COMP0119 CW2

March 2024

1 Uniform Laplace:

The uniform discretisation of the Laplace-Beltrami operator is shown in equation 1:

$$\Delta_{\text{uni}} f(v_i) := \frac{1}{|\mathcal{N}_1(v_i)|} \sum_{v_j \in \mathcal{N}_1(v_i)} (f(v_j) - f(v_i)) \tag{1}$$

This can be constructed as the form of $\mathbf{M}^{-1}\mathbf{C}$, where $\mathbf{M}$ is the diagonal matrix indicating number of neighbors (normalizing factor $\frac{1}{|\mathcal{N}_1(v_i)|}$) and $\mathbf{C}$ is the neighbors $(f(v_j) - f(v_i))$.

Here is a snippet of the code:

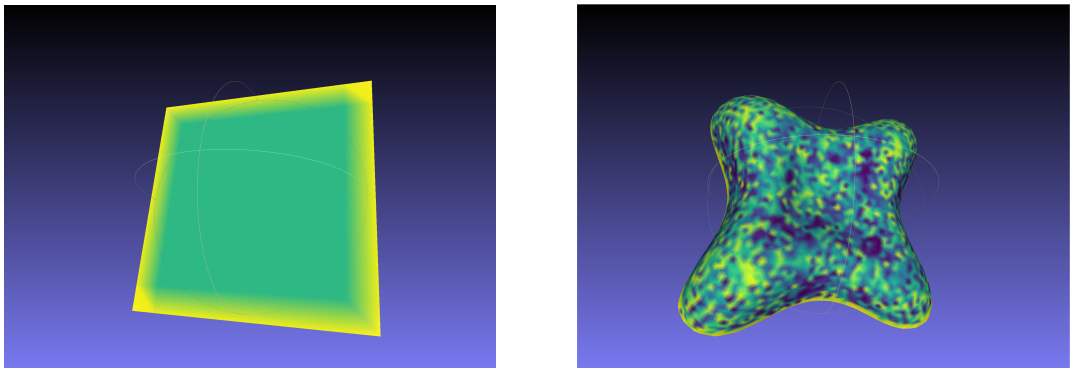
```
# Constructing the M and C matrix for the uniform discretization
def get_M_inv_uniform(mesh):
    neighbors = mesh.vertex_neighbors
    size = len(neighbors)
    nei_mat = sparse.spdiags([1/len(x) for x in neighbors], 0, size, size)
    return nei_mat

def get_C_uniform(mesh):
    size = len(mesh.vertices)
    neighbors = mesh.vertex_neighbors
    row = []
    col = []
    data = []
    for i in range(size):
        col.append(i)                          # for index of i
        col.extend(neighbors[i])               # for index of neighbors of i
        row.extend([i]*(len(neighbors[i])+1)) # at i th row, (len + 1) entries
        data.append(-len(neighbors[i]))        # first add diagonal value -N
        data.extend([1]*len(neighbors[i]))     # 1 at all neighbor indices
    mat = sparse.csr_matrix((data, (row, col)), shape=(size, size))
    return mat
```

The following part shows the computation of the mean curvature using the uniform Laplacian-Beltrami, and the Gaussian curvature. The mesh is colored by the curvatures using the *matplotlib viridis* color map and balanced using histogram equalization to prevent extreme values.

- a: Mean curvature

Since the operator is the approximation of $\Delta f = -2Hn$, where H is the mean curvature and n is a unit vector. H can be approximated by $H = length(f/2)$. The mean curvature for the plane and lilium is shown in figure 1a and 1b.



(a) Plane (b) Lilium

Figure 1: Uniform Laplacian-Beltrami mean curvature

- b: Gaussian curvature:

  The Gaussian curvature is defined as:

$$K = \left(2\pi - \sum_j \theta_j\right) / A \tag{2}$$

  , where A is the area (Barycentric is used) and $\sum_j \theta_j$ is the sum of the angles around a vertex shown in figure 2. Barycentric area is used that computes the third of the triangle area for each face, where the triangle area is calculated using the length of the cross product.

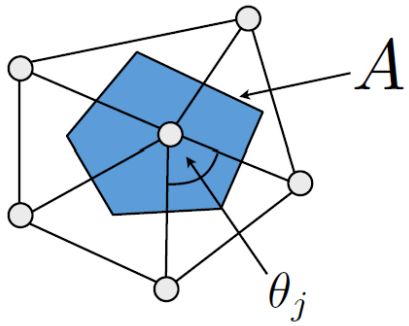
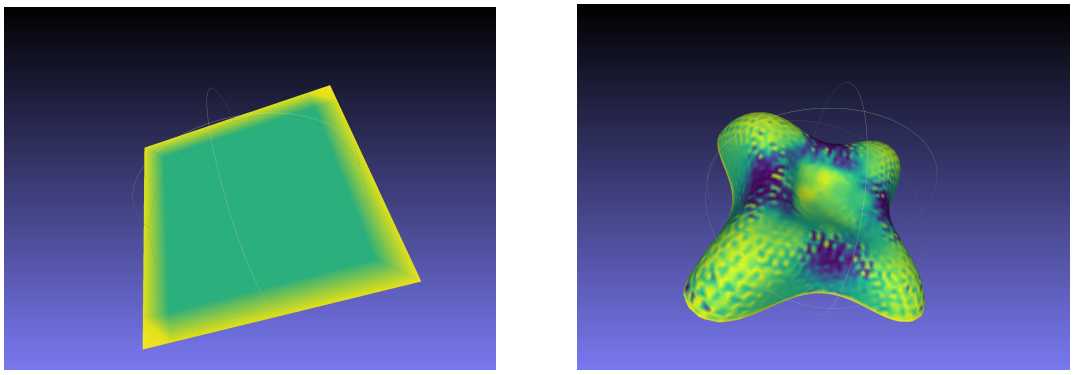


Figure 2: Gaussian curvature for one vertex

  The Gaussian curvature colored Plane and lilium is shown in figure 3a and 3b.

- c: Discussion: Gaussian curvature should be zero when one of the principle direction is zero, which is true for the plane. The mean curvature for the plane is also correct since the mean of



(a) Plane (b) Lilium

Figure 3: Gaussian curvature

two zeros is also zero. However, we can see discontinuous dots from figure 3b and 1b. Which shows that this is not a good approximation.

2 First and second fundamental form:

The Jacobian and Hessian of the parameterization can be derived:

$$\begin{pmatrix} x_u \\ x_v \end{pmatrix} = \begin{pmatrix} -a \times \sin(u) \times \sin(v) & b \times \cos(u) \times \sin(v) & 0 \\ -a \times \cos(u) \times \cos(v) & b \times \sin(u) \times \cos(v) & -c \times \sin(v) \end{pmatrix} \tag{3}$$

$$\begin{pmatrix} x_{uu} \\ x_{uv} \\ x_{vv} \end{pmatrix} = \begin{pmatrix} -a \times \cos(u) \times \sin(v) & -b \times \sin(u) \times \sin(v) & 0 \\ -a \times \sin(u) \times \cos(v) & b \times \cos(u) \times \cos(v) & 0 \\ -a \times \cos(u) \times \sin(v) & -b \times \sin(u) \times \sin(v) & -c \times \cos(v) \end{pmatrix} \tag{4}$$

The normal at a given parameter is defined as:

$$\mathbf{n} = \frac{\mathbf{x}_u \times \mathbf{x}_v}{\|\mathbf{x}_u \times \mathbf{x}_v\|} \tag{5}$$

The first and second fundamental form and the normal curvature equation can be found in the slides. The result of the normal curvature at the point (a,0,0) as a function of a direction vector on the tangent plane is shown in figure 4.

3 Cotangent Laplace-Beltrami:

The cotagent operator can be constructed as $M^{-1}C$ in a similar way. This time M contains the area normalization and C contains all the cotagent of the pairs of vertices. The code snippet can be found in the notebook.

```
# Constructing the M and C matrix for the cotagent discretization
def get_M_inv_cot(mesh):
    vertices_area = areas(mesh)
    size = len(vertices_area)
    nei_mat = sparse.spdiags(vertices_area, 0, size, size)
    return nei_mat
```

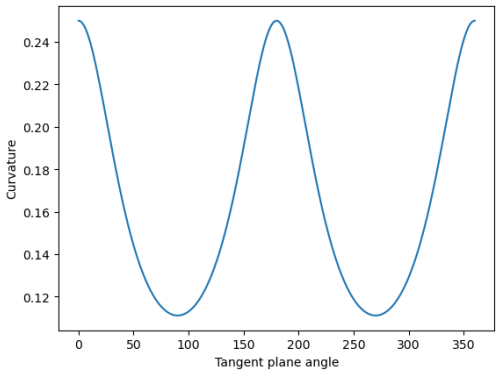


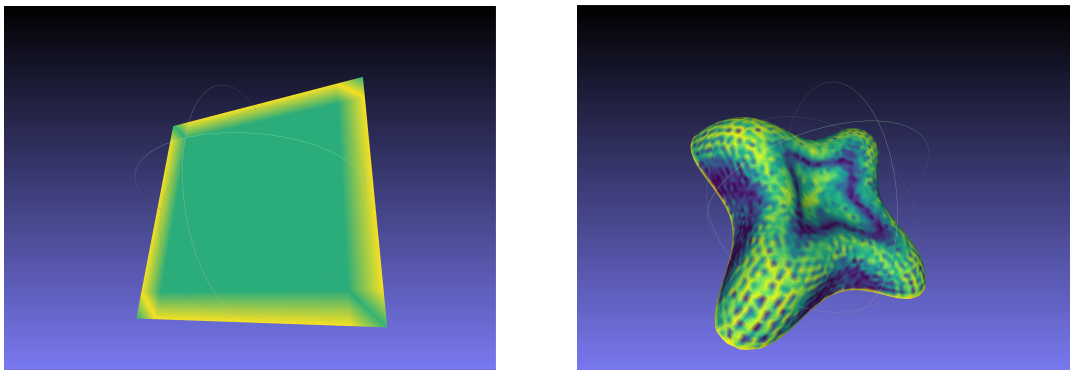
Figure 4: Normal curvature

```
def get_cot(v1, v2, mesh):
    '''
    get the 0.5 * cot alpha + cot beta for two vertices
    v1, v2: vertex index
    '''
    vfs = mesh.vertex_faces                    # get all the face index for each vertex
    ifaces = np.intersect1d(vfs[v1],vfs[v2]) # get the faces with the two vertex
    ifaces = ifaces[ifaces!=-1]                # -1 is padded so delete
    # print(ifaces)
    fvs = mesh.faces[ifaces]                   # get the two (or one) faces vertex list: like [[x,y,a],[x,b,y]]
    # print(fvs)
    num = np.array(fvs).shape[0]
    face_idx = np.array([0,1,2])               # for boolean calculation
    total_cot = 0
    for i in range(num):
        a = (fvs[i]!=v1)
        b = (fvs[i]!=v2)
        c = a & b                                  # get true false boolean array where the other vertex place is True
        # print(c)
        idx = face_idx[c]                          # get the location of the other face vertex wrt the fvs list
        theta = mesh.face_angles[ifaces[i],idx]      # get the angle according to the other vertex's index
        total_cot += 1/np.tan(theta[0])
    return 0.5*total_cot


def get_C_cot(mesh):
    size = len(mesh.vertices)
    neighbors = mesh.vertex_neighbors
    row = [] # stored row index non-zero
    col = [] # stored col index non-zero
    data = []
    for i in range(size):
        if i % 100 == 0:
            print(i)
        col.append(i)                           # for index of i
        col.extend(neighbors[i])                # for index of neighbors of i
        row.extend([i]*(len(neighbors[i])+1)) # at i th row, (len + 1) entries
        # compute cot alpha + cot beta for each pair of neighbor
        data_row = []
        for neighbor in neighbors[i]:
```

```
            cot = get_cot(i, neighbor, mesh)
            data_row.append(cot)
        data.append(-np.sum(data_row))        # -sum of non-diagonal in this row
        data.extend(data_row)                 # the cot value for the pairs
        # print(data)
    mat = sparse.csr_matrix((data, (row, col)), shape=(size, size))
    return mat
```

The mean curvature with cotangent operator is shown in figure 5a and 5b. This improves the mean curvature than the uniform one since the flat regions has a common curvature color now.



(a) Plane (b) Lilium

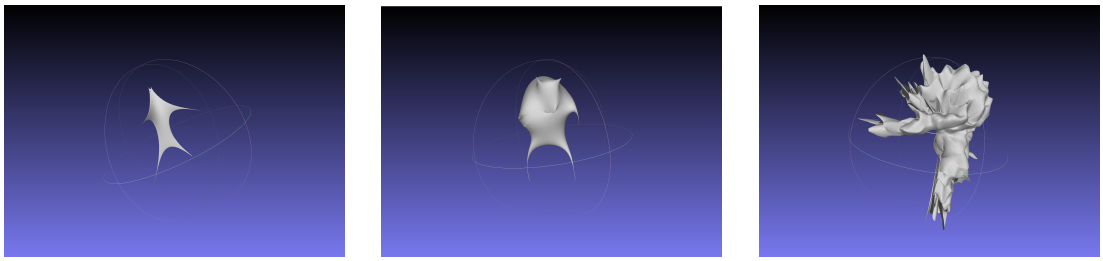
Figure 5: Gaussian curvature

4 Model analysis:

The reconstruction equation is:

$$\begin{array}{ccc} \mathbf{x} := [x_1, \ldots, x_n] & \mathbf{y} := [y_1, \ldots, y_n] & \mathbf{z} := [z_1, \ldots, z_n] \\ \mathbf{x} \leftarrow \sum_{i=1}^{k} \left(\mathbf{x}^T \mathbf{e}_i\right) \mathbf{e}_i & \mathbf{y} \leftarrow \sum_{i=1}^{k} \left(\mathbf{y}^T \mathbf{e}_i\right) \mathbf{e}_i & \mathbf{z} \leftarrow \sum_{i=1}^{k} \left(\mathbf{z}^T \mathbf{e}_i\right) \mathbf{e}_i \end{array} \tag{6}$$

The output mesh is included in the generated folder, with name *armadillo_kn.obj*. We can see that as k increase, more details is included since the large eigenvalue(frequency) means more detail/noise.



(a) k = 5 (b) k = 15 (c) Large k

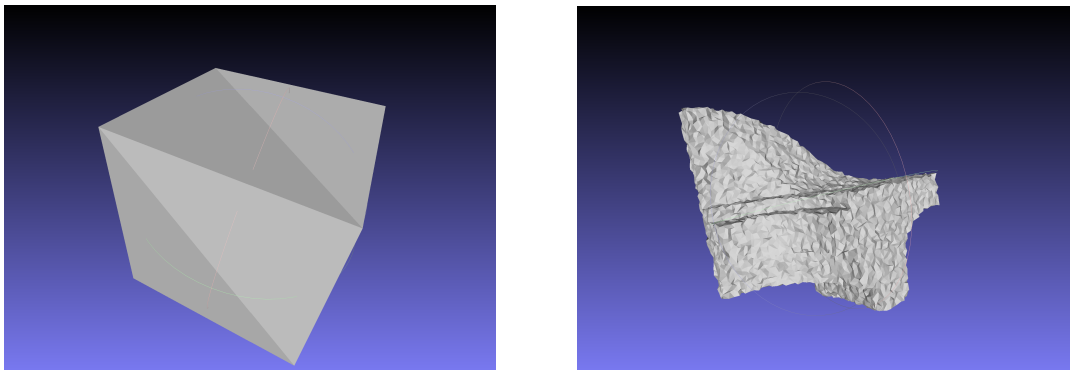
Figure 6: Model analysis

5 explicit smoothing:

Explicit smoothing is implemented:

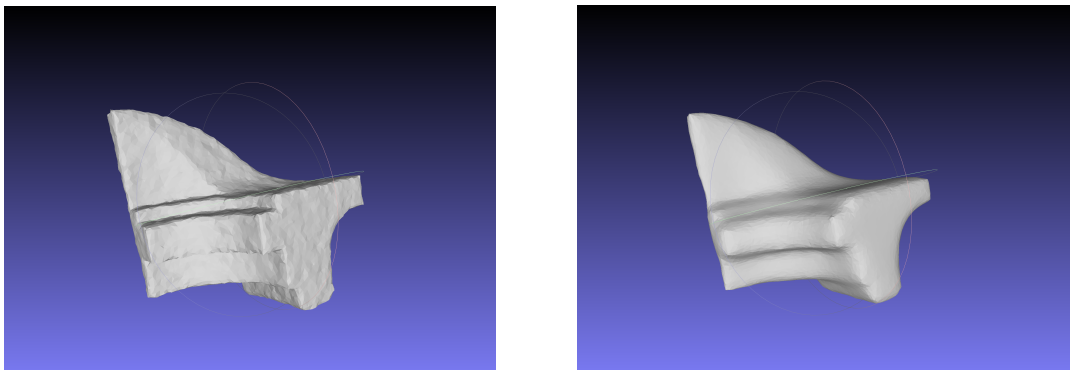
$$\mathbf{P}^{(t+1)} = (\mathbf{I} + \lambda \mathbf{L})\mathbf{P}^{(t)} \tag{7}$$

For the cube, the explicit smoothing is very unstable. With higher λ values resulting in the error in the meshlab viewer. The smoothed cube for small λ is shown in figure 7a. The same thing happens for the fandisk, where larger λ values results in error in the viewer, the smaller ones is shown in figures 7b, 8a and 8b.



(a) Cube (b) $\lambda = 0.01$

Figure 7: Explicit smoothing



(a) $\lambda = 0.1$ (b) $\lambda = 1$

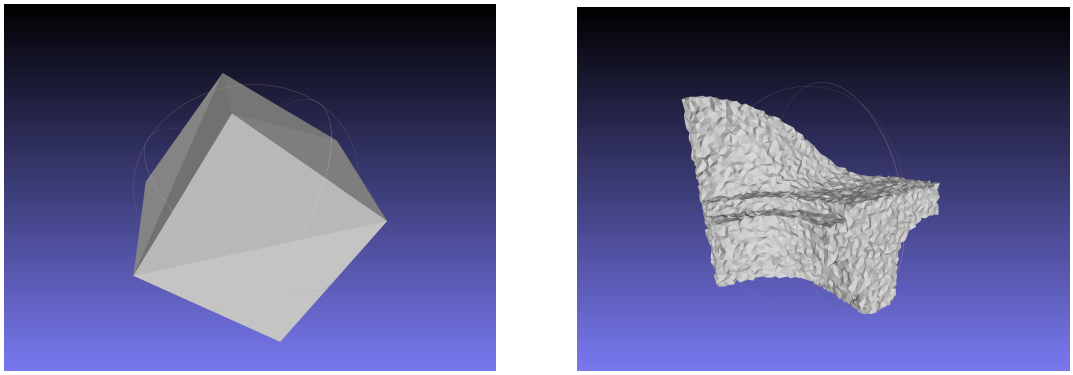
Figure 8: Explicit smoothing

6 Implicit smoothing:

The Implicit smoothing is implemented:

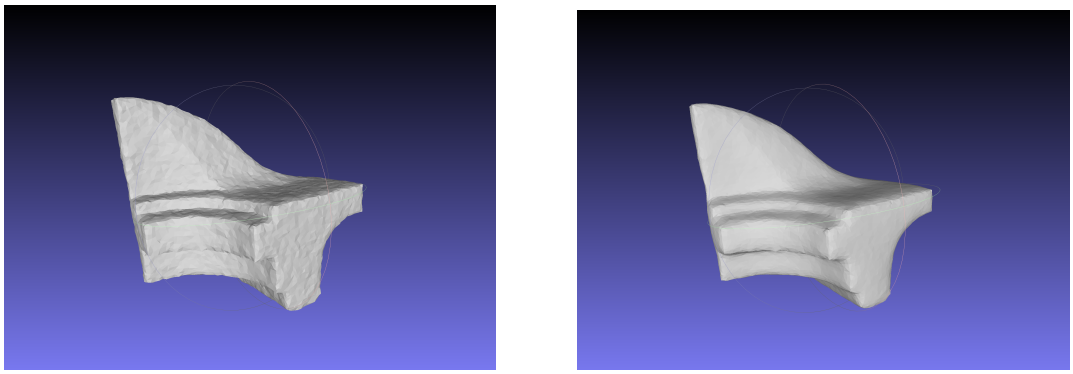
$$(\mathbf{I} - \lambda \mathbf{L})\mathbf{P}^{(t+1)} = \mathbf{P}^{(t)} \tag{8}$$

The result for the cube looks the same for all λ shown in figure 9a. As the implicit smoothing is unconditionally stable, all of the results are stable, shown in figure 9b, 10a and 10b.



(a) Cube with $\lambda = 500$ (b) $\lambda = 0.1$

Figure 9: Implicit smoothing



(a) $\lambda = 100$ (b) $\lambda = 500$

Figure 10: Implicit smoothing

Compared to the explicit smoothing, as implicit is stable, with higher λ value result in smalled iteration numbers.

7 Mesh Denoising:

The mesh of the noisy bunny is stored in the folder generated, with name *noisy_bunny_x.obj*. The smoothed bunny is named *smooth_bunny_x.obj*, with x being the noise amount.