Radial Basis Function Mesh Deformation

- Project Objective

Radial basis functions (“RBFs”) are the most versatile and commonly used scattered data interpolation techniques. In this project, we have two mesh objects: **source.obj** and **target.obj.** We need to use RBF interpolation algorithm to fit coordination of vertices in **source.obj**. Using the RBF class we gotten, we need to do mesh deformation on the **target.obj** to get output.obj which should have quite similar looking compared to the original source mesh.

- Development Environment & Library:

Ubuntu 18.04 LTS and above

Anaconda2 IDE Python 2.7

Numpy & Scipy & PyMesh and its dependency libraries (nose, Eigen, PyBind11)

- Building Compilation Environment:

1. Open terminal and type in “**bash Anaconda2-xxxx.xx-Linux-x86_64.sh**” to Install Anaconda2 for Linux (Python 2.7)

2. “**source ~/.bashrc”,** “**conda init**” and “**anaconda-navigator**” to initialize conda and open anaconda GUI

3. “**conda create -n <env_name> python=2.7**” to create another conda environment

4. “**conda activate <env_name>**” to activate conda environment

5. install Spyder from anaconda navigator

6. uninstall pip, setuptools, numpy, scipy shown in the library list of <env_name> (because of version conflicts)

7. “**conda install -c conda-forge pymesh2**” to install pymesh and dependency libraries

8. run script under directory of scripts and obj files

9. “**sudo apt-get install openctm-tools**” and “**ctmviewer output.obj**” to install ctmviewer to view .obj files

- Radial Basis Function Interpolation Algorithm:

- Radial Basis Function:

A radial basis function is a real-valued function whose value depends only on the distance from the origin and the norm is usually Euclidean distance.

Commonly used types of radial basis functions include:

Gaussians: $\varphi(r) = e^{-(\varepsilon r)^2}$

Multiquadric: $\varphi(r) = \sqrt{1 + (\varepsilon r)^2}$

Linear: $\varphi(r) = r$

Cubic: $\varphi(r) = r^3$

Thin plate spline: $\varphi(r) = r^2 log(r)$

Inverse quadratic: $\varphi(r) = \frac{1}{1+(\varepsilon r)^2}$

Inverse multiquadric: $\varphi(r) = \frac{1}{\sqrt{1+(\varepsilon r)^2}}$

In this script, we use thin-plate spline function as kernel function.

- Interpolation:

Given a set of n distinct data points $\{x_j\}_{j=1}^n$ and corresponding data values $\{f_j\}_{j=1}^n$, RBF interpolant is given by s(x) = $\sum_{j=1}^{n} \lambda_j \phi(\|x - x_j\|)$, where $\phi(r)$, $r \geq 0$, is some radial function. The expansion coefficients λ_j are determined from the interpolation conditions $s(x_j) = f_j$, j = 1, …, n, which leads to the following symmetric linear system:

$[A][\lambda] = [f]$, where the entries of A are given by $a_{j,k} = \phi(\|x - x_j\|)$.

For example, consider RBF estimation in two dimensions with an additional polynomial. A polynomial in one dimension is a + bx + cx^2 + …. Truncating after the linear term gives a +bx. A similar polynomial in two dimensions is of the form a + bx + cy. Adding this to the RBF synthesis equation gives $\hat{f}(P) = \sum_{k=1}^{n} c_k \phi(\|P - P_k\|) + c_{n+1} \cdot 1 + c_{n+2} \cdot x + c_{n+3} y$

Here p = (x, y) is an arbitrary 2D point and P_k are the training points.

- Implementation of RBF on Mesh Deformation:

Firstly, we call **pymesh.load_mesh()** to load **source.obj** and **target_source.obj**. We can get their vertices and faces in form of <numpy.ndarray shape(vertices_num, 3)> by calling .**vertices** and .**faces**. Then we choose small number of vertices as control points (50 vertices in the script). In source mesh, we calculate displacements between control points and rest points in x, y, x axis separately. For the two layers of loop (for i in range(control_num, vertex_num): for j in range(0, control_num)), in each subloop, we calculate distance between each points from one control points and add it into an 1-d array, after that, we form a 2d array [[control_point_x], [control_point_y], [control_point_z], [displacement_x]] and use it to initialize **RBF** class rbfX, rbfY and rbfZ for displacement_X, displacement_Y, displacement_Z. Then we can bring target vertices (x, y, z) into rbfX, rbfY, rbfZ to calculate corresponding displacement. After that it is easy to get corresponding coordination of vertex. After all loops, we get a 3 1-d arrays of coordination.

Because **faces** arrays of two mesh are the same, the sequence of vertices are also the same. Then we can easily use the face array gotten from target or source mesh to form output mesh with **pymesh.save_mesh_raw(vertices, faces, voxel = None)**.

- Error Analysis

We use a simple loop to calculate the error:

```
for i in range(0, size):
        percentage += abs(source[i]-output[i])/source[i]
percentage /= size
```

Based on the two mesh file in the directory, the error we get is 0.04% on average for each vertex (x/y/z)

- Alternative Solution

(1) The solution shown above is calculating the coordination in three axes separately, we can also control coordination on x and y axis as independent variables and use RBF class to fitting the value of z coordination.

The problem of this solution is the error is larger than the previous one and the effect on the fringe is quite bad based on our first try **deformation.py** in our directory.

(2) This problem can also be handled by OpenMaya api instead of PyMesh library and it can provide better visualization effect.