COMP0119 Acquisition and
Processing of 3D Geometry, 22/23
Assignment 2
Baptiste Moalic, [redacted]

20/03/23

In this coursework, we will investigate the different methods for mesh smoothing and denoising. We will look at their performance, the benefits they can have, their drawbacks, and to what extent they are dependent on the input mesh. We will concentrate on the uniform and non-uniform versions of the Laplace-Beltrami operator (they can both have advantages depending on what representation we want to achieve), the explicit and implicit versions of the Laplacian mesh smoothing (the latter is usually more efficient, but the trade-offs found between the different parameters can land us to the same conclusions), and their application to mesh denoising (they have a limited range of action, but we will discuss how we can possibly make them more efficient).

1 Uniform Laplace

- To compute the mean curvature H, we will use the formula found in slide 29 of lecture 06:

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

  where Δ_S represents the Laplace-Beltrami operator and $\boldsymbol{x}$ represents our mesh. We have different choices of implementation for this operator. In this part we will focus on uniform discretization (slide 23 lecture 06):

$$\Delta_S \boldsymbol{x_i} = \frac{1}{N_1(\boldsymbol{x_i})} \sum_{\boldsymbol{x_j} \in N_1(\boldsymbol{x_i})} (\boldsymbol{x_j} - \boldsymbol{x_i})$$

  where the $\boldsymbol{x_i}$ represent the vertices of our mesh.
  If we implement directly $\Delta_S \boldsymbol{x_i}$, we can simply iterate over all vertices and perform this sum. But, if we want to represent the matrix Δ_S, we need only to compute the factors in front of $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$. In this case, we have the inverse of the number of $\boldsymbol{x_i}$'s neighbors. It means that the row i will contain this factor for every neighbor in column j, and minus this factor times the number of neighbors for the vertex i (which is the same as -1).
  As seen in the lecture, this Laplacian is dependent on the mesh used.
  About the numerical implementation, let's note that we have used a sparse matrix. Indeed, as a vertex has on average 6 neighbors, we will only have on average 7 non-zero values per row.

- To compute the Gaussian curvature K, we use the formula found in slide 29 of lecture 06:

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

  where θ represents one angle between the vertex of interest and two of its neighbors, and A represents a "cell" around the vertex of interest. Here, we have used the model of barycentric cells (i.e the area around the vertex of interest is one third of the total triangle area made by this vertex and all of its neighbors).
  To implement this numerically, we have used the trimesh's built-in function **trimesh.vertex_neighbors** which returns the indices of the neighbors of the vertex of interest. However, the returned list of indices is not ordered, but to compute angles between vertices, we would like the indices to be ordered in a circular way: that is the goal of our function **choose_best_neighbors**. We compute the angle between each pair of vertices (two neighbors and the vertex of interest), and we can deduce that the smallest angles correspond to vertices that are closest to each other. Doing this iteratively, we obtain an ordered list of indices.

Here are our results (Figures 1 and 2):

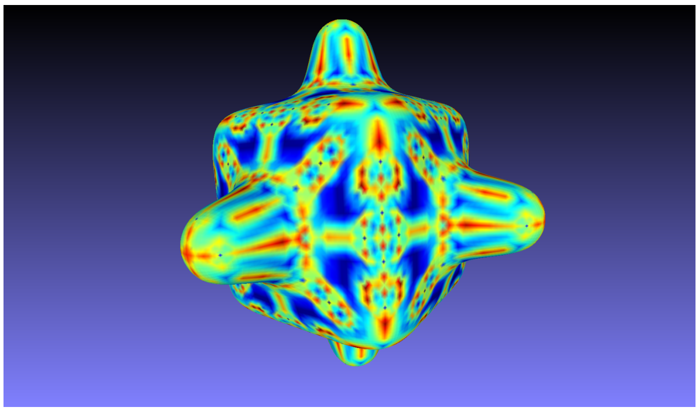


Figure 1: Bumpy cube mesh colored based on the value of its mean curvature, computed using an uniform Laplacian operator

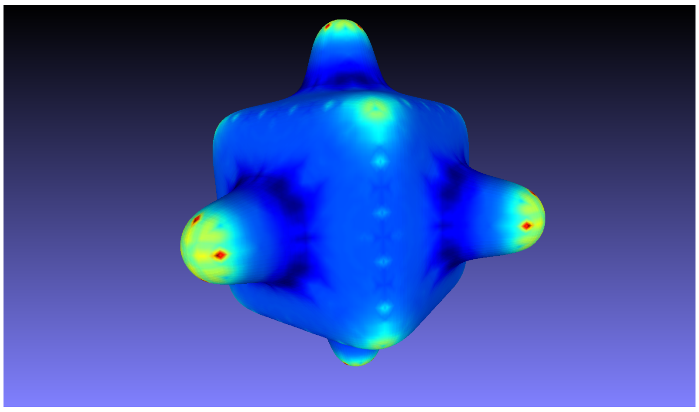


Figure 2: Bumpy cube mesh colored based on the value of its Gaussian curvature

We can see that in both cases (but even more so in the mean curvature one), there are a lot of small areas of multiple colors, which means that in these zones, the curvature must change quickly, which is not the case. It seems that our operator lacks global vision, and only concentrates on smaller patches one after another. Therefore, this is not a good approximation of continuous curvature.
Nevertheless, we cannot say that the results are false: we can clearly see in the mean curvature one that the most curved parts are more red, and in the Gaussian one, the dark-blue shades around the bumps of the cube show the difference of Gaussian curvature as we move along the bump.

2 First and Second Fundamental forms

As a reminder, here are the first and fundamental forms' formulas (found in slides 74 and 75 of lecture 06):

$$\boldsymbol{I} = \begin{pmatrix} E & F \\ F & G \end{pmatrix} = \begin{pmatrix} p_u^T p_u & p_u^T p_v \\ p_u^T p_v & p_v^T p_v \end{pmatrix} \qquad \boldsymbol{II} = \begin{pmatrix} e & f \\ f & g \end{pmatrix} = \begin{pmatrix} p_{uu}^T n & p_{uv}^T n \\ p_{uv}^T n & p_{vv}^T n \end{pmatrix}$$

where p_u and p_v are the first derivatives of p with respect to u and v respectively, p_{uu}, p_{uv} and p_{vv} are the second derivatives of p with respect to u, u and v (symmetrical to v and u), and v respectively, and n is the normal to the curve which can be computed as: $n = \frac{p_u \times p_v}{\|p_u \times p_v\|}$.
Here follow the analytical computations of the different derivatives of p needed to compute the first and second fundamental forms:

$$p(u,v) = \{a \cos(u) \sin(v), b \sin(u) \sin(v), c \cos(v)\}$$
$$p_u = \{-a \sin(u) \sin(v), b \cos(u) \sin(v), 0\}$$
$$p_v = \{a \cos(u) \cos(v), b \sin(u) \cos(v), -c \sin(v)\}$$
$$p_{uu} = \{-a \cos(u) \sin(v), -b \sin(u) \sin(v), 0\}$$

$$p_{vv} = \{-a\cos(u)\sin(v), -b\sin(u)\sin(v), -c\cos(v)\}$$

$$p_{uv} = \{-a\sin(u)\cos(v), b\cos(u)\cos(v), 0\}$$

$$(p_u \times p_v) = \{-bc\cos(u)\sin^2(v), -ac\sin(u)\sin^2(v), -ab\sin(v)\cos(v)\}$$

After that, we have everything to compute the normal curvature (formula found in slide 75 of lecture 06):

$$\kappa_n(\phi) = \frac{e(\cos(\phi))^2 + 2f\cos(\phi)\sin(\phi) + g(\sin(\phi))^2}{E(\cos(\phi))^2 + 2F\cos(\phi)\sin(\phi) + G(\sin(\phi))^2}$$

where ϕ represents a direction on the local tangent plane.
To realize what we are computing, here is what the ellipsoid and the tangent plane look like (Figure 3):

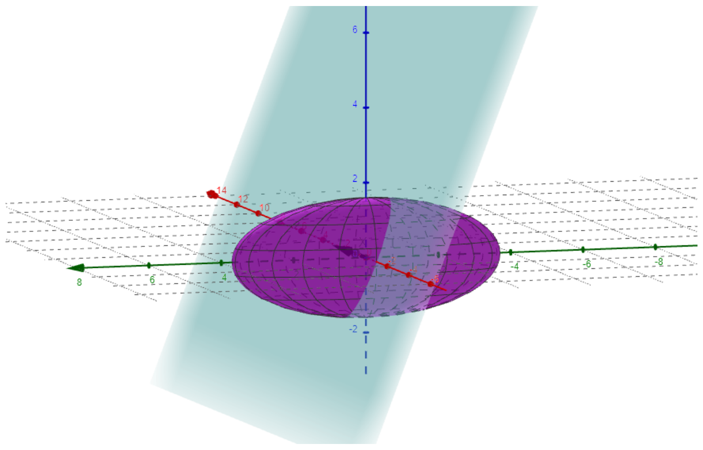
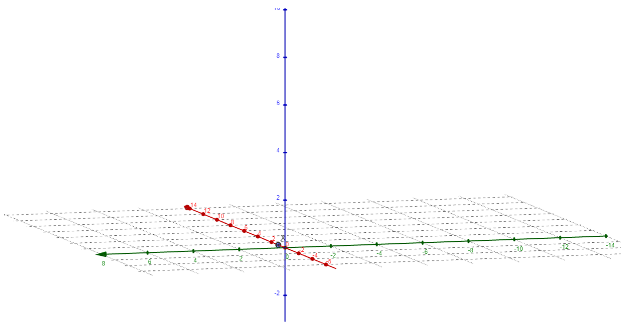
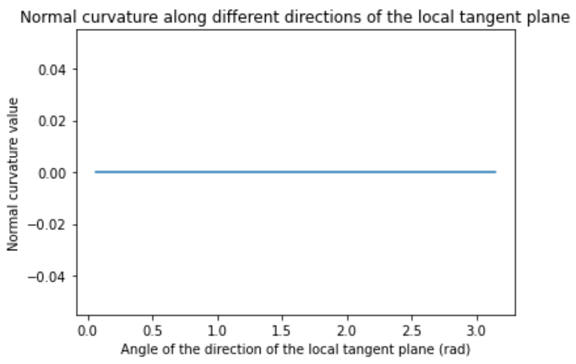


Figure 3: Ellipsoid (purple) and the local tangent plane to one of its points (blue) represented in GeoGebra

Now, here are our results for the normal curvature κ_n for two cases:

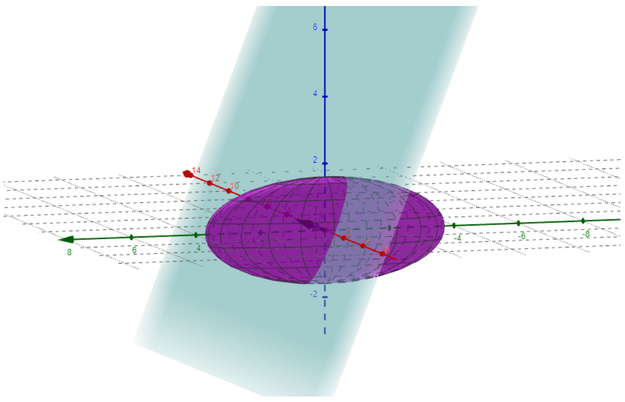


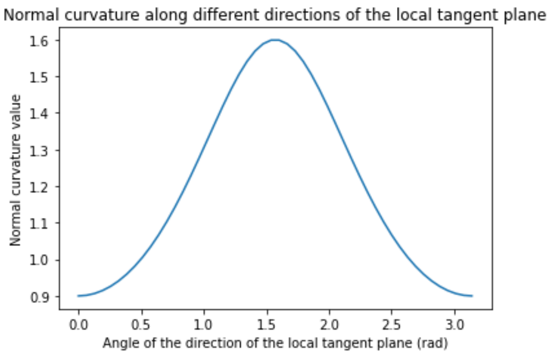
(a) Ellipsoid and its tangent plane (here, flat)



(b) Normal curvature graph

Figure 4: Normal curvature at one of the ellipsoid's point (right), and the visual representation of that ellipsoid for the requested parameters (left)


(a) Ellipsoid and its tangent plane



(b) Normal curvature graph

Figure 5: Normal curvature at one of the ellipsoid's point (right), and the visual representation of that ellipsoid for new parameters (left)

We were asked to compute κ_n at the point $(a, 0, 0)$, but in that case there is no ellipsoid: it is a line (Figure 4a). Therefore, the normal curvature is constant (Figure 4b). Then, we have decided to compute the normal curvature for non-zero values of b and c (Figure 5). We can clearly see a maximum and a minimum to the curve: these are the principal curvatures κ_1 and κ_2.

3 Non-uniform Laplace (Discrete Laplace-Beltrami)

In the computation of the Laplacian Δ_S, what changes from subsection 1 are the factors in the matrix:

$$\Delta_S \boldsymbol{x_i} = \frac{1}{2A(\boldsymbol{x_i})} \sum_{\boldsymbol{x_j} \in N_1(\boldsymbol{x_i})} (\cot \alpha_j + \cot \beta_j)(\boldsymbol{x_j} - \boldsymbol{x_i})$$

where α_j and β_j are angles in the triangles adjacent to the edge between the vertices $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$.
We use trimesh's built-in function **trimesh.triangles.angles** to retrieve these angles.

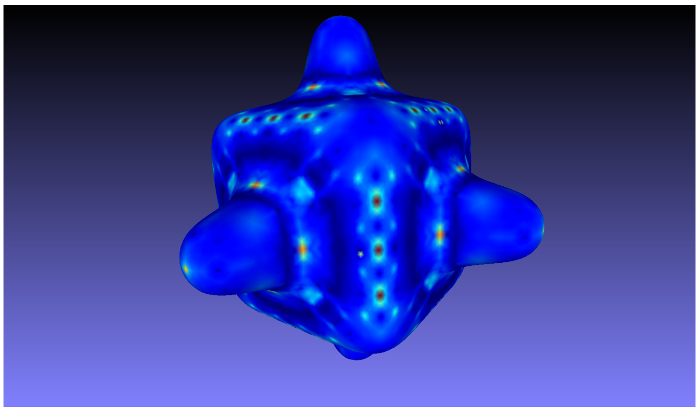
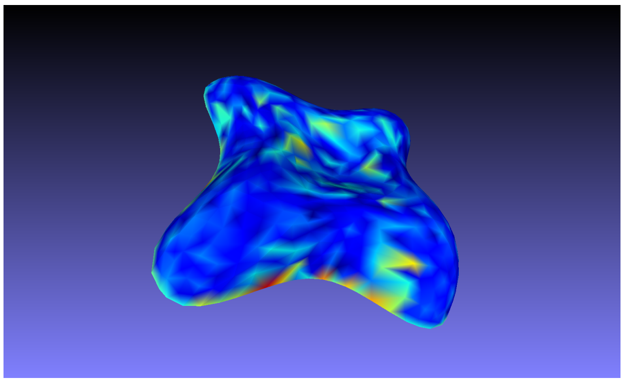
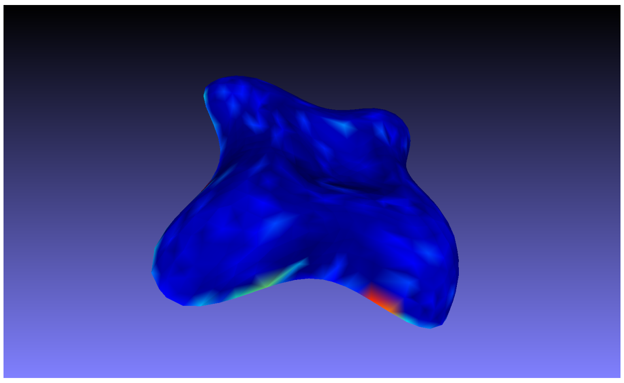


Figure 6: Bumpy cube mesh colored based on the value of its mean curvature, computed using a non-uniform Laplacian operator

The estimation is better if we wanted to have a more "global" estimation of the curvature. We can see that the bump exhibits clear curvatures at the top and at the bottom (Figure 6), whereas in the uniform case, we had curvatures all around it. We still have small areas of curvature, especially on the edges of the cube.
Let's note that we have had to threshold the values obtained in our Laplacian because depending on the area where we are located, the normalization area A can get small, and produce results too high in magnitude compared to the rest.
Let's now compare the uniform and non-uniform case for the requested meshes:

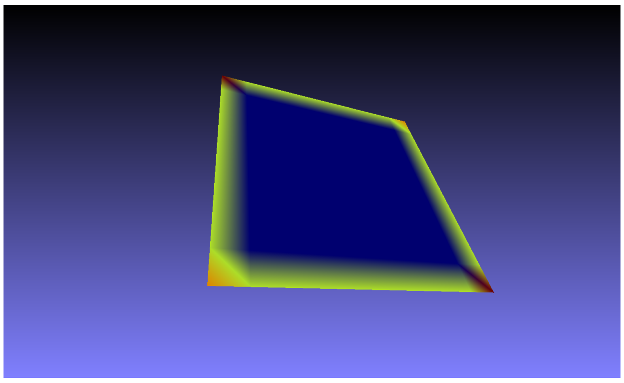


(a) Uniform Laplace-Beltrami operator

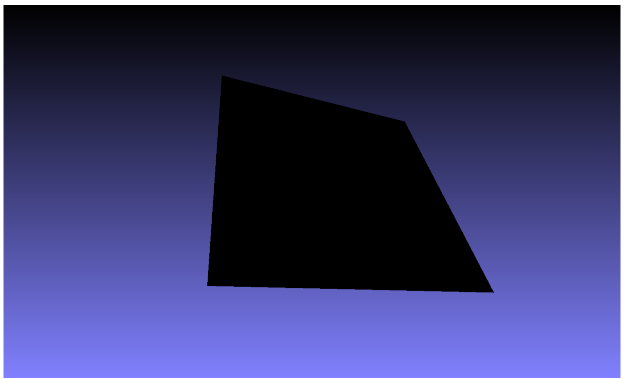


(b) Non-uniform Laplace-Beltrami operator

Figure 7: Comparison of the mean curvature for the lilium mesh for uniform (left) and non-uniform (right) Laplace-Beltrami operators



(a) Uniform Laplace-Beltrami operator



(b) Non-uniform Laplace-Beltrami operator

Figure 8: Comparison of the mean curvature for the plane mesh for uniform (left) and non-uniform (right) Laplace-Beltrami operators

The conclusions for the lilium mesh (Figure 7) are similar to the ones from the bumpy cube mesh: there are less irregularities, it is a better estimation, although we should use a new scaling to be able to see better the little variations.
For the plane mesh (Figure 8), it is the same. We are supposed to have the result on the right (Figure 8b) because the plane is flat, but we clearly see that in the uniform case (Figure 8a) the boundaries of the plane (which are seen as discontinuities) make the neighboring region higher in value, whereas the latter is still flat.
In both cases, we can conclude that the non-uniform version of the Laplace-Beltrami operator gives a better estimation of continuous curvatures.

4 Modal analysis

We use SciPy's built-in function **eigs** to compute the eigenvalues/vectors of our sparse Laplacian matrix.

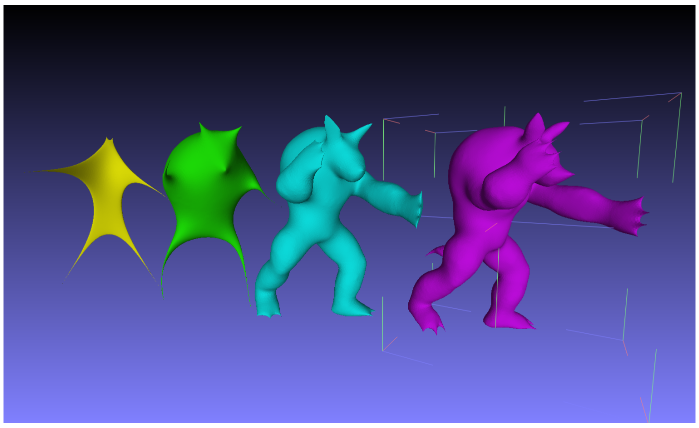


Figure 9: Different Armadillo meshes reconstructed with the 5 (yellow), 15 (green), 150 (blue) and 300 (purple) smallest eigenvectors issued from the uniform Laplacian operator

For low values of k, indeed we only see small parts of the whole mesh (Figure 9). Yet, they are quite fundamental: the low eigenvalues represent the low-frequency parts of the mesh. We can see that by progressively adding more eigenvectors, we can quickly have a sense of the shape of our mesh. In that sense, a larger k gives a better reconstruction.
The step between nothing and low-frequency-only is smaller (and gives visually a better idea of the mesh) than the step between low-frequency-only and low-frequency+high-frequency. We require way more eigenvectors to recover the high-frequency parts, whereas we can already have a good estimate with a correct number of eigenvectors.
This effect is emphasized in very detailed meshes, that as a result have a lot of vertices. It was discussed in the lectures: the bigger the mesh, the less efficient the modal reconstruction. All depends on the application we want to make of this modal analysis.

5 Explicit Laplacian mesh smoothing

The equation we want to solve in the explicit case is the one found in slide 82 of lecture 06:

$$\boldsymbol{X}^{t+1} = (\boldsymbol{I} + \lambda \boldsymbol{L})\boldsymbol{X}^{t}$$

where **X** represents the mesh's vertices, **I** is the identity matrix, **L** is the Laplace-Beltrami operator (not written as Δ_S to emphasize the fact that we are using the matrix representation), t represents the iteration number, and λ is the step size (scalar).
It is not really an equation, but rather a procedure to iterate. We obtain the new vertices by multiplying the current vertices by the matrix comprised of the identity and the Laplace-Beltrami operator.
The results for different iterations and λ values are as follows:

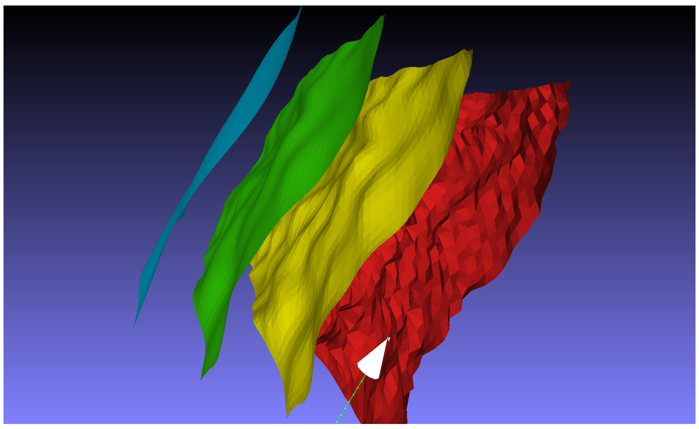


Figure 10: Different plane meshes where explicit smoothing has been performed for $\lambda = 0.05$ and for 500 (yellow), 1000 (green) and 5000 (blue) iterations. The original mesh is in red.

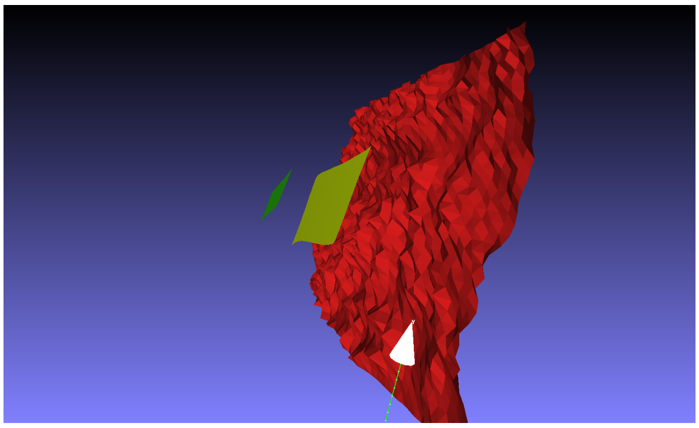


Figure 11: Different plane meshes where explicit smoothing has been performed for $\lambda = 1$ and for 500 (yellow) and 1000 (green) iterations. The original mesh is in red.

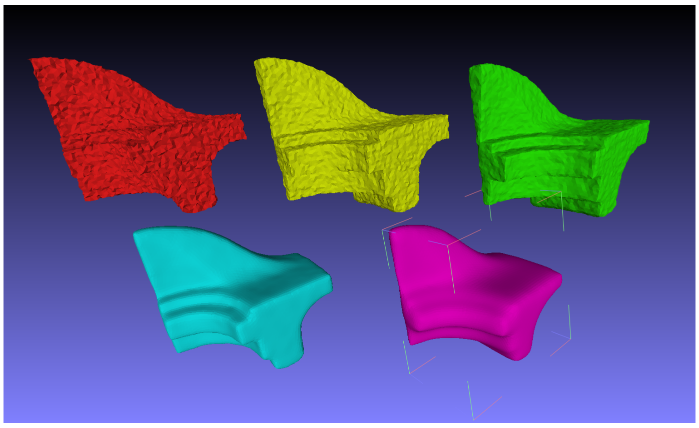


Figure 12: Different fandisk meshes where explicit smoothing has been performed for $\lambda = 0.001$ and for 500 (yellow), 1000 (green), 5000 (blue) and 20000 (purple) iterations. The original mesh is in red.

From what we can see, there is a trade-off to be found between the step size and the number of iterations. In the case where λ is small, we can see that taking more iterations will smooth correctly the noisy mesh (Figure 10).
The problem comes from higher values of λ (Figure 11). Yes, the mesh gets smoother, but it shrinks in size: we are deleting every edge and every detail of the surface. By iterating, we remove the features of the mesh, so that we obtain a surface as smooth as possible, but that does not resemble the original mesh at all. It is missing some kind of boundary conditions to keep the features of the original mesh.
Even for a small λ value, we can see that for the fandisk (Figure 12) the details are also removed. This problem should be fixed by considering a different version of the diffusion flow problem.

6 Implicit Laplacian mesh smoothing

This time, we really have an "equation" (see slide 83 of lecture 06 or equation 9 of Desbrun et al.: Implicit Fairing of Irregular Meshes using Diffusion and Curvature Flow, Siggraph '99):

$$(\boldsymbol{I} - \lambda \boldsymbol{L})\boldsymbol{X}^{t+1} = \boldsymbol{X}^{t}$$

what we mean by that, is that because the matrix on the left-hand side is sparse, inverting it to get an equation of the same form as in subsection 5 will result in a matrix absolutely not sparse, and likely to produce numerical errors (and possibly impossible to store on most computers). Therefore, we interpret this as a linear system of equations. To this effect, we use SciPy's built-in solver **spsolve** for sparse matrices.

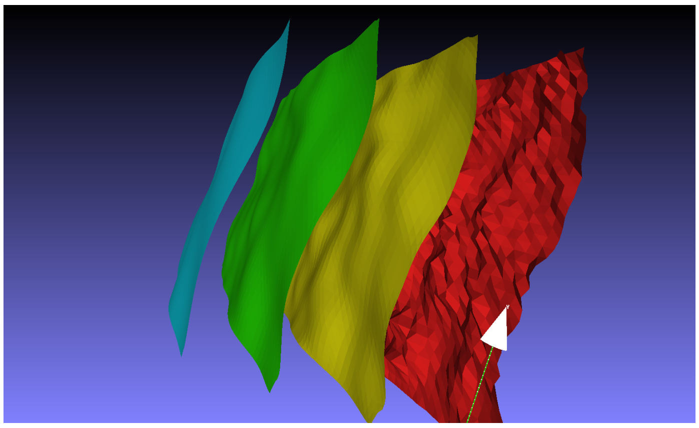


Figure 13: Different plane meshes where implicit smoothing has been performed for $\lambda = 0.01$ and for 500 (yellow), 1000 (green) and 3000 (blue) iterations. The original mesh is in red.

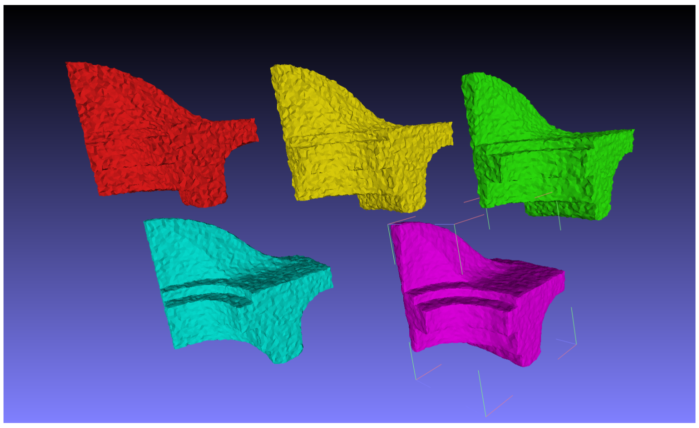


Figure 14: Different fandisk meshes where implicit smoothing has been performed for $\lambda = 0.001$ and for 100 (yellow), 250 (green), 500 (blue) and 1000 (purple) iterations. The original mesh is in red.

The results are similar for the plane (Figure 13), because it has not any notable features (it is flat). Yet, we see a real difference for the fandisk mesh (Figure 14). For similar values of λ (and a smaller number of iterations - this method is more efficient), the features of the mesh have been conserved, all while smoothing the other parts of the mesh. It is because looking at the equation of implicit smoothing, the fact that the matrix is on the left-hand side and that we have a (-) sign, we can interpret this as a gradient condition. Therefore, we can understand that this form of smoothing will be better at keeping the features of a mesh.
Eventually, if we take too high a value for λ, we will start to erase some features, but this is only an extreme case, and it has much less impact than in the explicit case.

7 Laplacian mesh denoising

For this part, we will stick to the implicit smoothing method from the previous subsection. We will add Gaussian noise (of different intensities) to every vertex of the mesh and see how well our smoothing procedure performs. We will use two methods: the first one using the Laplacian of the noisy mesh, and the second one using the Laplacian of the original noiseless mesh. The following figures show our results (15, 16, 17, and 18).

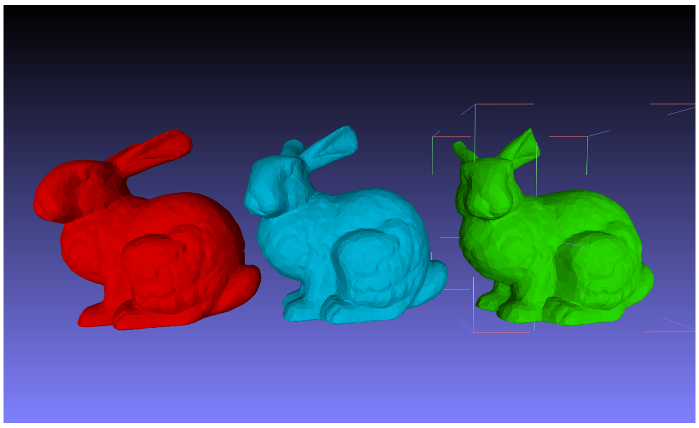


Figure 15: Noisy bunny mesh ($\sigma = 0.001$, red) denoised using the Laplacian from the noisy mesh (blue) and the Laplacian from the noiseless mesh (green).

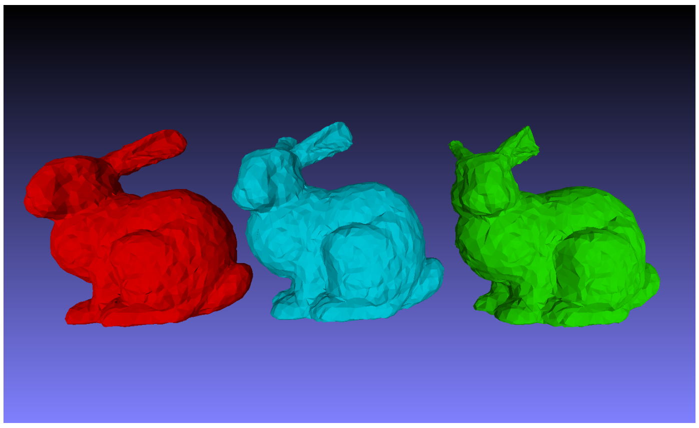


Figure 16: Noisy bunny mesh ($\sigma = 0.005$, red) denoised using the Laplacian from the noisy mesh (blue) and the Laplacian from the noiseless mesh (green).

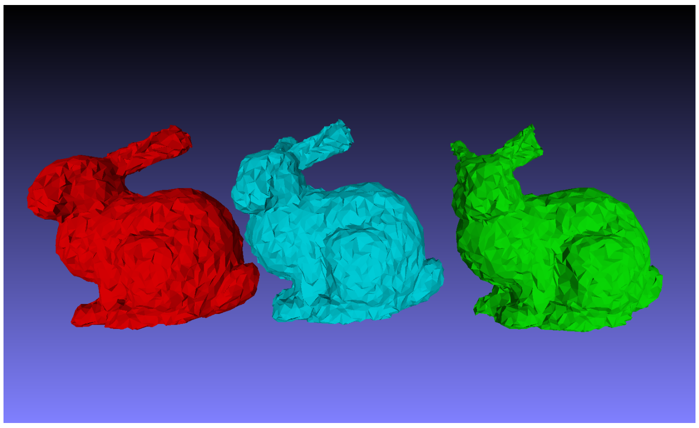


Figure 17: Noisy bunny mesh ($\sigma = 0.01$, red) denoised using the Laplacian from the noisy mesh (blue) and the Laplacian from the noiseless mesh (green).

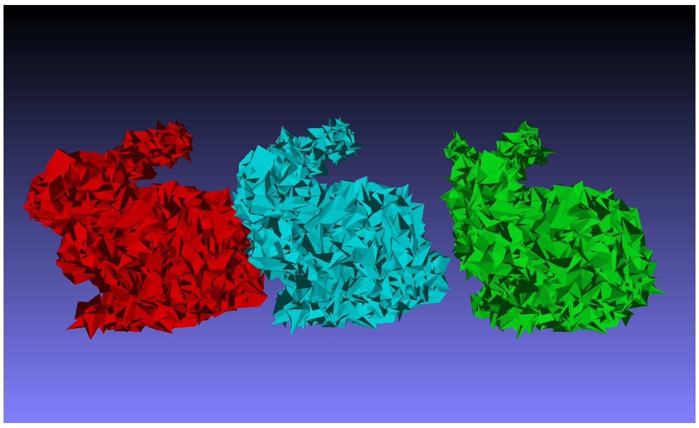


Figure 18: Noisy bunny mesh ($\sigma = 0.05$, red) denoised using the Laplacian from the noisy mesh (blue) and the Laplacian from the noiseless mesh (green).

Unfortunately, there does not seem to be any effect on the noisy meshes. We have tried to use either the implicit or the explicit smoothing functions, we have also tried varying the number of iterations of our smoothing methods. It might come from the fact that there are already too many details in the bunny mesh.
We have found an improvement using the cow_small_manifold2 mesh:

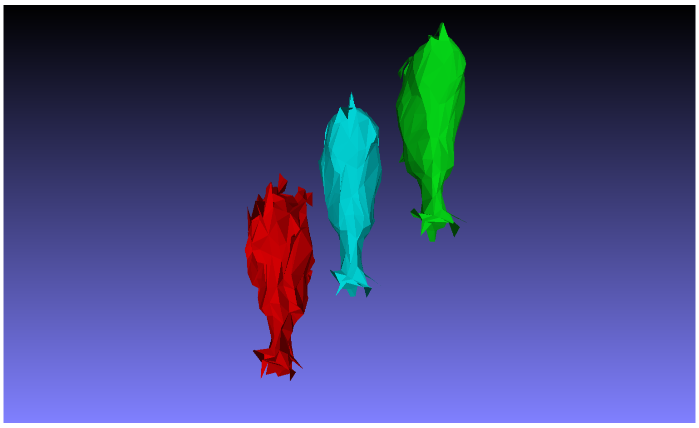


Figure 19: Noisy cow_small_manifold2 mesh ($\sigma = 0.05$, red) denoised using the Laplacian from the noisy mesh (blue) and the Laplacian from the noiseless mesh (green). Top view.

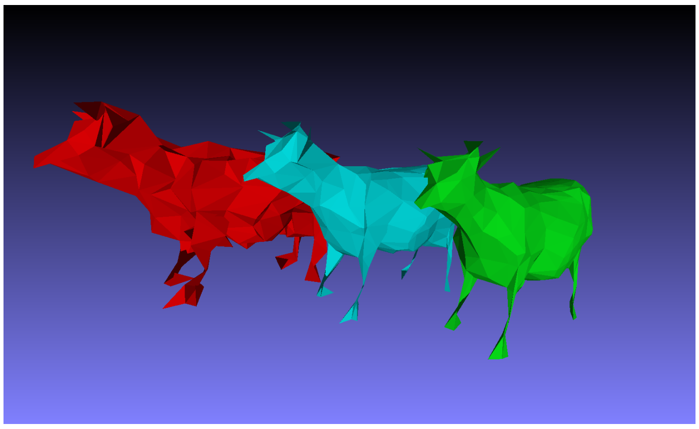


Figure 20: Noisy cow_small_manifold2 mesh ($\sigma = 0.05$, red) denoised using the Laplacian from the noisy mesh (blue) and the Laplacian from the noiseless mesh (green). Side view.

From these views (Figures 19 and 20), we can see that the noisy mesh comes back to a more natural shape. Following our results with the bunny mesh, we can deduce that the denoising operation works only on small meshes, containing only a small amount of details.
We have shown the results for both the Laplacian from the noisy mesh and the Laplacian from the noiseless mesh, because in some cases it is possible that we do not have access to the noiseless data in the first place.

We would therefore need to think about techniques like a Laplacian update after every iteration, or to use simpler meshes to compute the Laplacian to give a rough starting point to our noisy mesh, and then update the Laplacian as we iterate. Here, as the mesh does not contain a lot of vertices and the noise level stays reasonable, both Laplacian techniques work.