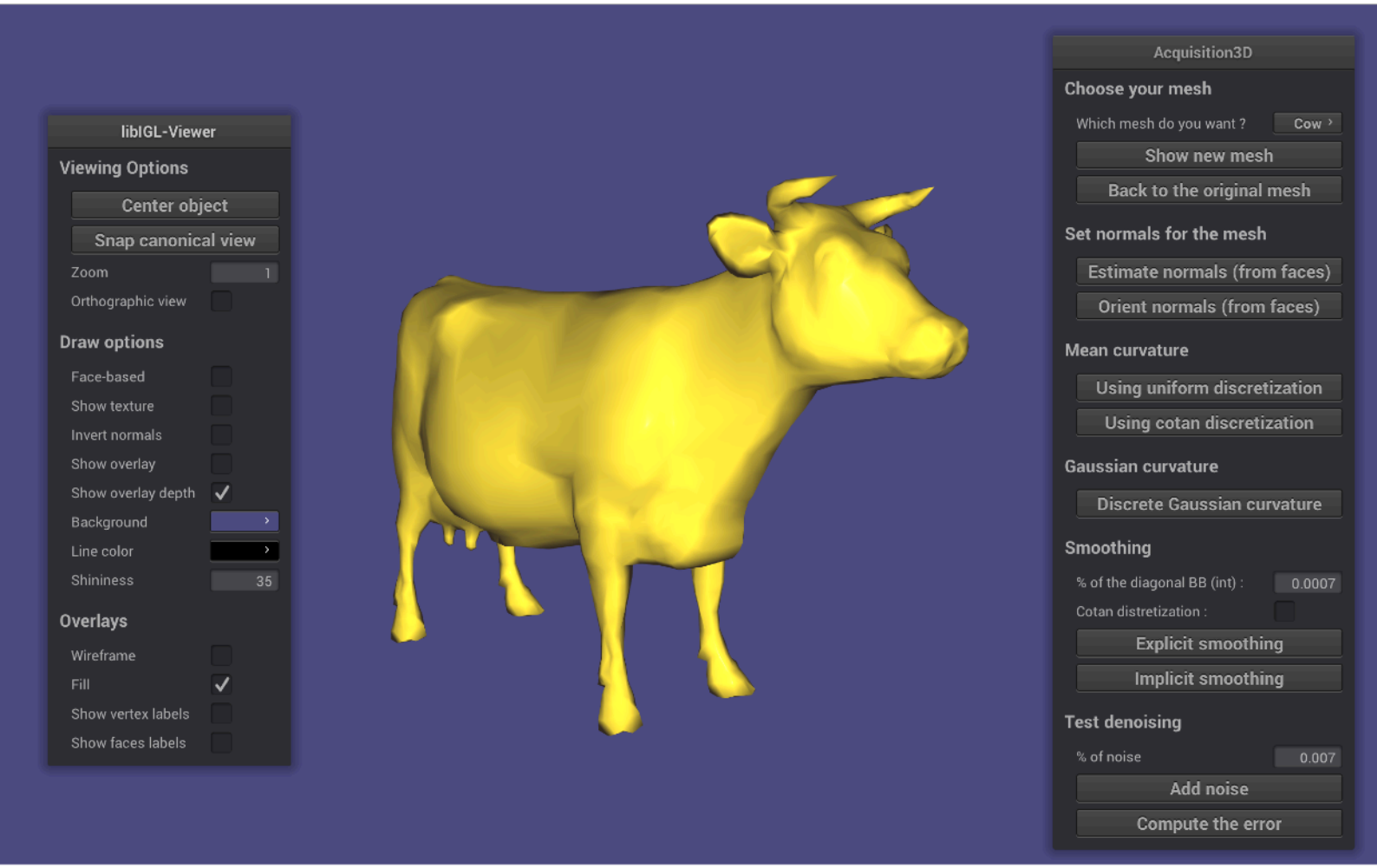
Assignment 2
Discrete Curvature and Laplacian mesh smoothing
Geometry Processing

For this assignment, I reused the framework given on Moodle. Here is the interface I computed :



You can see that it's possible to chose the mesh we want to perform smoothing on (4 choices, the bumpty, the bunny, the screwdriver and the cow). You can also choose with which curvature (mean or gaussian) you want to color the mesh but also which level of noise to add, the type of smoothing to perform and which step size to use. I'll go through an explanation of the different functions in this report.

For this coursework, I computed the functions in 4 files : discreteCurvature.h and discreteCurvature.cpp for the first part and smoothing.cpp and smoothing.h for the second part. I also try to code more efficiently with less functions hard coded than for the first coursework.

All my sparse matrices have been computed using sparse method into libGl, using a tripletList of type Triplet<double> from which I built my sparse matrices using 'setFromTriplets'.

Part 1 : Discrete Curvature

Question 1 : Uniform Laplace : mean curvature

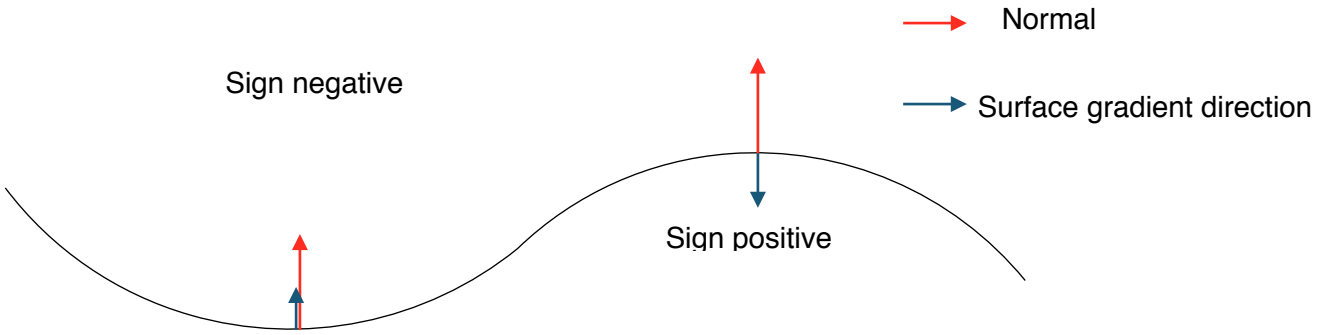
First, I compute the mean curvature using a uniform weighting. Indeed, the magnitude of the mean curvature is define this way :

$$H = \frac{1}{2} \|\Delta_{\mathcal{S}} \mathbf{x}\|$$

Hence, we need to choose the discretization to build the laplacian operator. In this first case, we just compute the weight of each neighbor as the inverse of the valence for the vertex's neighbors and -1 for the current vertex. This function is in discreteCurvature.cpp, *Eigen::MatrixXd meanCurvature(DecoratedCloud & cloud, int typeDiscretization)* which calls *Eigen::SparseMatrix<double> uniformLaplacian(DecoratedCloud & cloud)* to compute the L matrix. For each vertex this equation is satisfy :

$$\Delta_{\text{uni}} f(v_i) := \frac{1}{|\mathcal{N}_1(v_i)|} \sum_{v_j \in \mathcal{N}_1(v_i)} (f(v_j) - f(v_i))$$

Then, once we have the magnitude, we need to compute the sign of the mean curvature. Indeed, the sign is positive if the surface gradient's direction is opposite to the surface normal and negative is they are in the same direction.

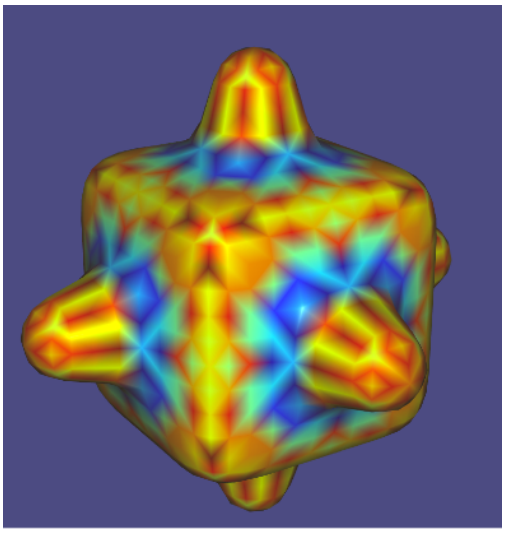


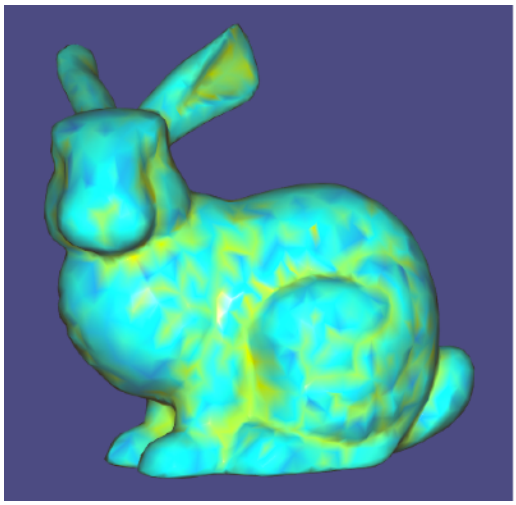
Indeed, in any case, the sign of the mean curvature must satisfy this equation :

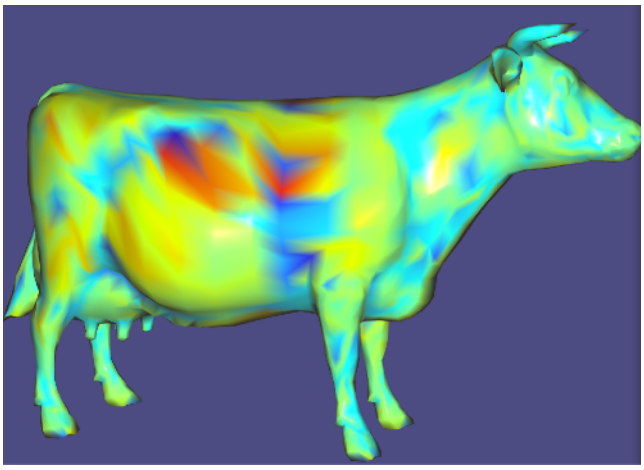
$$\Delta_{\mathcal{S}} \mathbf{x} = \text{div}_{\mathcal{S}} \nabla_{\mathcal{S}} \mathbf{x} = -2H\mathbf{n}$$

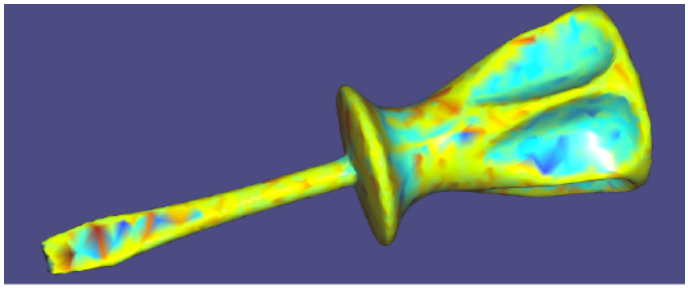
To compute the matrix, I go through all the vertices and then through all the faces. If I find a face with the current vertex, I store the next vertices taking advantages of the fact that triangles vertices are always stored in the same order. Then, I compute the valence by looking of the size of the vector I created and fill the matrix using the indices of neighbor and 1/valence.
Moreover, as the normals need to be computed accurately in order to obtain the right sign for H, I computed them using the function of the previous coursework to find the orientation with respect to the faces. It's why there are those 2 functions in the interface.

Here are the results for 4 meshes :

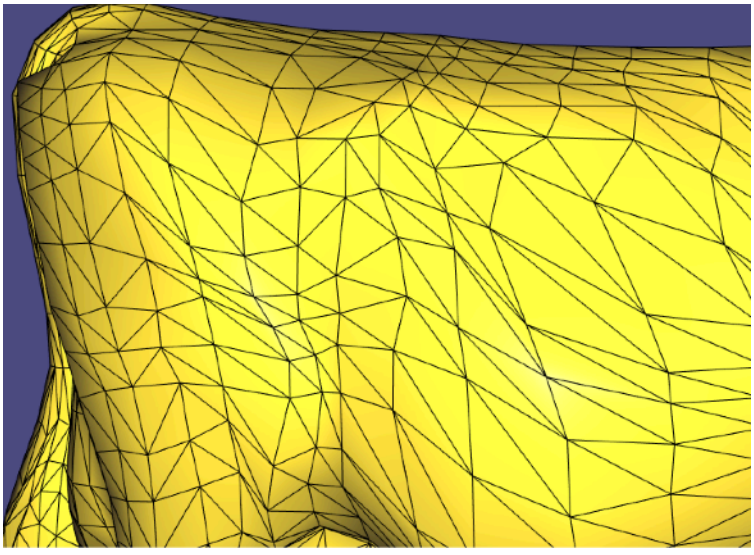








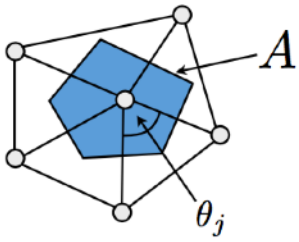
We can see the result isn't really convincing. Indeed, the result only depends on the connectivity, assuming that in places where the connectivity is high, the magnitude of the curvature is more importante because the mesh needs more details. However, this assumption doesn't always hold (see after). Then, the sign of the curvature is defined using the normal and surface gradient directions.
As I said, this discretization fails even more for mesh with irregular triangulation or bad tessellation. For example, the cow is in this case :



As the mesh is really irregular in this area, the uniform discretization isn't a good approximation. We need to add details about the angles of the triangles for example. It's done by computing the cotan weight.

Question 1 : Uniform Laplace : Discrete Gaussian curvature

Then, I computed the Gaussian curvature. For each vertex, I found the triangle it belongs to and I computed the angle for this triangle like that :

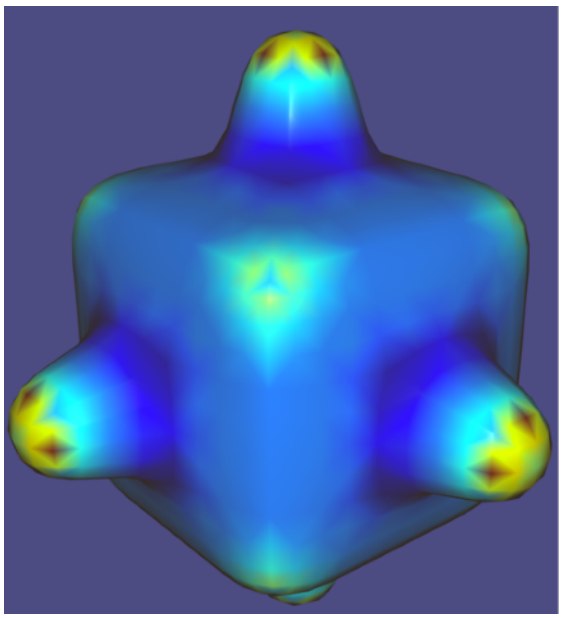


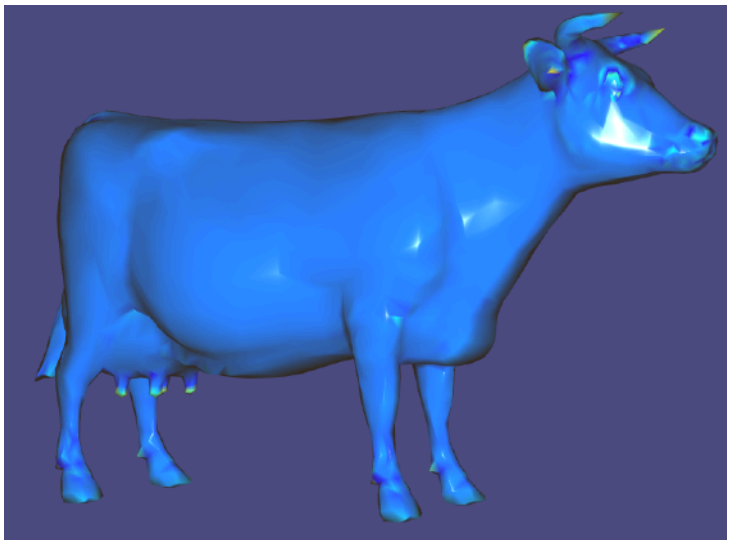
I also computed the area A for each vertex using the Barycentric cells. I simply computed the area of each vertex and divide it by 3. Then, for each vertex, I computed the gaussian curvature using this formula :

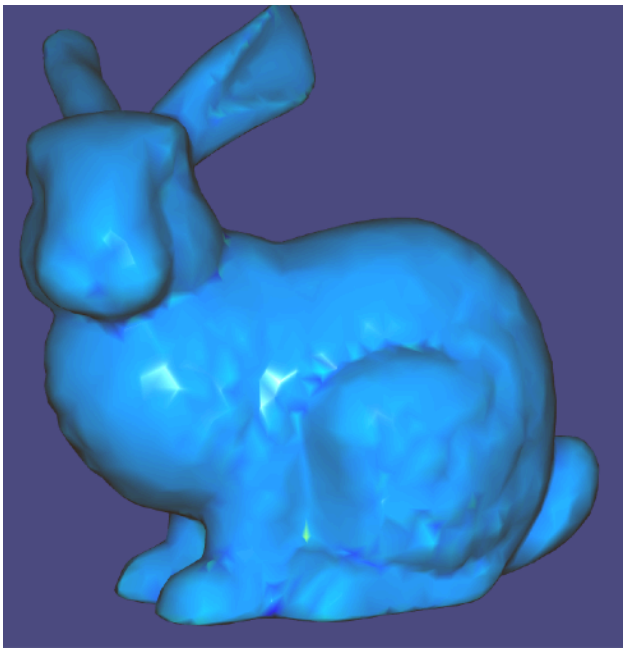
$$K = (2\pi - \sum_j \theta_j)/A$$

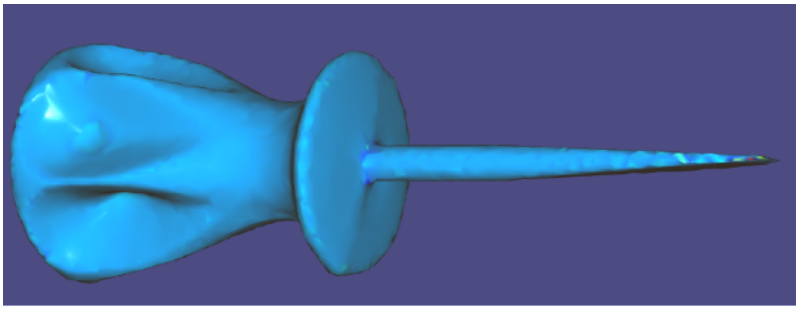
We can see that the gaussian curvature can be seen that an « angle deficit », as K is supposed to be 2pi in flat regions.
The result is computed in : Eigen::MatrixXd gaussianCurvUnifom(DecoratedCloud & cloud).

Here is the result :









The best mesh to observe the result is the bumpty. We can see that angles with a high deficit are on the high curvature area on the « bump » and low gaussian curvature area are in the valley.

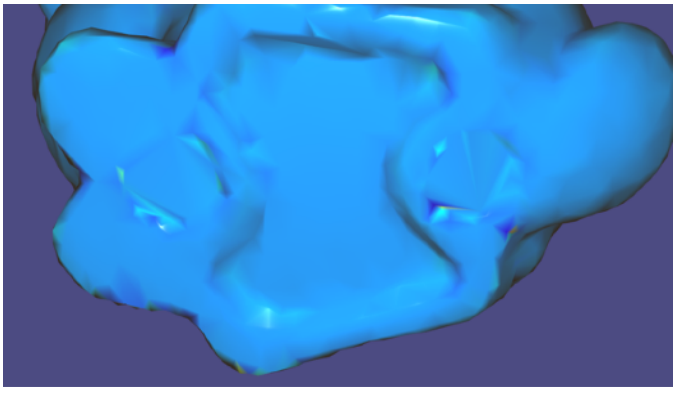
Indeed, the gaussian curvature is also define like that :

$$K = \kappa_1 \cdot \kappa_2$$

which explains that its value is high only in place where the value of the maximal and minimal curvatures are importante. For example, on the edge of the bumpty, the minimal curvature is zero so the value of K is low. It's the same for the bunny of the skew driver.

We can see high gaussian curvature area on the extremity of the cow horn and udder.

However, as the color map is stretched between the high and low curvature value (for exemple the ear of the bunny), most of the mesh is blue. For example, in the case of the bunny, because of area below the mesh like that and place of very high curvature as the eare :



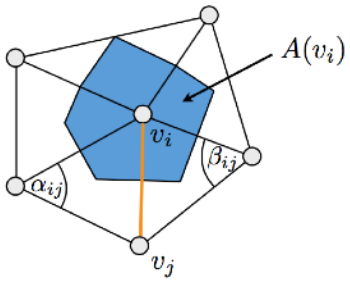
We can't see the difference of curvature on the body part.
You will see after how to obtain a better visualization after smoothing.

Question 2 : Non-Uniform Laplace Beltrami : cotan discretization

To compute this question, I used the separation of the L matrix into M, the diagonal matrix composed of 2*Area for each vertex (*Eigen::SparseMatrix<double> diagonalArea(DecoratedCloud & cloud)*) and the C matrix which contains the weight of each neighbor (*Eigen::SparseMatrix<double> weightCotan(DecoratedCloud & cloud))* computed as the sum of the cotan angles around its edge. Then, the function to compute the L matrix is in *computeCotanDiscretization(DecoratedCloud & cloud)*.
The function *meanCurvature* asks in parameter for the type of discretization wanted.
To compute the C matrix I used the *triangle_triangle_adjacency* function built in in libGl. From a matrix of faces, the function computes 2 matrices. The first one TT, store at each row i the indices of the neighboring triangle for the triangle i stored by in the same order as the edges. The second one TTi stores the indices of the edge of the triangle in TT in contact with the triangle i.
With those matrices and the fact that vertices in triangles are always stored in same order, we can find the position of the vertex and compute the cotton weight from that.

Indeed, this equation must be satisfied for each vertex :

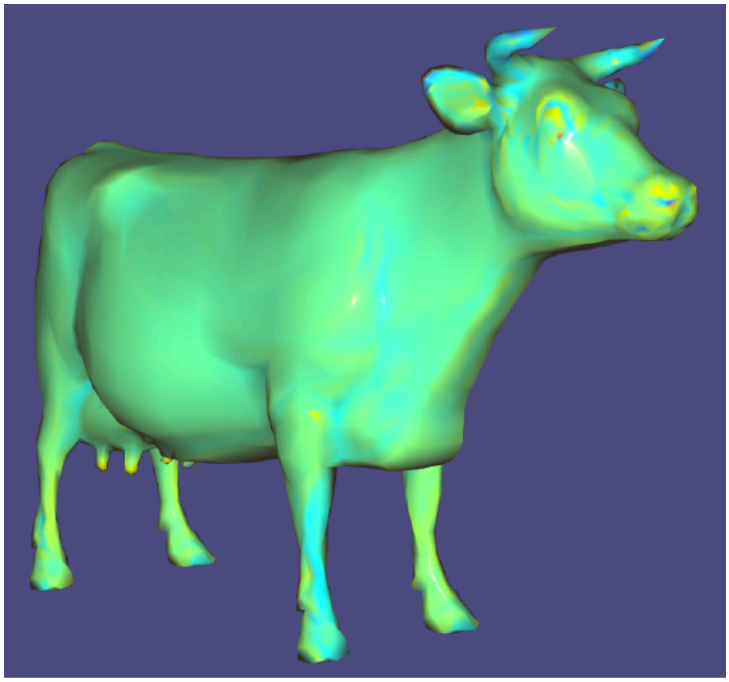
$$\Delta_{\mathcal{S}} f(v_i) := \frac{1}{2A(v_i)} \sum_{v_j \in \mathcal{N}_1(v_i)} (\cot \alpha_{ij} + \cot \beta_{ij}) (f(v_j) - f(v_i))$$

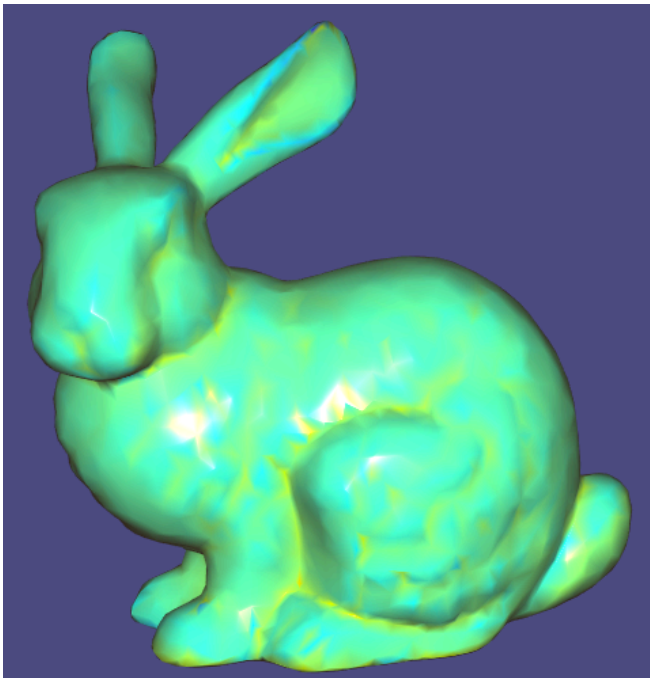


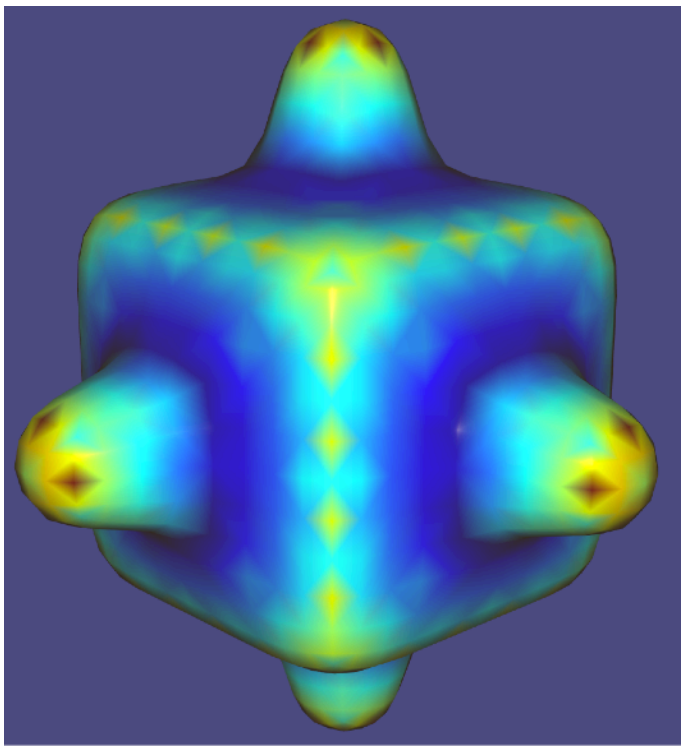
Then, the C matrix is construct this way :

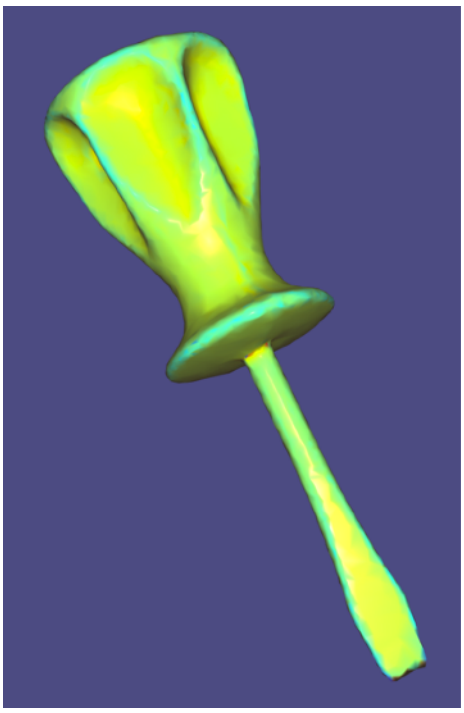
$$\mathbf{C}_{ij} = \begin{cases} \cot\alpha_{ij} + \cot\beta_{ij}, & i \neq j\ ,\ j \in \mathcal{N}_1(v_i) \\ -\sum_{v_j \in \mathcal{N}_1(v_i)} (\cot\alpha_{ij} + \cot\beta_{ij}) & i = j \\ 0 & \text{otherwise} \end{cases}$$

Here is the result for this part :









We can see the result is way more convincing using this discretization that with the uniform one. Indeed, the mean curvature express the mean between the minimal and maximal curvature. Then, it's supposed to be medium when 1 of the 2 is high, low when the 2 are low and high when both of them are high. The best mesh to visualized that is, once again, the bumpty. The highest value is for the bump and the lowest for the valley.

Moreover, artifacts due to bad triangulation have disappeared on the concerned meshes. The cow side is uniform and the bumpty isn't crossed by same value mean curvature lines (corresponding to triangles with the same valence value).

Hence, the best approximation for continuous curvature is definitely using the cotan discretization for the mean curvature. As the uniform one only uses the connectivity, the resulting colors aren't really showing the curvature (only in place where meshes obey the assumptions : high valence means high curvature area).

For the next part all the meshes are colored using the mean curvature computed with the cotan discretization

Part 2 : Laplacian mesh smoothing

Question 1 : Explicit Laplacian mesh smoothing

The function to perform smoothing is in « smoothing.cpp » and is called *explicitSmoothing(DecoratedCloud& cloud, double lambda, int typeDiscretization)*. Indeed, we can choose the type of discretization we want to use (uniform or cotan) in order to see the difference.

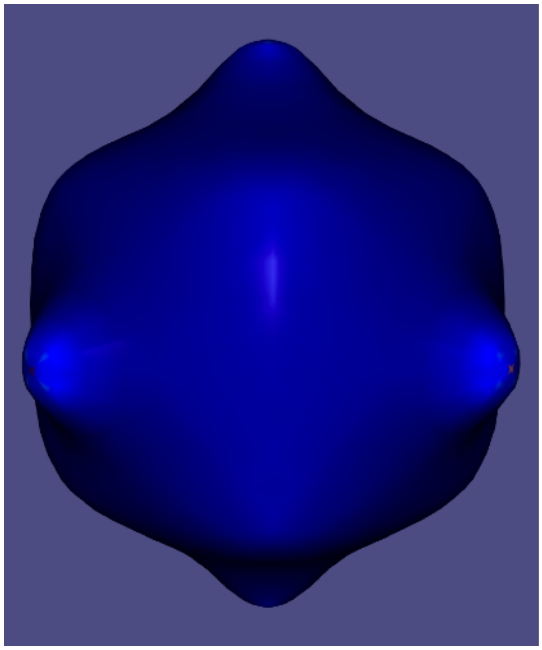
$$\mathbf{P}^{(t+1)} = (\mathbf{I} + \lambda \mathbf{L})\,\mathbf{P}^{(t)}$$

We must satisfy this equation to perform explicit smoothing. The size of lambda must be small to guarantee stability.
I used the size of the diagonal bounding box to define lambda. Indeed, the user is asked to enter the wanted pourcentage of the diagonal bounding box he wants to use as lambda. This way, I can try to find a pertinent pourcentage for all meshes independent of scale.

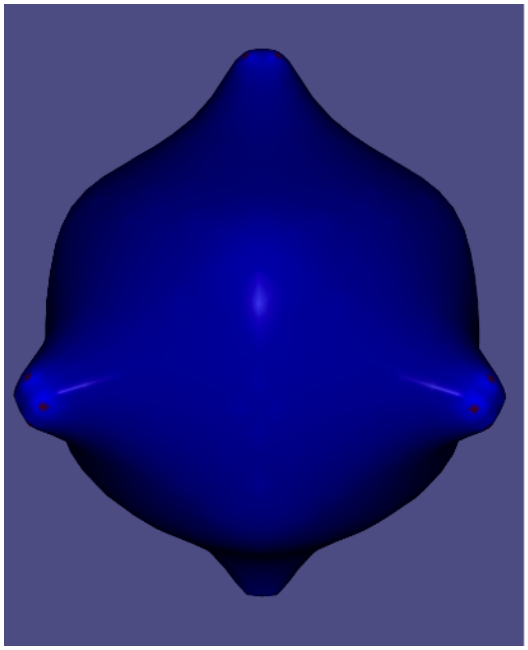
Uniform VS cotan discretization :
For this part I try to investigate the difference between uniform and cotan discretization when performing explicit smoothing. I only did it for one mesh (the Bumpty).

First of all, I noticed that the size of lambda needs to be higher when using the uniform than when using the cotan discretization. Indeed, the weights in the cotan discretization are bigger than in the uniform so the new vertices move faster already and need a smaller lambda size.

Then, the final result also differs. Using the uniform discretization, as we only take into account the connectivity, the mesh will never be as smoothed as using the cotan which considers also the angles. For example for the bumpty :



Using cotan with 0.02% of the diagonale bounding box as lambda with 20 iterations

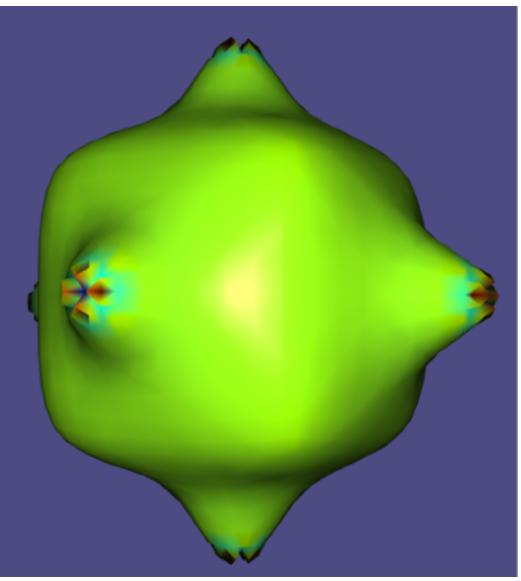


Using uniform with 1.6% of the diagonale bounding box as lambda with 20 iterations

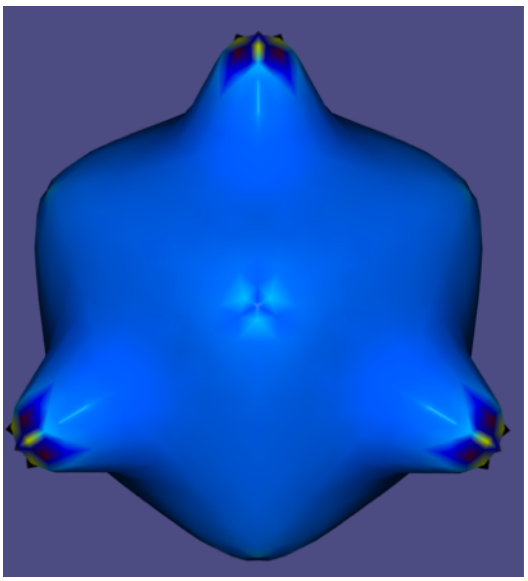
We can see that in both cases we won't arrive to a perfect sphere as we could have expected but, the smoothing using the cotan discretization is rounder on the bump than the one using the uniform discretization.
For now on, I will focus on the explicit smoothing using cotan discretization as we saw it obtains better results.

What happens if lambda is too big ? :

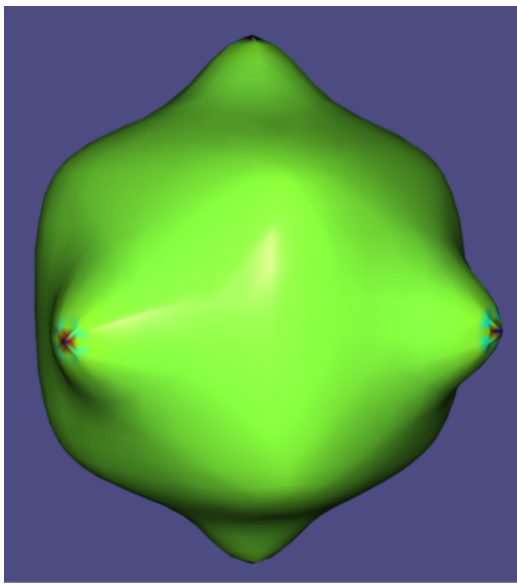
As I said, we can't arrive to a perfect sphere. It's because if we choose to increase the pourcentage of the bounding box used, some vertex «switch » position by overshooting and the result « explodes ». Hence, we can see that the efficiency and the stability of the explicit smoothing relies on the size of lambda.
Some example :



0.1% result after 10 iterations

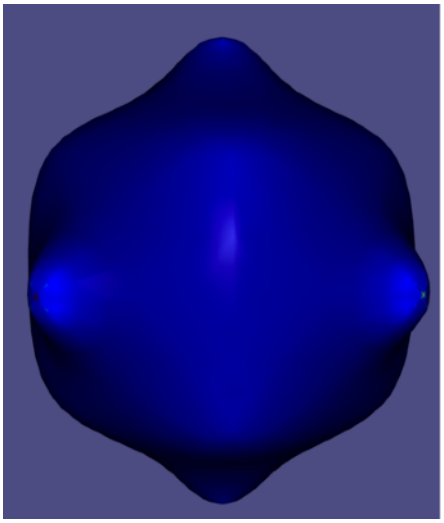


0.15% result after 4 iterations



0.05% result after 23 iterations

As you can see, artifacts appear at some points even if the size of lambda is really small (0.05%). The bigger the pourcentage of the bounding box is, the faster those « switches » happen. Moreover, if we want to obtain a good smoothing, we need to choose a very small size for lambda and apply many iterations :

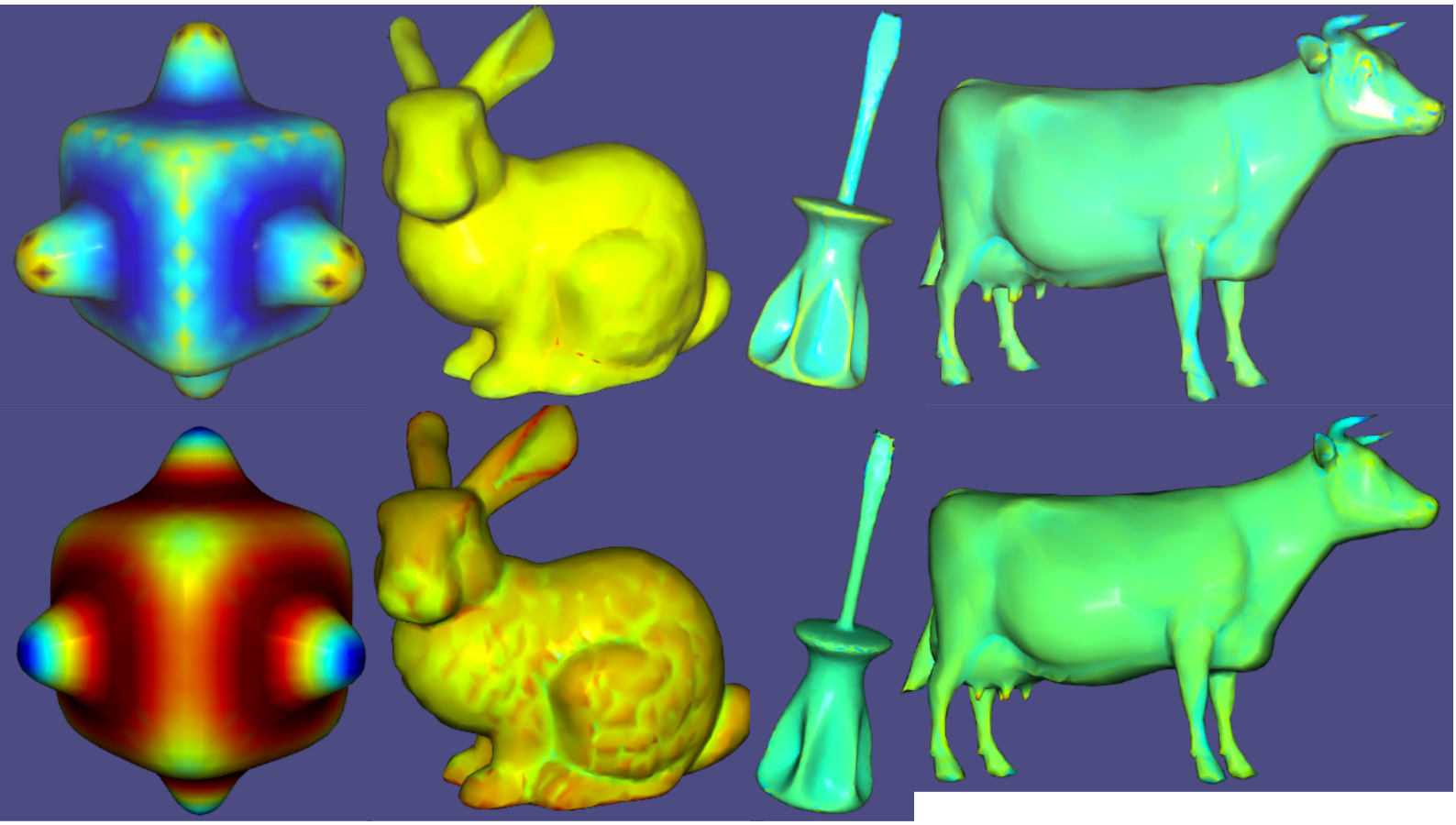


0.01% result after 103 iterations

To the left we have the result with a 0.01% of the bounding box. 103 iterations were required and I stoped when the bump were about to explode.
Hence, even with the cotan discretization the result of the smoothing isn't really convincing.
The colors aren't reliable because the artifacts create very high or very low values which stretch the color map.

What's a good size of lambda ? :
To perform this test, I investigated using the four meshes for the size of the bounding box which smoothed without any artifact for at least 10 iterations using the cotan discretization

	Bumpty	Bunny	Scew-driver	Cow
Size of the bounding box	0.05	1E-05	9E-05	0.0001



All the meshes are colored with the mean curvature using cotan discretization and smoothed also using cotan.
I would say there is not perfect size for lambda as it depends a lot on the mesh quality and structure. For example, the screwdriver requires a small lambda as its extremity is really thin, hence, « switches » of vertices can happen really fast. We can also observe that the colormap for the bumpty is inverted but I can't really explain why (except maybe bad scaling of the colormap).
For the bunny, on the bottom of it, the mesh isn't really nice as we saw before, the triangulation is really irregular, so the colormap of original mesh is stretched and we can't see the difference in curvature in the mesh. However, smoothing redistributed the bad vertices and it's now easier to observe the difference in curvature in the smoothed version.

In any case, lambda needs to be around 10-4, higher if the mesh includes thin part or isn't really of a good quality.

Question 2 : Implicit Laplacian mesh smoothing

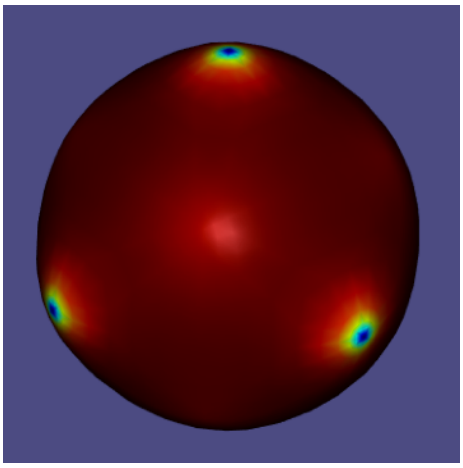
The code for this function is in «*Eigen::MatrixXd implicitSmoothing(DecoratedCloud& cloud, double lambda)*» in the smoothing.cpp file. It compute the Lw and the M matrix and uses the SimplicialLLT class of Eigen to solve this equation :

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

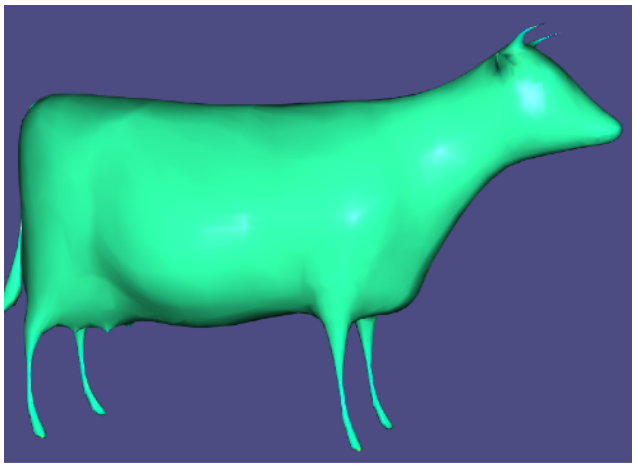
This method is more stable no matter the size of lambda because it optimizes the position of all of the new vertices in the same time using a non linear solver. In other word, we look for the solution which better satisfied our constraints, hence, vertices can't « switch ».

Hence, we can use lambda much higher and still obtain a consistent result. It also saves lots of iteration to obtain, at the end, the same result.

For example



2 iterations with lambda = 10%



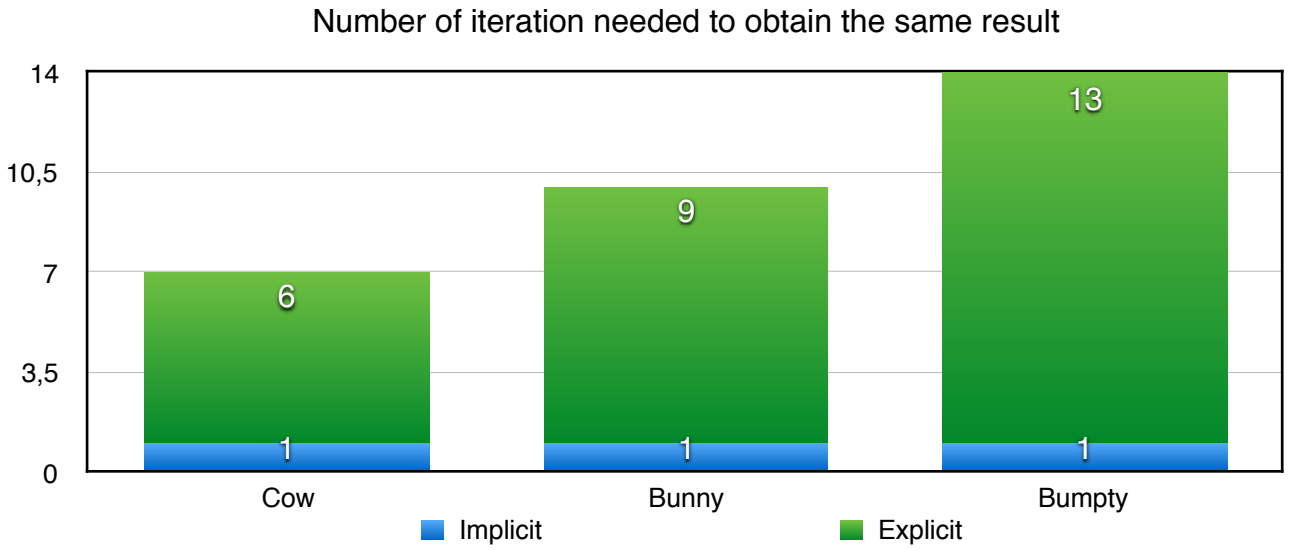
1 iteration with lambda = 0.09%

Even if the algorithm is stable, we can't put a lambda too high or the mesh decreases in size drastically.

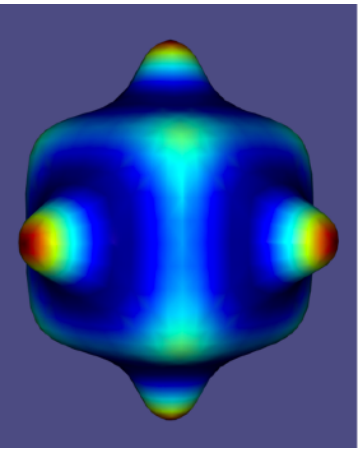
To compare my result I used the cow, the bunny and the bumpty.

I looked how many iterations were needed to obtain the same result using the 2 methods. However, with the first method, lot's of iterations are needed and vertices switch easily for the meshes with very thin parts so I couldn't perform many iterations. Hence, my objectif mesh wasn't very smoothed.

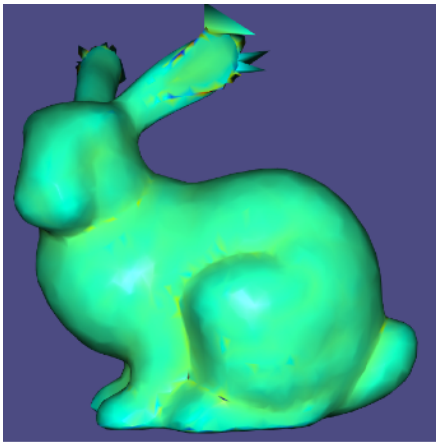
Indeed, as we saw earlier, when doing explicit smoothing we are not changing the topology of the mesh, hence, the flipping of triangles is a problem I tried to take into account when comparing the 2 methods.

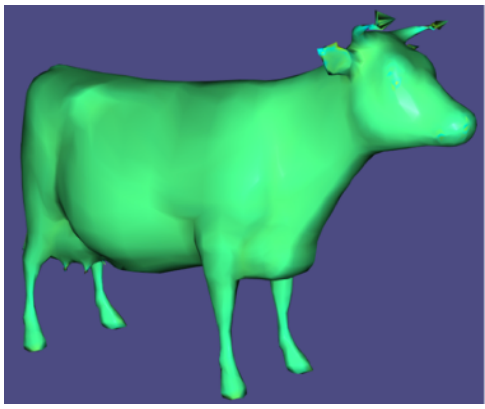


Explicit method :



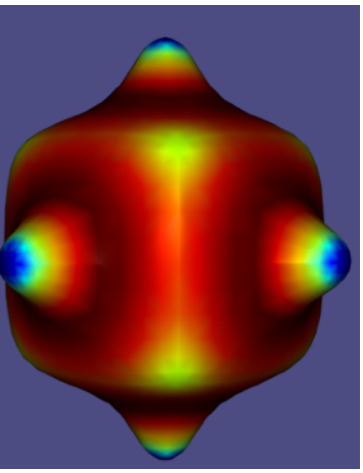
Exp 0.05%, Imp 2%



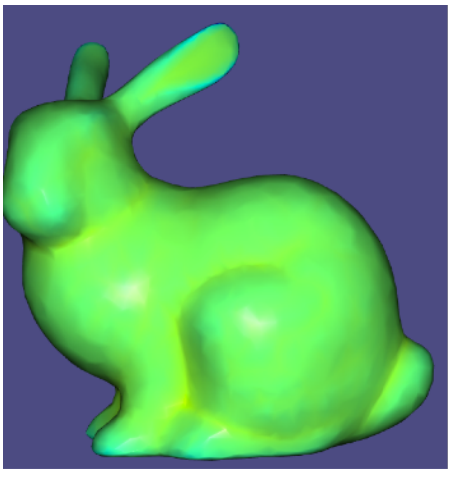


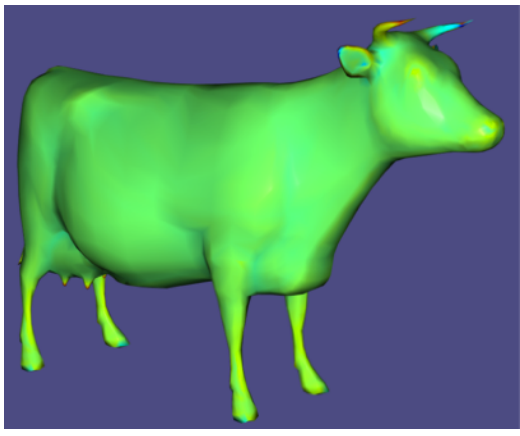
Exp 0.0005%, Imp 0.005%

Implicit method :



Exp 0.0005%, Imp 0.005%





Then, we can see that the implicit method is :

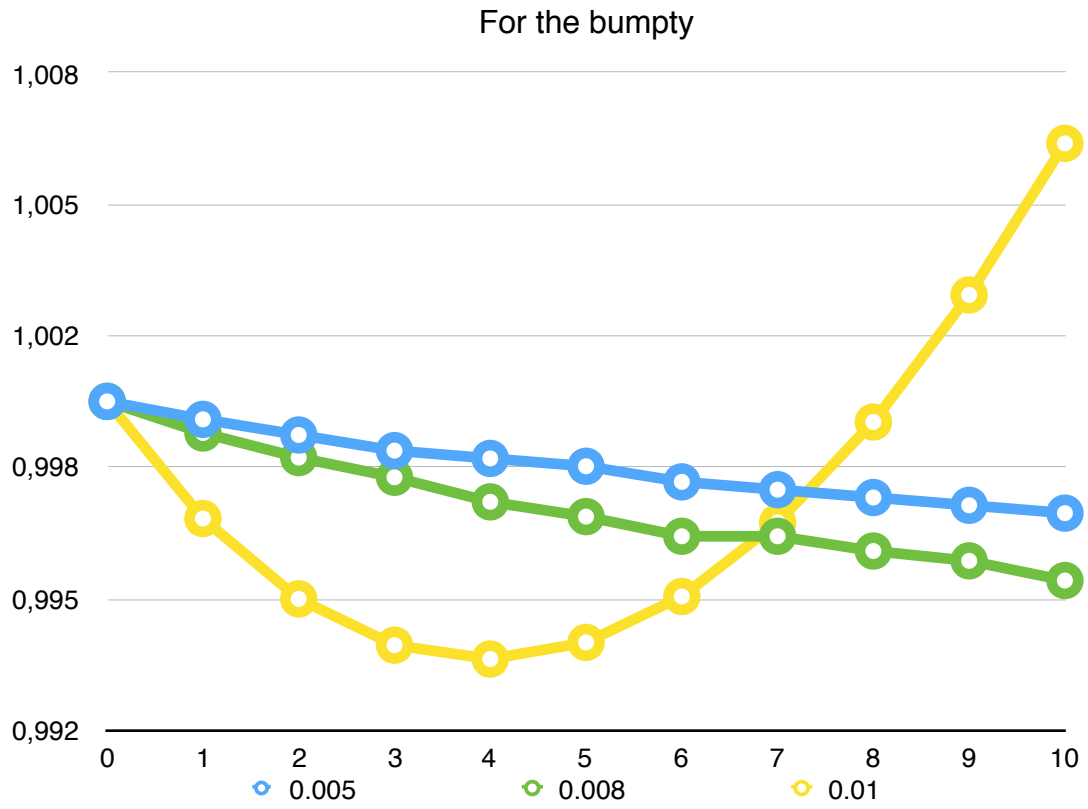
- more stable (we can see the some vertices switched on the ear of the bunny or at the end of the cow's horn)
- requires less iterations (cf table, I stoped when vertices start to switch)
- allows larger step sizes (cf information near images)

Question 3 : Evaluate Laplacian mesh denoising

In order to evaluate the denoising of a mesh, I computed a method to add noise (in smoothing.cpp) and a function to compute the difference between 2 meshes (computeError). The function to add noise asks the amount of noise the user wants to use by creating a Gaussian zero mean noise. The variance of the noise on each direction depends on a pourcentage (indicated by the user) of the bounding box size in each direction.

I will test those methods on the cow and the bumpty.
To try to recover from small noise I used a small lambda in the implicit method.

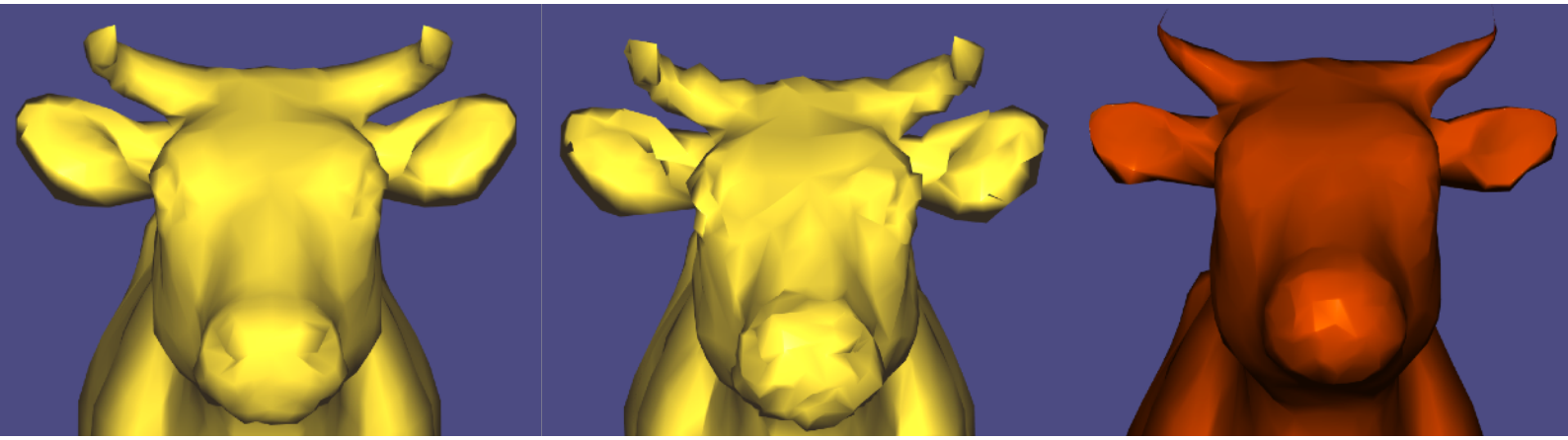
At each time, I used the same pourcentage of the bounding box to smooth than the one use to create noise.



I normalize the error by the error found at the first step after adding noise.
We can see for 2 small noises (0.005 and 0.008) the algorithm succeed reducing the error. However, for higher noise, the algorithm reduces the error much more before augmenting it again.

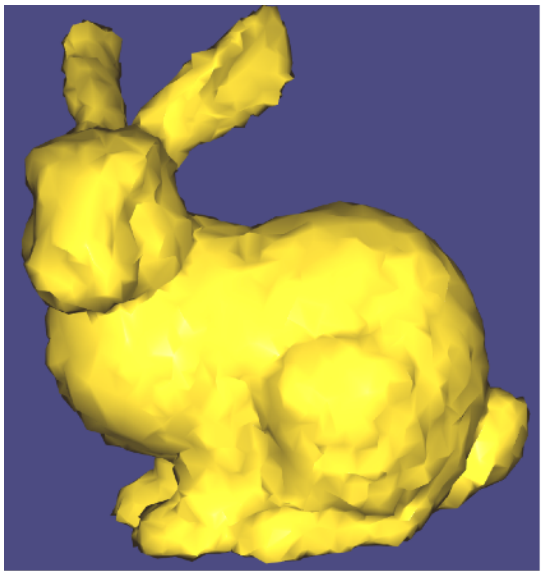
What we notice in this case is that the more importante the noise is, the higher lambda needs to be. However, if lambda is too high the general size of the mesh decreases so the error continues to grow.

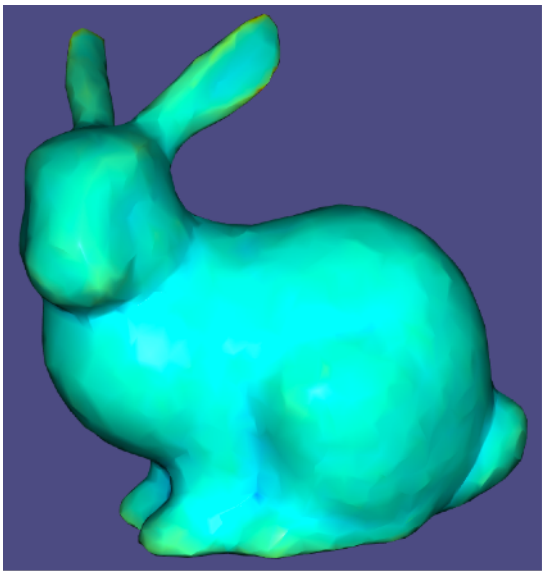
Hence, one of the main issue of this algorithm to denoise is that if the noise is too important, we need to increase lambda and by doing that, the size of the mesh will be reduced. The other main issue is the high frequency details.
Lets try with the cow now.
It's more interesting to look at images to show how evolves the high frequencies features :



Here we can observe the mesh before applying the noise, after applying a noise (0.007%) and the result after 4 iterations with lambda at 0.0007%. We can see we that the mesh red is smoother but the high frequency details have been lost in the process.
Hence, to obtain a good result, we should have a smoothing which differs in flat and high frequency regions.

An other example where the denoising process can get ride of lots of noise but also looses high frequencies details :





To conclude, I would say that the method works really well in flat region (like the bunny body) but fails to recover high frequency details if the noise is too high.
Moreover, I think the fact that the mesh decreases size when applying a lambda too important may be linked to the normalization area used. Indeed, this weight factor should try to preserve the area around a given vertex and the barycentric implementation I used isn't the most efficient and accurate.