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:

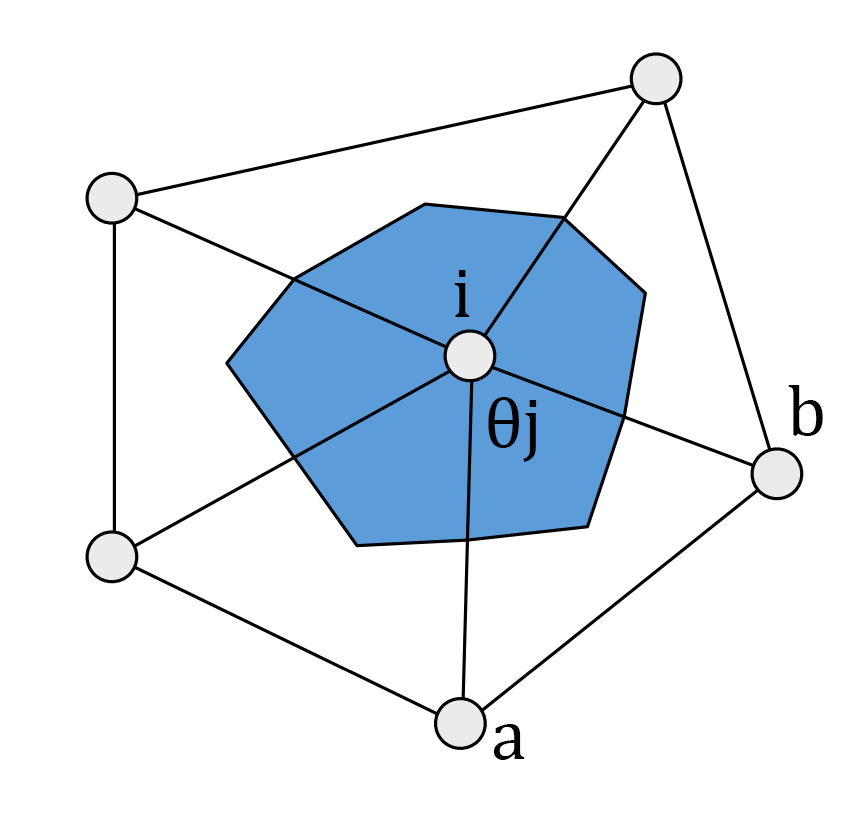
First, we need to construct a sparse matrix representing the mass. To compute the area (mass) of each face on the mesh, we can use the Heron’s formula since we have access to the face information and know each vertex coordinate within a triangle:

, 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 . 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().

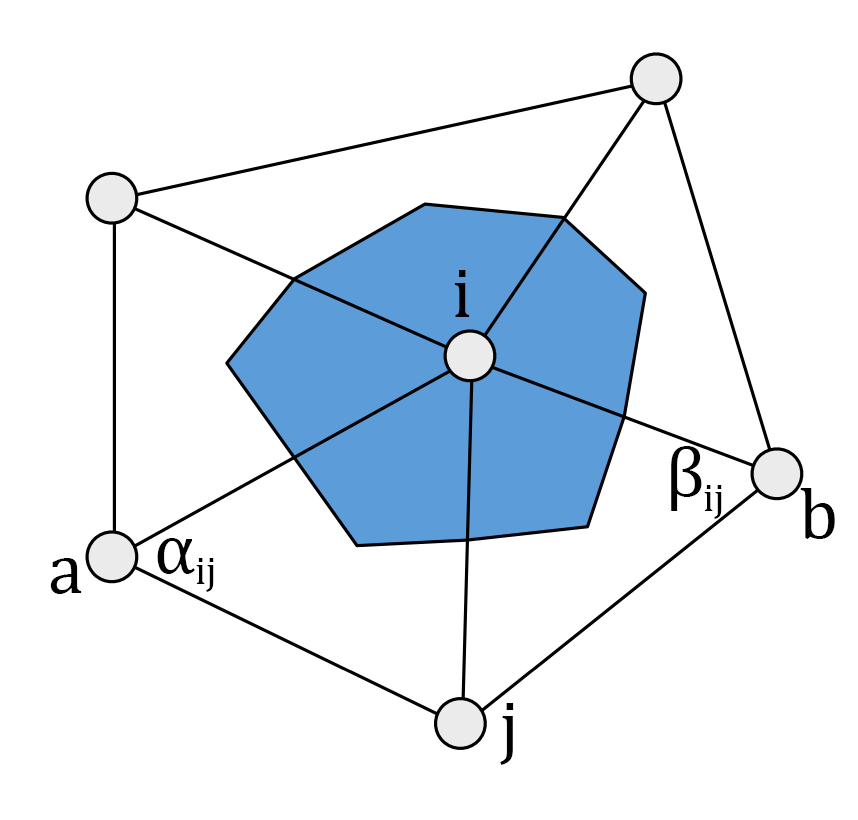
The obtained results are shown below:

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 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**. 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().

The obtained results are shown below:

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:

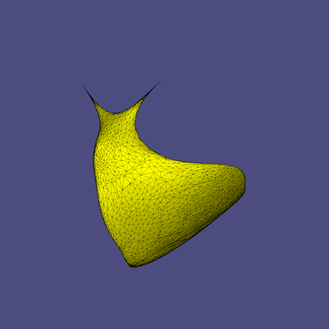
And now we have the symmetric .

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, we need to redefine 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:

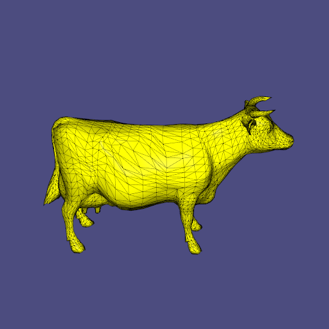
A close up of a plant

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 find 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:

If the is very large, the explicit method will become very unstable and the vertex will be shifted randomly (irregularly), which will produce a mesh that has a lot of broken faces. The suitable will be the max

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:

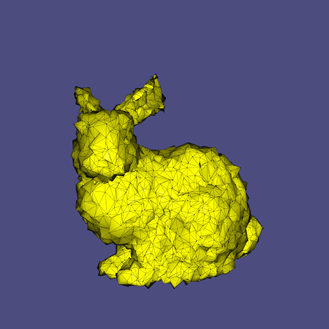
And solve the using Eigen::SimplicialCholesky in each iteration. The implementation can be found in MS::ImplicitSmoothing() and the results are provided below:

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.

The results are produced at 0th, 50th, 100th and 200th iteration:

Denoising results of the bunny (noise level = 1.0, )

A picture containing yellow

Description automatically generatedA picture containing yellow

Description automatically generatedA picture containing yellow

Description automatically generated

Noise Level = 1.0 and

A close up of a pear

Description automatically generatedA close up of a pear

Description automatically generatedA close up of a pear

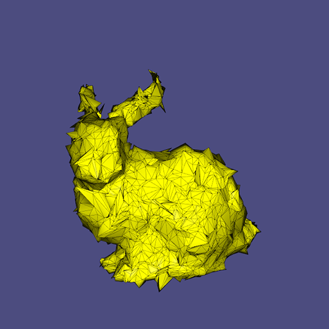
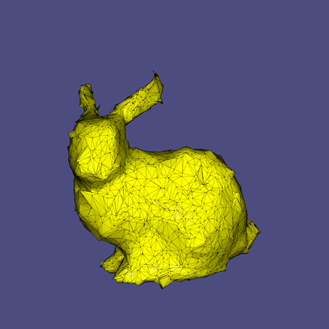
Description automatically generatedA close up of a pear

Description automatically generated

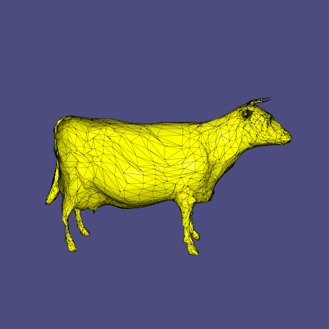
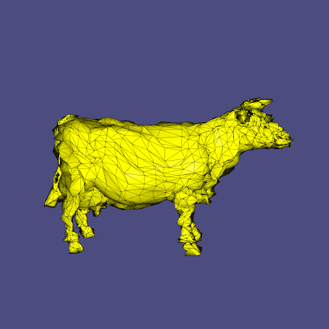
and Noise Level = 2.0, 3.0, 4.0, 5.0

A close up of a pear

Description automatically generatedA close up of a pear

Description automatically generated

Denoising results of the cow (noise level = 1.0, =0.000002)



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>