Report of Assignment 2

Chengkun Li

March 24, 2021

Abstract

This is the report for the coursework 2 of the course COMP119 of UCL on Acquisition and Processing of 3D Geometry. And the aim of this part is to implement and evaluate Laplacian filtering in the context of mesh denoising.

Discrete Curvature and Spectral Meshes

1. Uniform Laplace

The first task is to estimate Gaussian curvature at each vertex and implement Uniform Laplace operator to compute mean curvature. As for Uniform Laplace, its uniform discretization form function can be written as this way:

$$\Delta_{\text{uni}} f(v_i) \coloneqq \frac{1}{|N_1(v_i)|} \sum_{v_j \in N_1(v_i)} (f(v_j) - f(v_i))$$

where v_i is a vertex, $N_1(v_i)$ is the valence of this vertex which is the number of this vertex's the 1-ring neighbors (Fig.1) and $f(v_i)$ is a general function (it could be the color, heat or the position of the vertex v_i). So, in this place for computing the mean curvature of this mesh, the function converts to

$$\Delta_{\text{uni}}\boldsymbol{x}_i := \frac{1}{|N_1(v_i)|}\sum_{v_j \in N_1(v_i)}(\boldsymbol{x}_j - \boldsymbol{x}_i) \approx -2H\boldsymbol{n}$$

where $\boldsymbol{x}_i$ is the position vector (*x, y, z*) of this vertex, *H* is the mean curvature of vertex *i* and $\boldsymbol{n}$ is its normal vector.

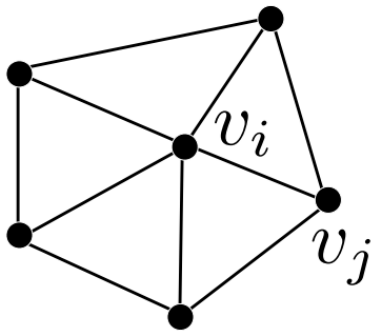


Fig.1 The example of valence of vertex *i*

Hence, we can get the mean curvature

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

For doing this we need to construct a Uniform Laplace matrix, we can build two matrixes, $\boldsymbol{M}$ and $\boldsymbol{C}$.

$$\boldsymbol{M} = diag(\dots, valence(v_i), \dots)$$

$$c_{ij} = \begin{cases} 1 & i \neq j, j \in N_1(v_i) \\ -valence(v_i) & i = j \\ 0 & otherwise \end{cases}$$

Therefore, the Uniform Laplace matrix $\boldsymbol{\Delta_{uni}}$ is

$$\boldsymbol{\Delta_{uni}} = \boldsymbol{M^{-1}C}$$

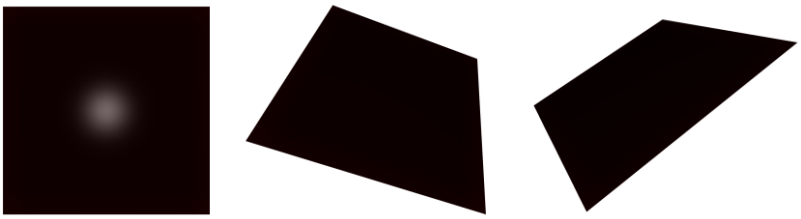


Fig.2 Magnitude of mean curvature by using Uniform Laplace. The first and second row shows the result of "plane.obj" and "lilium_s.obj" respectively.

For testing this part, you can run my code in this way.

```
# Argument '--obj' is used to indicate the 3D object we use,
# and use'--mode mean' is to show the mean curvature.
python part1.py --obj plane.obj --mode mean
python part1.py --obj lilium_s.obj --mode mean
```

As for estimating the Gaussian curvature K at each vertex, we can use the following function.

$$K = \frac{(2\pi - \sum_j \theta_j)}{A}$$

where A is an area normalization factor which is obtained by connecting edge midpoints and triangle barycenters. It is easily computed which is 1/3 sum of its neighbor triangles area. The meaning of θ_j and A could also be seen in Fig.3.

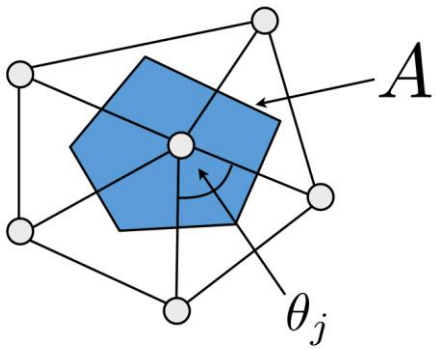


Fig.3 The area normalization factor A and angle θ_j

In Fig.4, I show my experiment results for Gaussian curvature.

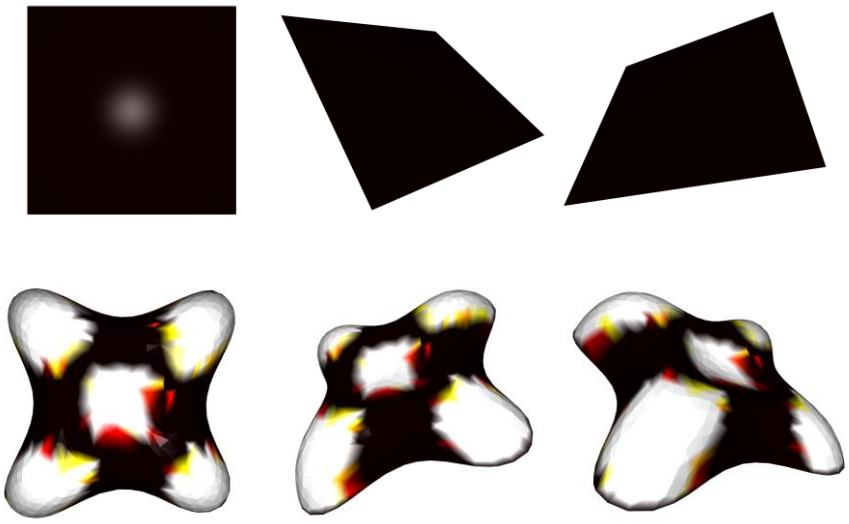


Fig.4 Magnitude of Gaussian curvature by using Uniform Laplace. The first and second row shows the result of "plane.obj" and "lilium_s.obj" respectively.

The code for running of this part is:

```
# Argument '--mode gaussian' is to show the mean curvature.
python part1.py --obj plane.obj --mode gaussian
python part1.py --obj lilium_s.obj --mode gaussian
```

In summary, for the plane, the mean and Gaussian curvature both are 0 at each vertex. However, for lilium_s, the texture is a little mottled that some area's color are more depth and some are less, but their colors are similar. So, I think it is not so good for approximation of continuous curvature. In terms of Gaussian curvature of lilium_s, the result seems good. For saddle part, the Gaussian curvature is negative where its color is dark red, and then the color of the ridge part is lighter whose gaussian curvature is positive, which is the same as our prediction. Thus, I'd like to say it is a good approximation of continuous curvature for Gaussian curvature.

(Worth to mention, I just set the mean and Gaussian curvature as zero for the boundary vertices.)

If you want to see the code of implementation of Gaussian curvature and Uniform Laplacian mean curvature, please find 'LaplaceOperator.py' and read the code of the function 'getMeanCurvature ()' and the function 'getGaussianCurvature' of the class 'LaplaceOperator'.

2. First and Second Fundamental forms:

We have known this surface's parameterization.

$$p(u,v) = \begin{pmatrix} aco\,s(u)\ sin(v) \\ b\ sin(u)\ sin(v) \\ c\ cos(v) \end{pmatrix}$$

Where $u \in [0,2\pi)$ and $v \in [0,2\pi)$, Computing the partial derivative of the mesh, we can get two tangent vectors $p_u(u,v)$ and $p_v(u,v)$.

$$p_u(u,v) = \frac{\partial p}{\partial u} = \begin{pmatrix} -a\ sin(u)\ sin(v) \\ b\ cos(u)\ sin(v) \\ 0 \end{pmatrix}, p_v(u,v) = \frac{\partial p}{\partial v} = \begin{pmatrix} a\ cos(u)\ cos(v) \\ b\ sin(u)\ cos(v) \\ -c\ \sin(v) \end{pmatrix}$$

So, we get the First fundamental form,

$$\mathrm{I} = \begin{pmatrix} E & F \\ F & G \end{pmatrix} := \begin{pmatrix} p_u^T p_u & p_u^T p_v \\ p_u^T p_v & p_v^T p_v \end{pmatrix}$$

$$\mathrm{I} = \begin{pmatrix} (a^2 \sin^2 u + b^2 \cos^2 u)\sin^2 v & \frac{b^2 - a^2}{4}\sin(2u)\sin(2v) \\ \frac{b^2 - a^2}{4}\sin(2u)\sin(2v) & (a^2 \cos^2 u + b^2 \sin^2 u)\cos^2 v + c^2 \sin^2 v \end{pmatrix}$$

The second fundamental form is

$$\mathrm{II} = \begin{pmatrix} e & f \\ f & g \end{pmatrix} := \begin{pmatrix} p_{uu}^T \boldsymbol{n} & p_{uv}^T \boldsymbol{n} \\ p_{uv}^T \boldsymbol{n} & p_{vv}^T \boldsymbol{n} \end{pmatrix}$$

where

$$p_{uu}(u,v) = \frac{\partial p_u}{\partial u} = \begin{pmatrix} -a\ cos(u)\ sin(v) \\ -b\ sin(u)\ sin(v) \\ 0 \end{pmatrix}$$

$$p_{uv}(u,v) = \frac{\partial p_u}{\partial v} = \begin{pmatrix} -a\ sin(u)\ cos(v) \\ b\ cos(u)\ cos(v) \\ 0 \end{pmatrix}$$

$$p_{vv}(u,v) = \frac{\partial p_v}{\partial v} = \begin{pmatrix} -a\ cos(u)\ sin(v) \\ -b\ sin(u)\ sin(v) \\ -c\ \cos(v) \end{pmatrix}$$

$\boldsymbol{n}$ is the normal vector of the vertex.

$$\boldsymbol{n} = \frac{p_u \times p_v}{\|p_u \times p_v\|} = \frac{1}{\|p_u \times p_v\|}\begin{pmatrix} -bc\cos(u)\sin^2 v \\ -ac\ sin(u)\sin^2 v \\ -\frac{ab}{2}\sin(2v) \end{pmatrix}$$

$$\|p_u \times p_v\| = \sqrt{b^2c^2\cos^2 u \sin^4 v + a^2c^2 \sin^2 u \sin^4 v + \frac{a^2b^2}{2}\sin^2 2v}$$

The normal curvature can be computed as

$$\kappa_n(\bar{\boldsymbol{t}}) = \frac{\bar{\boldsymbol{t}}^{\boldsymbol{T}} \text{II}\, \bar{\boldsymbol{t}}}{\bar{\boldsymbol{t}}^{\boldsymbol{T}} \text{I}\, \bar{\boldsymbol{t}}} = \frac{ea^2 + 2fab + gb^2}{Ea^2 + 2Fab + Gb^2}$$

Where

$$\boldsymbol{t} = mp_u + np_v = \frac{cos\phi}{\|p_u\|}p_u + \frac{sin\phi}{\|p_v\|}p_v$$

$$\bar{\boldsymbol{t}} = (m, n) = (\frac{cos\phi}{\|p_u\|}, \frac{sin\phi}{\|p_v\|})$$

For the point (*a, 0, 0*), it is easy to know $v = \frac{\pi}{2}$ and $u = 0$. Put them into the functions above and there are

$$E = p_u^T p_u = a^2$$
$$F = p_u^T p_v = 0$$
$$G = p_v^T p_v = c^2$$
$$e = p_{uu}^T \boldsymbol{n} = abc$$
$$f = p_{uv}^T \boldsymbol{n} = 0$$
$$g = p_{vv}^T \boldsymbol{n} = abc$$
$$\bar{\boldsymbol{t}} = (m, n) = (\frac{cos\phi}{|bc|}, \frac{sin\phi}{|bc|})$$

So, the vertex $(a, 0,0)$'s normal curvature is

$$\kappa_n(\phi) = \frac{abc}{a^2 \cos^2\phi + b^2 \sin^2\phi} \quad , where\ \phi \in [0,2\pi)$$

3. Non-uniform (Discrete Laplace-Beltrami):

In this part, I repeat what I do in question 1, but this time I use Cotangent Laplace to compute mean curvature, which is

$$\Delta_S f(v_i) \coloneqq \frac{1}{2A(v_i)} \sum_{v_j \in N_1(v_i)} (cot\alpha_{ij} + cot\beta_{ij})\big(f(v_j) - f(v_i)\big)$$

where the representations of α_{ij}, β_{ij} and A_{ij} are shown in the Fig.5. And here, $A(v_i)$ is the same with above.

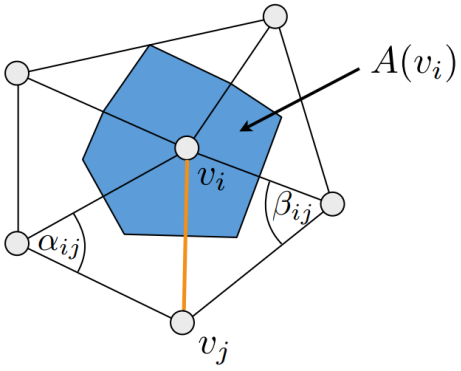


Fig.5 The interpretation of α_{ij}, β_{ij} and A_{ij} in Cotangent Laplace function

The same with before, Cotangent Laplace $\boldsymbol{\Delta_S}$ can also be represented by matrix $\boldsymbol{M}$ and $\boldsymbol{C}$.

$$\boldsymbol{M} = diag(\dots, A(v_i), \dots)$$

$$\boldsymbol{c}_{ij} = \begin{cases} \frac{1}{2}(cot(\alpha_{ij}) + \cot(\beta_{ij})) & i \neq j, j \in N_1(v_i) \\ -\sum_{j \in N_1(v_i)} c_{ij} & i = j \\ 0 & otherwise \end{cases}$$

$$\boldsymbol{\Delta_S} = \mathbf{M^{-1}C}$$

And also,

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

The experiment results are shown in Fig.6.

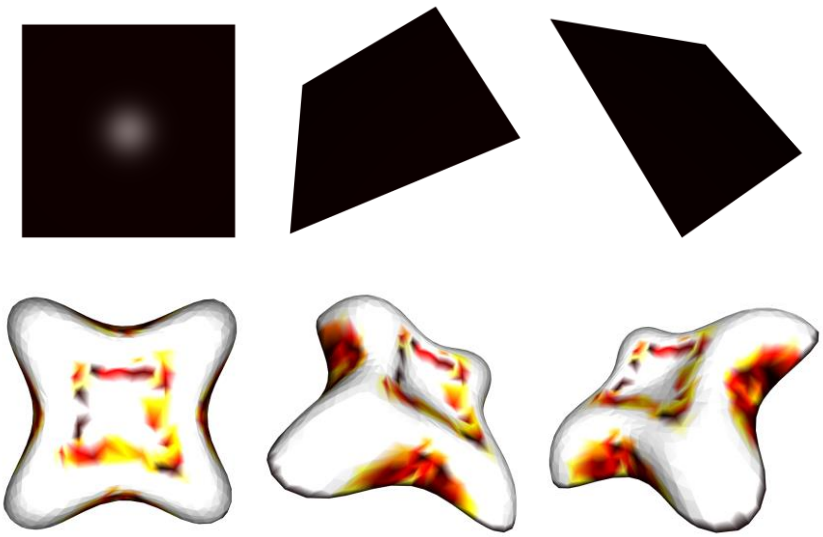


Fig.6 Magnitude of mean curvature calculated by Cotangent Laplace

For doing this, run:

```
# There is only one argument '--obj' here
# to indicate the 3D object which we use.
python part3.py --obj plane.obj
python part3.py --obj lilium_s.obj
```

As a result, the performance of Cotangent Laplace is better, because of taking account into the shape of mesh triangles. For the plane, the results of Uniform and Cotangent Laplace are the same. But for the lilium_s, Cotangent Laplace's result is more correct and satisfies our prediction.

If you want to see the code of implementation of Cotangent Laplacian mean curvature, please find 'LaplaceOperator.py' and read the code of the function 'getMeanCurvature ()' of the class 'LaplaceOperator'.

4. Spectral Analysis

There are three steps to do spectral analysis. Firstly, you should to setup Laplace-Beltrami matrix Δ. And then, compute k smallest eigenvectors $\{\boldsymbol{e}_1, \dots, \boldsymbol{e}_k\}$. Finally, reconstruct the mesh by using these eigenvectors.

For compute Laplace matrix, we can use function $\boldsymbol{\Delta} = \boldsymbol{M^{-1}C}$ to get. Here, we use Cotangent Laplace which we have done in the previous task.

In terms of the eigenvectors, we solve the function.

$$\boldsymbol{\Delta e}_i = \lambda_i \boldsymbol{e}_i$$

However, the Laplace matrix is not symmetric ($\Delta \neq \Delta^{\mathbf{T}}$). Hence, we do as following.

$$\boldsymbol{M^{-1}Ce}_i = \lambda_i \boldsymbol{e}_i$$
$$\left(\boldsymbol{M^{-\frac{1}{2}}CM^{-\frac{1}{2}}}\right)\boldsymbol{M^{\frac{1}{2}}e}_i = \lambda_i \boldsymbol{M^{\frac{1}{2}}e}_i$$
$$\tilde{\Delta}\boldsymbol{y}_i = \lambda_i \boldsymbol{y}_i$$

Where $\tilde{\Delta} = \boldsymbol{M^{-\frac{1}{2}}CM^{-\frac{1}{2}}}$ and $\boldsymbol{y}_i = \boldsymbol{M^{\frac{1}{2}}e}_i$. $\tilde{\Delta}$ is symmetric and $\{\mathbf{y}_i\}$ will be orthogonal. So, for $\{\mathbf{e}_i\}$ we try to do solve these:

$$\boldsymbol{y}_i = eigenvectors(\tilde{\Delta})$$
$$\boldsymbol{e}_i = M^{-1/2}\boldsymbol{y}_i$$

And then, basing on the absolute value of $\{\lambda_i\}$ to find the k smallest eigenvectors $\{\boldsymbol{e}_1, \dots, \boldsymbol{e}_k\}$.

The finally step is to reconstruct. Although $\{\mathbf{y}_i\}$ are orthogonal, $\{\boldsymbol{e}_i\}$ are not. Because

$$< \boldsymbol{e}_i, \boldsymbol{e}_j >= \boldsymbol{e}_i^T \boldsymbol{e}_j = \boldsymbol{y}_i^T \boldsymbol{M}^{-1} \boldsymbol{y}_j \neq \boldsymbol{y}_i^T \boldsymbol{y}_j$$

Hence, we redefine the inner product as

$$< \boldsymbol{e}_i, \boldsymbol{e}_j >= \boldsymbol{e}_i^T \boldsymbol{M} \boldsymbol{e}_j$$

So, the projecting vertex position vector $\boldsymbol{x} \coloneqq (x_1, \dots, x_n)^T$ becomes (the same for y-axis and z-axis)

$$\tilde{\boldsymbol{x}} = \sum_{i=1}^{k} < \boldsymbol{x}, \boldsymbol{e}_i > \boldsymbol{e}_i = \sum_{i=1}^{k} (\boldsymbol{x}^T \boldsymbol{M} \boldsymbol{e}_i) \boldsymbol{e}_i$$

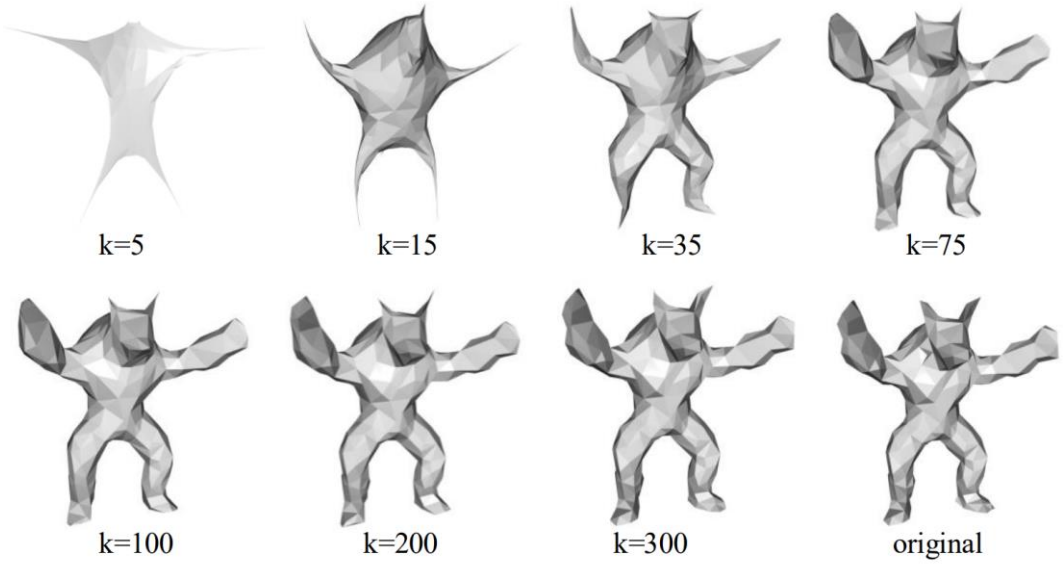


Fig.7 Spectral analysis reconstruction by different number of eigenvectors

Fig.7 shows the reconstructed results by several different numbers of eigenvectors. For running this part:

```
1.  # Use '--k' to set the number of eigenvectors
2.  pyhon part4.py --k 60
```

In conclusion, the idea for spectral analysis is to decompose the mesh in principle components space. Just like what we do in Fourier transform, it is to decompose an arbitrary signal into a sequence of periodic signals with different frequency. In spectral analysis of the 3D mesh, the eigenvector with smaller eigenvalue represents the low frequency feature of the mesh. From Fig.7, we notice the bigger k we use, the more detail the reconstructed mesh will have. Even if we set $k = N$ (N is the number of all the vertices), the reconstructed will be the same with the original.

If you want to see the code of spectral analysis, please find 'LaplaceOperator.py' file and read

the code of the function 'decompose ()' of the class 'LaplaceOperator'.

Laplacian Mesh Smoothing

5. Implement explicit Laplacian mesh smoothing

In this part, we are doing explicit Laplacian mesh smoothing. Exactly, we iteratively move each vertex to its negative normal vector's direction. For each vertex $\boldsymbol{p}_i$, we solve the function below to get the new position of the next iteration.

$$\boldsymbol{p}_{\mathrm{i}}^{t+1} = \boldsymbol{p}_i^t + \lambda \Delta \boldsymbol{p}_i^t$$

Where λ is step size for every iteration. Also, we can rewrite this function in matrix notation.

$$\boldsymbol{P}^{t+1} = (\boldsymbol{I} + \lambda \Delta)\boldsymbol{P}^t$$

Where $\boldsymbol{P}^t = (\boldsymbol{p}_1^t, \dots, \boldsymbol{p}_n^t)^T$.

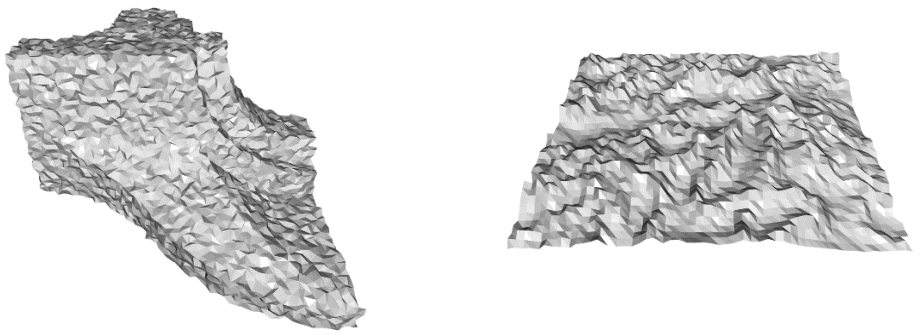
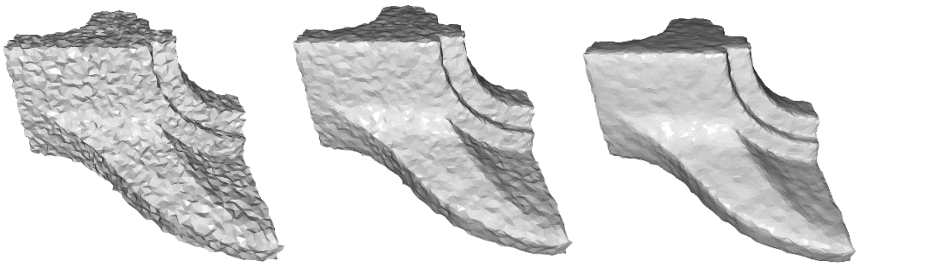
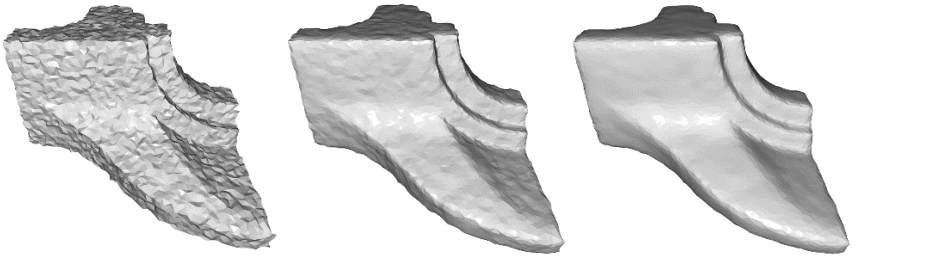


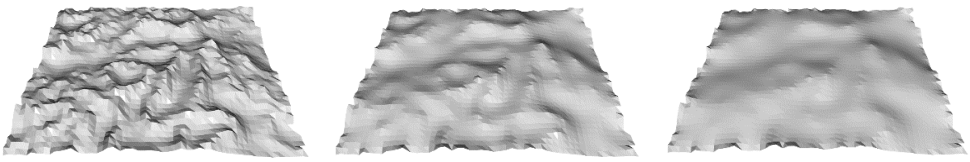
Fig.8 Original 3D objects that need smoothing. Left: fandisk. Right: plane



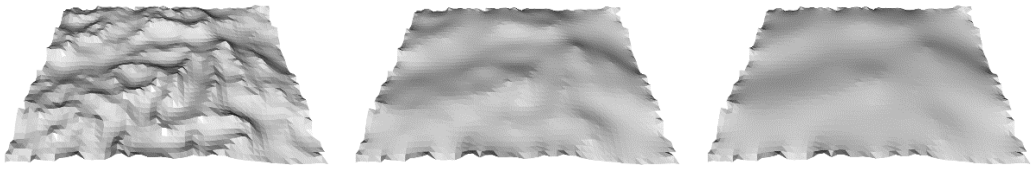
(a) λ=1e-5, iter=50 (b) λ=5e-5, iter=50 (c) λ=1e-4, iter=50



(d) λ=1e-5, iter=100 (e) λ=5e-5, iter=100 (f) λ=1e-4, iter=100

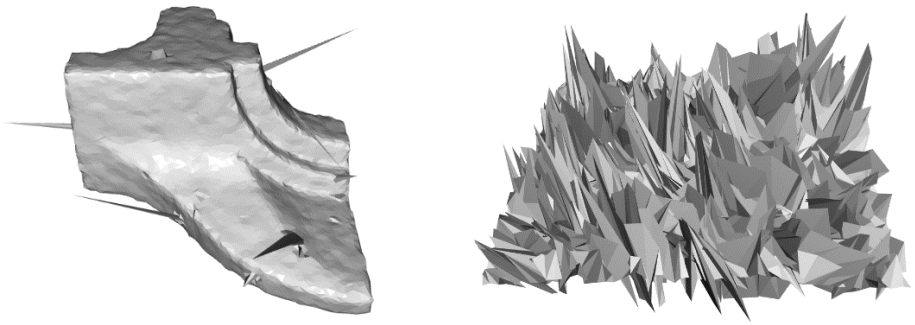


(g) λ=1e-7, iter=50 (h) λ=5e-7, iter=50 (i) λ=1e-6, iter=50



(j) λ=1e-7, iter=100 (k) λ=5e-7, iter=100 (l) λ=1e-6, iter=100

Fig.9 Explicit Laplacian mesh smoothing



λ=1e-3, iter=5 λ=1e-5, iter=5

Fig.10 Explicit Laplacian mesh smoothing with large step size

In my experiment, step size λ play an important role in mesh smoothing. Fig.9 has shown the results with different step size and iteration. It seems $\lambda = 10^{-4}$ is a good value for "fandisk" mesh, but for "plane" mesh $\lambda = 10^{-6}$ is much better. That is to say that the choice of the good value of λ depends on the object itself. Apart from this, the more iteration, the more smooth it will be. But if the iteration number is too big, the smoothed mesh will lose some details. Besides, I select some very large value of λ to test. However, there are some spines in the results (Fig.10), and the smoothing does not work in this case.

For testing,

```
# '--step_size' is to set the step size lambda of iteration
# '--iteration' is to set the number of iteration
python part5.py --obj fandisk_ns.obj --step_size 1e-5 --iteration 50
```

If you want to see the code of implementation of explicit Laplacian mesh smoothing, please find 'LaplaceOperator.py' and read the code of function 'smooth()' of class 'LaplaceOperator'.

6. Implement implicit Laplacian mesh smoothing

For this task, we use the implicit Laplace method to smooth mesh, and it is unconditionally stable. It can be written as

$$\boldsymbol{p}_{\mathrm{i}}^{t+1} = \boldsymbol{p}_i^t + \lambda \Delta \boldsymbol{p}_i^{t+1}$$

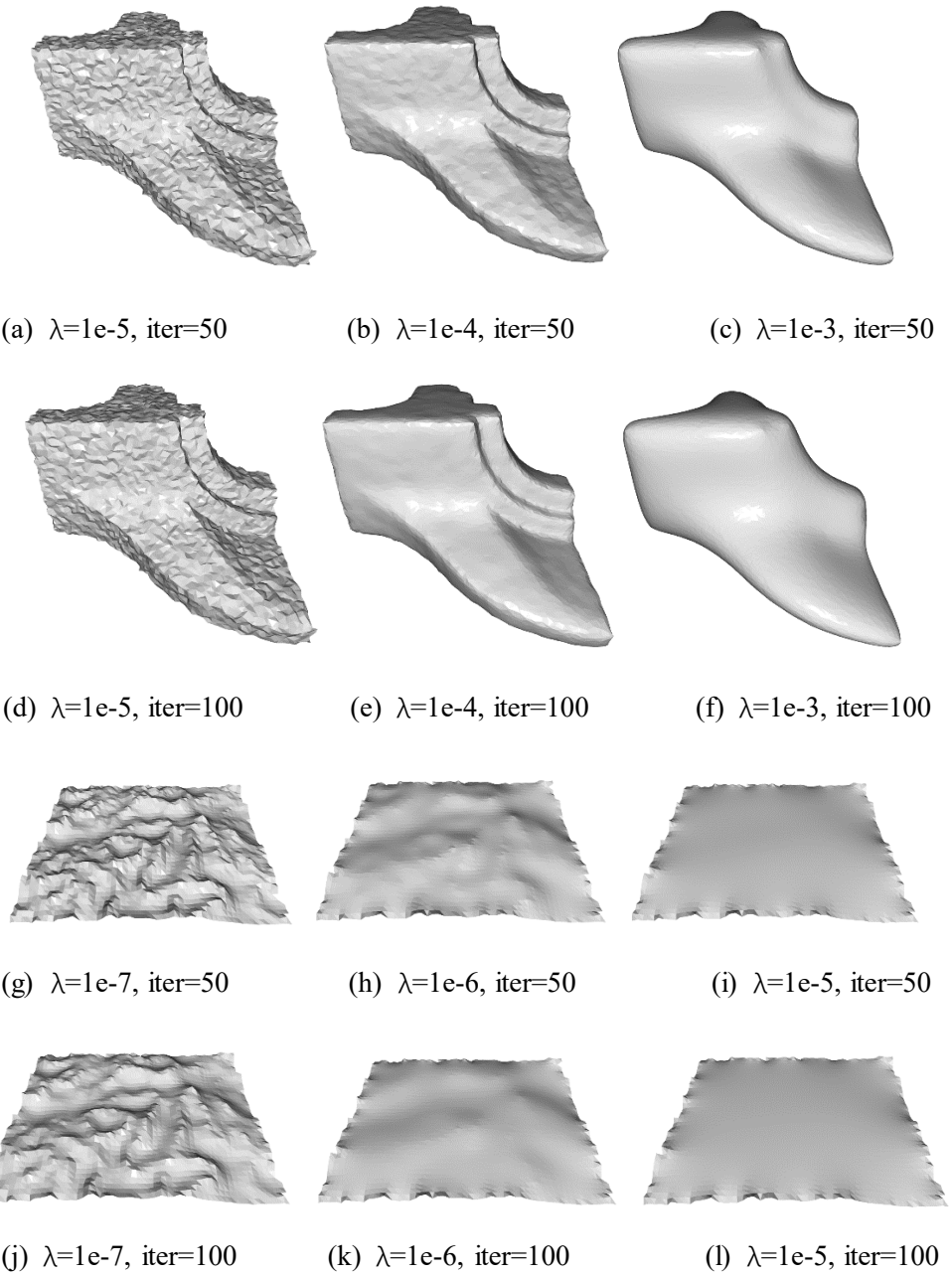
For matrix notation,

$$(\boldsymbol{I} - \lambda \Delta)\boldsymbol{P}^{t+1} = \boldsymbol{P}^t$$

$$\left(\boldsymbol{I} - \lambda \boldsymbol{M}^{-1}\boldsymbol{C}\right)\boldsymbol{P}^{t+1} = \boldsymbol{P}^t$$

$$(\boldsymbol{M} - \lambda \boldsymbol{C})\boldsymbol{P}^{t+1} = M\boldsymbol{P}^{t}$$

Finally, solve this sparse linear function.



(a) λ=1e-5, iter=50 (b) λ=1e-4, iter=50 (c) λ=1e-3, iter=50

(d) λ=1e-5, iter=100 (e) λ=1e-4, iter=100 (f) λ=1e-3, iter=100

(g) λ=1e-7, iter=50 (h) λ=1e-6, iter=50 (i) λ=1e-5, iter=50

(j) λ=1e-7, iter=100 (k) λ=1e-6, iter=100 (l) λ=1e-5, iter=100

Fig.11 Implicit Laplacian mesh smoothing

Comparing Fig.9 and Fig.11, we notice that there is almost no difference between explicit and implicit smoothing method when the value of λ is in a suitable range (See the first column of Fig.9 and Fig.11, or the third column of Fig.9 with the second column of Fig.11). The implicit method is more stable and we can even use very large λ to smooth without error. However, the explicit method is invalid in this case (Compare Fig.10 and the third column of Fig.11). Besides, we notice that the more iteration the more smooth it will be, but will cause some details

lost.

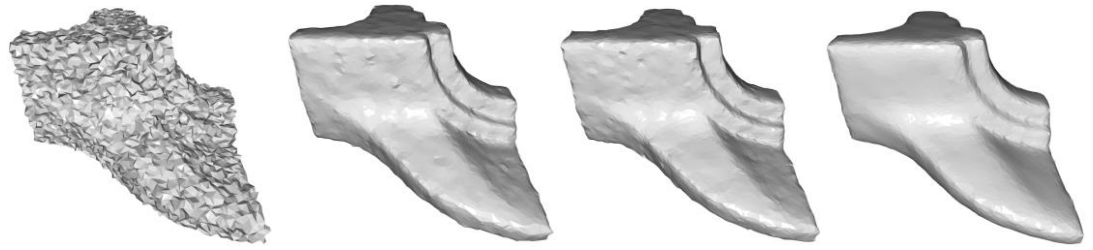
For running this part:

```
# '--step_size' is to set the step size lambda of iteration
# '--iteration' is to set the number of iteration
python part6.py --obj fandisk_ns.obj --step_size 1e-5 --iteration 50
```

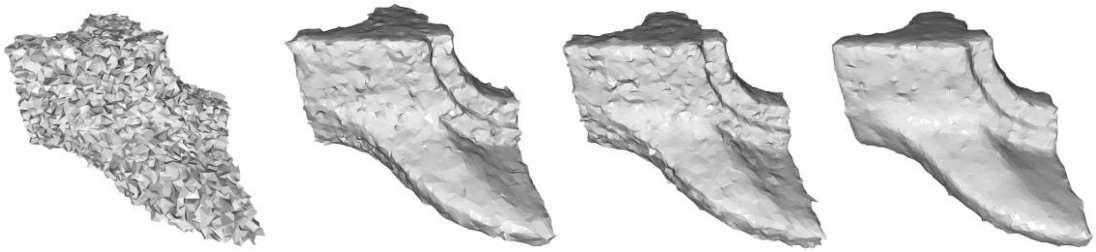
If you want to see the code of implementation of implicit Laplacian smoothing, please find 'LaplaceOperator.py' and read the code of the function 'smooth()' of class 'LaplaceOperator'.

7. Evaluate the performance of Laplacian mesh denoising

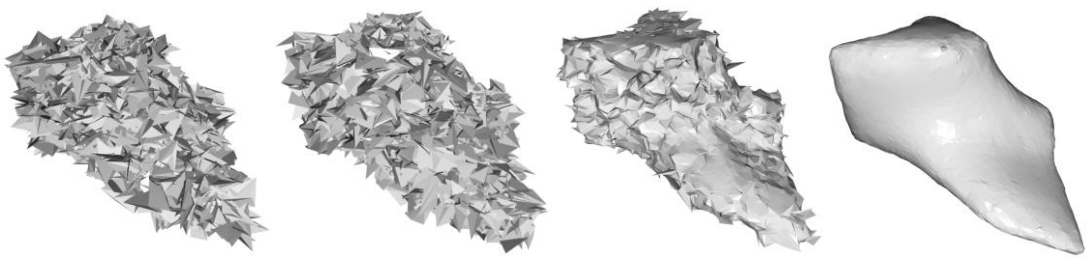
For evaluating the performance of Laplacian mesh denoising, I add Gaussian noise to perturb vertices of mesh. And the noise's scale is proportional with the bounding box.



(a) noise=0.005 (a1) λ=5e-5, iter=200 (a2) λ=1e-4, iter=100 (a3) λ=1e-4, iter=200



(b) noise=0.010 (b1) λ=5e-5, iter=200 (b2) λ=1e-4, iter=100 (b3) λ=1e-4, iter=200



(c) noise=0.050 (c1) λ=1e-4, iter=200 (c2) λ=1e-3, iter=200 (c3) λ=1e-2, iter=200

Fig.12 Denoising mesh using implicit Laplace

And then, I do implicit Laplacian mesh smoothing just like what we do in task 6 with different step size and iteration number. You can run this part as:

```
# Eg. we add a Gaussian noise
# whose standard variance is 0.05 and mean is zero.
# For smoothing, I set step size as 1e-3 for every iteration,
# and the total iteration is 200
python part7.py --obj fandisk_ns.obj --noise 0.05 \
--step_size 1e-3 --iteration 200
```

In conclusion, the final smoothing result depends on λ and iteration's number. With the same iteration, the bigger λ is, the more smooth the result will be (Fig.12 (a1) with (a3), or (b1) with (b3)). If λ is the same, the noise will reduce more with bigger iteration (Fig.12 (a2) with (a3), or (b2) with (b3)). Besides, λ makes the same influence on the denoising result with the number of iteration. Seeing (b1) and (b2) of Fig.12 (Also, (a1) and (a2) are fine), the value of λ of (b1) is only half of the (b2), while the iteration is twice the value of (b2), however these two results are similar and reduce the noise into a same level.

If the noise is not so big, the smoothing works well (The first and second row of Fig.12 show this). However, when the noise is much large. it does not work (showing at third row of Fig.12). Although the mesh is smoothed (if we use big step size and iteration), the details also lose and only the basic shape is saved.