COMP0119: Acquisition and Processing of 3D Geometry

Coursework 2: Curvature Discretisation and Mesh Smoothing

This report will demonstrate the algorithm implemented for the curvature discretisation and mesh smoothing.

Section 1: Curvature Discretisation

Task 1: Uniform Mean and Gaussian Curvature

Uniform Mean Curvature

To compute the uniform mean curvature, we need to first construct the Laplacian operator in sparse matrix form using the number of neighbouring vertices:

The number of neighbouring vertices can be retrieved using igl::adjacency\_list() which returns the index of connected vertices for each vertex. For each connected vertex, we assign the to it and at last we need to make sure the diagonal direction is filled with negative 1. Next, we need to take the part into the equation by multiplication, which is just the vertex information of the input mesh.

The discrete mean curvature is given as and the second half part has already been calculated before. Therefore, we can simply compute the norm of each element in the  and multiply it by 0.5 to get the uniform mean curvature **H**. The implementation explained above can be found in MS::LaplacianMatrix() and MS:: UniformMeanCurvature().

Gaussian Curvature

To compute the Gaussian curvature, we need to know the face area of the one-ring neighbours of each vertex and the angle between two continuous edges. The equation is given as:

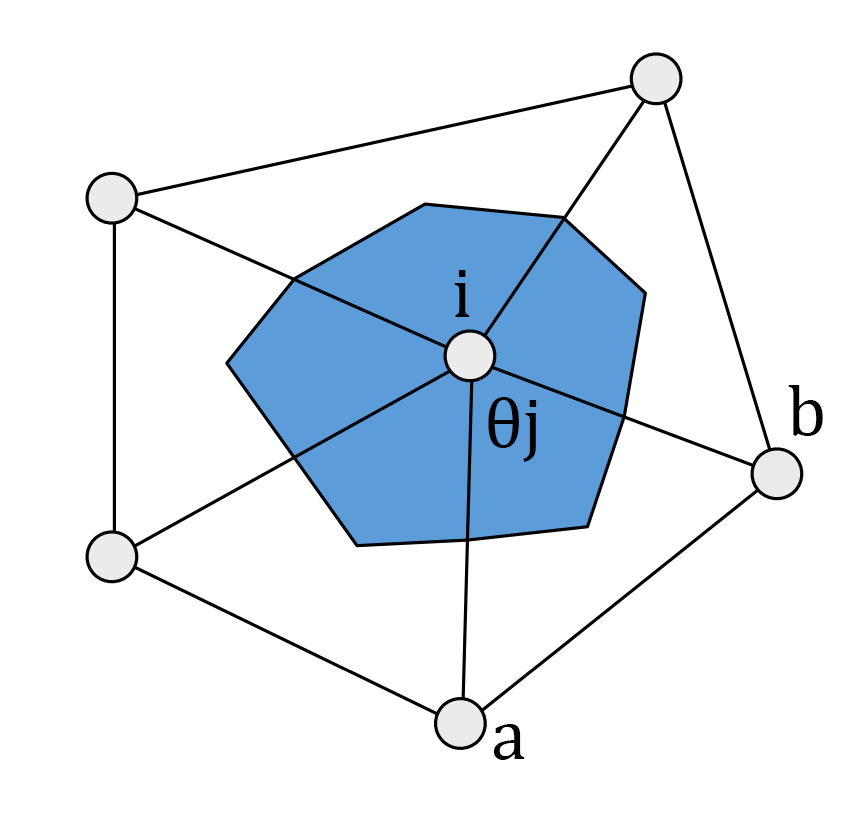
First, we need to construct a sparse matrix representing the mass. To compute the area (mass) of each face on the mesh, we can use the Heron’s formula since we have access to the face information and know each vertex coordinate within a triangle:

, where *a*, *b*, *c* is the length of each side and

We store all the value into an area list and can easily access the value using the face index as many times as we want.

The next step is to identify those faces that connect to the current vertex. We can use the igl::vertex\_triangle\_adjacency() to obtain a list of connected face for each vertex, then use the face index to lookup the value we compute in the previous step. Because we want to use the Barycentric cells, we sum up the area of connected faces for each vertex and divide it by 3. We apply each value to the diagonal direction, and this will give us the **M** matrix.

The next step is to calculate the angle between each two edges using the vector dot product. The angle can be computed as:



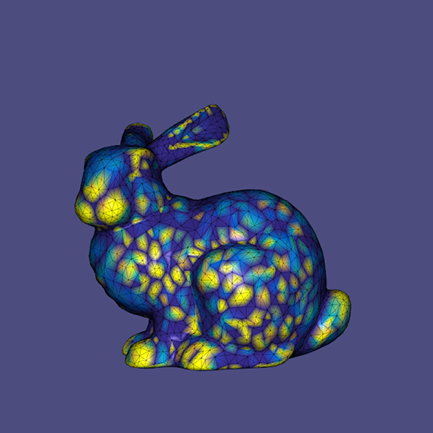
, where ,

Because we already have the connected face information of each vertex, we can immediately know the index of and vertex and thus perform the calculation above. By summing up the angle in radians and putting it back to the Gaussian curvature equation, we obtain the **K** for the input mesh. The implementation demonstrated above can be found in MS::BarycentricMassMatrix() and MS::GaussianCurvature().

Result Comparison and Discussion

The obtained results are shown below:

A picture containing yellow

Description automatically generated

*Uniform mean curvature and gaussian curvature of the bunny*

A close up of a horse

Description automatically generatedA close up of a horse

Description automatically generated

*Uniform mean curvature and gaussian curvature of the cow*

As we can see from the figure above, the result of the uniform mean curvature does not look promising because the Laplacian operator only takes the connectivity with neighbours into account and it produces very bad results for irregular triangulation. As we can notice, there is a lot of planar surface around the cow’s wrest which should return a value of zero, but it gives non-zero values for these planes instead. Though the algorithm is simple and efficient, it is not applicable for all meshes. Therefore, the uniform mean curvature is not a good approximation for continuous curvature.

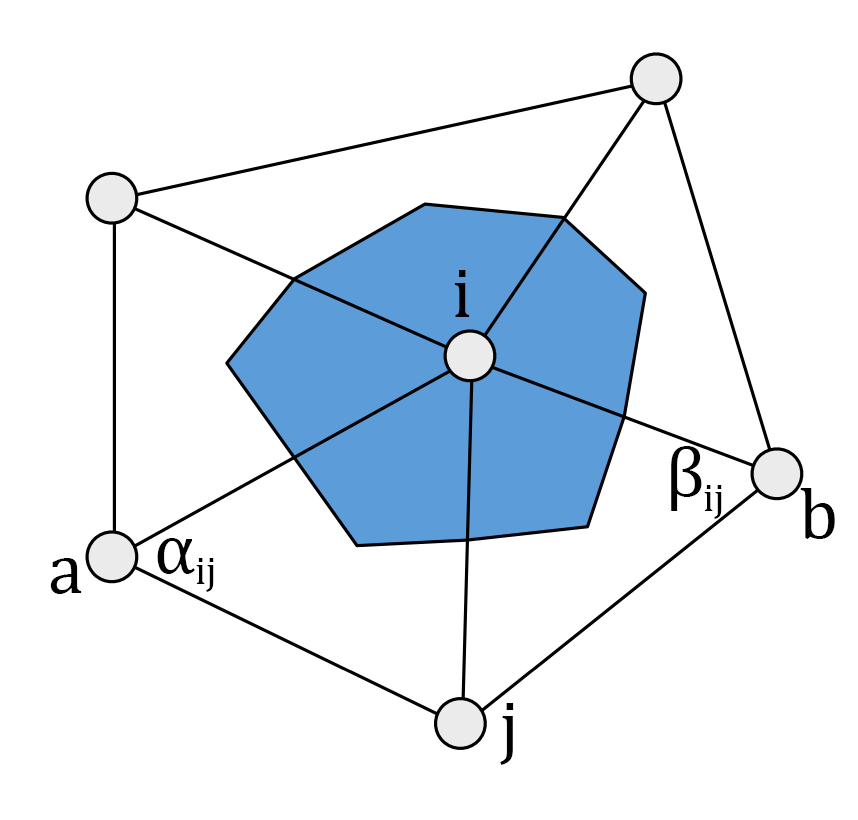
On the other hand, the Gaussian curvature yields a totally different result. It highlights the area with both high maximum and minimum curvature value, such as the bunny’s mouth and the cow’s horn and feet. The planar surface (bunny’s back and cow’s wrest) and cylindric surface (cow’s legs) are all in dark blue which means these areas have very low magnitude of the Gaussian curvature. The approximation is not accurate, but it does return a better result we want. Therefore, compared with the uniform mean curvature, the Gaussian curvature has done a better job, but still, it is not good enough for approximating the continuous curvature.

Task 2: Non-Uniform Mean Curvature

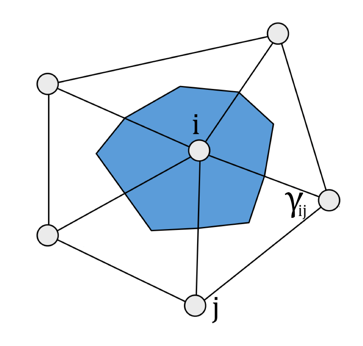
To calculate the non-uniform mean curvature, we will need to apply the Laplace-Beltrami matrix to the input mesh using the following equation:

The Laplace-Beltrami operator is formed by two matrix that we need to implement here:. The **M** matrix has already been implemented and we just need to take an inverse version of it using coefficient-wise inverse function in Eigen: cwiseInverse(). The cotangent matrix **C** is given by:

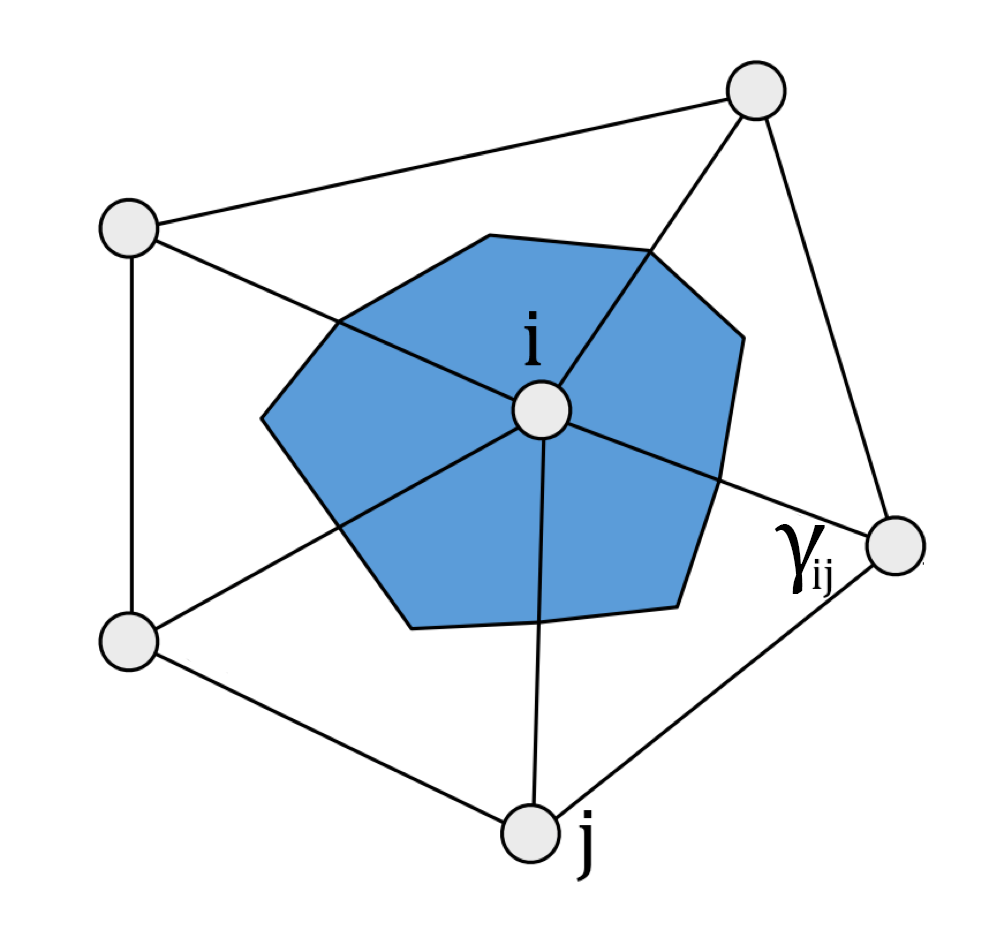
The implementation of the cotangent matrix **C** is similar to the process we take to find the angle in Gaussian curvature: we have to know the connected vertices and faces in order to find the edges. They can be found using igl::adjacency\_list() and igl::vertex\_triangle\_adjacency(). The algorithm is explained below:



Since we have the connected vertices of each vertex , we can use the known vertex , or edge to find the third vertex that forms each face, and , using the adjacent face information (this method **ONLY** applies to **enclosed manifold** meshes and it is guaranteed to return two vertices). Then the and value can be computed using dot product from , and , respectively so that we are able to compute the cotangent value of obtained angles. Once we fill the values into the sparse matrix based on the , condition, we can get the cotangent matrix **C**.



If we want to handle a non-manifold mesh (one edge can have *n* connected faces) or a non-enclosed manifold mesh (one edge may only have one connected face), we need to process the vertex separately instead of as a pair of and . First, we need to find all the vertices that are located opposite to the edge and we record their index in a list. Then for each vertex, we calculate the cotangent value as shown above, and accumulate it to the sum, then divide by the number of founded vertices *n*. Then the cotangent matrix **C** is defined as:

**

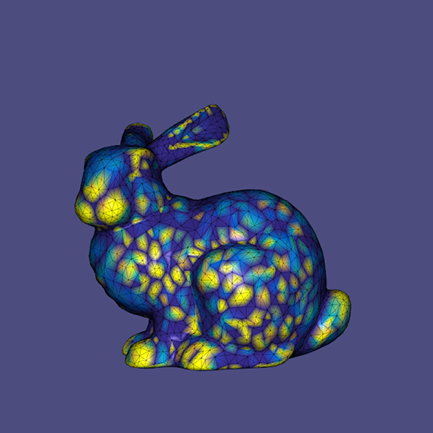
The last step is exactly same as the uniform mean curvature except that we replace the uniform Laplacian operator with Laplace-Beltrami operator:

The implementation illustrated above can be found in MS::CotangentMatrix(), MS::LaplaceBeltramiMatrix() and MS::NonUniformMeanCurvature().

Result Comparison and Discussion

The obtained results are shown below:

A picture containing yellow

Description automatically generated

*Non-uniform mean curvature and gaussian curvature of the bunny*

A close up of an animal

Description automatically generatedA close up of a horse

Description automatically generated

*Non-uniform mean curvature and gaussian curvature of the cow*

As we can observe from the figure above, the results from the non-uniform mean curvature improves a lot. Compared with the Gaussian curvature, this method is even more accurate, given the fact that it further shows the curvature of the bunny’s head and ears, cow’s face and feet correctly and more precisely (clearer layers of colour). Therefore, the non-uniform mean curvature is a good approximation for continuous curvature.

Task 3: Reconstruction

This section requires us to perform spectral analysis. The mesh can be reconstructed using *k* smallest eigenvectors from the Laplace-Beltrami matrix. The basic reconstruction equation is given by:

We will utilise the Spectra library to help us solve the eigenvalue problem. To find *k* number of smallest eigenvectors, we need to first pass in a symmetric sparse matrix in order to produce real values. However, the Laplace-Beltrami matrix is not symmetric because the cotangent matrix **C** is not symmetric. We need to convert the into a symmetric matrix so that . To begin with, the eigenequation can be written as:

Then we can rewrite the as:

And now we have the symmetric .

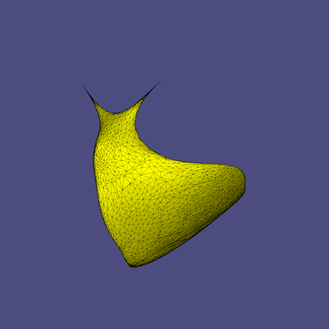
By passing in the symmetric Laplace-Beltrami matrix into the Spectra’s symmetric eigen solver Spectra::SymEigsSolver with the k, it will return us the eigenvectors as a complex matrix, and we can acquire the real eigenvector from them using the real() function. Therefore, we now have:

Because the  is not orthogonal, , thus we need to normalise it by redefining the inner product by plugging the mass matrix **M** into the equation so that we can obtain the correct value:

The implementation explained above can be found in MS::Reconstruction().

Reconstruction results for k=5, 10 and 30 as well as the original mesh are provided below:

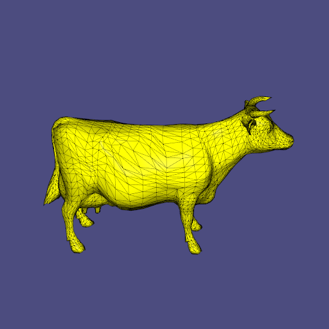
A close up of a plant

Description automatically generatedA close up of a pear

Description automatically generatedA close up of a pear

Description automatically generated

*The original mesh “bunny.off” and its reconstruction using k = 5, 10, 30*

A picture containing outdoor, sky, kite, flying

Description automatically generatedA picture containing sky, yellow, water, outdoor

Description automatically generatedA picture containing sky, yellow, looking

Description automatically generated

*The original mesh “cow.off” and its reconstruction using k = 5, 10, 30*

By observing the results, we can notice that with only few numbers of smallest eigenvectors, the reconstruction is able to recreate a rough shape of the original mesh, for example, it roughly reconstructs the flat shape (side view) of the cow with only 5 smallest eigenvectors, and later reconstructs the head. Similarly, the bunny recovers the rough shape from 30 smallest eigenvectors. However, the cow’s legs and the bunny’s ears still look like skeletons due to insufficient smallest eigenvectors. Though it is not required, we have also tried using a relatively large k value (approximately 150) for the construction; although it takes a lot time to produce the result, the results are quite surprising: though still lacking some details, they are extremely close to the original mesh. Therefore, we can conclude that the reconstruction result becomes more detailed each time when a bigger k value is provided.

Section 2: Mesh Smoothing

Task 5: Explicit Laplacian

The explicit linear diffusion scheme is given as:

Where the ***P*** represents the mesh that changes over time by a scalar diffusion constant , ***I*** represents the identity matrix, ***L*** represents the Laplacian operator. The implementation of the explicit Laplacian mesh smoothing is very straightforward, we simply plug all known variables into the equation in the loop and then can obtain the result. In order to compare this method with the implicit method later, we need to make sure the ***L*** we use in both methods are the same. Thus, the smoothing process will be based on the Laplace-Beltrami operator.

The implementation can be found in MS::ExplicitSmoothing(), and the results are shown below. The results are produced at 100th iteration with increasing , and we use Gaussian curvature here since it gives the best visibility to see the change of the mesh.

A picture containing colorful, yellow

Description automatically generatedA picture containing colorful, sky, sitting, yellow

Description automatically generatedA picture containing sky, colorful, sitting, yellow

Description automatically generatedA picture containing kite

Description automatically generated

A picture containing animal, sky

Description automatically generatedA close up of an animal

Description automatically generatedA close up of an animal

Description automatically generated

As we can see from the results, there appears to be a very weird artefact in the smoothed result 3. If we further increase the diffusion constant, the smoothing operation will fail as it produces a line. We can conclude that if the is very large (varies based on the mesh), the explicit method will become very unstable thus breeds abnormal values and collapses the entire mesh. A small gives a small step for smoothing which preserves more details thus the result will not be very obvious, while a big boosts the smoothing process which can take only few iterations to reach the desired result and may also crash due to the stability. Therefore, for the best result and efficiency (less iteration), the suitable for the explicit smoothing should be the largest which still keeps the explicit smoothing stable, theoretically. For 100 iterations, we have estimated the for each used mesh: for the bunny and for the cow.

Task 6: Implicit Laplacian

The implicit linear diffusion scheme is given as:

Where the parameters are the same as the explicit method. However, we can ignore the effect of a large because it always remains stable in the implicit scheme. Though the equation may be rewritten as , the computation would become extremely expensive due to the large sparsity (in 2D we also have to take semi-implicit scheme to improve the efficiency), thus the best method is to solve the SLE with Cholesky decomposition. We can rewrite the equation as:

And solve the using Eigen::SimplicialCholesky in each iteration. The implementation can be found in MS::ImplicitSmoothing() and the results are provided below. Since we need to compare this method with the previous one, the parameters used here will be the same:

A picture containing sky, colorful

Description automatically generatedA picture containing sky, colorful, sitting, green

Description automatically generatedA picture containing sky, colorful, green, light

Description automatically generatedA picture containing colorful, sitting, sky

Description automatically generated

A picture containing animal, sky

Description automatically generatedA close up of an animal

Description automatically generatedA close up of an animal

Description automatically generatedA close up of an animal

Description automatically generated

As we can see from the result, the implicit smoothing does not produce any weird artefact or collapse even if we increase the diffusion constant to a very large value. This indicates that the implicit smoothing method can remain stable regardless of the value of . Here we also pick results from both meshes to compare those two implemented smoothing methods:

A picture containing sky, colorful, sitting, green

Description automatically generatedA picture containing sky, colorful, sitting, green

Description automatically generated

*Explicit and implicit smoothing result using*

A picture containing animal, sky

Description automatically generatedA picture containing animal, sky

Description automatically generated

*Explicit and implicit smoothing result using*

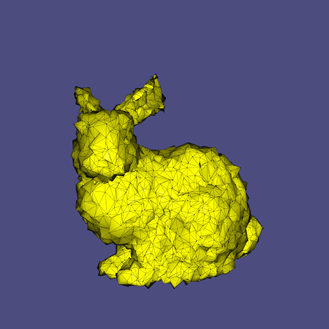
If we take a close look at the processed mesh from both methods, the results that use the same diffusion constant are visually identical. Therefore, we can conclude that the implicit Laplacian smoothing is more superior than the explicit because the diffusion constant has no influence on its stability which means it can smooth the mesh with a higher rate without collapse.

Task 7: Denoising

In order to benchmark the performance of denoising with the diffusion flow, we need to add different levels of noise to the original mesh and see how it can be smoothed back to a well-shaped mesh. The noise adding function is similar to the one we did in the coursework with regard to the integrative closest point: we simply add zero-mean gaussian noise with different values of standard deviation and scale the noise with the bounding box for each axis. Since we want to minimise the effect of choosing a bad diffusion constant, the implicit Laplacian smoothing method is utilised for all the following tests. We turn off the curvature in order to have a better view of shape.

Denoising Result (Basic)

The results are all produced at 0th, 50th, 100th and 200th iteration using fixed noise level and diffusion constant:

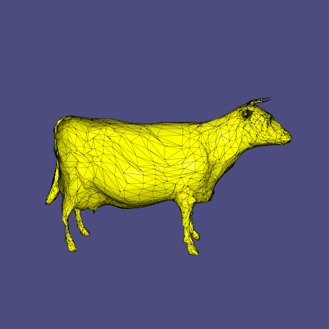
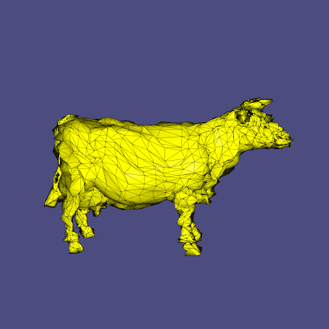
A picture containing yellow

Description automatically generatedA picture containing yellow

Description automatically generatedA picture containing yellow

Description automatically generated

*Noise level = 1.0,*



*Noise level = 1.0,*

From the results above, we can notice that the Laplacian mesh denoising is relatively effective given a low noise level and a reasonable diffusion constant. However, because the original mesh has already been distorted before the denoising, it is not possible to recover the details such as the bunny’s face and cow’s legs.

Denoising Benchmark A

This benchmark aims to find the relation between the noise level and the diffusion constant of the denoising method. We will run the method using one fixed variable and one changing variable (noise level or diffusion constant). The results are produced at 200th iteration:

A close up of a pear

Description automatically generatedA close up of a pear

Description automatically generatedA close up of a pear

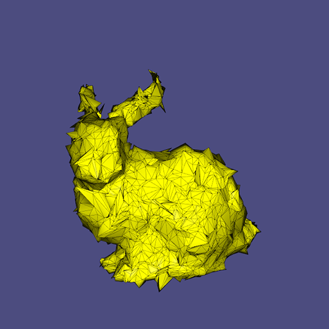
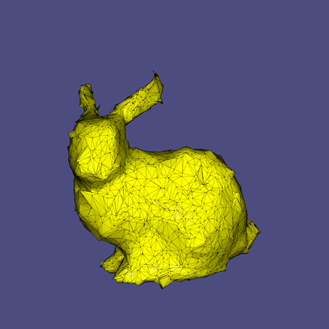
Description automatically generatedA close up of a pear

Description automatically generated

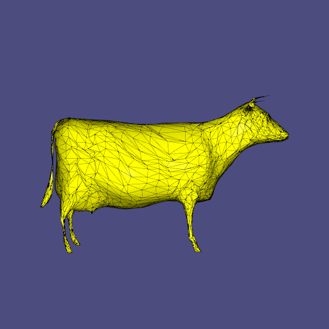
*Noise level = 1.0 and*

A close up of a pear

Description automatically generatedA close up of a pear

Description automatically generated

*and Noise Level = 2.0, 3.0, 4.0, 5.0*

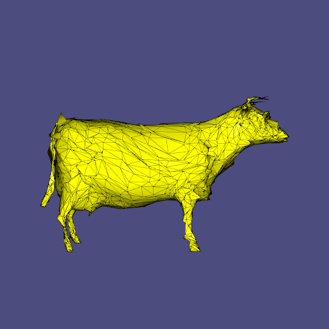
A picture containing yellow

Description automatically generatedA black and yellow umbrella

Description automatically generatedA picture containing sky, yellow

Description automatically generated

*Noise level = 1.0 and*

A picture containing parrot, animal, reptile

Description automatically generatedA picture containing reptile, animal

Description automatically generatedA close up of a flower

Description automatically generated

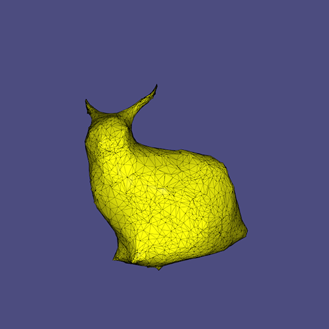
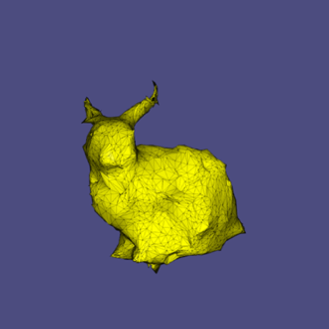
and noise level = 2.0, 3.0, 4.0, 5.0

As we can observe from the results above, when we keep the noise level fixed, the higher diffusion constant removes more noises as well as smooth the surface a lot. If we keep a high diffusion constant and add more noise to the mesh, the final quality decreases a lot. Therefore, we may make an assumption that the method can handle the highly noised mesh if we increase the diffusion constant at the same time.

Denoising Benchmark B:

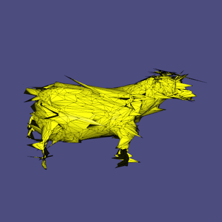
This benchmark aims to find the limitation of the denoising method. This time, we will use a very large noise level and at the same time, a very large diffusion constant to keep the balance and see how it performs on two different meshes:

A close up of a yellow flower

Description automatically generated

*Noise level = 20.0 and*

A close up of a flower

Description automatically generated

*Noise level = 20.0 and*

From the figures above, we can see that the denoising result is not good. The method recovers most of the bunny but judging from the overall shape, we may say that the denoising process does not reach our expectation. The cow has very obvious distortion even after the denoising and the mesh looks very broken.

Conclusion

From the benchmark results, we can conclude that the Laplacian mesh denoising works very well when the mesh does not have too much noise. With a high diffusion constant, it can still produce a reasonably good denoising result. However, the method fails when the original mesh is too noisy that the surface is totally distorted and the attributes (or characteristics) are no longer visible, even a high diffusion constant cannot recover the detail from it, though the mesh can still be “smoothed”.

Reference

Desbrun, M., et al. (1999). Implicit Fairing of Irregular Meshes using Diffusion and Curvature Flow. *SIGGRAPH '99 Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, 317-324. doi: 10.1145/311535.311576

Lu, T. (2013). *Lecture 12: Discrete Laplacian*. Stanford University. Retrieved from <https://graphics.stanford.edu/courses/cs468-13-spring/assets/lecture12-lu.pdf>