Acquisition and Processing of 3D Geometry Coursework 2

Hugo Germain

March 26^{th} 2017

In this Coursework, we will first be computing Mean and Gaussian Curvatures on 3D meshes. We will start with a uniform discretization, and then use the cotangent discretization to see how the results improve. Secondly, we will be reusing our Laplace operator to perform mesh smoothing, both explicitly and implicitly.

Like in the previous Coursework, we will base our program on the provided IGL Framework and extend the C++ files to implement our core functions. In particular, you will find the essential functions under *decoratedCloud.cpp* and *main.cpp*.

Here is the list of the covered questions :

1. **Uniform Laplace :** The Mean Curvature and Gaussian Curvature algorithms were both implemented. The area computation was chosen as $\frac{1}{3}$ of the total neighbouring faces area.
2. **Non-Uniform Laplace :** The Discrete Laplace-Beltrami operator was implemented, using a cotangent discretization
3. **Explicit Laplacian Smoothing** was implemented as well, and performs reasonably well for small λ steps.
4. **Implicit Laplacian Smoothing** was implemented, and provides satisfying results even for a large amount of noise and larger λ steps.

Discrete Curvature

1 Uniform Laplace

We begin our Coursework by estimating the Mean and Gaussian Curvatures for a 3D Mesh. As done in the previous Coursework, we use the provided IGL-Framework as a starting point, and work on `.OFF` meshes.

As a reminder, the Mean Curvature H and the Gaussian Curvature K can be defined as such :

$$H = \frac{\kappa_1 + \kappa_2}{2} \ , \quad K = \kappa_1 . \kappa_2$$

where $\kappa_1 = max_\phi \kappa_n(\phi)$, $\kappa_2 = min_\phi \kappa_n(\phi)$.

Alternatively, by deriving from the Laplace Operator functions, we can express those as

$$-2H\boldsymbol{n} = \boldsymbol{\Delta_s x} \ , \quad K = \frac{2\pi - \sum_j \theta_j}{A}$$

These are the expressions we will be using. To color code the mesh vertices, we will be using the `igl::jet` method, which provides a color range from blue to red while also normalizing the input vectors.

Mean Curvature

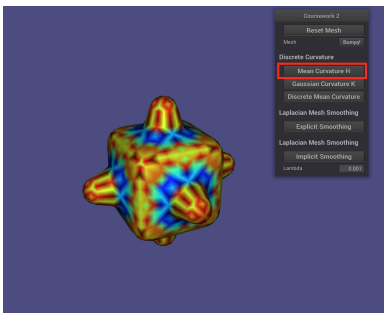


FIGURE 1 – Mean Curvature on *bumpy.off*

To compute the mean curvature, we first need to construct the Laplace Operator. It consists of a sparse matrix, embodying the distance between a vertex and the average of its neighbours : $\frac{1}{|N_1(v_i)|} \sum_{v_j \in N_1(v_i)} (\boldsymbol{x_j} - \boldsymbol{x_i})$

As a result, we first need to **find all the neighbours** of the all the vertices in the mesh. To do so, we browse iteratively through all the vertices, find their appearances in the face matrix, and store their neighbours.

Then, for each row in the Laplace sparse matrix (ie. each vertex in the mesh), we **insert corresponding weights**. That is, either -1 on the diagonal, or $\frac{1}{|N_1(v_i)|}$ on the corresponding neighbour indices.

Once our matrix is constructed, we can compute the **absolute value** of the Mean Curvature using $H = \frac{1}{2}||\boldsymbol{\Delta}_s \boldsymbol{x}||$. Now, we still have to find its correct **sign**. We do so by estimating the normal at each vertex (using the provided `calculateCloudNeighboursFromFaces` and `orientCloudNormalsFromFaces` methods), and compare its orientation with the actual average of the one-ring neighbours (using a simple dot product).

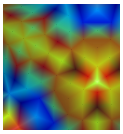


FIGURE 2 – Detail on Mean Curvature of*bumpy.off*

Observations : We observe some quite noisy results. Indeed, on rather planar portions of the model, we observe important variations while we should be expecting smoother values. This can be explained by the fact that the Mean Curvature gives a rather poor approximations for irregular triangulation, as it only takes into account the surrounding vertex positions.
Nonetheless, we can still see a certain consistency in the color code on concave and convex regions. On regions where either one of the max or min curvature are high (ie. non-planar surfaces) we do get some stronger mean curvature. As we will see later on, using the Cotangent discretization allows for a more accurate computation of the mean curvature.

Nb : The computation of the oriented normals can take a while to compute on larger meshes

Gaussian Curvature

To compute the Gaussian Curvature, we don't need to construct the Laplace Operator. We simply compute the **angle deficit** for each vertex, and normalize it by the **area** of the surrounding faces.

As for the mean curvature, we start off by finding all the neighbouring vertices. Then, we compute the angle deficit by retrieving each angle θ_j defined as :

$$\theta_j = acos \frac{\boldsymbol{AB}.\boldsymbol{AC}}{|\boldsymbol{AB}|.|\boldsymbol{AC}|}$$

Similarly, we compute the area of a surrounding triangle as

$$A_j = \frac{1}{2}|\boldsymbol{AB}|.|\boldsymbol{AC}|.sin(\theta_j)$$

Lastly, the Area factor is computed as $A = \frac{1}{3}\sum_j A_j$

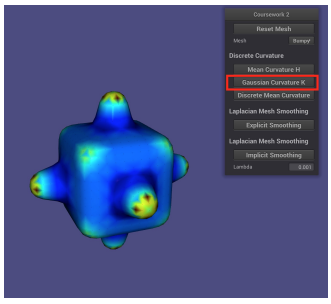


FIGURE 3 – Gaussian Curvature on *bumpy.off*

Observations : We observe coherent results. Indeed, the Gaussian curvature being defined as the product of the Maximum and Minimum curvature at a given point, any point that doesn't have a strong minimum **and** maximum curvature will have a small curvature. That is, points on planar and cylindric-like regions. The remaining points which will have a strong curvature are therefore those located on locally-spherical regions, which is what we observe on the *bumpy* model.

2 Non-uniform (Discrete Laplace-Beltrami)

We know try to estimate the mean-curvature using the Cotangent-Discretization. The only difference with our previous computation is the way we build our Laplace operator. This time, we define $\boldsymbol{L}$ as :

$$\boldsymbol{L} = \boldsymbol{M^{-1}C}$$

where $\boldsymbol{M} = diag(2A_j)$ and $\boldsymbol{C} = \begin{cases} cot(\alpha_{ij}) + cot(\beta_{ij}) \text{ where } i \neq j, j \in N_1(v_j) \\ -\sum_{v_j \in N_1(v_i)}(cot(\alpha_{ij}) + cot(\beta_{ij})), \text{ where } i = j \\ 0 \text{ otherwise} \end{cases}$

To compute α_{ij} and β_{ij}, we retrieve the adjacent triangles using the method `igl::triangle_triangle_adjacency`. We then compute the angles using the same formula as earlier. The areas are also estimated as previously using a third of the surrounding faces area.

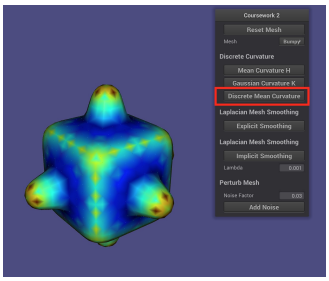


FIGURE 4 – Discrete Mean Curvature (Cotan) on *bumpy.off*

nb : When computing the cotangent, we make sure to write it as cotan(x) = tan($\frac{\pi}{2} - x$). That way, we make sure it stays definite within [0,2π] and doesn't return NaN values.

Observations : The results we obtain are again coherent with the meaning of the mean curvature. The results seem a lot smoother than with the Uniform discretization, and overall more consistent. The edges and 'tips' of the model present a high response as expected, while the flat regions give a lower mean curvature. Note that the color coding seems to differ from the Uniform Mean Curvature, but it is only due to the fact that we have higher values on the tips, which offset the whole color grading.

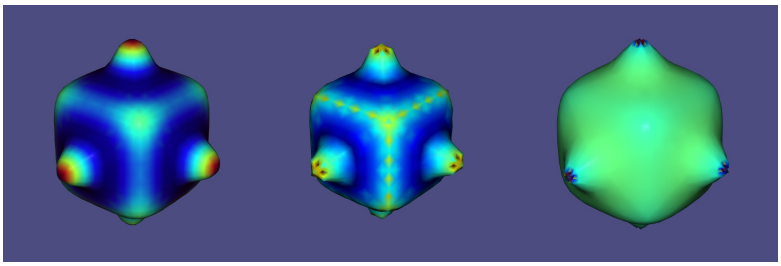
Laplacian Mesh Smoothing

Now that we have successfully constructed the Laplace operator, we can use it to smooth our mesh. The idea behind mesh smoothing is that we use the Mean Curvature information as a direction to move each vertex.

3 Explicit Laplacian Mesh Smoothing

Having already computed the Laplace operator, performing mesh smoothing is pretty straight-forward. We simply refresh the vertices $\boldsymbol{P}^{(t)}$ with :

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



(a) 10 iterations, $\lambda = 0.01$ (b) 1 iterations, $\lambda = 0.1$ (c) 20 iterations, $\lambda = 0.01$

FIGURE 5 – Explicit smoothing iterations for *bumpy.off*

Observations : When applying a small step size λ, we notice a rather good improvement over the first iterations. The Mesh gets smoother, and so does the mean curvature (*fig.a*). However, if we carry on we will see appearing some artifacts on the extremities of the mesh (*fig.c*). These artifacts most likely come from an "overshooting" of the smoothing, creating overlaps and making the mesh then diverge as we carry on smoothing.
We can notice that applying a step size that is too large, such as $\lambda = 0.1$ will lead to similar anomalies, and cause the mesh to diverge even more quickly (*fig.b*).

4 Implicit Laplacian Mesh Smoothing

To perform Implicit Smoothing, again we reuse the previously built Laplace Operator, only this time we refresh the vertices using :

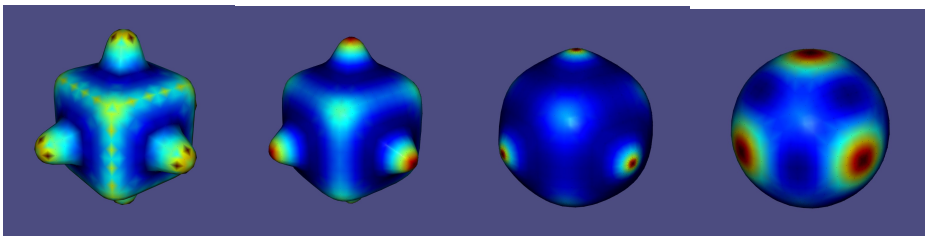
$$(\boldsymbol{I} - \lambda \boldsymbol{L})\boldsymbol{P}^{(t+1)} = \boldsymbol{P}^{(t)}$$

This equation being unconditionally stable, we can take much larger λ steps. However this comes at a cost, as it requires computing a matrix inversion (ie. solving a linear system). To make the computation more efficient, we make use of the fact that we can write $\boldsymbol{L}$ as $\boldsymbol{M}^{-1}\boldsymbol{C}$. $\boldsymbol{M}$ being symmetric, rewriting the equation as :

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

allows us to solve a sparse symmetric positive definite system. This can be done efficiently using Eigen and the sparse Cholesky decomposition.

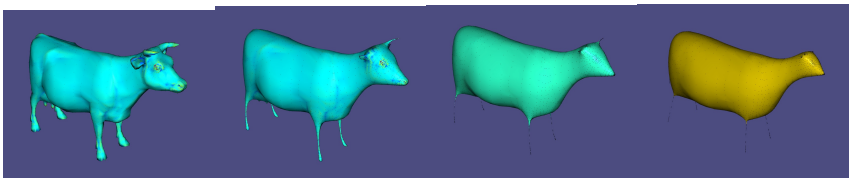
Here are the obtained results :



(a) (b) (c) (d)

FIGURE 6 – Implicit smoothing iterations for *bumpy.off*, $\lambda = 0.1$

Observations : With much larger λ steps, we obtain a much smoother result. The *bumpy.off* model morphs into a sphere after only a couple of iterations, and we observe no artifacts whatsoever.
mbox Here are the results on a more complex mesh :



(a) (b) (c) (d)

FIGURE 7 – Implicit smoothing iterations for *cow.off*, $\lambda = 0.1$

Again, the smoothing operates nicely and with no anomalies. Overall, we can see the benefits of Implicit Laplacian Smoothing : although it requires more computation, it provides much more satisfying results.
Please note that the color coding appears as plain as areas like the legs become extremely stretched and narrow, outputing high mean curvature values and offsetting the color coding of `igl::jet`.

5 Mesh Denoising

Let us now reuse the Explicit and Implicit Laplacian Smoothing in the context of mesh *denoising*. As we did in the previous Coursework, we apply synthetic noise to our mesh and evaluate how well these methods perform.

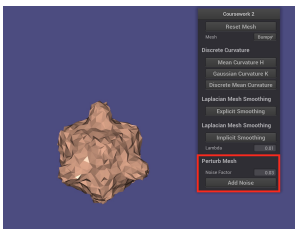
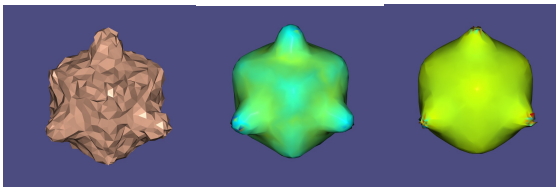


FIGURE 8 – Applying noise on *bumpy.off*

Explicit Denoising

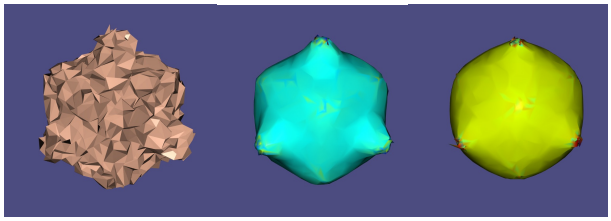
We first apply a reasonably low amount of noise ($f = 0.03$). Using rather small λ steps, we can observe the mesh smoothing into a shape that is similar to the one observed earlier. However, it does contain a higher amount of artifacts, and we can distinguish irregularities in the mesh wireframe. This can become a problem if we consider aspects like shading, texturing or reflections. It also makes further steps of the explicit denoising less accurate.



(a) (b) (c)

FIGURE 9 – Explicit Denoising Iterations, Noise Factor $f = 0.03$ $\lambda = 0.01$

Adding more noise ($f = 0.06$) leads to even stronger artifacts, and the smoothed mesh does not appear present many irregularities in its meshing and triangulation.

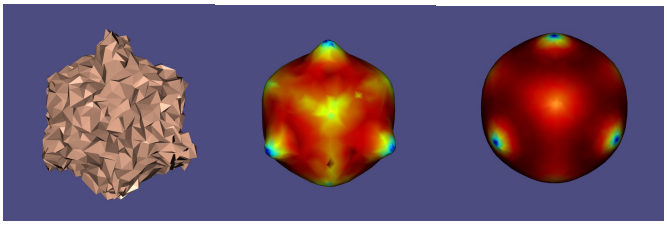


(a) (b) (c)

FIGURE 10 – Explicit Denoising Iterations, Noise Factor $f = 0.06$ $\lambda = 0.01$

Implicit Denoising

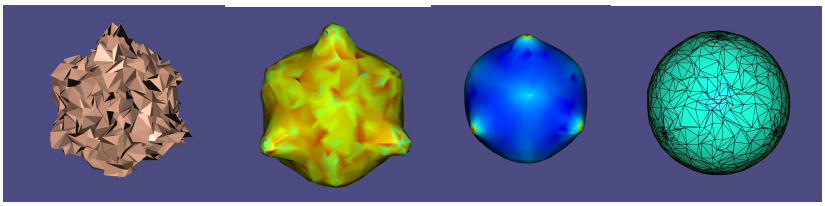
We now proceed similarly with implicit smoothing. As expected, this method yields better results. With a noise factor of $f = 0.03$, the mesh smoothes nicely and merges into a sphere as earlier.



(a) (b) (c)

FIGURE 11 – Implicit Denoising Iterations, Noise Factor $f = 0.03$ $\lambda = 0.1$

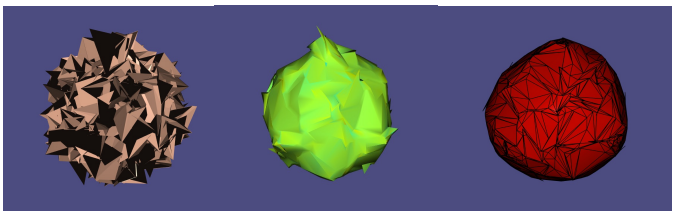
For a stronger noise factor $f = 0.06$, we still manage to merge into a sphere but as observed with the explicit smoothing, the mesh presents many irregularities. Again, this can become more of a problem if we consider shading, texturing or reflections.



(a) (b) (c) (d)

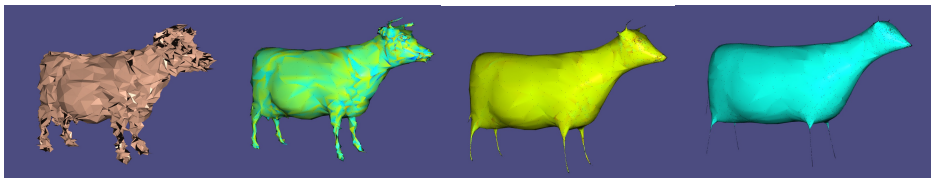
FIGURE 12 – Implicit Denoising Iterations, Noise Factor $f = 0.06$ $\lambda = 0.1$

Surprisingly however, if we consider very high amount of noise ($f = 0.3$), we still manage to merge the mesh into a somewhat spherical shape. This shape obviously contains many irregularities and would most likely not be exploitable, but we can see that implicit denoising can be used to a much larger extent than explicit denoising.



(a) (b) (c)

FIGURE 13 – Implicit Denoising Iterations, Noise Factor $f = 0.3$ $\lambda = 0.1$



(a) (b) (c) (d)

FIGURE 14 – Implicit Denoising on *cow.off*, Noise Factor $f = 0.06$ $\lambda = 0.1$