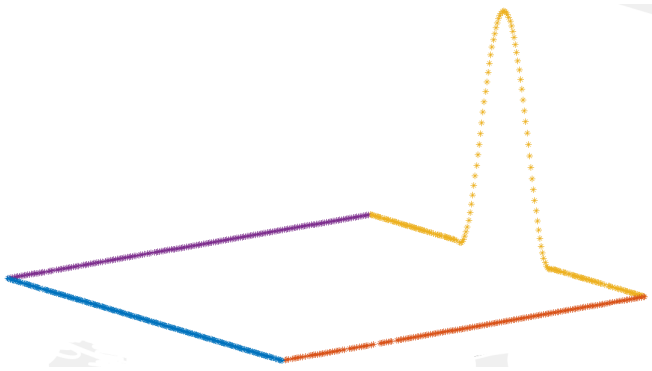

Figure 3.1. An example on layer 3 of boundary edges that have been found.

3.2 Ordering each edge points

When we end up with 4 sets of points forming the 4 edges, we need to order them in a consequential order such that close points in the set are actually close in terms of euclidean distance.
This is the algorithm:

1. Let us take an edge set E_i. Let us consider a random point of it P_i and the tangent vector of it $tang_i$

2. For each point Q_i in E_i, we compute the scalar product between $tang_i$ and the vector $P_i - Q_i$. If it is positive we add it to a list of "positive points" with respect to P_i and we store the distance between the points. Otherwise we save it to a list of "negative points".

3. Then we order the points in both lists with respect to their distance from P_i. And we simply merge the lists by taking the negative point list, the point P_i, and the positive list.

After this procedure the edge has ordered points following the sense of its tangent vector.

3.3 Non classical approach used for layers 8 and 9

As it will be clear in second part of this thesis, the most problematic layers are layer 8 and layer 9. Such layers did not present a regular division of the surface boundary into 4 distinct edges which could easily be separated by the algorithm.

3.3.1 Merging boundaries in a different way

In this case, a more brave procedure has been implemented starting from the merged boundary pieces (which in this case are not 4 unfortunately) with all the points that have been already ordered:

1. First of all let us set a flag to 0 for all the pieces which means that they have not been checked yet. Take the first boundary piece and set its flag to 1. Let us create a list of points with his points.
2. Let us take the final point of this list P_f and search between the other pieces with flag 0 for the one which has the initial point or the final point closer to P_f.
3. Let us consider the winning piece in this competition. If the point closer to P_f is his starting point, simply add it as it is to the list. Otherwise add it with reverse order of points. Set its flag to 0 and start again by searching at the previous step until each piece has been analyzed.

In this way the final list of points is the list of boundary points ordered in a precise way (clockwise or viceversa) and we simply take equal portions of points starting from the beginning. Those will be our 4 edges for transfinite surface and the couples will be the first and third edge, and the second and fourth.

Sometimes this procedure has shown to present critical problems due to the fact that we are using boundary pieces which could be already corrupted by mistakes during ordering of points. So a second way has been tried.

3.3.2 Re-ordering of original boundary points: naive baricenter

The idea is that the original boundary points can be ordered clockwise or viceversa by taking their baricenter, which is the average point, and then by taking a system of axes centered around such point and with two orthogonal axes found like the phase of detection of boundary points. And then, we order the points with respect to their angle formed with the first of those axes. The main issue of this idea is that probably since the points do not lie on a plane the approximating plane created by the two axes does not describe well the angular situation.

3.3.3 Re-ordering of original boundary points: PCA baricenter

Now, the most natural thing to think is that in order to project the points onto a plane that shows the angular situation with respect to the baricenter we need to find the best possible plane that explains the variation of our 3D data points forming the boundary. Since this sentence recalls itself the algorithm, we decided to use the plane formed by the first two principal components. And so the algorithm is the following:

1. Let us take the boundary points $\{p_i\}_{i=1}^{N}$

2. Let us take their 2D projections onto the plane formed by the two principal components computed as eigenvectors of the covariance matrix of boundary points dataset.
3. Let us take the baricenter of this set of 2D points and let us define as the axes of a baricenter centered system the two canonical vectors (0,1) and (1,0).
4. Let us compute the angle formed by each vector linking the baricenter to a point with (1,0) as discussed in boundary points detection.
5. Let us order the points with respect to such angle. The result is the wanted one.

This has been the best idea so far in order to reorder the points.

3.3.4 Finding 4 cracking points

After having ordered our points, we need to divide them in 4 set of points which will be our edge pieces for transfinite surface.
The most naive way (told before) is to crack the boundary points into 4 equally sized consecutive sets. But it is also important to notice that it could be important to crack them as best as possible since they will become the four describing lines of an entire surface.

It could be better to try a different method. The first alternative was to take them as the points which presented the highest angular gap between two consecutive edges. Both for original 3D coordinates and 2D PCA projected points. The algorithm is the following:

- For each point p_i let us take the vector formed with its following $v := p_{i+1} - p_i$ and also the vector $w := p_{i+2} - p_{i+1}$. Let us take their scalar product and divide it for their norms.
- Let us order the scalar products in ascending order. The lowest ones represent the steepest angular gap between vectors. The first 4 will be the searched points

Otherwise, it could be possible to think of a more creative procedure by using Triangle (Jonathan Richard Shewchuk). Since when Triangle gets used on a set of points, it creates a so called **Conver hull** to cover the boundary points and to enclose the dataset, it can be a good idea to triangulate the projected 2D boundary points on the principal components and, by following the convex hull, take the points which present the steepest angular gap as seen as above as the 4 cracking points.

3.4 Making sure that opposite edges are oriented the same way

The concept of this part is quite simple. Two boundary edges are considered opposite if their tangent vectors are quite parallel under a certain tolerance. If we find two opposite edges we simply make sure that their tangent vectors are oriented the same way. If not, we invert the list of point and change the sign of one of the two edges.

Chapter 4

Projection of points and Mesh generation

4.1 Creation of transfinite surface

Let us summarize the situation we are in. We have 4 edge lists of points that surround our surface, we have isolated the couples and we have made sure that they were oriented the same way. Let us take a look at the transfinite interpolation formula:

$$S(u,v) := (1-v)c_1(u)+vc_3(u)+(1-u)c_2(u)+uc_4(v)-[(1-u)(1-v)P_{1,2}+uvP_{3,4}+u(1-v)P_{1,4}+(1-u)vP_{3,2}]$$

We have $P_{i,j}$ which is simply the point in common with two edge boundarys that touch each other (we manually add a point to the other one in order to link them). We need a [0,1] parametrization of each edge and all the ingredients for the transfinite surface are ready. We thought of creating a cubic 3d spline in the sense of least squares that fitted each edge by using a c++ library called "splines". It automatically creates what we need. So now we have finally created our transfinite surface.

It is important to notice that there are two input dataset which represents faults that are not given as surfaces, but only two opposite curves of boundary points are given in order. In this case a simple thing can be done. Each curve can be approximated by a cubic spline parametrized in [0,1]. Those for example could represent our c_1 and c_3. The remaining two are taken as straight lines linking two final points of the curves and they can simply be parametrized as a segment with the following formula:

$$c_2(u) := uP_{1_{in}} + (1-u)P_{2_f} \quad c_4(u) := uP_{2_{in}} + (1-u)P_{1_f} \;\; \forall u \in [0,1]$$

Where $P_{i_{in}}$ and P_{i_f} represent initial and final point of curve i.

4.2 Finding worst points

The spline is C^2 by definition , so the formula for the transfinite surface defines a C^2 functional from R^2 to R^3. Infact, if we take a point P_i belonging to the original data set, we can define:

$$F(u,v) := ||S(u,v) - P_i||_2^2$$

which is indeed a C^2 functional.

Thus, we can apply the gradient method in order to minimize such functional in u and v in order to find the projection of the point onto the transfinite surface. The details about the algorithm are discussed above at general principles paragraph. After having obtained the projection for each point we can then simply take a portion of them in order of descending distance from their projection (which is the value of the functional) and consider them as **worst points**.

4.2.1 How many points?

In order to understand how many points to take in order to achieve a good mesh, we could think of an iterative procedure which creates a progressive number of meshes formed by more and more triangles until a certain criterion has been reached.

The idea is to compute at each iteration the volume under the 3D triangular mesh by summing toghether each volume define by each triangle T_i :

$$V = \sum_{i=1}^{numbOfTriangles} h_i A(T_i)$$

where $A(T_i)$ is the are of each triangle and h_i is the average z coordinate of the three points of the triangle.

If we continue until a relative tolerance has been reached, we find the optimal number of points to add to our mesh.

4.3 Triangular Mesh generation

After having a list of worst points we can manually add a list of edge points from transfinite interpolation and pass everything to Jonathan Richard Shewchuk which will generate a 2D triangular mesh of the square $[0,1]^2$ by following the instructions given above. In fact, for the worst point, the u and v coordinates passed to triangle are the ones of their projection, and the edge ones are chosen by us.
If we choose to have a quality mesh new points will be generated to the mesh, and our job is to evaluate such 2D points in 3D space by using transfinite interpolation formula. We keep such evalutations and the original worst points with their coordinates.

On the other hand, we could choose to not add steiner points to our mesh. This could be the case when we do not want to risk to have a confusing mesh and we want only to consider original points.

After this, a 3D triangular mesh of space has been created. This mesh is very dense on triangles on areas that are populated by worst points (which are points badly approximated by transfinite interpolation) and less dense on areas in which the transfinite interpolation does its job.

4.3.1 Particular cases and mesh management

For layers 8 and 9 the classical method used to generate a mesh dows not work properly. The reason is that the surfaces are not well approximated by transfinite interpolation. The only way in order to create a proper triangular 3D mesh with triangles that do not intersecate each other is to consider as 2D domain the plane x, y, not the plane u, v of the surface. In this way tha triangulation will generate only non intersecting cells. The number of points given to the mesh is chosen based on the order given by gradient method, as the previous cases.

4.3.2 Cheap selection of boundary points

In some cases it is good to choose the least possible amount of boundary points in order to create a more coarse triangular mesh of the surface. In such cases it is enough to do this:

- Take the initial point of the edge. Take its second point and compute the unit tangent vector v_1 of the line that links them
- Check the next edge. If the unit tangent vector formed by the second edge v_2 and the previous v_1 are parallel enough, which means that $||v_1 \mathrm{x} v_2|| < tol$ ($tol = 10^{-5}$), the point is useless. Otherwise it becomes the next point to choose for the edge and the tangent vector v_2 the new term of comparison for the next ones.
- Iterate over all the points until the edge has been seen completely.

In this way, we obtain a proper amount of boundary points which describe well the curvatures on the border of the surface.

4.4 Tethraedal Mesh generation

In order to generate a tethraedal mesh of space containg all layers and faults, The simplest idea has been to enclose the space into a cube created by respecting the min and max coordinates of the points of the data.
The meaning of this sentence is that the vertices of the cube are chosen as the one containing the extremal coordinates. For example one vertex is $(\min(x), \min(y), \min(z))$, another one is $(\max(x), \min(y), \min(z))$ and so on by considering each combination.

In this way , the software **tetgen** (Hang Si) is able to work properly without giving any error, because the geometry is well defined. But if the cube touches some points of the layers (this is highly probable since it has been chosen on the boundary), each face that gets touched must be divided into sub faces in order to be geometrically defined for the program.

In order to do so the idea was to create a triangular 2D mesh of the face by considering all the points who touched it as input data. In this way each face was divided into triangles well defined.

Then, a simple application of **tetgen** (Hang Si) is the key of success.

4.4.1 Subfaces refining

Each face of the cube gets divided in several triangles using the software triangle. After having projected the point of the mesh which were close enough to the face, the optimizing step is to make sure only the necessary amount of sub faces remain. The objective is to merge togheter the polygons which share an edge that is not one of the pre existing edges before the usage of triangle.

In order to do so, these are the steps applied to the created 2D mesh on each face of the square:

- Set each triangle to a state 0 which means it has not been controlled.
- For each polygon of the mesh, create a queue which contains all its neighbors. Set the initial triangle to state 1.
- Start scrolling the queue. For each neighbour, control if the triangle has been controlled. If yes, control if the shared edge is a pre-mesh edge. If so, discard the neighbour. If yes merge the polygon to that neighbour and set its state to 1.
- Do that until each triangle gets controlled and all the new polygon are created.
- Take the new list of polygons as the updated one of the mesh.

In this way tetgen will have to make way less computational effort to create the tethraedal mesh.

.

Part II

Second Part

Chapter 5

Results

5.1 Computational aspects

I decided to make a preprocessing of raw data by centering around their sample mean x, y and z coordinates. This has been done in order to avoid dealing with too big numbers. Also, with layers 3,4 and 5, I decided to make a rescaling of x and y coordinates in order to make the computation of gradient more stable, since they varied too much with respect to z coordinates. The most successful rescaling was multiplying them by $10^{-2.4}$. The rescaling was succesful with gradient method but poorly behaving with the part of finding border points. For this reason, It is applied after that phase.

Sometimes, since the subtraction of two vectors could lead to numerical cancellation, I decided to enlarge their norm before making the operation.

5.2 Hyperparameter tuning

Hyperparameter tuning was not a technical driven procedure. In this context it is too difficult to grid parameters in order to find the best because the best way to achieve a good degree of precision is to try different combinations after seeing repeatedly the image of point cloud. Now I will present the most important parameters that have been tuned and their best configuration for each fault and layer.

1. K1: number of neighbors of each cloud point to be considered in the process of finding border points.

2. DT: distance threshold for ransac algorithm in order to fit the best plane (used in finding border points)

3. K2: the number of neighbors to be considered for each cloud point in the process of creating border pieces

4. DT2: distance threshold for ransac algorithm in order to fit the best line (used in border pieces creation)

5. tolj: relative tolerance on functional used in Gradient Method

6. tolx: absolute tolerance on u and v coordinates used in Gradient Method

7. NMAX: maximum number of iterations used in gradient method

8. nmax: maximum number of alpha reductions used in wolfe condition control in gradient method

9. h: used in the approximation of gradient with finite differences

The number of neighbors is a very important parameter because if chosen too high or too low it changes a lot the way a point is seen.
If too high for example a point that is a border point could be mistaken for an internal point due to close boundaries of the surface.
If chosen too low an internal point could be seen as a border point simply because it does not have enough neighbors to avoid angular gap.
Distance threshold is a really sensitive parameter. If it increases more points will be included in the plane/line fitted and we will get more inliers.
This could eventually lead to new border points never found before when it was too strict and did not find a good model.

c_1, the coefficient used in armijo controls is chosen as standard value 10^{-5}.

5.3 Output and results

Here I present all the layers in exam from a graphical point of view. Each colored image from now on has been produced thanks to the MATLAB software (MathWorks).

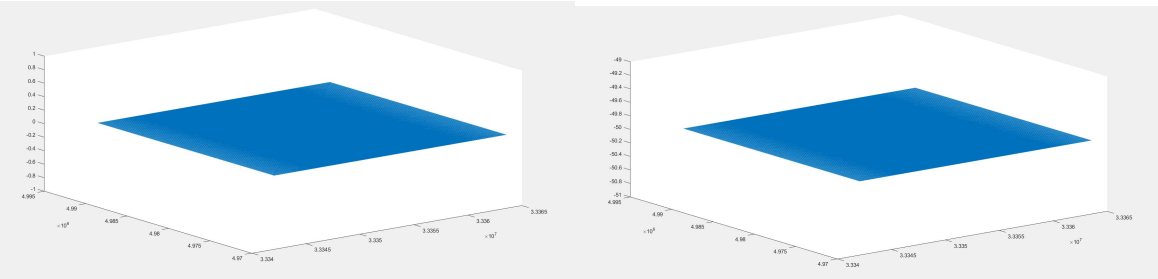

Figure 5.1. Layer 0 and 1.

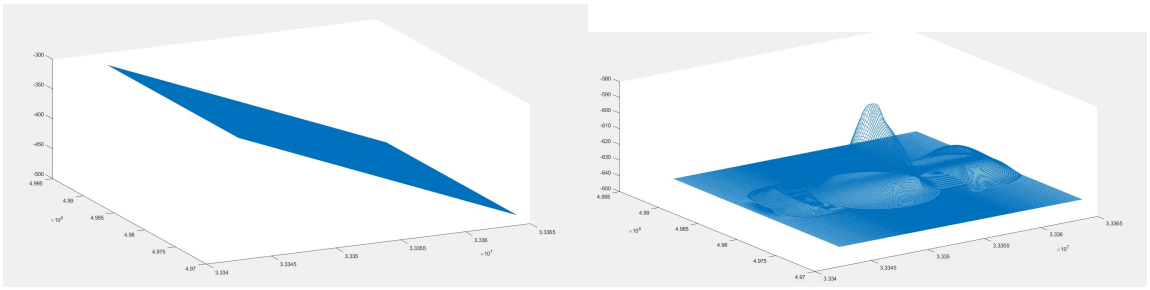

Figure 5.2. Layer 2 and 3.

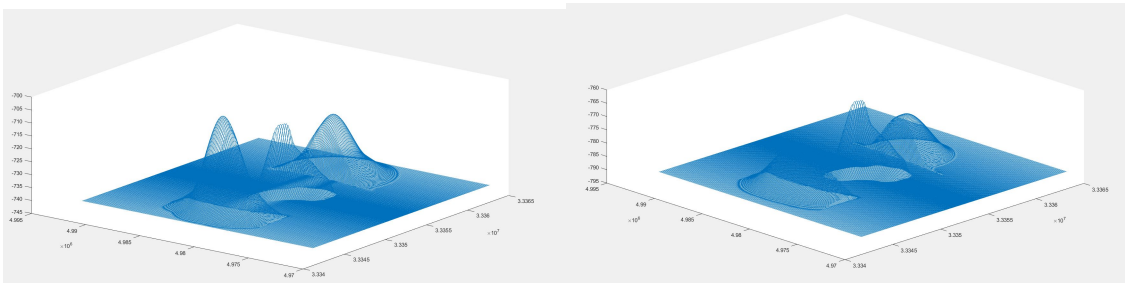

Figure 5.3. Layer 4 and 5.

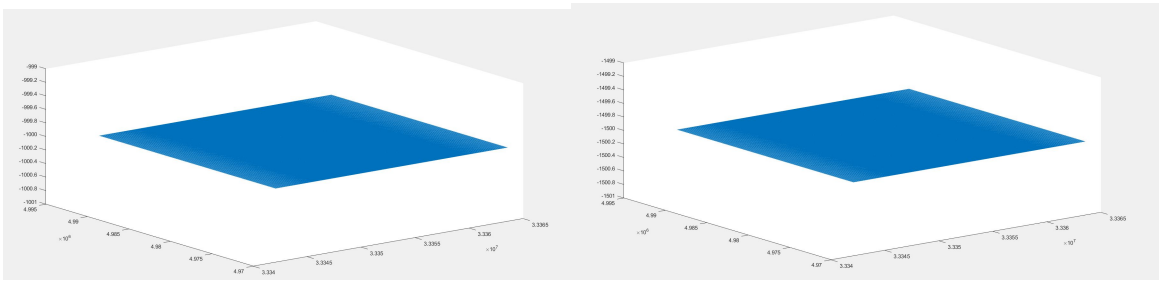

Figure 5.4. Layer 6 and 7.

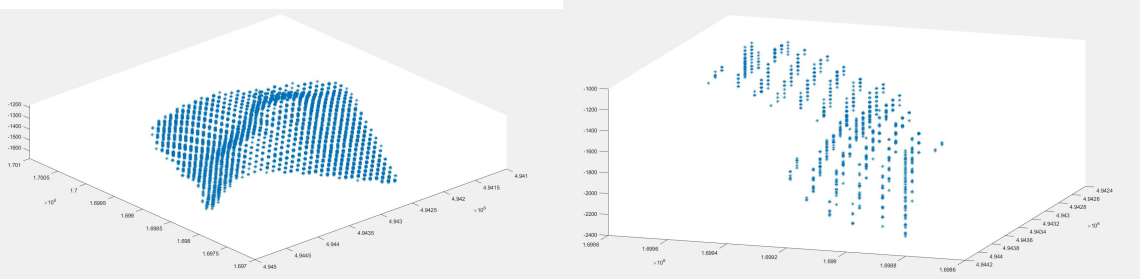

Figure 5.5. Layer 8 and 9.

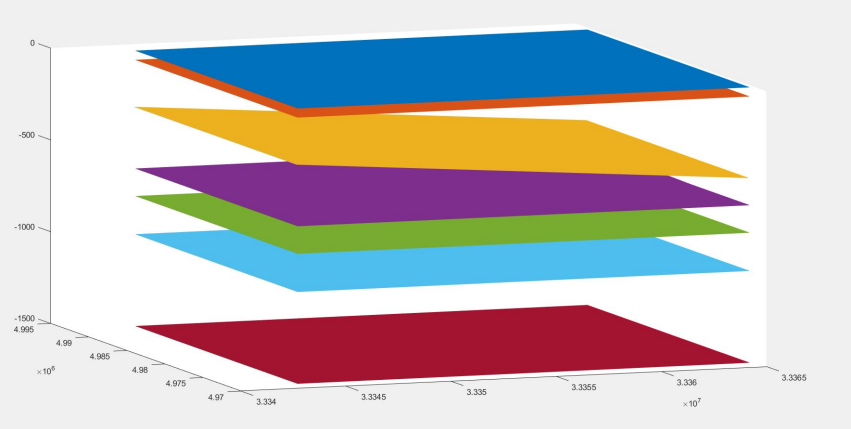

Figure 5.6. All the layers between 0 and 7.

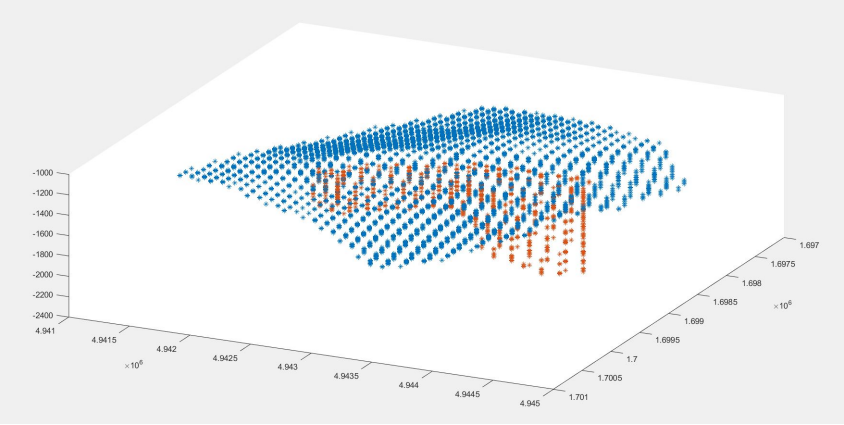

Figure 5.7. Layer 8 and 9 together.

5.3.1 Best Hyperparameters

For each layer we will now present the best hyperparameter values found to work better.

	K1	DT	K2	DT2	tolj	tolx	NMAX	nmax
Layers between 0 and 7	100	100	200	2	10^{-5}	10^{-3}	100	300
Layer 8	300	600	40	50000	10^{-9}	10^{-9}	200	300
Layer 9	200	600	500	400	10^{-9}	10^{-9}	200	300

Good Hyperparameters value presented for layers.

5.3.2 Border points

And know let us see how the algorithm to find border points on the different layers behaved graphically. We can notice that layers 8 and 9 are more unstable than the others because they represent a way more irregular figure. In this sense sometimes border points are discarded even if it would be better to keep them. It is a counter effect of using an algorithm which uses a lot of special features and has a lot of hyperparameters.

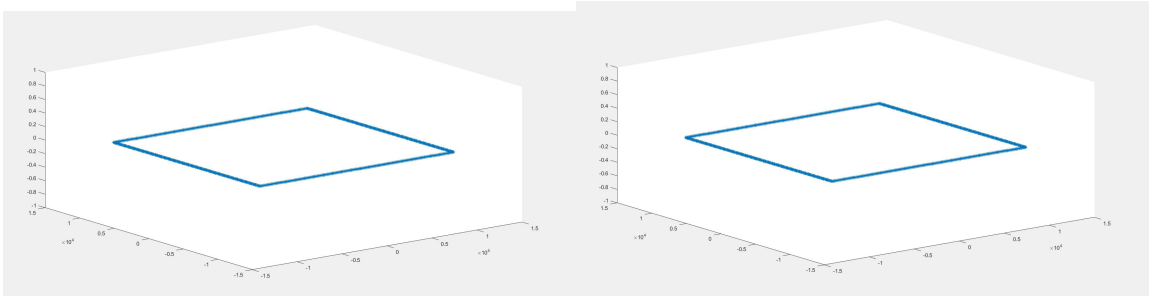

Figure 5.8. Border points for 0 and 1.

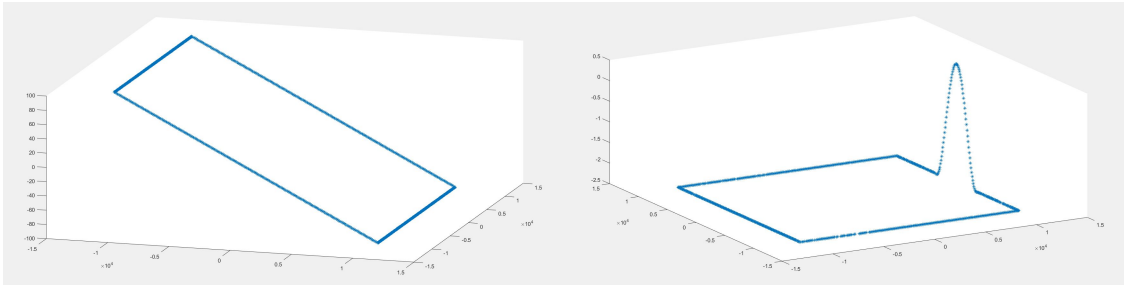

Figure 5.9. Border points for 2 and 3.

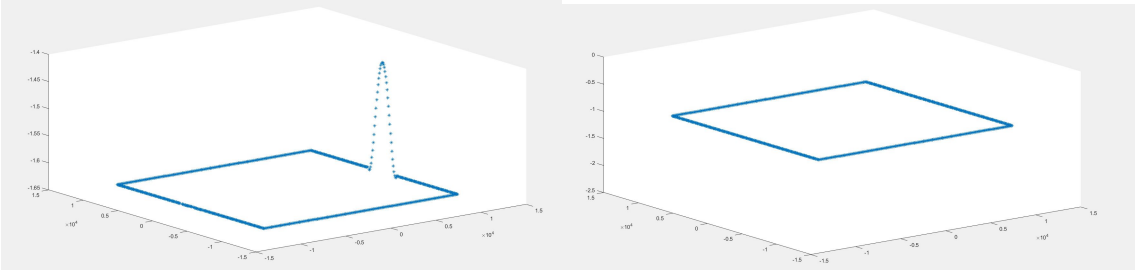

Figure 5.10. Border points for 4 and 5.

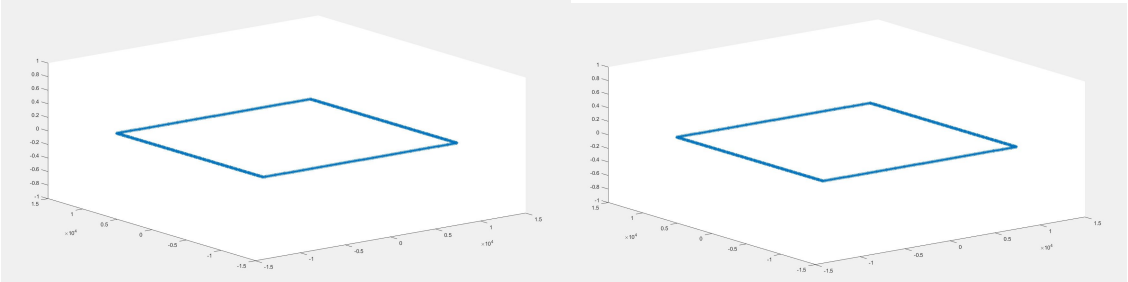

Figure 5.11. Border points for 6 and 7.

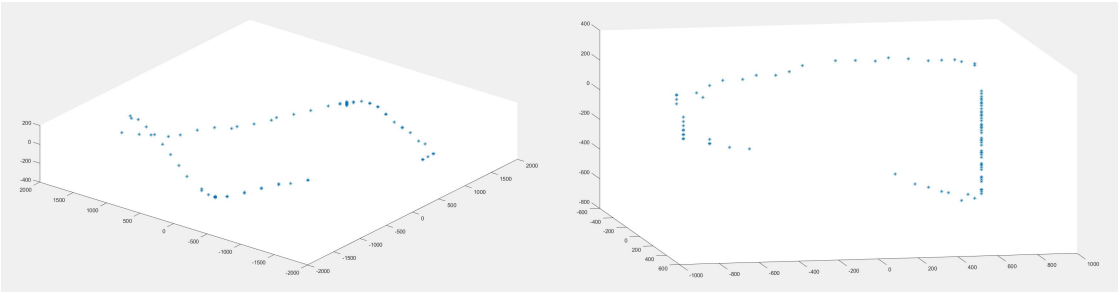

Figure 5.12. Border points for 8 and 9.

5.3.3 Gradient method and outputs

A good way of choosing the total number of original points that are approximated the worst to be taken into the mesh is to see if there is a good elbow of the graph of the function "squared distance from projection onto transfinite surface" with respect to each point ordered in descending value of it.

Another, which is the one that has been used more frequently, is to simply take a look at

first m bad approximated points compared to their projection onto transfinite surface in an image, and decide to take a proper amount of them based on the figure.

Now let us see each situation taking the first 5000 badly approximated points from each layer.

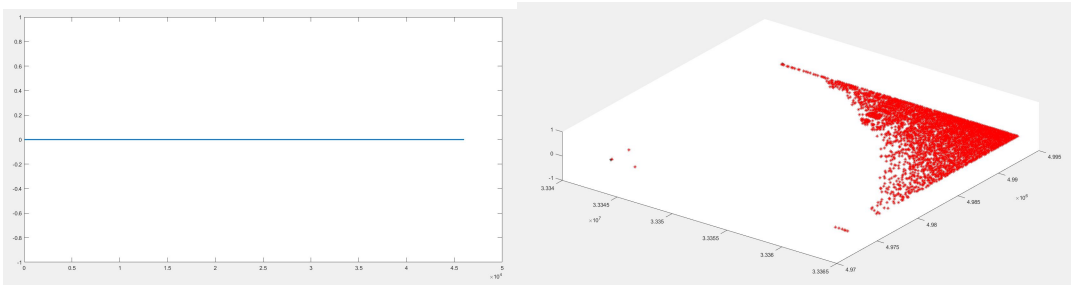

Figure 5.13. Distance function and graphical situation for layer 0.

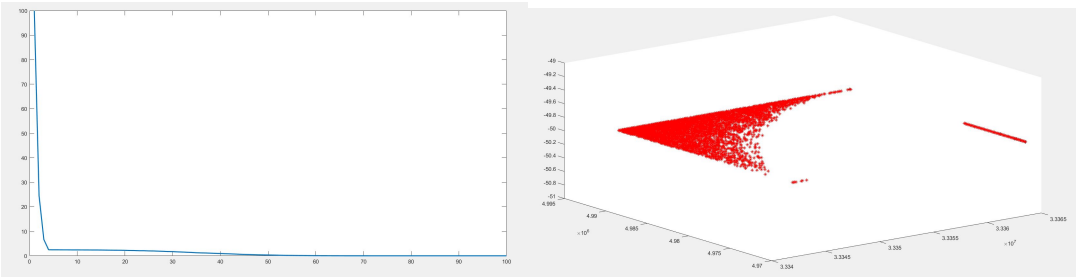

Figure 5.14. Distance function and graphical situation for layer 1.

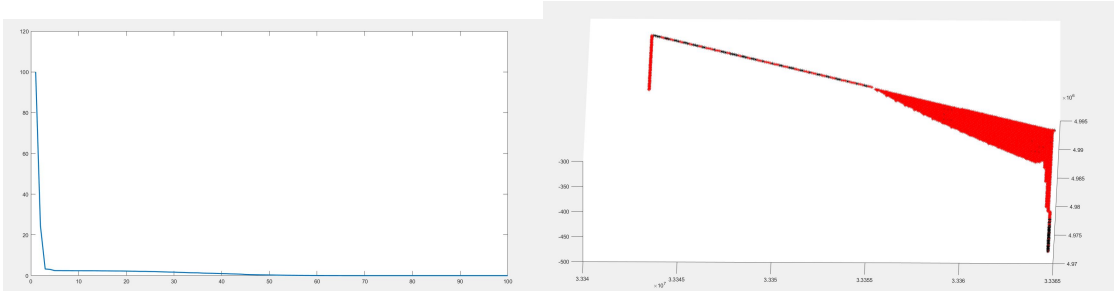

Figure 5.15. Distance function and graphical situation for layer 2.

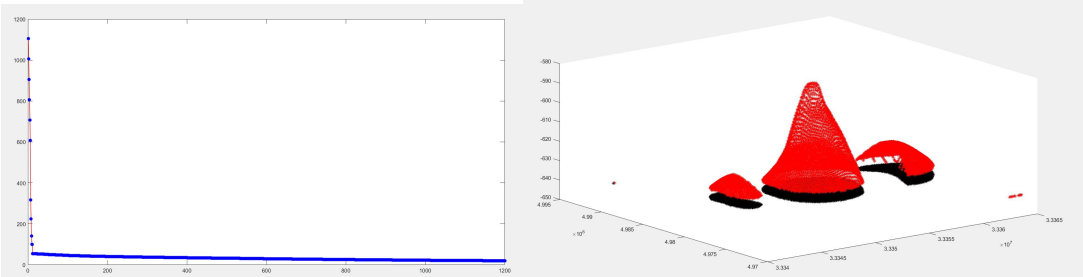

Figure 5.16. Distance function and graphical situation for layer 3.

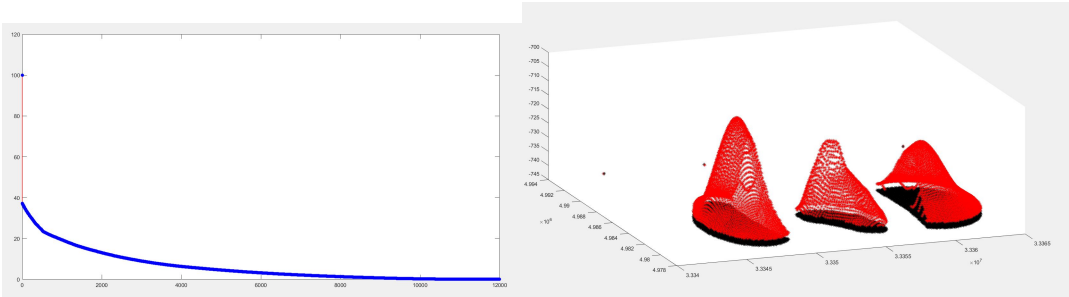

Figure 5.17. Distance function and graphical situation for layer 4.

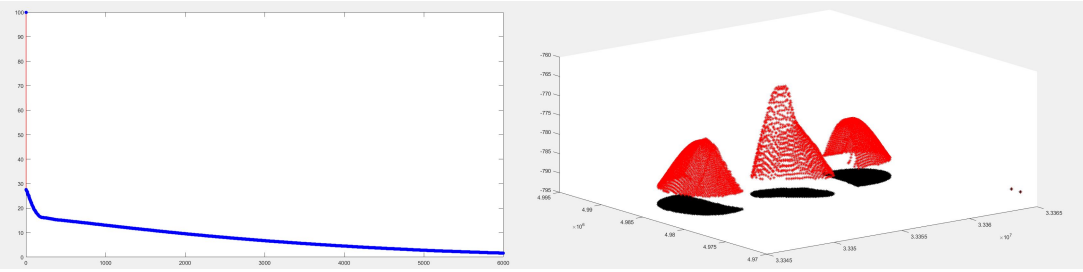

Figure 5.18. Distance function and graphical situation for layer 5.

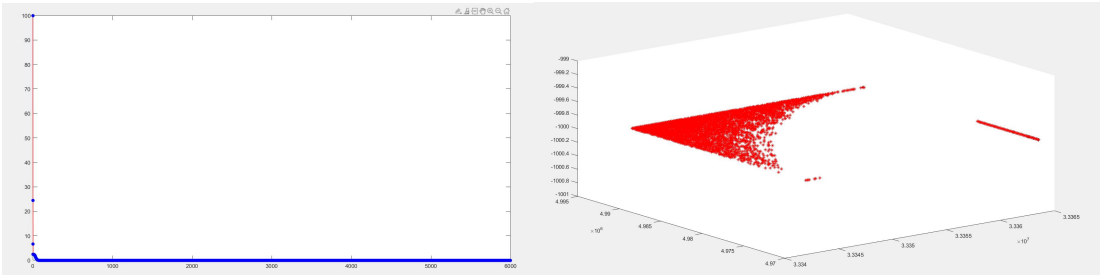

Figure 5.19. Distance function and graphical situation for layer 6.

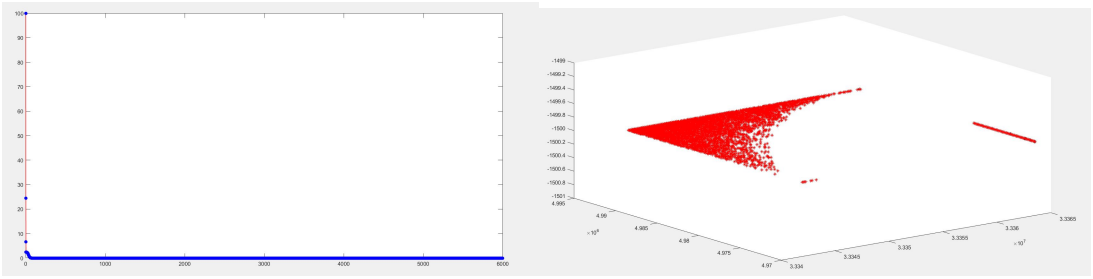

Figure 5.20. Distance function and graphical situation for layer 7.

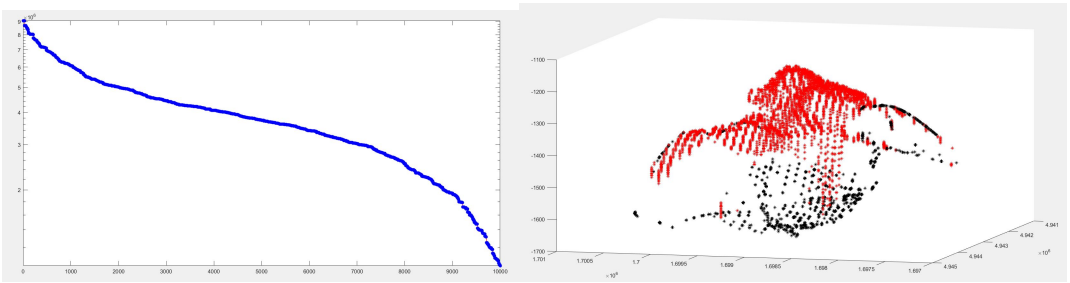

Figure 5.21. Distance function and graphical situation for layer 8.

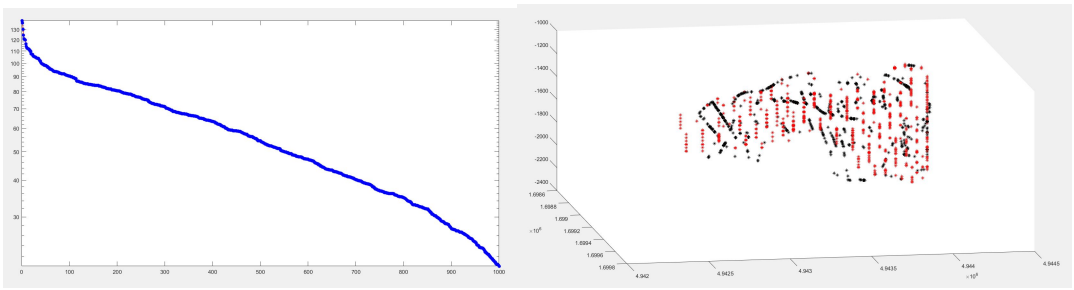

Figure 5.22. Distance function and graphical situation for layer 9.

We can see clearly that the gradient method finely approximated each layer unless some points which were obviously over the comprehension of the transfinite surface. Therefore the method did his work.

For tolj and tolx chosen for layers between 0 and 7 the gradient method always converged before NMAX iterations.
Only 24 times with NMAX=500 and same tolerances the method did not converge in Layer 9. It is a good result indeed.

5.3.4 Triangular Meshes

Here are the meshes created by triangle:

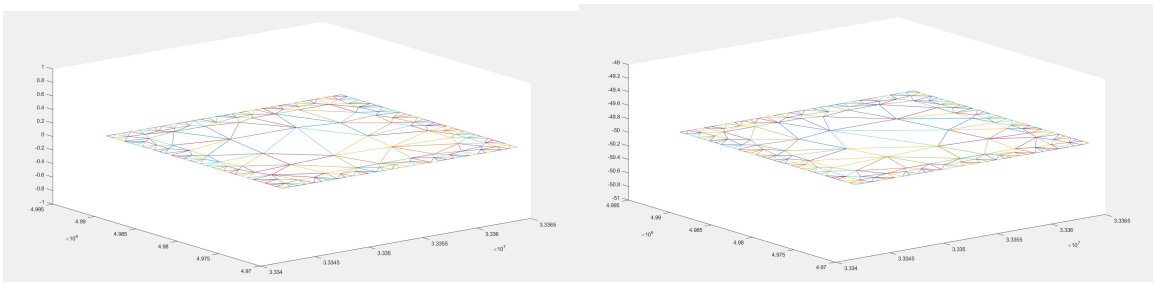

Figure 5.23. Meshes 0 and 1.

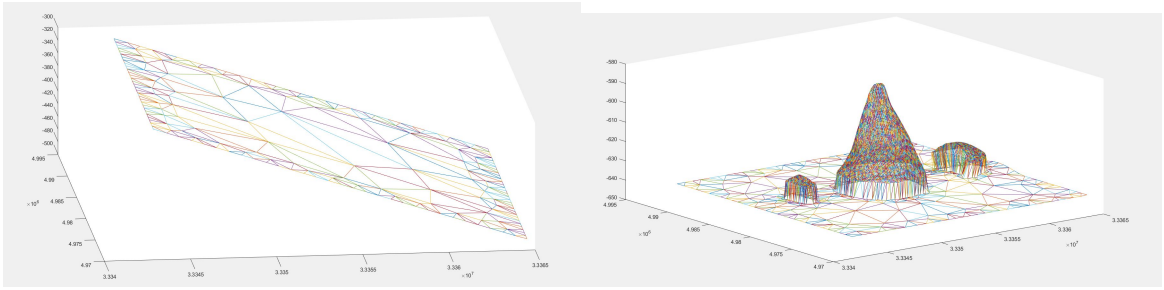

Figure 5.24. Meshes 2 and 3.

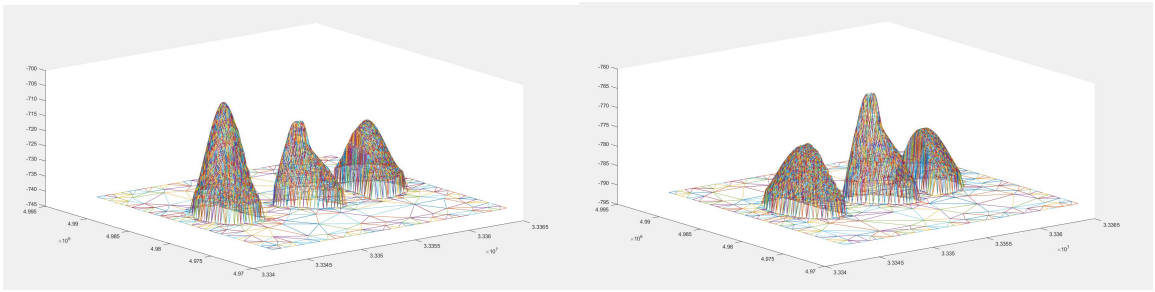

Figure 5.25. Meshes 4 and 5.

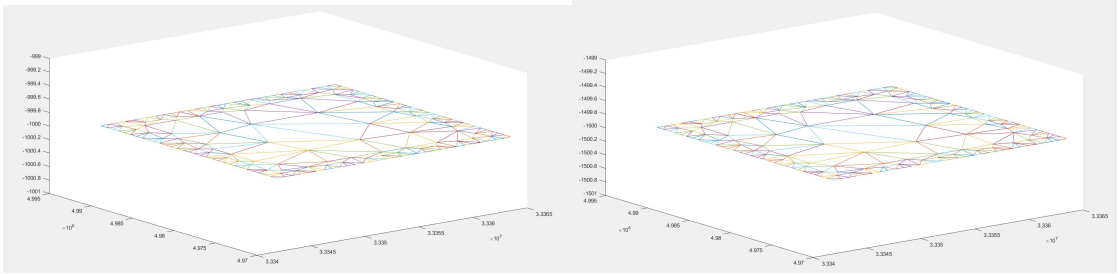

Figure 5.26. Meshes 6 and 7.

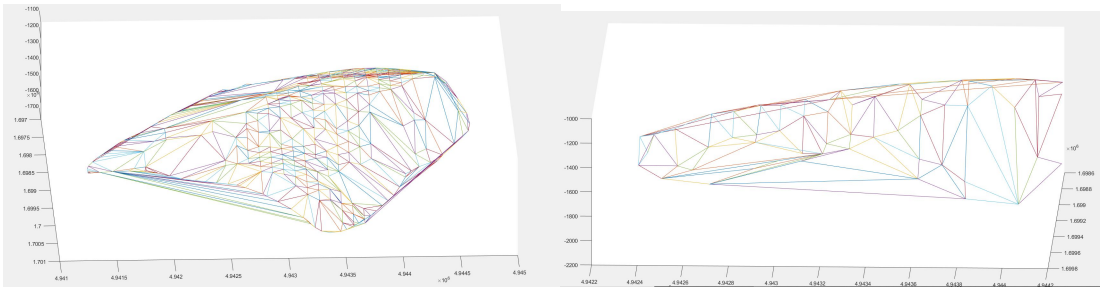

Figure 5.27. Meshes 8 and 9

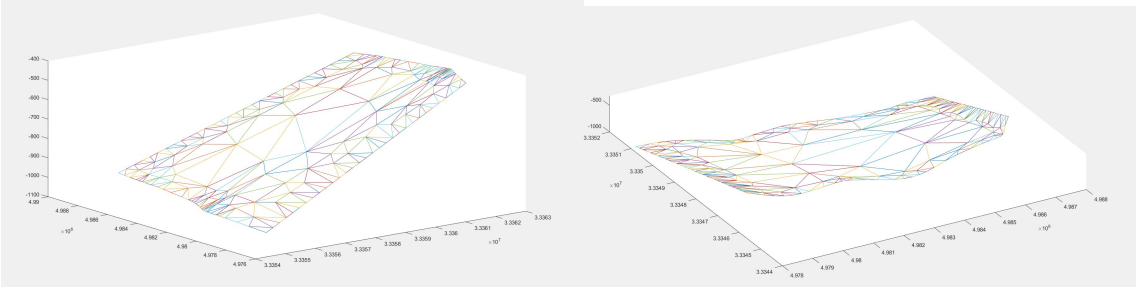

Figure 5.28. Meshes of fault 1 and 2.

5.3.5 Tetrahedal meshes

After having retrieved the meshes of all the surfaces we have several files that describe them, all of them generated by Triangle (Jonathan Richard Shewchuk).
In order to create the tetrahedal mesh we first consider for each triangulated surface two files which are **ele** file and **node** file. **Node** file describes all the points of a triangulation, by attributing them their index into the mesh, their 2D coordinates which are associated to the transfinite surface, and also as attributes their 3D coordinates into space, togheter with a number which indicates if they are on the boundary of the surface or not.

Ele file describes all of the elements of the mesh (in this case all the triangles). For each element there is a specific index and the set of n indexes of the points that form the elements, where n is the number of vertices of it.

The output of the process is a *poly* file which contains the list of points related to all of the faces of the meshes and of the cube which containts them all, followed by the list of faces which form the mesh which are in general convex polygons.
This file is then processed by Tetgen (Hang Si) with the flag **p** which simply tells to process a poly file without any kind of quality optimizer.
Here is an example of a **Poly** file:

```
   breaklines
24 3 0 0
%number_of_points    dimensionality_of_points    number_of_attribute
presence_of_border_flag
     1 33343254.257096 4971589.283837 0.000000
     %index   x   y   z
     2 33364954.257096 4971589.283837 0.000000
     3 33364954.257096 4992789.283837 0.000000
     4 33343254.257096 4992789.283837 0.000000
     5 33343254.257096 4971589.283837 0.000000
     6 33364754.257096 4971689.283837 -50.000000
 ..........................................
35 0
%number_of_faces number_of_attributes
     1
     % one polygon
     3   4    5    2
     %number_of_vertices vertex_1 vertex_2 .... vertexN
     1
     3   2    3    4
     1
     3   10    11    7
 ..........................................
```

Here is an example of a **Node** file:

```
   breaklines
6   2   4   1
%number_of_nodes     dimensionality_of_nodes     number_of_attributes
presence_of_border_flag
    1     0   0   33364754.257096   4971689.2838369999   -50   0     1
    %index   u    v    x    y    z    attribute    border_flag
    2     0.02   0   33364370.257096   4971589.2838369999   -50   0     1
    3     1   0   33343054.257096   4971589.2838369999   -50   0     1
    4     1   1   33343054.257096   4992789.2838369999   -50   0     1
    5     0   1   33364754.257096   4992789.2838369999   -50   0     1
    6     0   0   33364754.257096   4971689.2838369999   -50   0     1
```

Here is an example of a **Ele** file:

```
   breaklines
3   3   0
%number_of_ele   dimensionality   presence_of_border_flag
    1         5      6      2
    %index_of_ele     index_first_point    index_second_point   index_third_point
    2         2      3      4
    3         2      4      5
```

Below we present an example of how the space gets described by a caged set of layers into a cube which is optimized as described in the first part. Layers 0,1 and 2 have been chosen for the process. Since all of the chosen layers are basically plane shaped, their mesh after the optimization are formed by only 4 points which represent only the vertices of the rectangle that define them. The resulting situation after the enclosure into a polygon with the creation of a **poly** file is 24 nodes and 35 faces. After the creation of a tetrahedal mesh we find only 50 tetrahedra. The important fact is that all of the layers originally presented 46008 point each, so in total 138024 points which have been reduced to only 24.

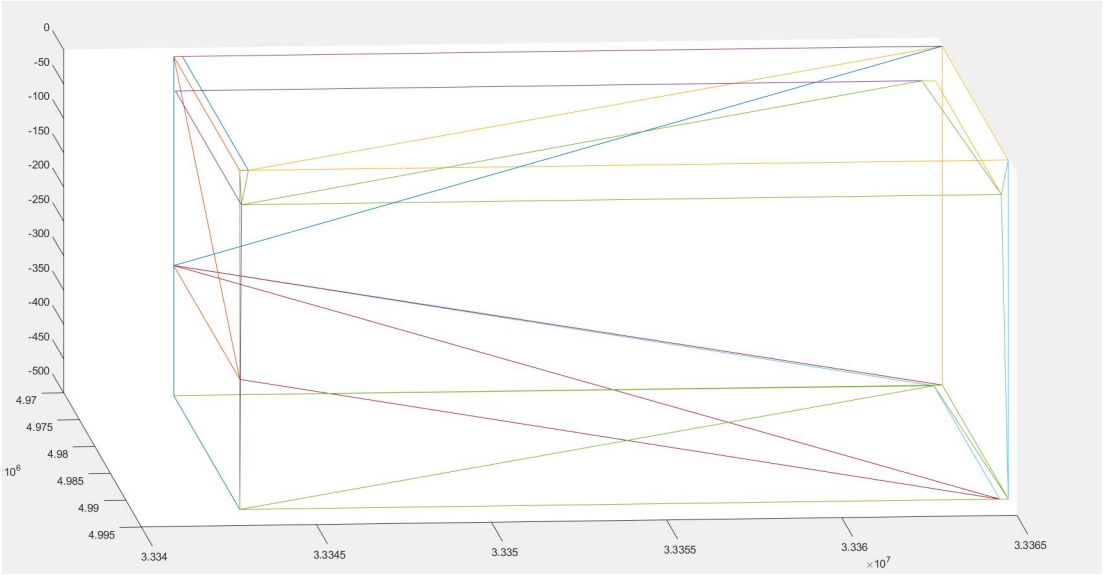

Figure 5.29. Several layers grouped togheter.

Also, we show the relative tethraedal corresponding mesh

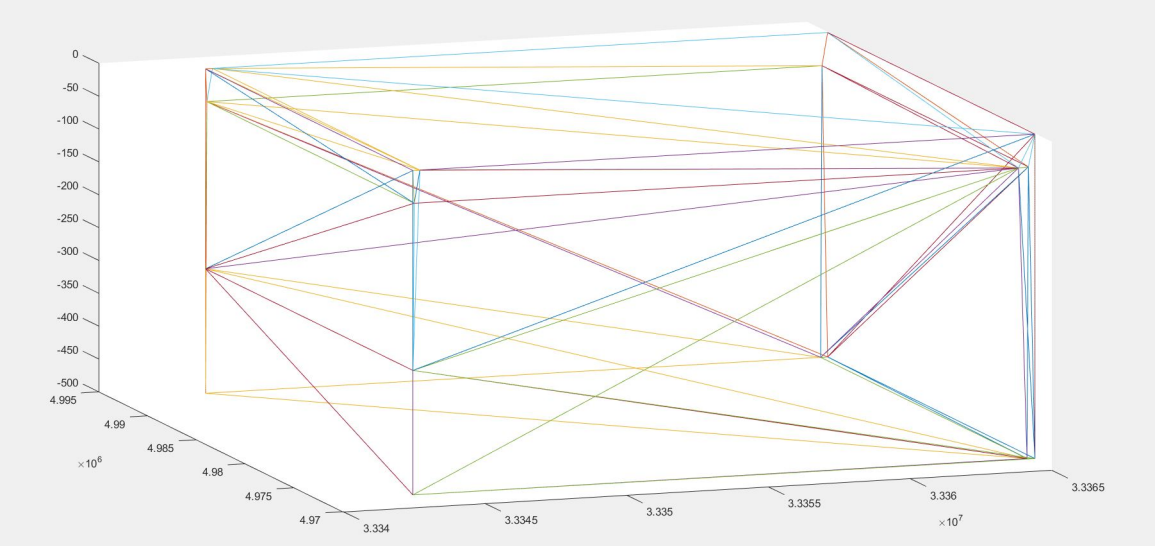

Figure 5.30. Several layers grouped togheter.

The mesh is represented with several files which recall the same structure of the files which represent a triangular mesh of a surface.

5.4 Conclusions and future developments

This algorithm has been shown to be functional with respect to complex geometrical surfaces but it still lacks of precision and efficiency. This is the point from which it is necessary to continue in order to create a finite structure which can be really used by several companies which need it for many purposes. The tethraedal mesh generation has not been completed yet, it will be investigated in future research about the topic and hopefully in few months acceptable results will be produced. It is a good idea to automize everything with several shell scripts which call the files automatically. Also, Artificial intelligence has a lot of growth potential and it could show itself to be effective in no time.

Bibliography

Hang Si. Tetgen. URL http://wias-berlin.de/software/tetgen/.

Jonathan Richard Shewchuk. Triangle mesh generator. URL https://www.cs.cmu.edu/~quake/triangle.html.

MathWorks. Matlab documentation center. URL http://www.mathworks.it/it/help/matlab/.

Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

Huan Ni Xiangguo Lin Xiaogang Ning Jixian Zhang. Edge detection and feature tracing in 3d-point clouds by analyzing geometric properties of neighborhoods. *Remote Sensing*, 2016.