本次作业选择用python实现网格平滑任务，实现了Bilateral Normal Filter和Laplacian Smoothing 两种方法并做了比较。

Method 1:Bilateral Normal Filter

1 Introduction

实现代码在BilateralNormalFilter.py中，将网格初始化为一个trimesh对象，依次迭代调整三角形面的法相，然后再去调整顶点位置。该算法借鉴2维图像处理中双边滤波的思想，在Smoothing时考虑相邻Mesh的空间临近性和法向量值的相似性，可以有效保留网格的几何细节。

2 Some details

update faces

首先对每一个面进行双边滤波，调整面的法向。
$n_i^{k+1} = \frac{1}{K_i}\sum_j A_j W_{\sigma_s}(||c_i - c_j||) * W_{\sigma_r}(||I_i^k - I_j^k||) * n_j^k$
$K_i = \sum_j W_{\sigma_s}(||c_i - c_j||) * W_{\sigma_r}(||I_i^k - I_j^k||)$
其中，n_i^k表示第k步面i的法向量，求和项是遍历与面i相邻的面j求和，A_i是三角形j的面积，$W_{\sigma_s}(|||c_i - c_j||)$是空间临近性，计算几何上的接近，$c_i, c_j$分别指面i和面j 的重心，$W_{\sigma_s}$是$\sigma$为$\sigma_s$的高斯函数，$W_{\sigma_r}(||I_i^k - I_j^k||)$计算法向量的相似性。

update vertex

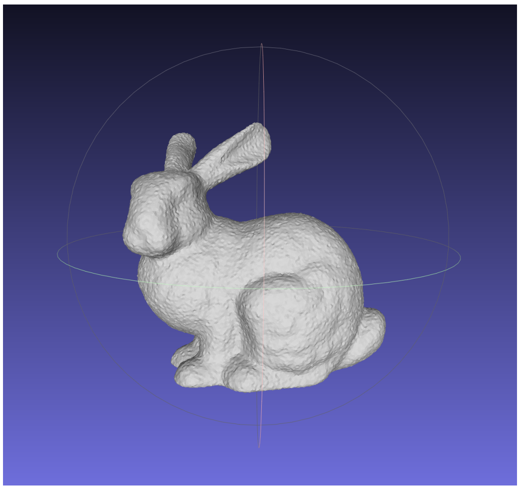
根据调整得到的面法向量要与每一个三角形的三边垂直，使用投影优化更新顶点位置：
$v_i^{new} = v_i + \frac{1}{\mathcal{F}}\sum_{j\in\mathcal{F}(i)}\delta_j$
$\delta_j = \langle c_j - v_i, n_j\rangle n_j$
对于面i的每一个相邻面j，c_j代表面j的法向量，该公式把$c_j - v_i$投影到n_j方向上，让顶点在这个方向上移动一小段距离，加权平均得到整体的偏移量。

3 Code

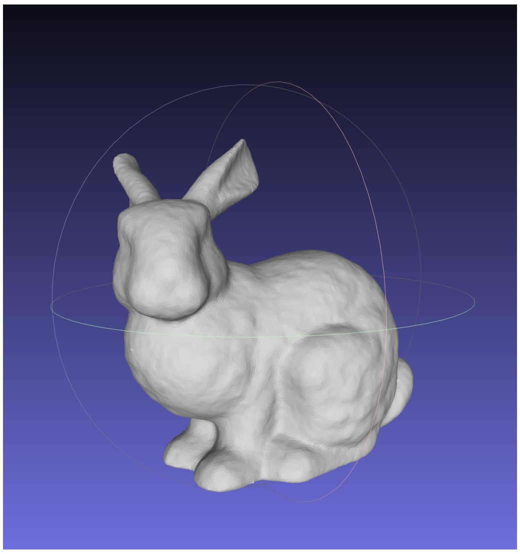
在BilateralNormalFilter.py中实现了一个类，传入一个trimesh对象作为初始化，在smoothing时，先迭代更新面法向量(update_faces())，然后再迭代更新顶点(update_vertex())，更新公式见上一部分。为了提高迭代更新效率，在开始时先做预处理算出每一个面的相邻面，然后计算所有相邻面之间的中心点距离的平均值，作为空间核的标准差σ_s，查阅资料知在图像中这个值通常是人为设定，但在网格中采用几何平均可以自适应缩放。在迭代时，按公式依次更新面和顶点即可，最后输出得到的mesh。

4 Results

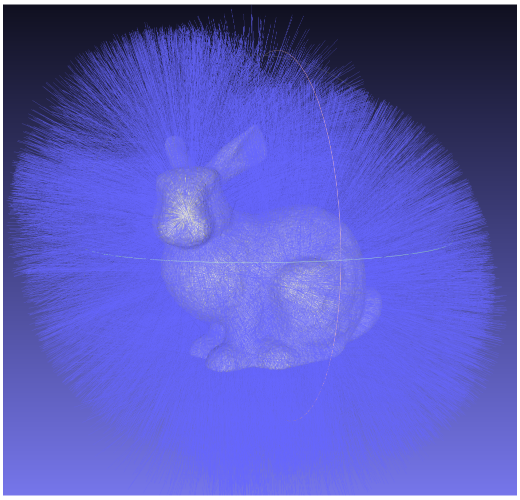
选择法向量高斯核为0.5，面迭代次数20，顶点迭代次数15，可以得到比较不错的结果：
对比处理前后法向量情况，看以看出算法处理后，表面法向量确实比原来要更加规整。



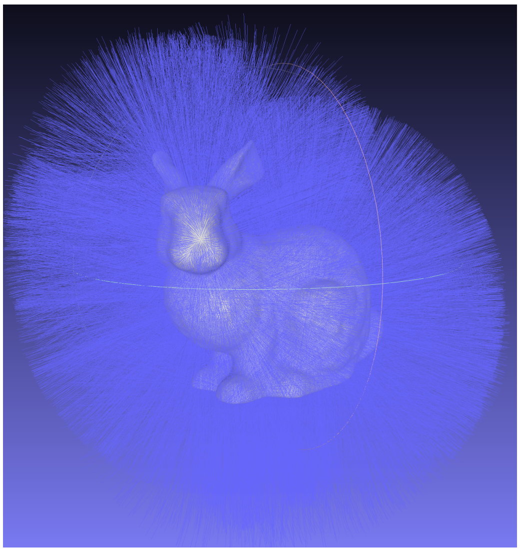
(a) Original



(b) After Bilateral Normal Filter



(a) Original



(b) After Bilateral Normal Filter

Method 2:Laplacian Smoothing

1 Introduction

实现代码在LaplacianSmoothing.py中。因为显式欧拉存在数值不稳定，对步长的选取没有鲁棒性等问题，本任务采用了隐式欧拉对网格顶点位置进行迭代计算。

参考了Polygon mesh processing中第3、4两章内容，实现了等权重和余切两种拉普拉斯算子，尝试了Barycentric cell 和Voronoi cell两种局部区域的选取方式。

2 Some details

Uniform Laplacian

$\Delta f(v_i) = \frac{1}{|N_1(v_i)|}\sum_{v_j \in N_1(v_i)}(f(v_j) - f(v_i))$

v_i代表某个点，$f()$是以点为自变量的函数，在这个问题中可以理解为点的坐标，$N_1(v_i)$代表与v_i相邻的周围一圈点。

以这个形式构成的拉普拉斯算子记为L，即

$$
L \begin{bmatrix} f(v_1) \\ f(v_2) \\ . \\ . \\ . \\ f(v_n) \end{bmatrix} = \begin{bmatrix} \Delta f(v_1) \\ \Delta f(v_2) \\ . \\ . \\ . \\ \Delta f(v_n) \end{bmatrix} \tag{1}
$$

观察到可以把L拆成一个对角矩阵D与另一个矩阵A的乘积，即$L = DA$。
其中，D对角线上元素为$|N_1(v_i)|$，A是这个Mesh的邻接矩阵。

Cotangent Laplacian

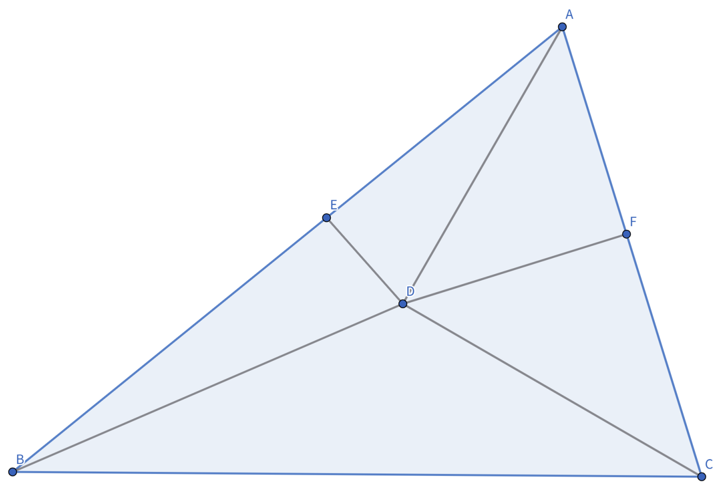
$\Delta f(v_i) = \frac{1}{2A_i} \sum_{v_j \in N_1(v_i)} (cot\alpha_{i,j} + cot\beta_{i,j})(f(v_j) - f(v_i))$
$v_i, f(v_i), N_1(v_i)$与Uniform Laplacian中含义相同，A_i代表了局部区域的面积，$\alpha i, j$，$\beta_{i,j}$表示以(v_i, v_j)为边的两个三角形的其他两个角。
记拉普拉斯算子为L，则L也可表示为对角矩阵D和邻接矩阵A的乘积，D对角线上是局部区域面积A_i。

implicit Euler

$f(t + h) = f(t) + hf(f + h)$
写成矩阵形式有
$(I - hL)f(t + h) = f(t)$
由于 $L = DA$
$(D^{-1} - hA)f(t + h) = D^{-1}f(t)$
这是$Ax = b$的形式，采用稀疏矩阵求解即可得到$f(t + h)$

Barycentric cell 和 Voronoi cell

一个顶点周围一圈三角形的重心围成的区域就是这个顶点的Barycentric cell，一个顶点周围一圈三角形外心围成的区域就是这个顶点的Voronoi cell
对于Barycentric cell，根据重心的性质很快得出局部区域面积是这一圈三角形面积的$\frac{1}{3}$
对于Voronoi cell，分开计算每个三角形中的区域并求和，以下面这个三角形为例，假设voronoi区域与该三角形的交是四边形AEDF。



$R = DA = DB = DC = \frac{abc}{4S}$

$\angle ADE = \angle ACB$ (圆心角等于圆周角的2倍)
$DE = AD * cos\angle ADE = AD * cos\angle ACB = AD * \frac{a^2+b^2-c^2}{2ab}$
所以 $S_{\triangle ADE} = \frac{1}{2}AE * DE = \frac{1}{4}AB * AD * cos\angle ADE = \frac{1}{4}AE * \frac{abc}{4S} * \frac{a^2+b^2-c^2}{2ab}$
同理可以算出$S_{\triangle ADF}$
这样就求出了每一个三角形中的voronoi cell的面积。

3 Code

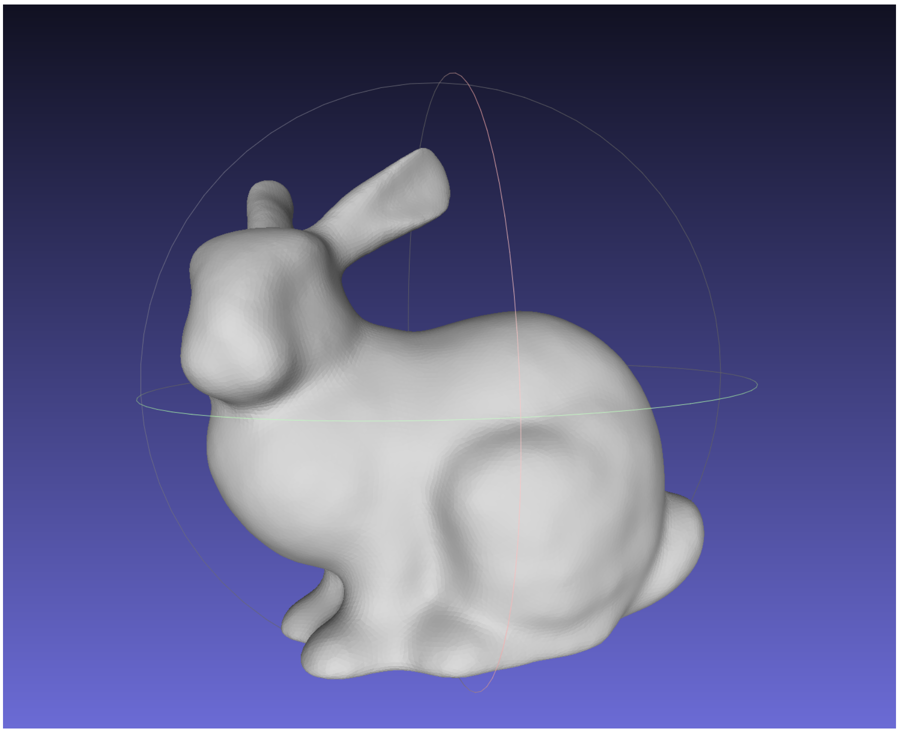
在LaplacianSmoothing.py中实现了一个类，主程序运行时每次迭代调用Compute_D和Compute_A，进而计算出拉普拉斯矩阵L，这里的D实际上是上述提到的D的逆，这么做是为了编码方便，防止直接计算逆矩阵。构造出拉普拉斯矩阵后，使用scipy.sparse.linalg中的spsovle方法求解稀疏矩阵方程组$Ax = b$。

在构建D时，采用Uniform Laplacian只用统计与该点相邻的元素个数；采用Contangent Laplacian要根据mesh中的信息，找到与该顶点相邻的面，计算局部区域的面积A_i。

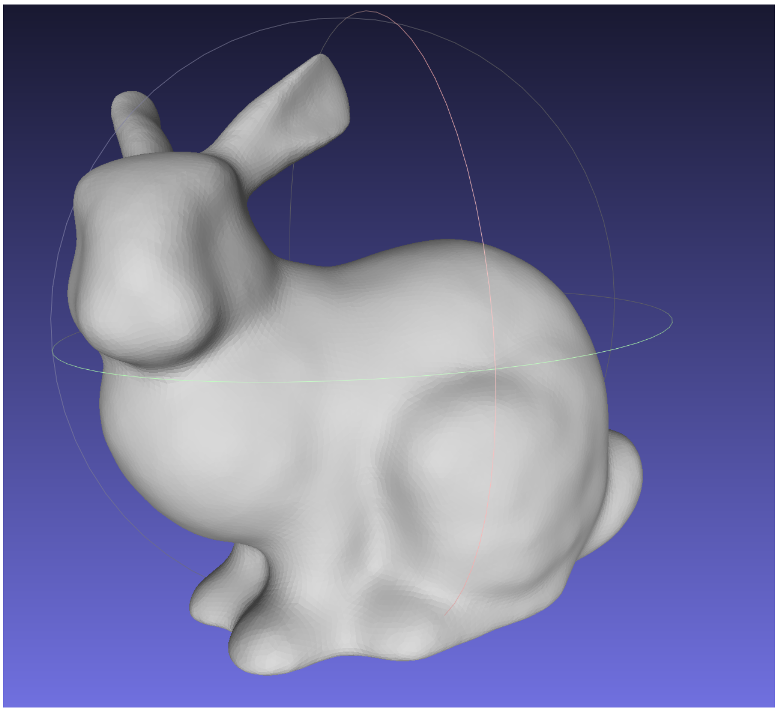
在构建A时，采用Uniform Laplacian只用在矩阵中，把将与当前顶点相邻点的位置设为1，对角线设为相邻点个数之和的相反数；采用Contangent Laplacian要遍历相邻点，然后找到这两点决定的边构成的三角形中的另一个点，然后计算余切值，为了防止数值不稳定，我限定了余切值范围只能在-10到10，然后类似Uniform Laplacian一样处理对角线元素即可。

遇到最大的bug是，因为在smooth迭代中，三角形的连接关系不会改变，但是三角形的形状几乎不会保持不变，所以Uniform Laplacian矩阵一经算出就不用在计算，每次都可以调用相同的矩阵。但是，Contangent Laplacian矩阵与三角形的具体形状相关，必须在每一次迭代时重新计算，才能得到准确结果，否则，迭代步数稍大就会发生顶点聚集，造成拓扑结构改变，算法失败，甚至迭代比较少（比如10步）得到的网格也被严重光滑，效果很差。虽然计算一次拉普拉斯矩阵的耗时很长，但是在选择余切拉普拉斯时迭代一次就可以得到比较不错的效果；在选择平均拉普拉斯时，拉普拉斯矩阵只需要计算一次，所以迭代步数我选择了20，比起原先带噪声的网格效果都不错。

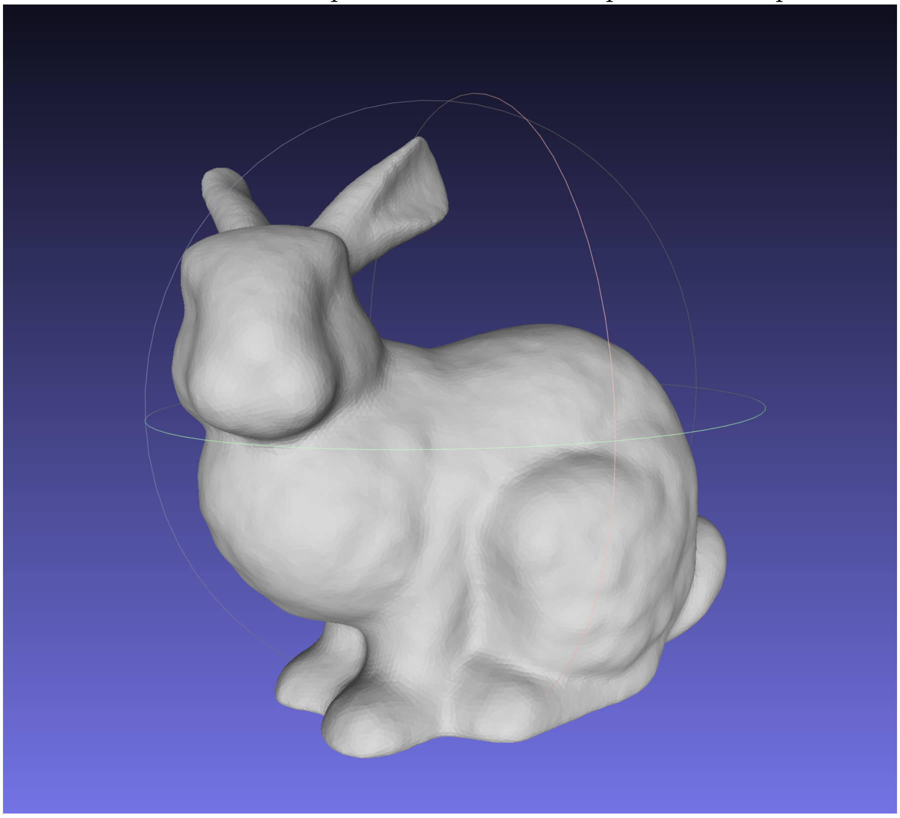
4 Results



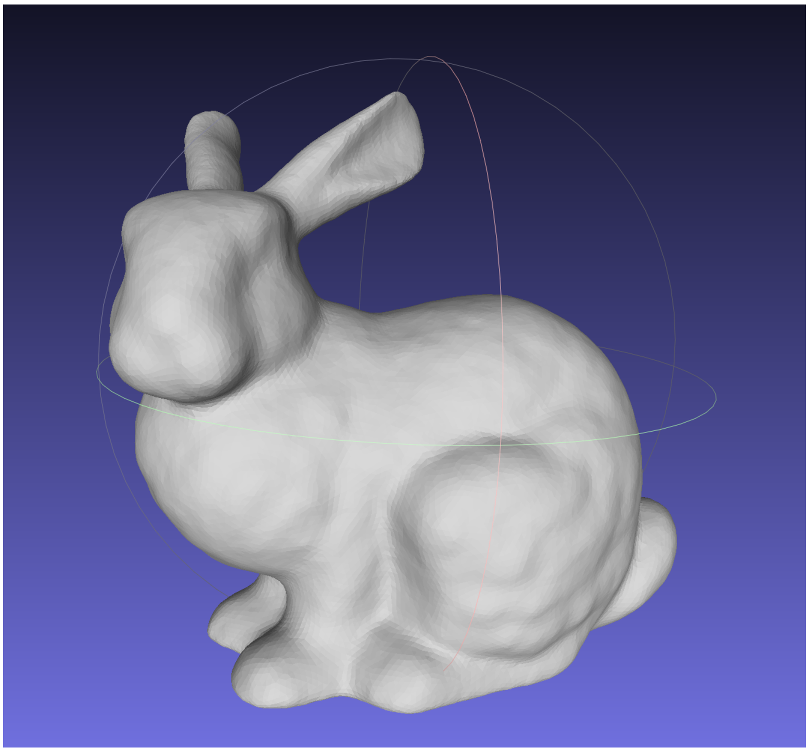
Barycentric cell + Uniform Laplacian + iteration step is 100 + step size is 0.1



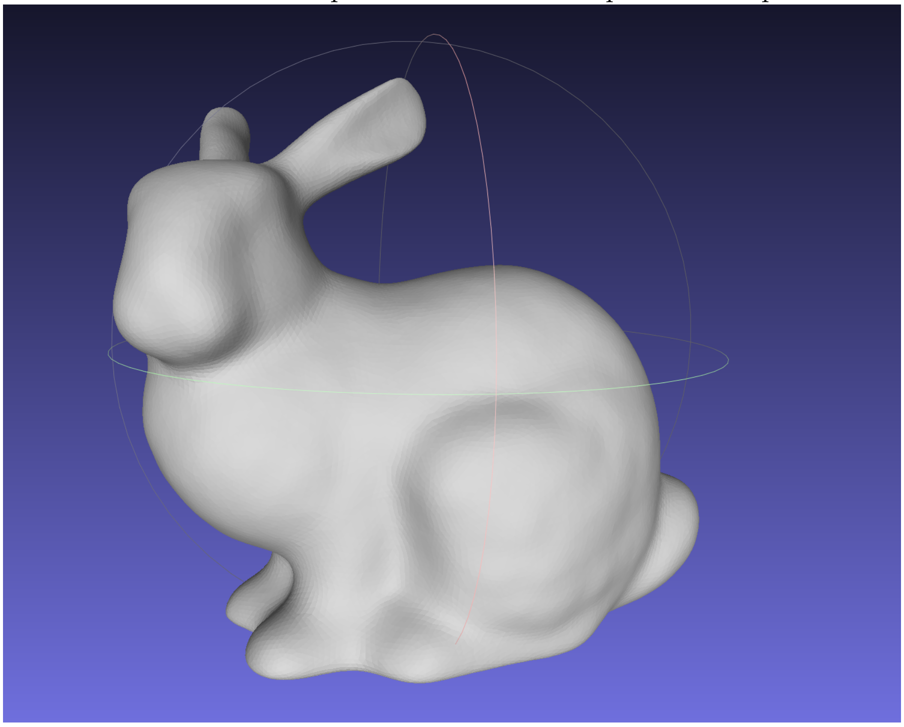
Voronoi cell + Uniform Laplacian + iteration step is 100 + step size is 0.1



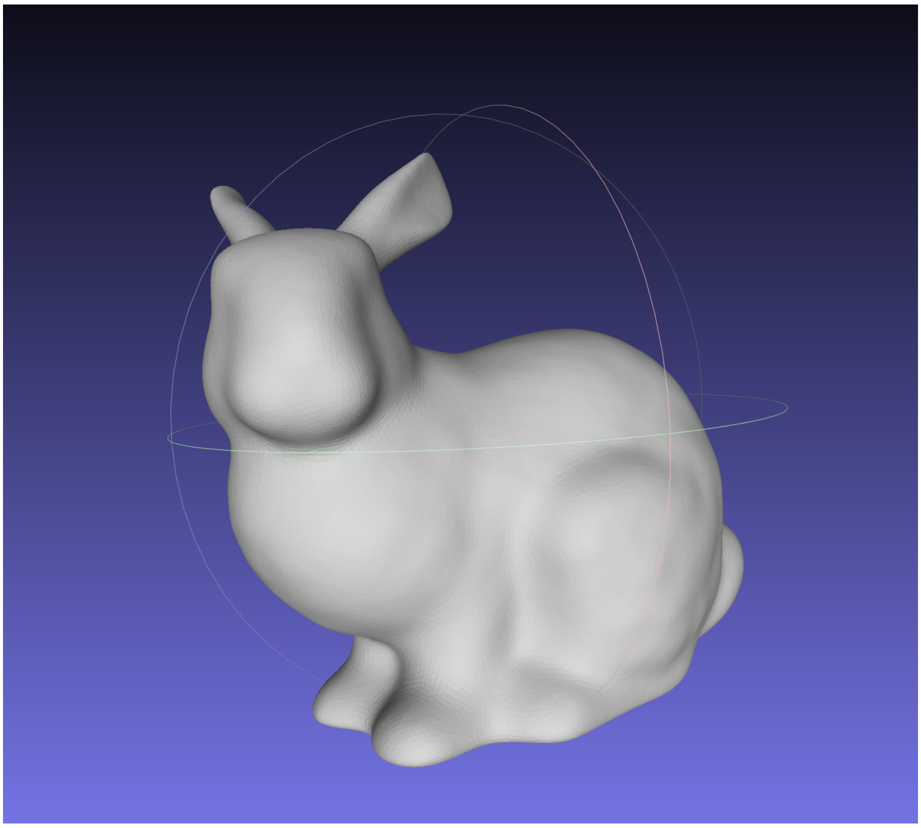
Barycentric cell + Uniform Laplacian + iteration step is 20 + step size is 0.1



Voronoi cell + Uniform Laplacian + iteration step is 20 + step size is 0.1



Barycentric cell + Cotangent Laplacian + iteration step is 1 + step size is 1e-5



Barycentric cell + Cotangent Laplacian + iteration step is 1 + step size is 1e-5

Summary

比较我实现的两种算法，Bilateral Normal Filter在运行效率和结果上我认为都优于Laplacian Smoothing，我在提交的作业中放了我认为处理效果最好的几个结果。
result1.obj 使用Laplacian Smoothing 参数为 stepsize=0.1，mode=Uniform，cell=Voronoi，iterationstep=20
result2.obj 使用Bilatera Normal Filter 参数为 sigma_r=0.5，faces_iteration=20，vertex_iteration=15

Reference

法向量双边滤波原理
投影优化
Polygon Mesh Processing