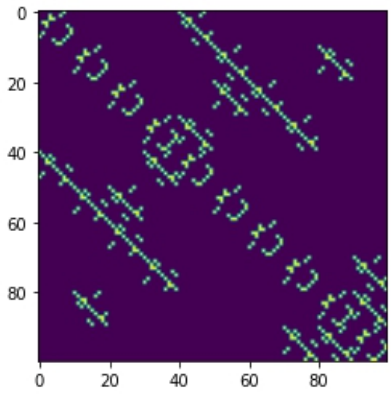
Report

Question 1

In order to create a uniform discretized Laplacian matrix, I have manually created M-1 and C matrix separately. First, I designed a "vertex connectivity table" by looping through all edges to find the two vertices' index i and j, and set the table's [i,j] and [j,i] element to 1. This table provides the connectivity information - for example, the query of neighbours of a vertex with index k could be done by looking up the table's kth row and find indices of non-zero element(this feature is applied in later questions). Another feature of this table is calculating the valence, which is done by summing up each row. The following plot demonstrates that this vertex connectivity table is symmetric, hence the C matrix derived from this table is also symmetric.



Therefore, in the next loop through all vertex, I copied the table to C matrix and have set the diagonal element of C to be the sum of each row(valence), and implemented the M-1 matrix by filling the diagonal with 1.0/valence.

The normal curvature is then calculated by the following formula:

$$H = \frac{1}{2} \|\Delta_S \mathbf{x}\|$$

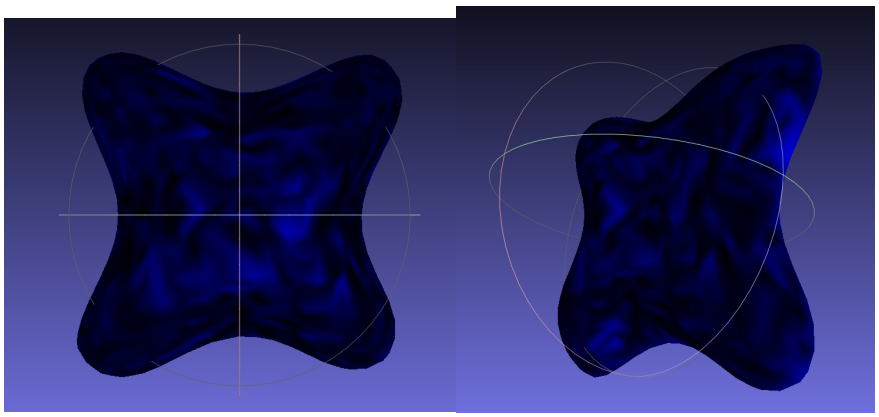
In addition, I have defined a function area to calculate the area of a triangle given by three 3-D vertices. I initialized an array to store the sum of ring-area around each vertex. By looping through all faces on the mesh, I have computed this current face's area and add it to the related vertex index's element in ring-area holder array. After this step, the face area is accumulated for each vertex. The Gaussian Curvature is then computed using the following formula:

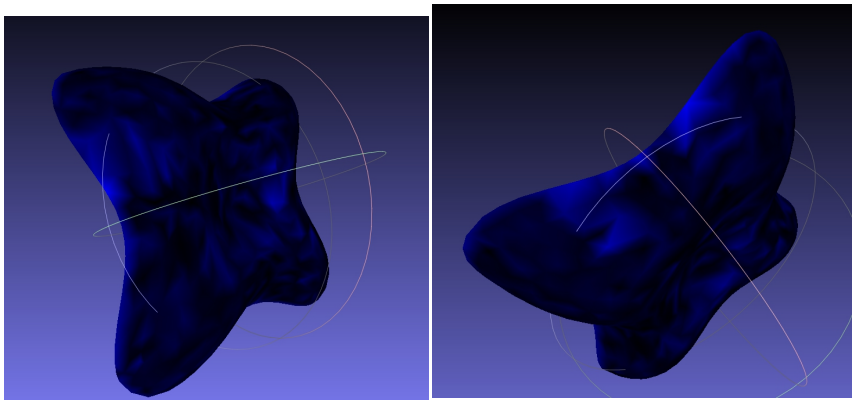
$$K = (2\pi - \sum_j \theta_j)/A$$

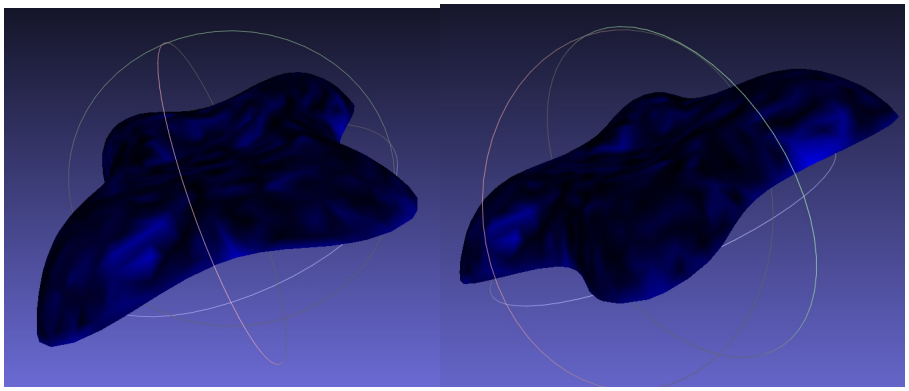
The curvatures are visualized in color intensity in blue channel.

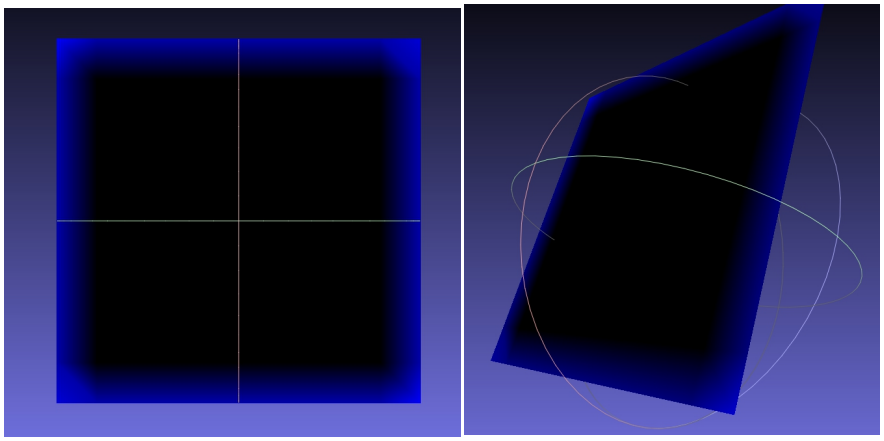
Result:

Normal Curvature:

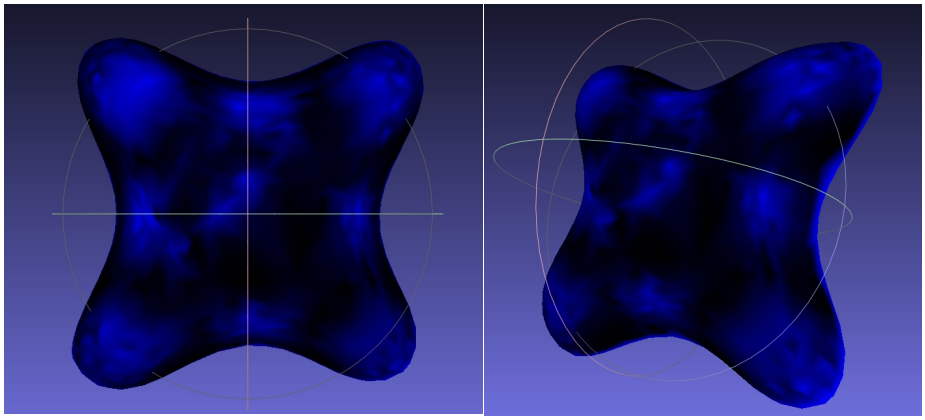
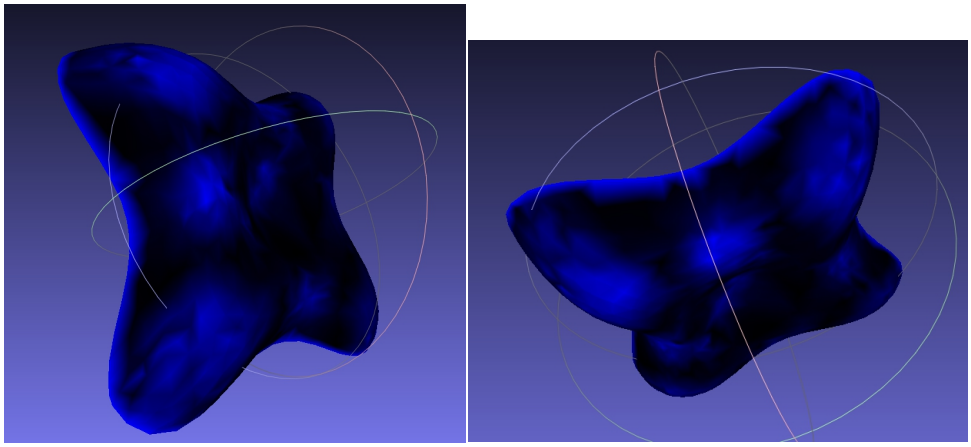
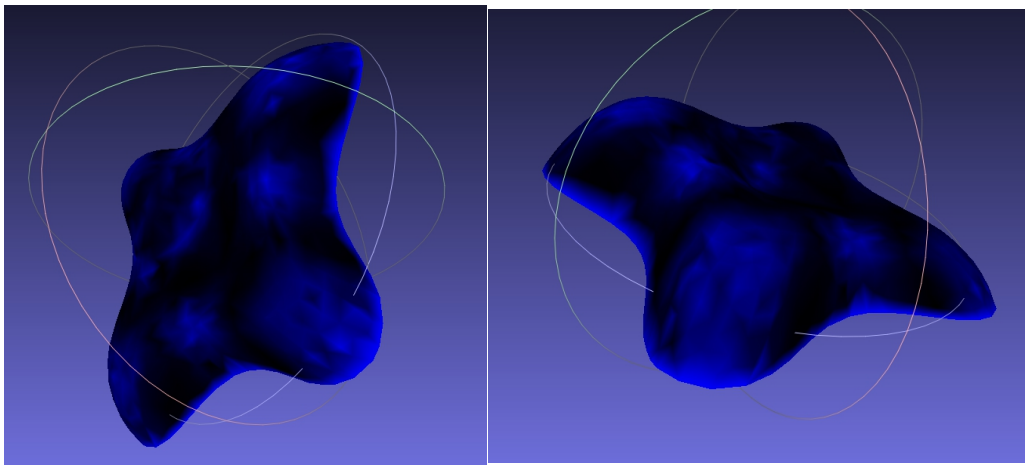
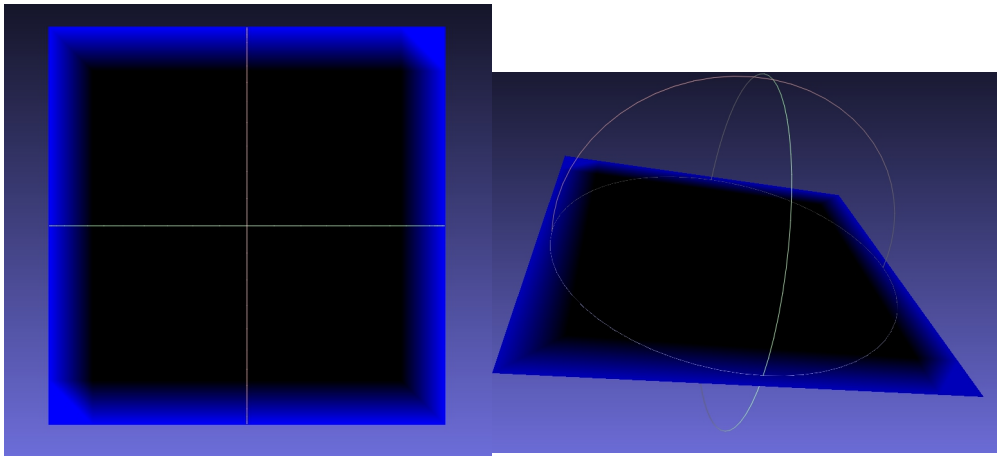








Gaussian Curvature:

From the above pictures, both curvature results are not continuous and depends highly on triangulation. Since this first mesh is not properly and uniformly triangulated, this curvature is not accurate in most places.
For the plane mesh's both curvature computation, the valence at the boundary vertices is less than those vertices in the center, therefore the ring-area is less and M^-1 matrix has a larger value, leading to a large curvature which is not accurate in this case.

Question 2

P = a*cos(u)sin(v), b*sin(u)sin(v), c*cos(v)

X_u = dp/du = -a*sin(u)sin(v), b*cos(u)sin(v), c*cos(v)

X_v = dp/dv = a*cos(u)cos(v), b*sin(u)cos(v), -c*sin(v)

First fundamental form:

$X_u^T X_u = a^2\sin^2(u)\sin^2(v) + b^2\cos^2(u)\sin^2(v) + c^2\cos^2(v)$

$X_u^T X_v = -a^2\sin(u)\sin(v)\cos(u)\cos(v) + b^2\cos(u)\sin(v)\sin(u)\cos(v) - c^2\cos(v)\sin(v)$

$= (b^2 - a^2)\,(\sin(u)\sin(v)\cos(u)\cos(v)) - c^2\cos(v)\sin(v)$

$X_v^T X_v = a^2\cos^2(u)\cos^2(v) + b^2\sin^2(u)\cos^2(v) + c^2\sin^2(v)$

$$\begin{bmatrix} XuTXu & XuTXv \\ XuTXv & XvTXv \end{bmatrix}$$

=

$$\begin{bmatrix} a2\sin2(u)\sin2(v) + b2\cos2(u)\sin2(v) + c2\cos2(v) & b2 - a2)\,(\sin(u)\sin(v)\cos(u)\cos(v)) - c2\cos(v)\sin(v) \\ b2 - a2)\,(\sin(u)\sin(v)\cos(u)\cos(v)) - c2\cos(v)\sin(v) & a2\cos2(u)\cos2(v) + b2\sin2(u)\cos2(v) + c2\sin2(v) \end{bmatrix}$$

Second fundamental form:

X_{uu} = -a*cos(u)sin(v), -b*sin(u)sin(v), c*cos(v)

X_{uv} = -a*sin(u)cos(v), b*cos(u)cos(v), -c*sin(v)

X_{vv} = -a*cos(u)sin(v), -b*sin(u)sin(v), -c*cos(v)

$X_u \times X_v$ = b*cos(u)sin(v)*-c*sin(v) - c*cos(v)b*sin(u)cos(v),
c*cos(v)a*cos(u)cos(v) - -a*sin(u)sin(v)-c*sin(v),
-a*sin(u)sin(v)b*sin(u)cos(v) - b*cos(u)sin(v) a*cos(u)cos(v)

= - b*cos(u)sin(v)*c*sin(v) - c*cos(v)b*sin(u)cos(v),
$a*c*\cos^2(v)*\cos(u) - a*c*\sin^2(v)\sin(u)$,
$a*b*\sin^2(u)\sin(v)\cos(v) - a*b*\cos^2(u)\sin(v)\cos(v)$

Normal = $\frac{Xu \times Xv}{\|Xu \times Xv\|}$

Question 3

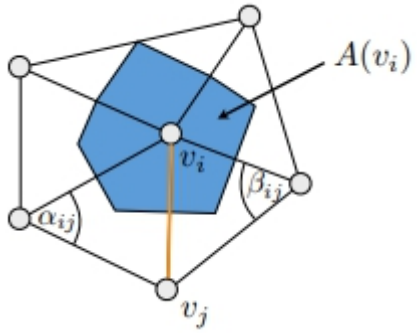
I have implemented the function Get_LB (Question 3, L130 Get_LB). to calculated a proper cotangent discretization Laplacian-Beltrami, with two helper function area(Question 1-2, L55 area) and find_common_triangle (Question 3, L112 find_common_triangle).
In this function, the "vertex connectivity table" is computed using the same method as question 1. Then, I created two arrays to store the ring-area and cotangent value. The ring-area is

calculated using the same method as question 1.

After that, a “vertex-face table” is a n by m matrix where n is the number of faces and m is the number of vertices. Looping through all the faces and its three vertices, for ith face and its jth vertex with index k, the vertex-face table’s [i, k] element is filled with one. This table will be used in finding the common triangle shared by two vertices and corresponding cotangent values.

Next, for each vertex, I have looped through all other vertices and checked in vertex connectivity table. If the tables indicates that the vertex i and vertex j are connected, these vertices’ indices will be passed into function find_common_triangle to find the index of common face that these two vertices share. This helper function finds all the face(triangle) indices containing each input vertices, and check if there are any indices are in common. For non-boundary vertex pairs, there will be two common face indices found because these vertices share two faces in this case. In the other hand, boundary edge only has one face connected to it. Hence, [-1,-1] is returned for boundary vertices, and triangle indices are returned for non-boundary vertices.

If the vertex i and j are not on the boundary, this algorithm loops through all triangles found and all their vertices. It checks if the current vertex’s index is the same as i or j. If it isn’t, the vertex is the right corner to compute the cotangent value, shown as the angle alpha and beta in following graph:



The cotangent value is stored in the [i,j] of cotangent table, which is the same structure as C matrix in previous question.

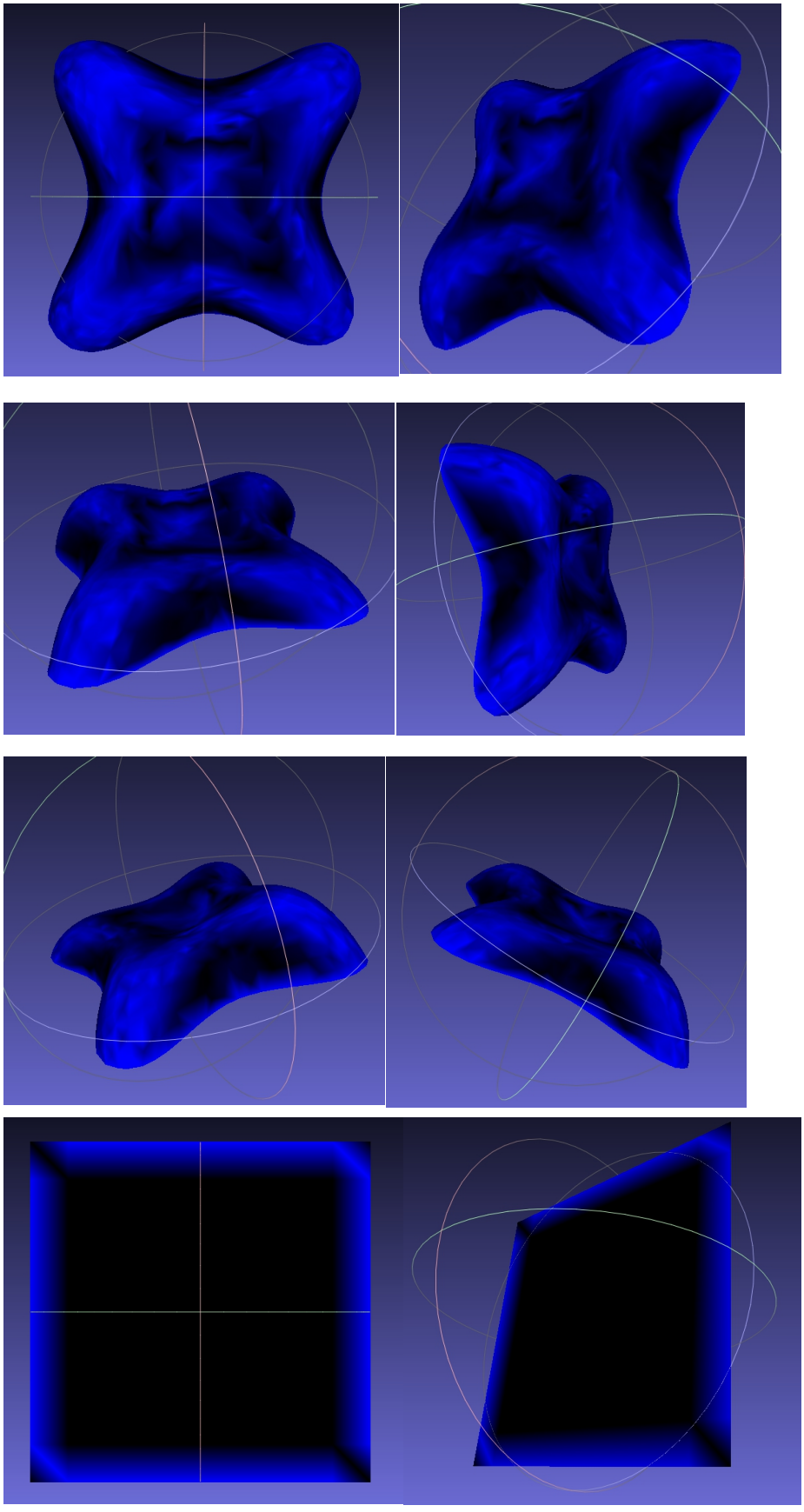
After all the cotangent are computed, I filled the diagonal of cotangent table with the sum of each row, according to the formula shown below:

$$\Delta_S f(v_i) := \frac{1}{2A(v_i)} \sum_{v_j \in \mathcal{N}_1(v_i)} (\cot \alpha_{ij} + \cot \beta_{ij}) (f(v_j) - f(v_i))$$

Then, I have set the M matrix as 1/3 of total ring area and inverse it. By multiplying the M^-1 and C, the final Laplace-Beltrami operator is computed.

Noticeably, this function returns the diagonal of M^-1 and C matrix as well. These will be useful in later questions.

Result:



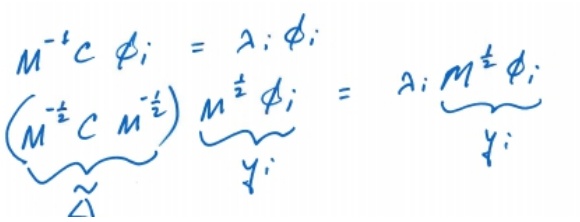
The above results are much more continuous than uniform discretization. The cotangent

weighting has reduced the impact of bad triangulation.

Question 4

First, I have computed the Laplace-Beltrami matrix of the given mesh, as well as the M^-1 and C matrix. A M matrix is also computed by dividing the M^-1 matrix by 1.

The derived Laplace term $\tilde{\Delta}$ in the equation below is calculated by dot multiplication of two M^-(1/2) matrices and a C matrix.



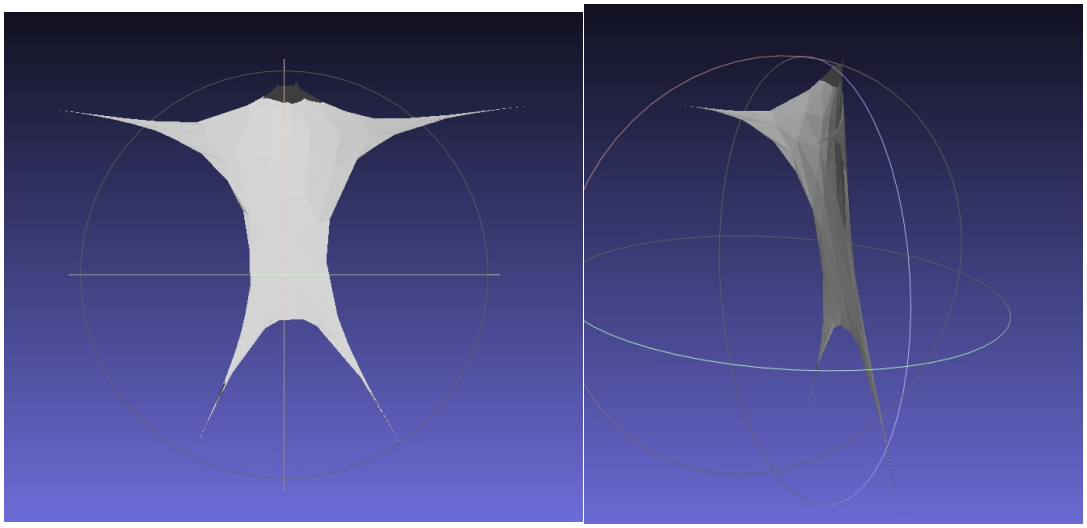
Notice that the C matrix here is twice as big as in lecture note, therefore I have multiplied C matrix by 0.5.

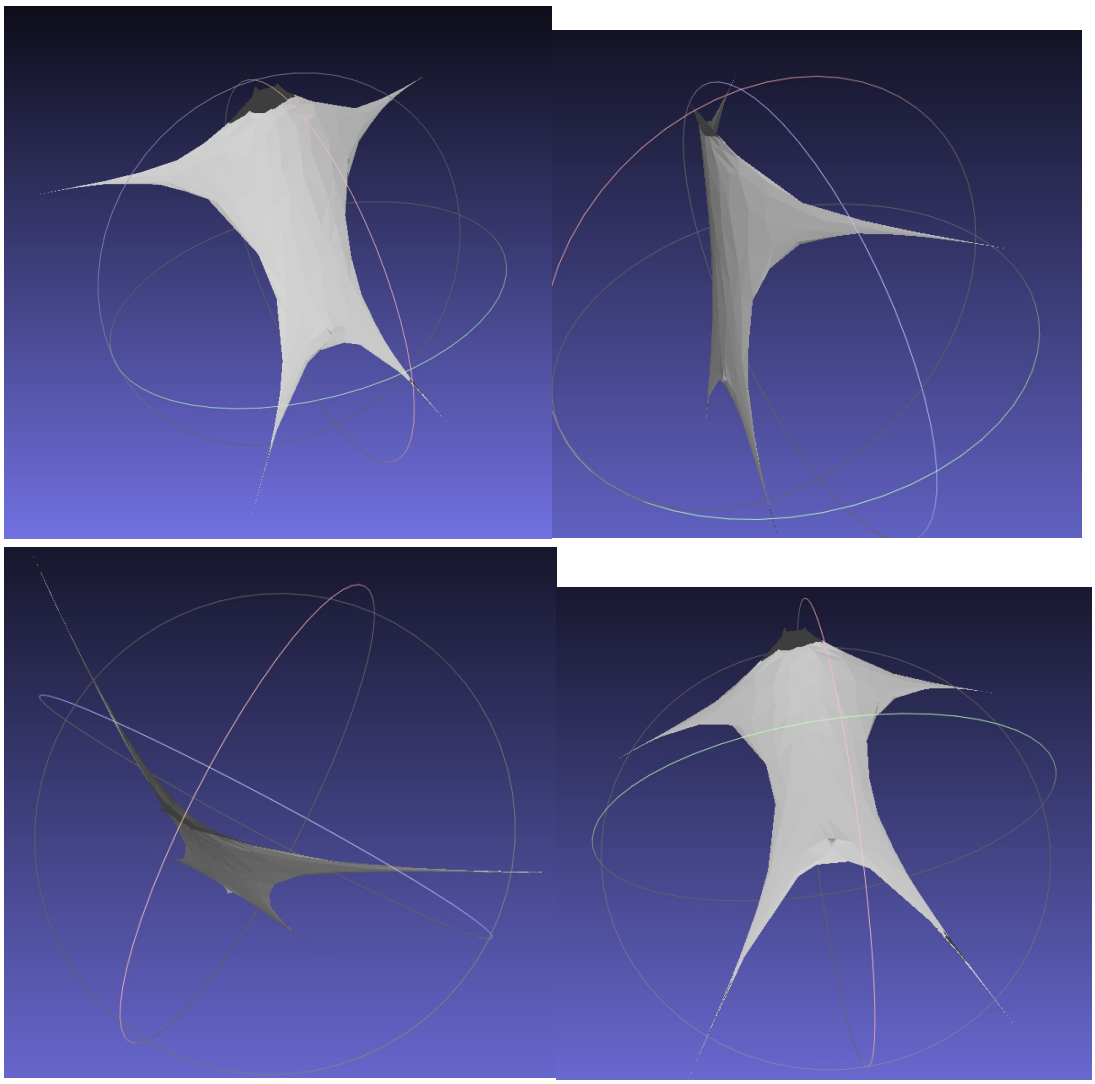
Then, I used scipy.sparse.linalg to compute the k smallest eigen vectors of $\tilde{\Delta}$. Next, I defined a function mesh_reconstruction to reconstruct the mesh from eigen vectors. For each eigen vector, I first compute the dot product of a single component of vertex with M matrix and current eigen vector. This results in a scalar value as a projection on the eigen vector. Then, this scalar is multiplied with the eigen value, and accumulated for all eigen values.

By replacing the vertex position with the horizontally stacked new x,y and z values, the mesh is successfully reconstructed with symmetric feature preserved. I took k=5, 15 and 200 in this experiment.

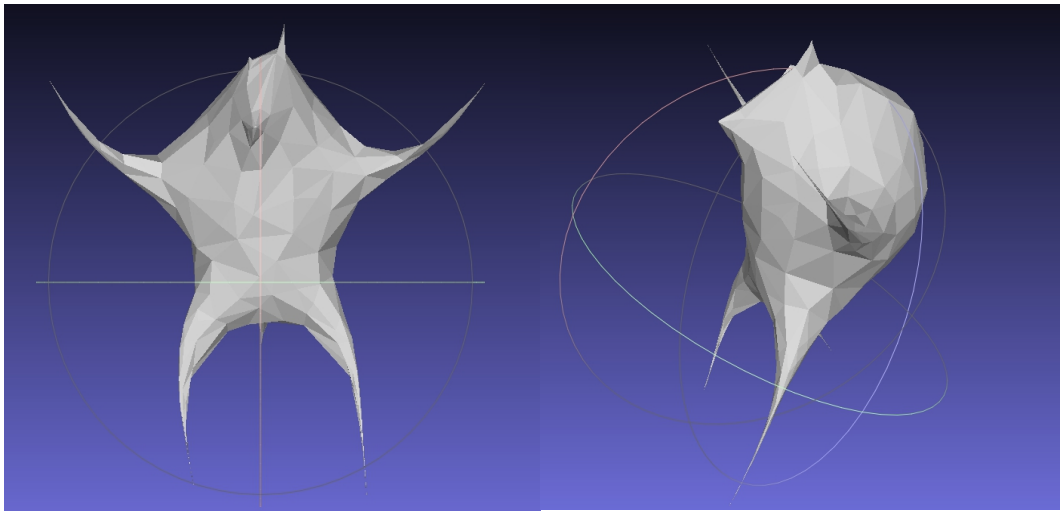
Result:

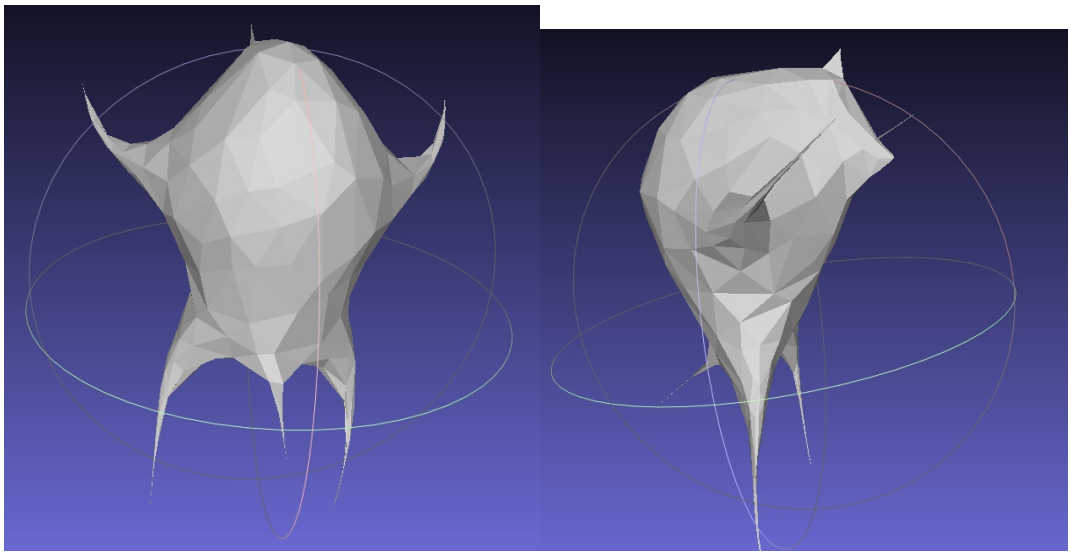
K=5

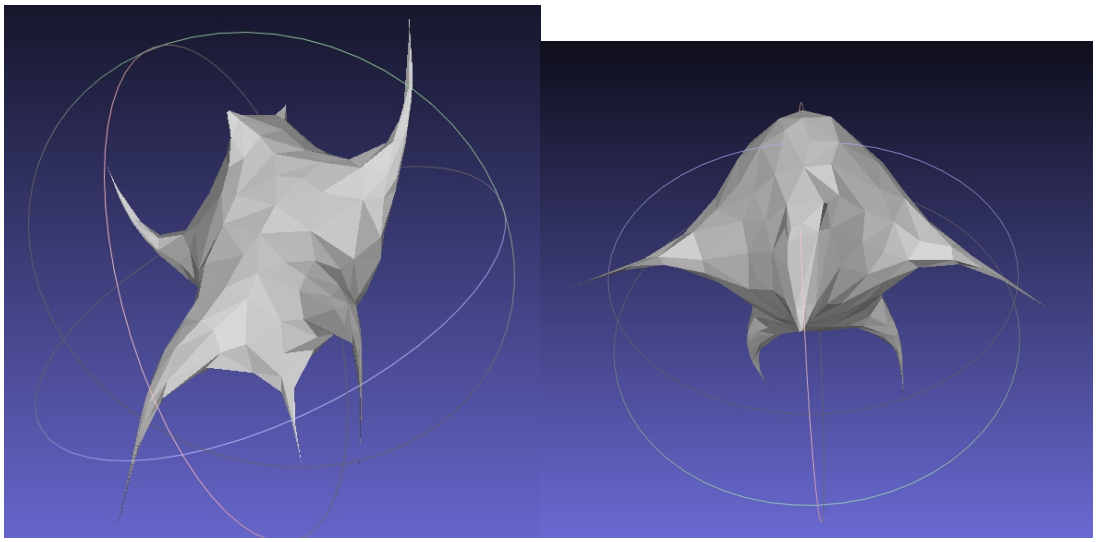




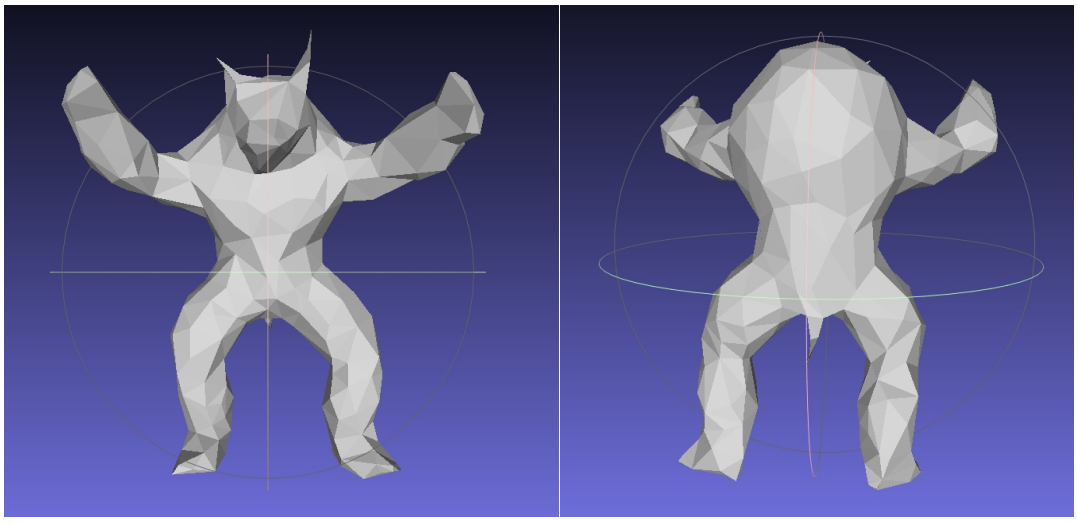
K = 15

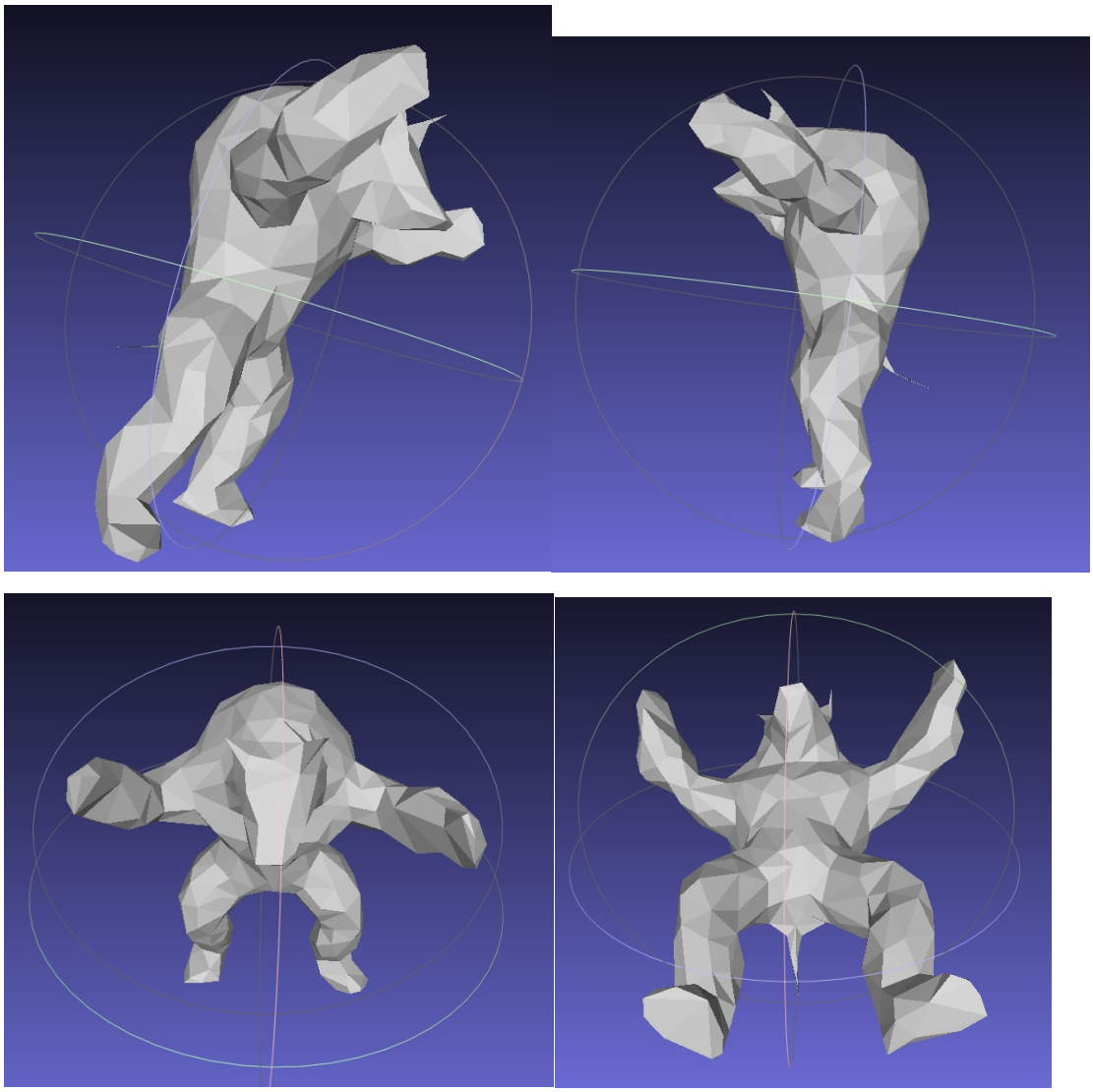






K = 200





The more eigen vectors used, the closer the result is compared to the original mode. Also, the thickness of the model grows with the number of eigen vectors used.

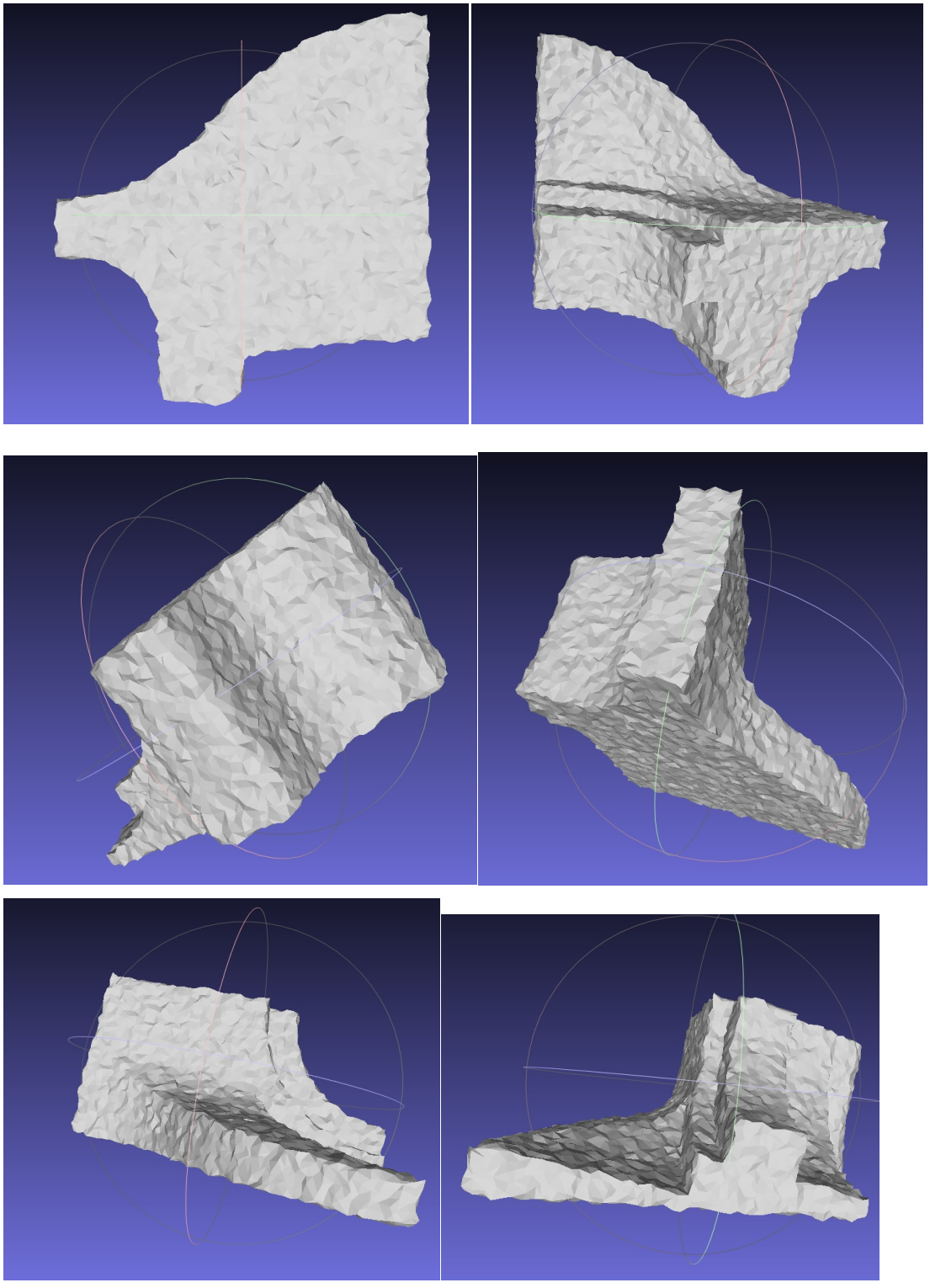
Question 5

This question implements the formula below, in an iterative way:

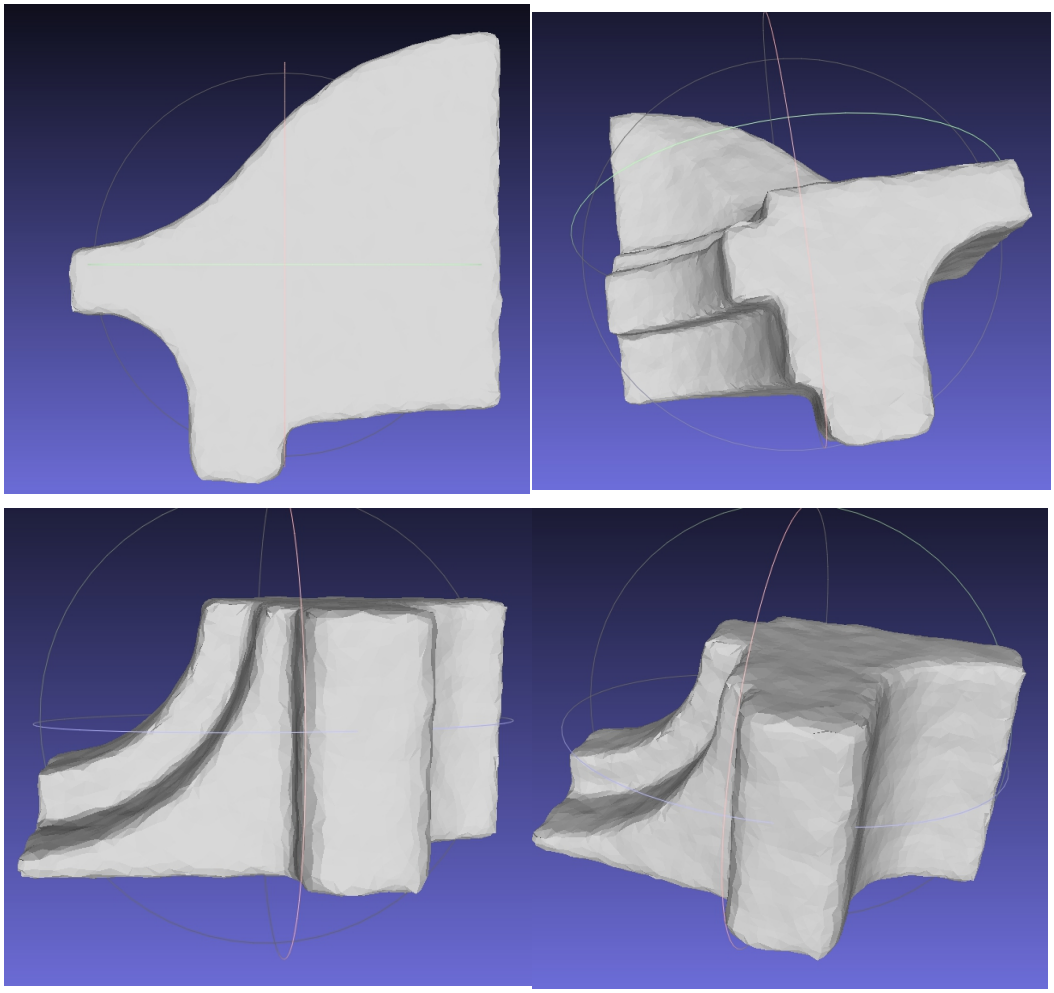
$$\mathbf{p}_i^{(t+1)} = \mathbf{p}_i^{(t)} + \lambda \Delta \mathbf{p}_i^{(t)}$$

For each iteration, the new vertices position is computed by adding the product of Laplace-Beltrami, λ and old position onto old position. At the end of iteration, the old position becomes the updated position. I choose λ = 0.001 in this case, and tested 1 iteration and 5 iteration.

Iteration = 1

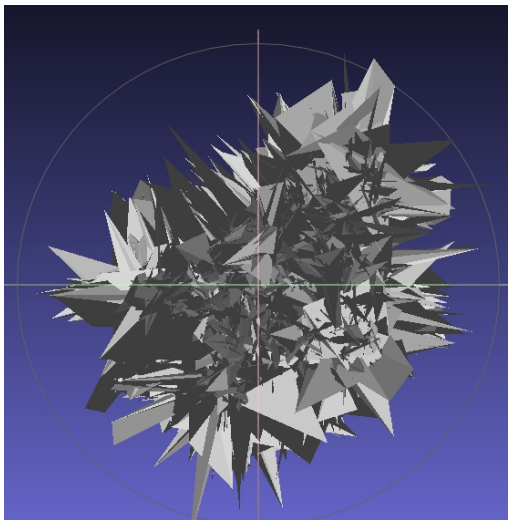


Iteration = 5:



The effect of one iteration of smoothing might not be very obvious, but there's a significant difference after iteration 5. The mesh becomes reasonably smooth.

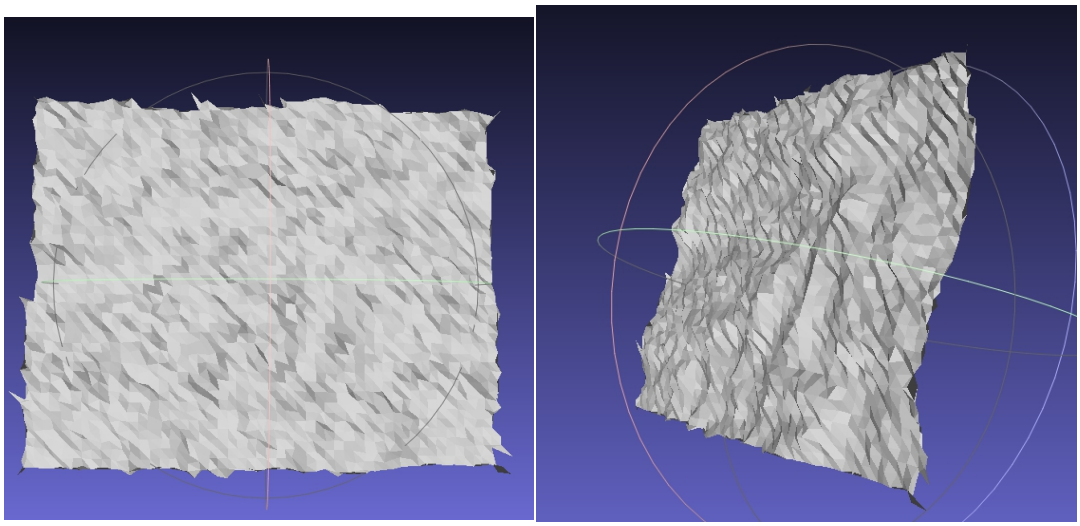
If the λ value is too big (0.1 in this case), the mesh will look exploded. This is because the laplace-beltrami matrix itself is already quite large due to the small face area. Therefore the lambda value should keep small.

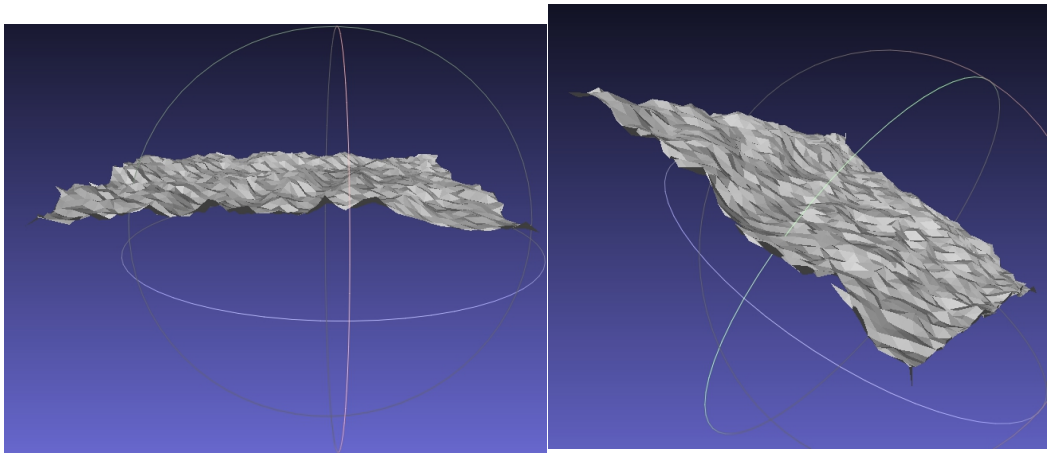


For the mesh "plane_ns", I choose λ = 0.00005 to prevent the mesh from exploding. However,

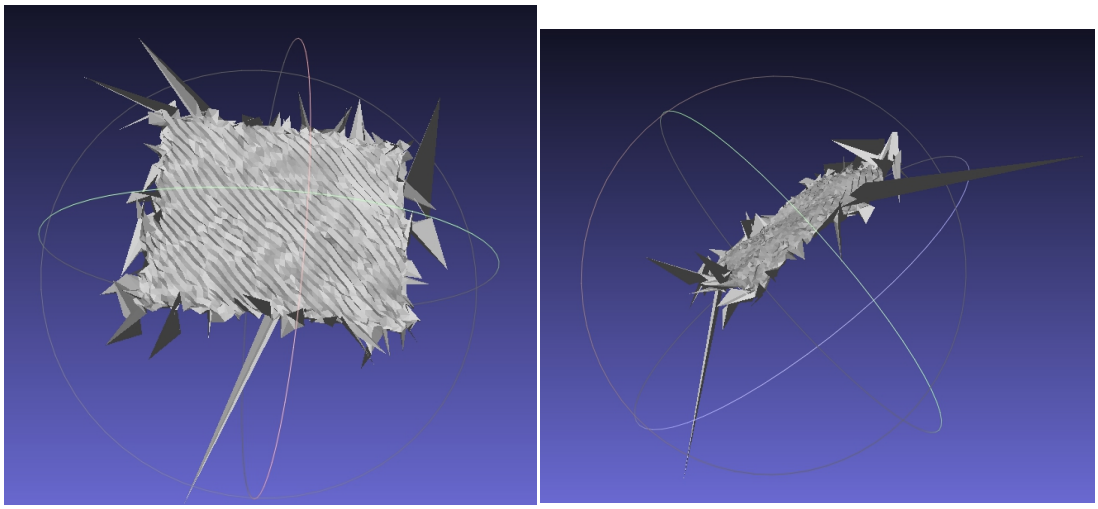
after iteration 5, the edge has been exaggerated too much that the shape has lost.

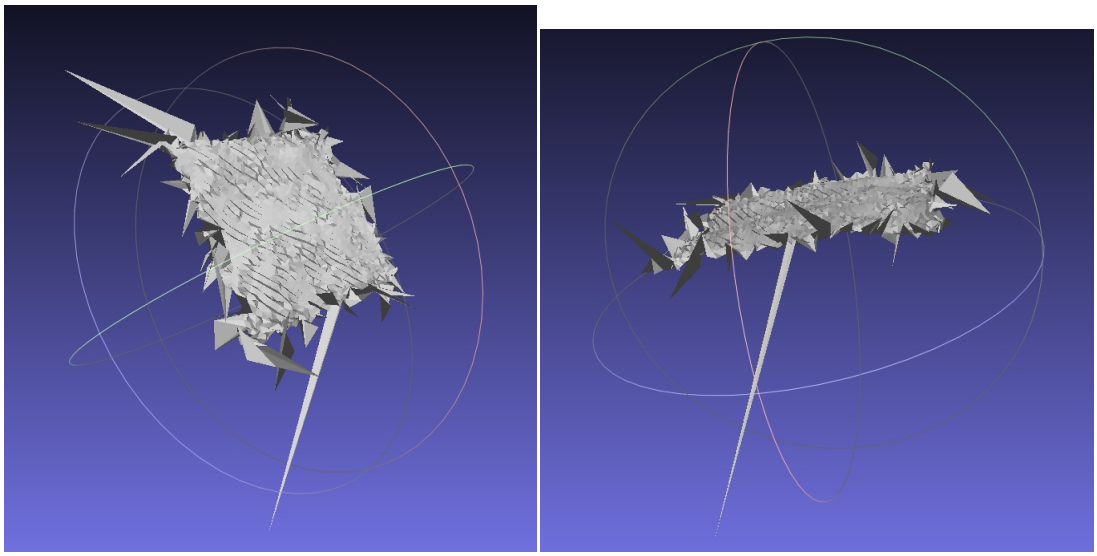
Iteration = 1





Iteration = 5





This smoothing algorithm works well for non-planar objects wile some details are preserved. It is not plane friendly even with a very small λ.

Question 6

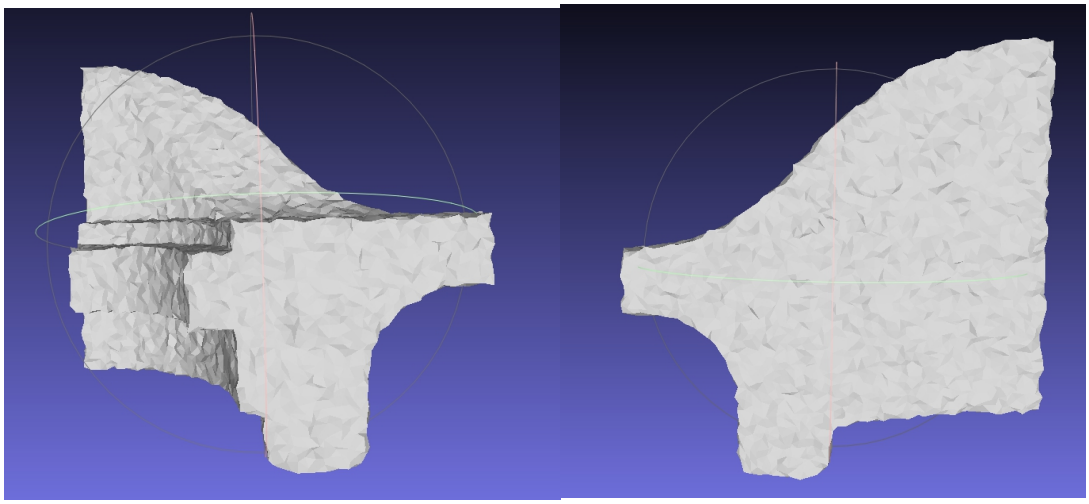
This question implements the formula below, in an iterative way:

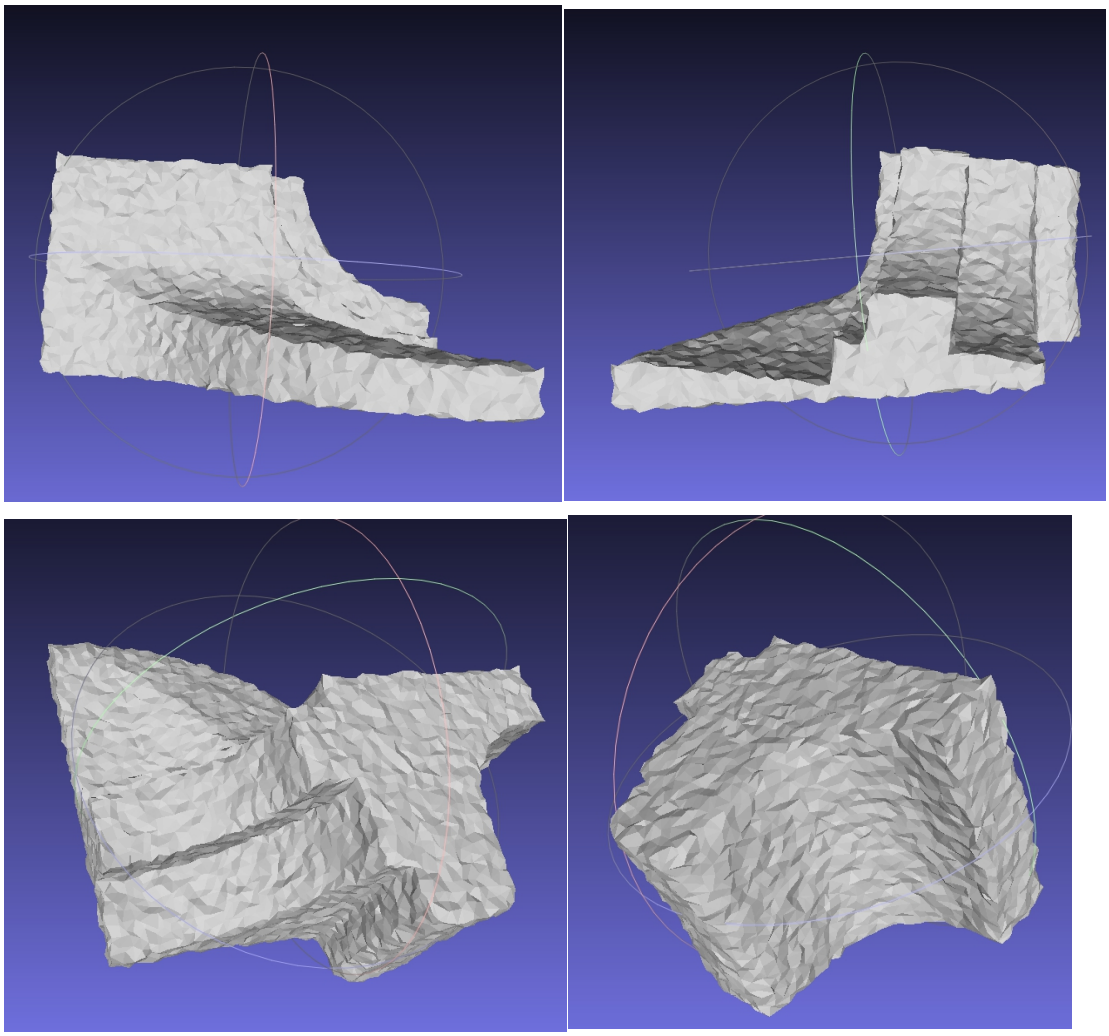
$$(\mathbf{M} - \lambda \mathbf{L}_w)\mathbf{P}^{(t+1)} = \mathbf{M}\mathbf{P}^{(t)}$$

The L_w matrix in equivalent to the C matrix in my previous implementation. After computing the laplace-beltrami operator, the M^-1 matrix has been inversed to get the M matrix. Then, the formula becomes an Ax = b problem, where A is (M - λL_w), x is the new position $P^{(t+1)}$, b is $MP^{(t)}$. I have used scipy.linalg.solve to get the solution, and replace the old vertex position by the result at the end of each iteration.

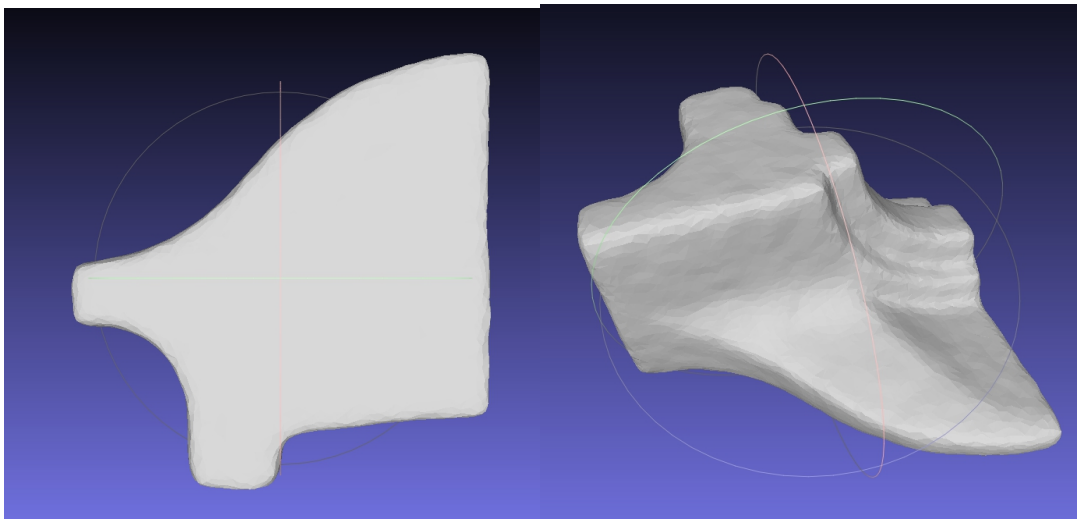
Result:

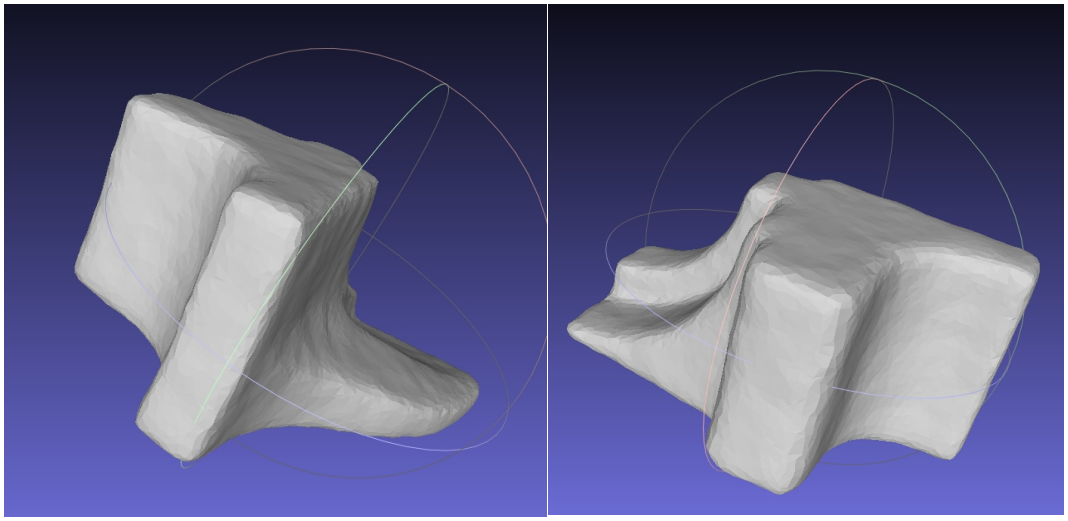
Iteration = 1, λ = 0.001

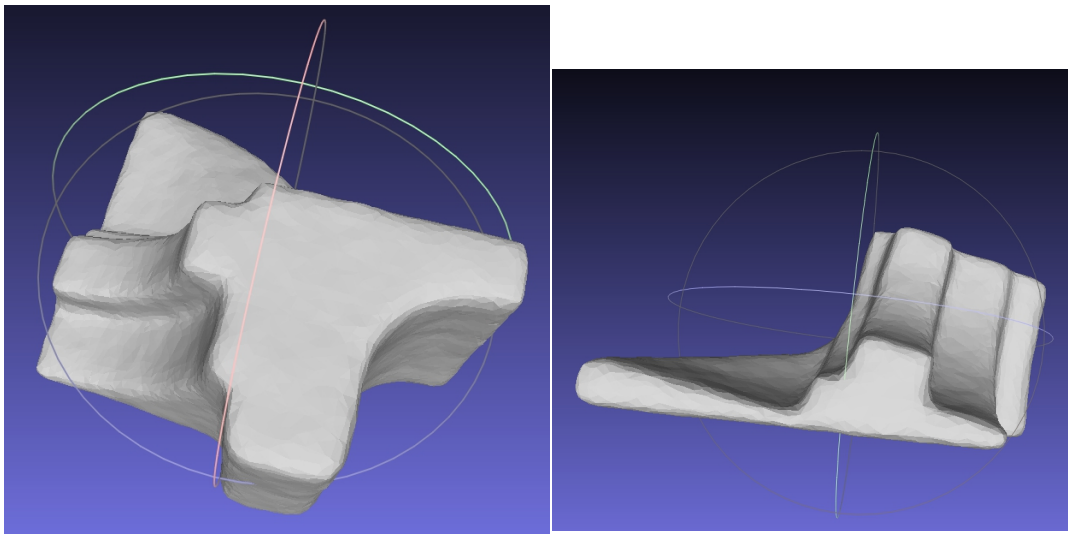




Iteration = 5, λ = 0.001:



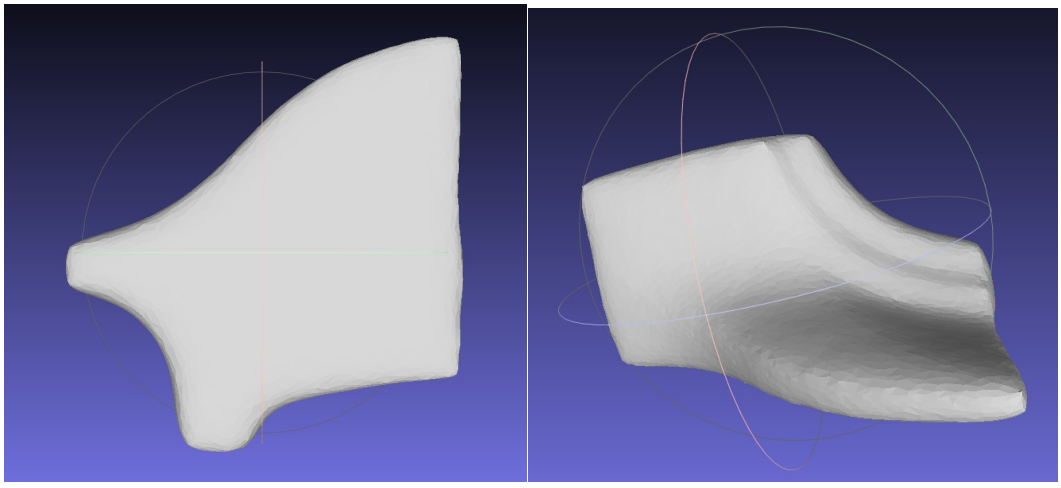


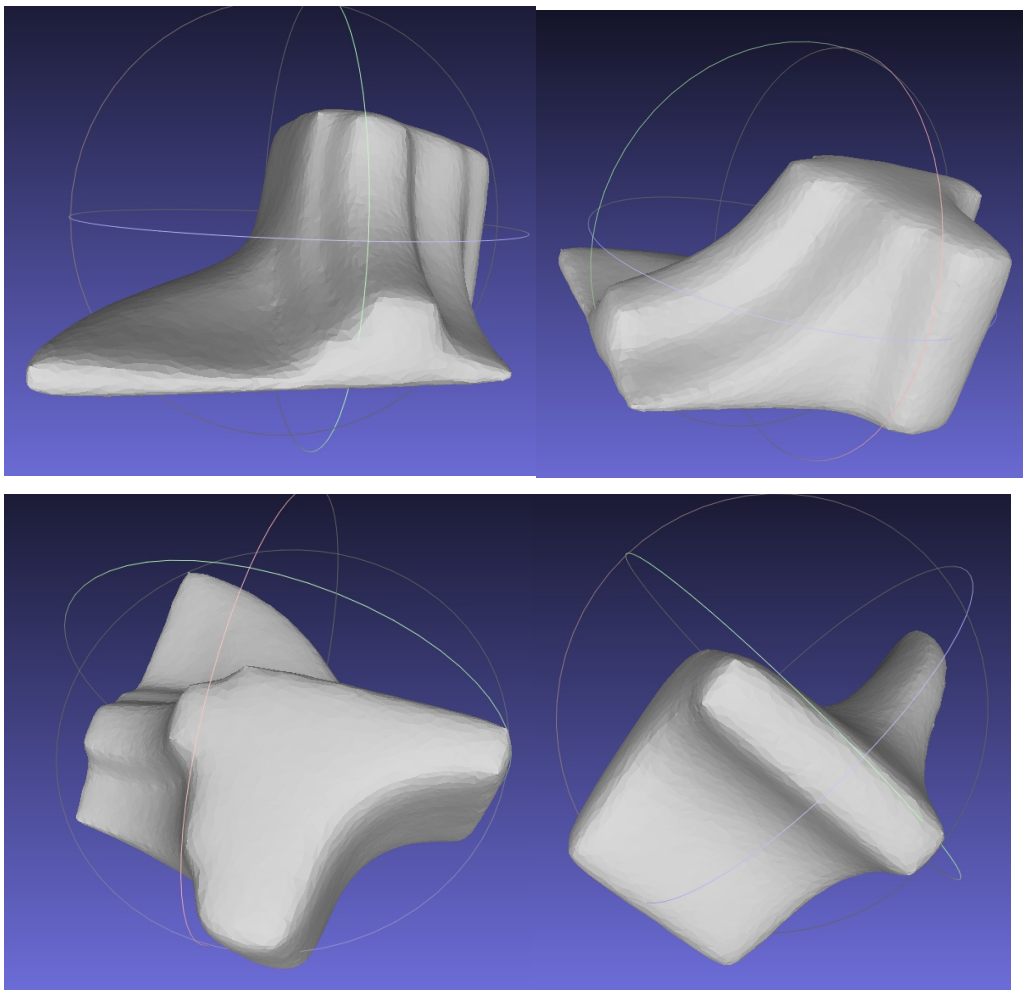


The other mesh “plane_ns” has the same result as question.

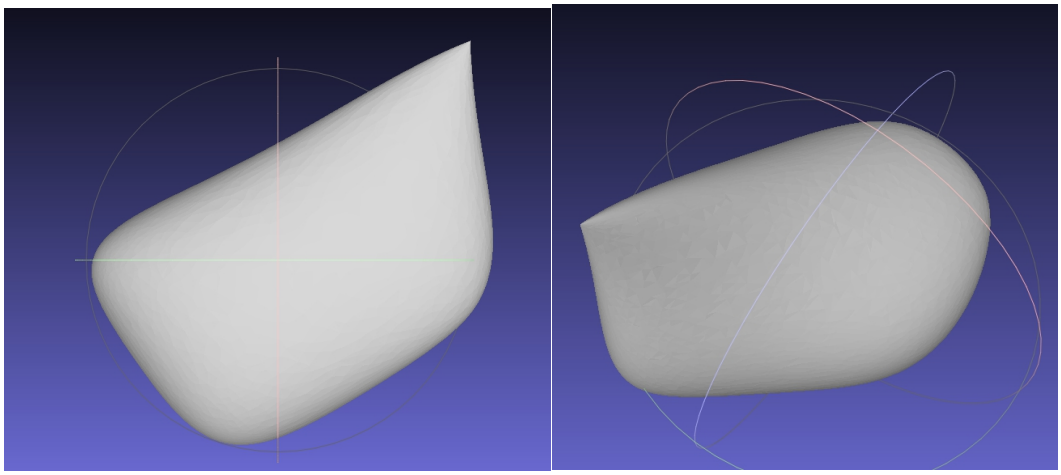
Then I have increased the λ and have following results:

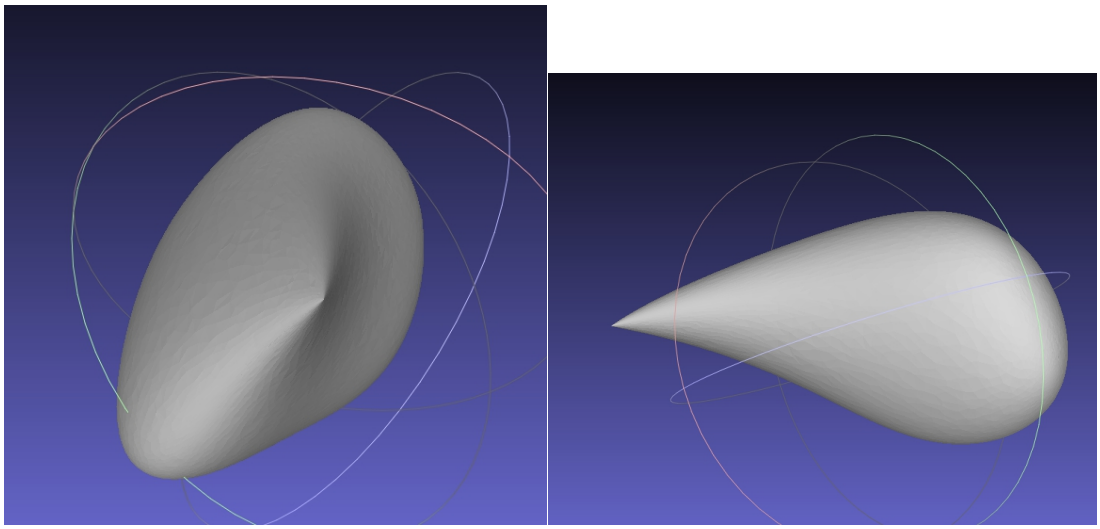
Iteration = 1, λ = 0.05:





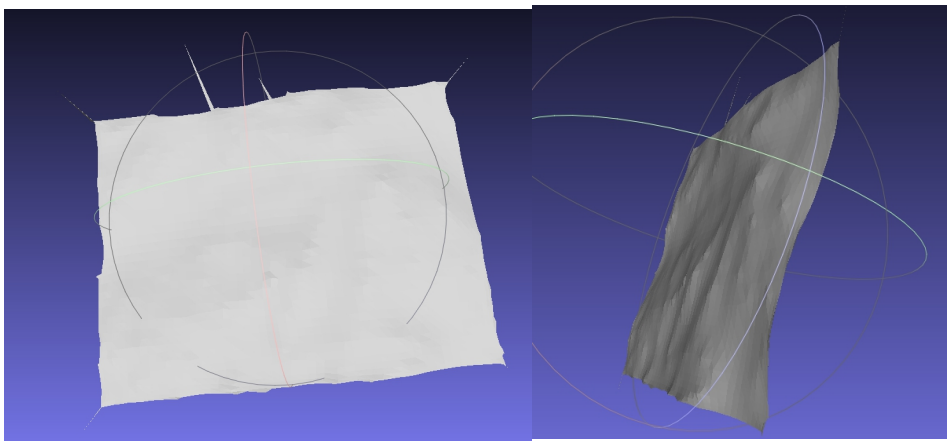
Iteration = 5, λ = 0.05:

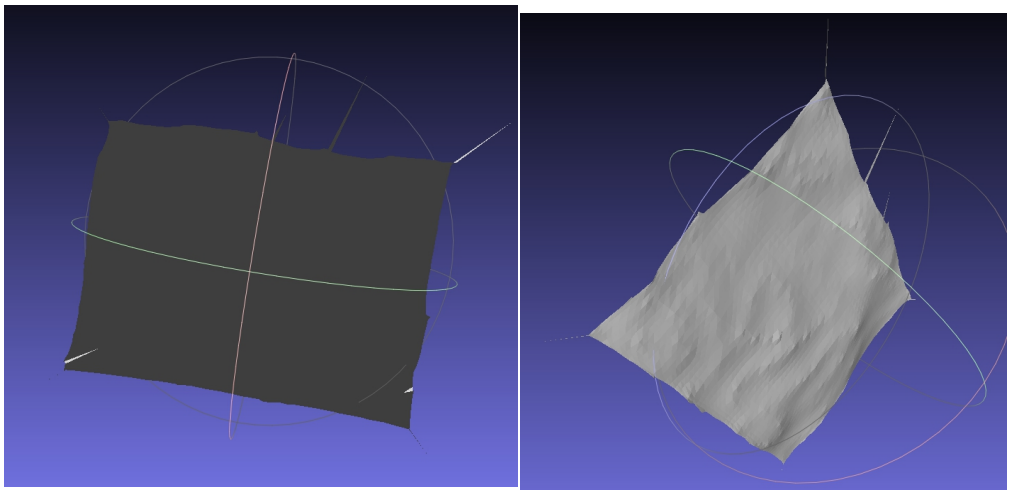




We can see that when λ is larger, the smooth effect is already quite obvious after a single iteration. After 5 iteration, the shape is totally lost. In general, when λ is larger and iteration is smaller, the mesh details are not preserved very well compared to smaller λ but larger iteration number.

For the other model, the λ needs to be even smaller for the planar shape. I chose λ = 0.0001 for 1 iteration, the planar mesh has been flattened dramatically.





In conclusion, the choice of λ and iteration number has to be carefully designed to balance the

smooth efficiency and quality.