## Problem 1.1

$$M = (V, E, F)$$

$A \in \mathbb{R}^{n \times n}$ , $D \in \mathbb{R}^{n \times n}$ & $D = A\,1_{n \times 1}$

where $1_{n \times 1}$ is a vector of all ones.

Laplacian $L = D - A$

(a) To prove: $\sum_{(i,j) \in E} \|x_i - x_j\|^2 = x^T L x$ for $x \in \mathbb{R}^n$

$$RHS = x^T L x = x^T (D - A) x$$

Data matrix $P = \begin{bmatrix} x & y & z \end{bmatrix}_{n \times 3}$ , $x, y, z \in \mathbb{R}^{n \times 1}$

→ $x$ denotes the vector containing the $x$-coordinates of all the points in data matrix $P$

→ Similarly $y, z \in R^{n \times 1}$ denotes the vector containing the $y$ & $z$ coordinates of all the points in data matrix $P$.

$$x = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^T \in \mathbb{R}^{n \times 1}$$

$x_1$: $x$-coordinate of $p_1$

$x_n$: $x$-coordinate of $p_n$

Now,

$$x^T L x = x^T (D - A) x =$$

$$= x^T D x - x^T A x$$

$$= x^T \begin{bmatrix} d_1 & & & 0 \\ & d_2 & & \\ & & \ddots & \\ 0 & & & d_n \end{bmatrix} x - x^T \begin{bmatrix} - a_1^T - \\ - a_2^T - \\ \vdots \\ - a_n^T - \end{bmatrix} x$$

$$= x^T \begin{bmatrix} d_1 x_1 \\ d_2 x_2 \\ \vdots \\ d_n x_n \end{bmatrix}_{n \times 1} - x^T \begin{bmatrix} a_1^T x \\ a_2^T x \\ \vdots \\ a_n^T x \end{bmatrix}_{n \times 1}$$

$$= \left( d_1 x_1^2 + d_2 x_2^2 + \cdots + d_n x_n^2 \right) -$$

$$\left( x_1 a_1^T x + x_2 a_2^T x + \cdots + x_n a_n^T x \right)$$

$$d_i = a_i^T 1_{n \times 1}, \quad 1 \le i \le n$$

$$= \sum_{i=1}^{n} d_i x_i^2 - \sum_{i=1}^{n} x_i a_i^T x$$

$$= \sum_{i=1}^{n} a_i^T 1_{n\times 1}\, x_i^2 - \sum_{i=1}^{n} x_i \sum_{(i,j)\in E} x_j$$

$$= \sum_{i=1}^{n} x_i^2 \sum_{(i,j)\in E} 1 - \sum_{i=1}^{n} x_i \sum_{(i,j)\in E} x_j$$

$$= \sum_{i=1}^{n} x_i^2 \sum_{\{j:\, a_{ij}=1\}} 1 - \sum_{i=1}^{n} x_i \sum_{\{j:\, a_{ij}=1\}} x_j$$

if $(i,j) \in E$, then adjacency matrix $(A)$ element $a_{ij} = 1$. Hence $(i,j) \in E$ is equivalent to $a_{ij} = 1$

Now, Let's look at the 1st term $= \sum_{i=1}^{n} x_i^2 \sum_{\{j:a_{ij}=1\}} 1$

→ for a fixed $1 \le i \le N$,

$$x_i^2 \sum_{\{j : a_{ij}=1\}} 1 = x_i^2 \times \#\text{ of } 1\text{'s in } i^{th} \text{ row of adjacency matrix } A$$

$$= x_i^2 \, d_i$$

$$\sum_{i=1}^{n} x_i^2 \sum_{\{j : a_{ij}=1\}} 1 = \sum_{(i,j) \in E} (x_i^2 + x_j^2)$$

Similarly, for the 2nd term

$$\sum_{i=1}^{n} x_i \sum_{\{j : a_{ij}=1\}} x_j = \sum_{(i,j) \in E} 2 x_i x_j$$

Thus, $x^T L x = \sum_{(i,j) \in E} (x_i^2 + x_j - 2 x_i x_j)$

$$\boxed{x^T L x = \sum_{(i,j) \in E} \|x_i - x_j\|^2}$$

Proved

For the 2nd term $\sum_{i=1}^{n} x_i \sum_{(i,j)\in E} x_j$.

For any $(i,j)$ pair in the edge set $E$, the 2nd term will have 2 $x_i x_j$ terms for any $(i,j) \in E$.

Ex-

$x_1 \sum_{(1,j)\in E} x_j = x_1 x_3 + x_1 x_6 + x_1 x_{15}$

$x_3 \sum_{(3,j)\in E} x_j = x_3 x_1 + \cdots$

$x_6 \sum_{(6,j)\in E} x_j = x_6 x_1$

$2x_1x_6$

$2x_1x_3$

Hence $\sum_{i=1}^{n} x_i \sum_{(i,j)\in E} x_j = \sum_{(i,j)\in E} 2x_i x_j$

(b) To prove: $L \in S_n^+$ (symmetric & PSD)

$$L = D - A$$

$$L^T = D^T - A^T = D - A^T$$ [since D is diagonal]

Now, in the mesh, for any edge $(i,j) \in E$, we have

$A_{ij} = A_{ji} = 1$. Otherwise, remaining elements are all zeros.

$A^T = A$ since A is an adjacency matrix

$$L^T = D - A^T = D - A = L \rightarrow L \text{ is symmetric matrix.}$$

Now, for any $x = [x_1 \; x_2 \; \dots \; x_n]^T \in \mathbb{R}^{n \times 1}$

from part (a), $x^T L x = \sum_{(i,j) \in E} \|x_i - x_j\|^2 \geq 0$

Condition holds for any mesh $M = (V, E, F)$

Thus $x^T L x \geq 0 \quad \forall \; x \in R^n$

$L$ is PSD matrix $\Rightarrow L \in S_n^+$

Proved

## Problem 1.1

(c) $P \in \mathbb{R}^{n\times 3}$ $\quad P = \begin{bmatrix} -p_1^T- \\ -p_2^T- \\ \vdots \\ -p_n^T- \end{bmatrix}_{n\times 3} = \begin{bmatrix} | & | & | \\ x & y & z \\ | & | & | \end{bmatrix}_{n\times 3}$

$x = [x_1 \quad x_2 \cdots \cdots x_n]^T \in \mathbb{R}^n$

$y = [y_1 \quad y_2 \cdots \cdots y_n]^T \in \mathbb{R}^n$

To show:

$$\sum_{(i,j)\in\mathcal{E}} \|p_i - p_j\|_2^2 = x^T L_x + y^T L_y + z^T L_z$$

$p_i = [x_i \quad y_i \quad z_i]^T \in \mathbb{R}^{3\times 1}$

$p_j = [x_j \quad y_j \quad z_j]^T \in \mathbb{R}^{3\times 1}$

$$\|p_i - p_j\|_2^2 = (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2$$

$$= \|x_i - x_j\|^2 + \|y_i - y_j\|^2 + \|z_i - z_j\|^2$$

$$\text{LHS} = \sum_{(i,j)\in\mathcal{E}} \|p_i - p_j\|_2^2$$

$$= \sum_{(i,j)\in\mathcal{E}} \left( \|x_i - x_j\|^2 + \|y_i - y_j\|^2 + \|z_i - z_j\|^2 \right)$$

$$= \quad x^T L x + y^T L y + z^T L z$$ [proved in part (a)]

$$= \text{RHS}$$

## Problem 1.2 (Normalized Laplacian)

(a)

$$L_{norm} = D^{-1}L = D^{-1}(D-A)$$

$$= I - D^{-1}A$$

Let $A = \begin{bmatrix} — a_1^T — \\ — a_2^T — \\ \vdots \\ — a_n^T — \end{bmatrix}_{n \times n}$ $a_i^T$ : rows of $A$

Let $D = \begin{bmatrix} d_1 & & 0 \\ & d_2 & \\ & \ddots & \\ 0 & & d_n \end{bmatrix}_{n \times n}$ be the diagonal matrix

with diagonal entries $d_1, d_2 \ldots\ldots d_n$

$d_i$ = degree of vertex $i$ = $a_i^T 1_{n \times 1}$

where $1_{n \times 1}$ is a $(n \times 1)$ vector containing all 1's

$$D^{-1} = \begin{bmatrix} 1/d_1 & & 0 \\ & 1/d_2 & \\ & \ddots & \\ 0 & & 1/d_n \end{bmatrix}_{n \times n}$$

To prove: sum of each row of $L_{norm} = 0$

$$L_{norm} = I_{n \times n} - D^{-1}A$$

$$D^{-1}A = \begin{bmatrix} 1/d_1 & & 0 \\ & 1/d_2 & \\ 0 & & \ddots \; 1/d_n \end{bmatrix} \begin{bmatrix} — \; a_1^T \; — \\ — \; a_2^T \; — \\ \vdots \\ — \; a_n^T \; — \end{bmatrix}$$

$$D^{-1}A = \begin{bmatrix} — \; a_1^T/d_1 \; — \\ — \; a_2^T/d_2 \; — \\ \vdots \\ — \; a_n^T/d_n \; — \end{bmatrix}_{n \times n} \rightarrow \left[ \text{row } i \text{ of } A \text{ is multiplied by } \frac{1}{d_i} \right]$$

$$L_{norm} = I - D^{-1}A$$

$$= \begin{bmatrix} 1 & & 0 \\ & 1 & \\ 0 & & \ddots \; 1 \end{bmatrix}_{n \times n} - \begin{bmatrix} — \; a_1^T/d_1 \; — \\ — \; a_2^T/d_2 \; — \\ \vdots \\ — \; a_n^T/d_n \; — \end{bmatrix}_{n \times n}$$

$$\text{Sum of } i^{th} \text{ row of } L_{norm} = 1 - \frac{a_i^T 1_{n \times 1}}{d_i}$$

$$= 1 - \frac{a_i^T 1_{n \times 1}}{a_i^T 1_{n \times 1}}$$

Sum of $i^{th}$ row of $L_{norm} = 1 - 1 = 0$

The above applies to any row of $L_{norm}$. $1 \leq i \leq n$

Thus, sum of each row of $L_{norm} = 0$

Proved

## Problem 1.2

(b) $$\Delta p_i := p_i - \underbrace{\frac{1}{|N(p_i)|} \sum_{p_j \in N(p_i)} p_j}_{\text{average of 1-ring neighborhood points}}$$

To prove: $\Delta p_i = [L_{norm} P]_i$;

$$L_{norm} P = (I - D^{-1} A) P = P - D^{-1} A P$$

$$P \in \mathbb{R}^{n \times 3}, \quad P = \begin{bmatrix} — & p_1^T & — \\ — & p_2^T & — \\ & \vdots & \\ — & p_n^T & — \end{bmatrix}_{n \times 3} = \begin{bmatrix} | & | & | \\ p_x & p_y & p_z \\ | & | & | \end{bmatrix}_{n \times 3}$$

each $p_i \in R^{3 \times 1}$ $1 \le i \le n$

$p_x, p_y, p_z \in R^n$

Now, $L_{norm} P = P - D^{-1} A P$

$$= \begin{bmatrix} — & p_1^T & — \\ — & p_2^T & — \\ & \vdots & \\ — & p_n^T & — \end{bmatrix} - \begin{bmatrix} 1/d_1 & & 0 \\ & 1/d_2 & \\ & \ddots & \\ 0 & & 1/d_n \end{bmatrix} \begin{bmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_n^T & — \end{bmatrix} P$$

$$L_{norm} P = \begin{bmatrix} — p_1^T — \\ — p_2^T — \\ \vdots \\ — p_n^T — \end{bmatrix}_{n\times 3} - \begin{bmatrix} — a_1^T/d_1 — \\ — a_2^T/d_2 — \\ \vdots \\ — a_n^T/d_n — \end{bmatrix}_{n\times n} \begin{bmatrix} — p_1^T — \\ — p_2^T — \\ \vdots \\ — p_n^T — \end{bmatrix}_{n\times 3}$$

$$p_x = \begin{bmatrix} p_{1x} & p_{2x} & \cdots & p_{nx} \end{bmatrix}^T$$

$$p_y = \begin{bmatrix} p_{1y} & p_{2y} & \cdots & p_{ny} \end{bmatrix}^T$$

$$p_z = \begin{bmatrix} p_{1z} & p_{2z} & \cdots & p_{nz} \end{bmatrix}^T$$

$$L_{norm} P = \begin{bmatrix} — p_1^T — \\ — p_2^T — \\ \vdots \\ — p_n^T — \end{bmatrix} - \begin{bmatrix} \frac{a_1^T p_x}{d_1} & \frac{a_1^T p_y}{d_1} & \frac{a_1^T p_z}{d_1} \\ \frac{a_2^T p_x}{d_2} & \frac{a_2^T p_y}{d_2} & \frac{a_2^T p_z}{d_2} \\ \vdots & \vdots & \vdots \\ \frac{a_n^T p_x}{d_n} & \frac{a_n^T p_y}{d_n} & \frac{a_n^T p_z}{d_n} \end{bmatrix}$$

Now for $1 \le i \le n$

$$\frac{a_i^T p_x}{d_i} = \frac{a_i^T p_x}{a_i^T 1_{n\times 1}} =$$ average of x-component of all vertices $p_j$ which are connected by an edge to vertex $p_i$

1-ring neighborhood

Similar definitions for $\frac{a_i^T p_y}{d_i}$ & $\frac{a_i^T p_z}{d_i}$

Thus,

$$\Delta p_i = [L_{norm} P]_i = p_i - \begin{bmatrix} a_i^T p_x / d_i \\ a_i^T p_y / d_i \\ a_i^T p_z / d_i \end{bmatrix}$$

$$d_i = |N(p_i)|$$

$$\Delta p_i = p_i - \frac{1}{d_i} \sum_{p_j \in N(p_i)} p_j$$

$$\boxed{\Delta p_i = p_i - \frac{1}{|N(p_i)|} \sum_{p_j \in N(p_i)} p_j}$$

Proved.

# hw2_Q1

November 17, 2022

## 1 Problem 1.3 (Chamfer, Curvature & Normal Loss)

[1]:
```
import sys
import time
import numpy as np
from tqdm.notebook import tqdm

# You can use other visualization from previous homeworks, like Open3D
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

import torch
from torch import nn
from torch.functional import F

import trimesh
print("Trimesh version:", trimesh.__version__)
```

```
Trimesh version: 3.16.2
```

[2]:
```
"""Visualization utilies."""
def show_points(points):
    fig = plt.figure(figsize=(14,8))
    ax = fig.gca(projection='3d')
    ax.set_xlim3d([-2, 2])
    ax.set_ylim3d([-2, 2])
    ax.set_zlim3d([0, 4])
    ax.scatter(points[:, 0], points[:, 2], points[:, 1])


def compare_points(points1, points2):
    fig = plt.figure(figsize=(14,8))
    ax = fig.gca(projection='3d')
    ax.set_xlim3d([-2, 2])
    ax.set_ylim3d([-2, 2])
    ax.set_zlim3d([0, 4])
```

```
    ax.scatter(points1[:, 0], points1[:, 2], points1[:, 1])
    ax.scatter(points2[:, 0], points2[:, 2], points2[:, 1])
```

[3]:
```
# define device type - cuda:0 or cpu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

### 1.0.1 Load data

[4]:
```
"""Load data."""
source_pcd = trimesh.load("../data/source.obj")
target_pcd = trimesh.load("../data/target.obj")
print("Source PCD:", source_pcd)
print("Target PCD:", target_pcd)

source_vertices = source_pcd.vertices
source_faces = source_pcd.faces
target_vertices = target_pcd.vertices
target_faces = target_pcd.faces

num_src_pts = source_vertices.shape[0]
num_tgt_pts = target_vertices.shape[0]
num_src_faces = source_faces.shape[0]
num_tgt_faces = target_faces.shape[0]
```

```
Source PCD: <trimesh.Trimesh(vertices.shape=(962, 3), faces.shape=(1920, 3))>
Target PCD: <trimesh.Trimesh(vertices.shape=(1502, 3), faces.shape=(3000, 3))>
```

### 1.0.2 Convert numpy data to torch tensor

[5]:
```
src_vtx_ten = torch.clone(torch.tensor(source_vertices)).to(device)
src_faces_ten = torch.clone(torch.tensor(source_faces)).to(device)
tgt_vtx_ten = torch.clone(torch.tensor(target_vertices)).to(device)
tgt_faces_ten = torch.clone(torch.tensor(target_faces)).to(device)
```

[6]:
```
def plot_mesh_3d(vertices, faces, plt_title):
    # Creating figure
    fig = plt.figure(figsize=(14, 9))
    ax = plt.axes(projection ='3d')

    # Creating color map
    my_cmap = cm.get_cmap('rainbow')

    surf_plot = ax.plot_trisurf(vertices[:,0], vertices[:,1], vertices[:,2],
                                triangles=faces,
                                cmap=my_cmap,
                                linewidth = 0.1,
                                antialiased = True)
```

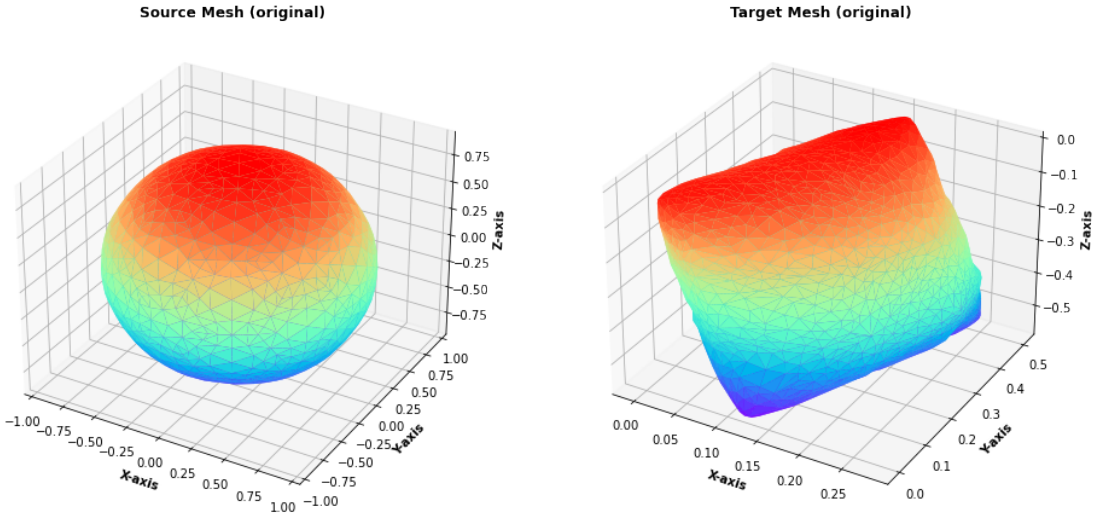
```
#     fig.colorbar(surf_plot, shrink=0.5, aspect=10)
    ax.set_title(plt_title, fontweight ='bold')
    ax.set_xlabel('X-axis', fontweight ='bold')
    ax.set_ylabel('Y-axis', fontweight ='bold')
    ax.set_zlabel('Z-axis', fontweight ='bold')
    plt.show()
```

[7]:
```
def plot_meshes_3d(vertices1, faces1, vertices2, faces2, plt_title1,␣
 ↪plt_title2):
    # Creating figure
    fig = plt.figure(figsize=(16, 10))
    # Creating color map
    my_cmap = cm.get_cmap('rainbow')

    ax = fig.add_subplot(1, 2, 1, projection='3d')
    surf_plot = ax.plot_trisurf(vertices1[:,0], vertices1[:,1], vertices1[:,2],
                                triangles=faces1,
                                cmap=my_cmap,
                                linewidth = 0.1,
                                antialiased = True)
    ax.set_title(plt_title1, fontweight ='bold')
    ax.set_xlabel('X-axis', fontweight ='bold')
    ax.set_ylabel('Y-axis', fontweight ='bold')
    ax.set_zlabel('Z-axis', fontweight ='bold')

    ax = fig.add_subplot(1, 2, 2, projection='3d')
    surf_plot = ax.plot_trisurf(vertices2[:,0], vertices2[:,1], vertices2[:,2],
                                triangles=faces2,
                                cmap=my_cmap,
                                linewidth = 0.1,
                                antialiased = True)
    ax.set_title(plt_title2, fontweight ='bold')
    ax.set_xlabel('X-axis', fontweight ='bold')
    ax.set_ylabel('Y-axis', fontweight ='bold')
    ax.set_zlabel('Z-axis', fontweight ='bold')
#     fig.colorbar(surf_plot, shrink=0.5, aspect=10)

    plt.show()
```

### 1.0.3 Display source and target point clouds

[8]:
```
plot_meshes_3d(source_vertices,
               source_faces,
               target_vertices,
               target_faces,
               "Source Mesh (original)",
```

```
                "Target Mesh (original)")
```




### 1.0.4 Function to compute adjacency matrix for any mesh

[9]:
```
def get_adjacency_mat(faces, num_pts):
    num_faces = faces.shape[0]
    adj_mat = torch.zeros((num_pts, num_pts))

    for i in range(num_faces):
        vertex_lst = faces[i]
        adj_mat[vertex_lst[0], vertex_lst[1]] = 1
        adj_mat[vertex_lst[1], vertex_lst[0]] = 1

        adj_mat[vertex_lst[0], vertex_lst[2]] = 1
        adj_mat[vertex_lst[2], vertex_lst[0]] = 1

        adj_mat[vertex_lst[1], vertex_lst[2]] = 1
        adj_mat[vertex_lst[2], vertex_lst[1]] = 1

    return adj_mat
```

[10]:
```
def get_Lnorm_mat(A_mat):
    num_pts = A_mat.shape[0]
    D_mat = torch.diag(torch.sum(A_mat, axis=1))
    L_mat = D_mat - A_mat
    D_inv = torch.linalg.inv(D_mat)
    Lnorm_mat = torch.eye(num_pts) - torch.matmul(D_inv, A_mat)

    return Lnorm_mat
```

[11]:
```
# adjacency matrix not changing for src and tgt pcd. so just one time computation
src_adj_mat = get_adjacency_mat(src_faces_ten, num_src_pts)
tgt_adj_mat = get_adjacency_mat(tgt_faces_ten, num_tgt_pts)
print(src_adj_mat.shape, tgt_adj_mat.shape)

# Lnorm matrix not changing for src and tgt pcd. so just one time computation
src_Lnorm_mat = get_Lnorm_mat(src_adj_mat)
tgt_Lnorm_mat = get_Lnorm_mat(tgt_adj_mat)
print(src_Lnorm_mat.shape, tgt_Lnorm_mat.shape)

src_Lnorm_mat = src_Lnorm_mat.type(torch.DoubleTensor).to(device)
tgt_Lnorm_mat = tgt_Lnorm_mat.type(torch.DoubleTensor).to(device)

# deltaP not changing for tgt pcd. so just one time computation
src_deltaP = torch.matmul(src_Lnorm_mat, src_vtx_ten)
tgt_deltaP = torch.matmul(tgt_Lnorm_mat, tgt_vtx_ten)
```

```
torch.Size([962, 962]) torch.Size([1502, 1502])
torch.Size([962, 962]) torch.Size([1502, 1502])
```

### 1.0.5 Function to compute chamfer loss

[12]:
```
# helper functions for computing Chamfer distance
def bpdist2(feature1, feature2, data_format='NWC'):
    """This version has a high memory usage but more compatible(accurate) with optimized Chamfer Distance."""
    if data_format == 'NCW':
        diff = feature1.unsqueeze(3) - feature2.unsqueeze(2)
        distance = torch.sum(diff ** 2, dim=1)
    elif data_format == 'NWC':
        diff = feature1.unsqueeze(2) - feature2.unsqueeze(1)
        distance = torch.sum(diff ** 2, dim=3)
    else:
        raise ValueError('Unsupported data format: {}'.format(data_format))
    return distance
```

[13]:
```
# params:
#   xyz1: 1 x points x 3, point cloud 1 (pc1)
#   xyz2: 1 x points x 3, point cloud 2 (pc2)
# output:
#   dist1: 1 x points in point cloud 1, the nearest neighbor distance for each point of pc1 to pc2
#   idx1: 1 x points in point cloud 1, for each point of pc1, index of the nearest neighbor in pc2
#   dist2: 1 x points in point cloud 2, the nearest neighbor distance for each point of pc2 to pc1
```

```
#   idx2: 1 x points in point cloud 2, for each point of pc2, index of the␣
 ↪nearest neighbor in pc1

def Chamfer_distance_torch(xyz1, xyz2, data_format='NWC'):
    assert torch.is_tensor(xyz1) and xyz1.dim() == 3
    assert torch.is_tensor(xyz2) and xyz2.dim() == 3
    if data_format == 'NCW':
        assert xyz1.size(1) == 3 and xyz2.size(1) == 3
    elif data_format == 'NWC':
        assert xyz1.size(2) == 3 and xyz2.size(2) == 3
    distance = bpdist2(xyz1, xyz2, data_format)
    dist1, idx1 = distance.min(2)
    dist2, idx2 = distance.min(1)
    return dist1, idx1, dist2, idx2
```

[14]:
```
def compute_chamfer_loss(src_vertices, tgt_vertices):
    dist1, idx1, dist2, idx2 = Chamfer_distance_torch(src_vertices.
 ↪unsqueeze(0), tgt_vertices.unsqueeze(0))
#     print(dist1.shape, idx1.shape, dist2.shape, idx2.shape)
    chamfer_loss = torch.sum(torch.square(dist1)) + torch.sum(torch.
 ↪square(dist2))

    return chamfer_loss
```

[15]:
```
# original chamfer loss written by me

# def compute_chamfer_loss(src_vertices, tgt_vertices):
#     num_src_pts = src_vertices.shape[0]
#     num_tgt_pts = tgt_vertices.shape[0]

#     chamfer_loss = 0

#     for i in range(num_src_pts):
#         dists = torch.square(torch.linalg.norm(tgt_vertices -␣
 ↪src_vertices[i], axis=1))
#         chamfer_loss += torch.min(dists)

#     for j in range(num_tgt_pts):
#         dists = torch.square(torch.linalg.norm(src_vertices -␣
 ↪tgt_vertices[j], axis=1))
#         chamfer_loss += torch.min(dists)

#     return chamfer_loss
```

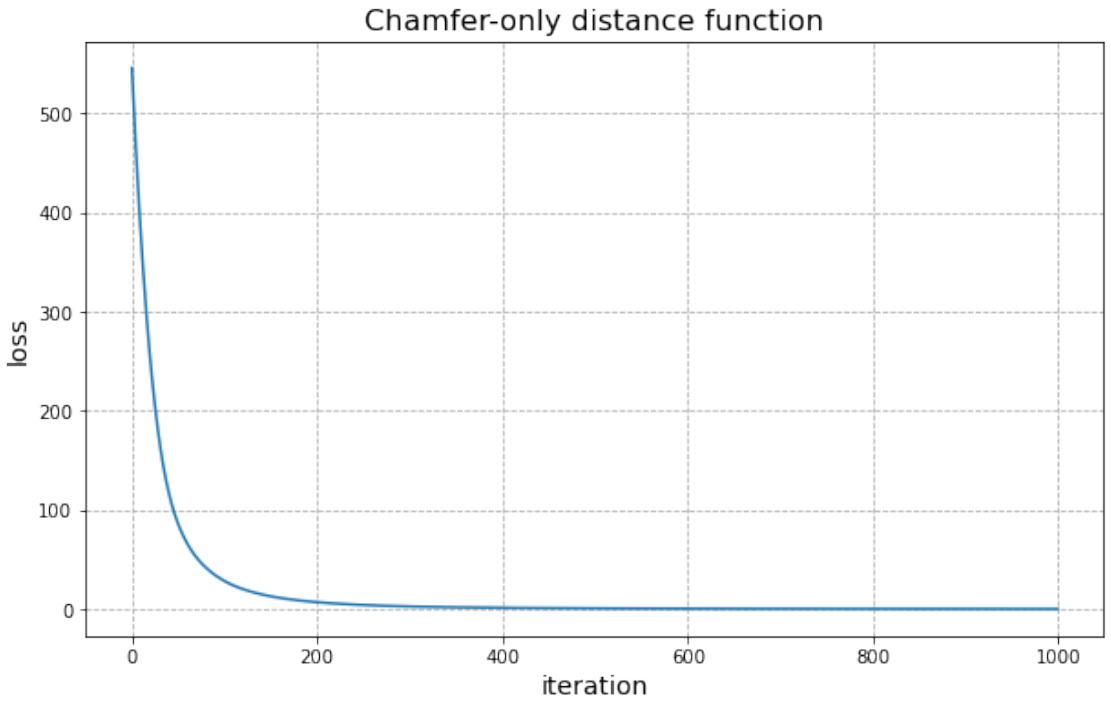
### 1.0.6 Below code iterates and deforms the source vertices towards target mesh using Chamfer Loss

[16]:
```
chamfer_losses = []
num_itr = 1000

weights_chamfer = nn.Parameter(torch.clone(src_vtx_ten)).to(device)

# Instantiate optimizer
optimizer = torch.optim.Adam([weights_chamfer], lr=0.005)

for _ in tqdm(range(num_itr)):
    loss = compute_chamfer_loss(weights_chamfer, tgt_vtx_ten)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    chamfer_losses.append(loss.item())

deformed_chamfer_vertices = weights_chamfer.cpu().detach().numpy()
```

```
  0%|          | 0/1000 [00:00<?, ?it/s]
```

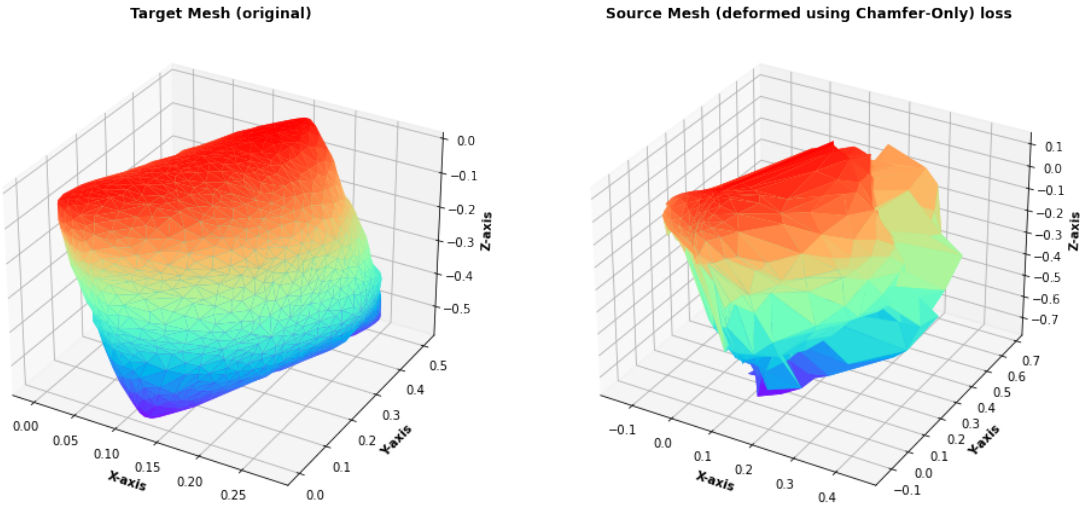
### 1.0.7 Plot the variation of chamfer distance function with optimization iteration

[17]:
```
fig = plt.figure(figsize=(10, 6))
plt.plot(chamfer_losses)
plt.grid(linestyle='--')
plt.title("Chamfer-only distance function", fontsize=16)
plt.xlabel("iteration", fontsize=14)
plt.ylabel("loss", fontsize=14)
plt.show()
```




### 1.0.8 Result: Comparison between Target Mesh and Deformed Mesh using Chamfer Loss

[18]:
```
plot_meshes_3d(target_vertices,
               target_faces,
               deformed_chamfer_vertices,
               source_faces,
               "Target Mesh (original)",
               "Source Mesh (deformed using Chamfer-Only) loss")
```



### 1.0.9 Describe what has happened in the deformation process by language

- The source mesh is spherical (slightly elliptical) in shape, whereas the target mesh is rectangular at the boundaries but slighly spherical at the middle.
- The chamfer loss measures the sum of the squared distances of each point in the source set to its nearest neighbour in the target set and vice-versa. The objective function in the part (a) optimization problem is the chamfer loss itself whereas the vertices of the source mesh are the optimization variables.
- The adam optimizer optimizes the vertices to minimize the chamfer distance loss. Therefore, for each point in source mesh, the distance to nearest point in the target mesh is minimized which results in source vertices moving more closer to their corresponding nearest neighbors in target mesh. The loss value saturates near 0 after which we do not see any significant improvement.
- The result using only chamfer loss is far from perfect. But, we can see that the mesh is being deformed in the direction of target mesh shape.

### 1.0.10 Function to compute Curvature + Chamfer loss

[19]:
```
def compute_curvature_loss(src_vertices, tgt_vertices):
    dist1, idx1, dist2, idx2 = Chamfer_distance_torch(src_vertices.
 ↪unsqueeze(0), tgt_vertices.unsqueeze(0))
    chamfer_loss = torch.sum(torch.square(dist1)) + torch.sum(torch.
 ↪square(dist2))

    # compute deltaP for src vertices / weight matrix
    src_deltaP = torch.matmul(src_Lnorm_mat, src_vertices)

    # curvature + normal loss
    deltaP_src = torch.sum(torch.square(torch.linalg.norm(src_deltaP -␣
 ↪tgt_deltaP[idx1], axis=1)))
    deltaP_tgt = torch.sum(torch.square(torch.linalg.norm(tgt_deltaP -␣
 ↪src_deltaP[idx2], axis=1)))

    return chamfer_loss + deltaP_src + deltaP_tgt
```

[20]:
```
# code written by me

# def compute_curvature_loss(src_vertices, tgt_vertices):
#     num_src_pts = src_vertices.shape[0]
#     num_tgt_pts = tgt_vertices.shape[0]

#     # compute deltaP for src vertices / weight matrix
#     src_deltaP = torch.matmul(src_Lnorm_mat, src_vertices)
```

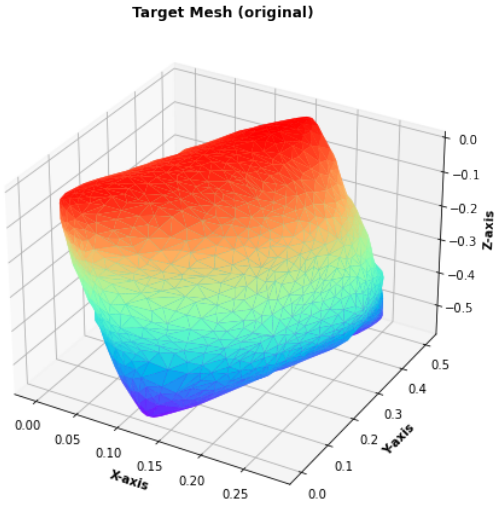
```
#     chamfer_loss = 0
#     min_idx_lst = []

#     for i in range(num_src_pts):
#         dists = torch.square(torch.linalg.norm(tgt_vertices -␣
 ↪src_vertices[i], axis=1))
#         min_idx_lst.append(torch.argmin(dists))

#         chamfer_loss += torch.min(dists)

#     deltaP_src = torch.sum(torch.square(torch.linalg.norm(src_deltaP -␣
 ↪tgt_deltaP[min_idx_lst], axis=1)))

#     min_idx_lst = []

#     for j in range(num_tgt_pts):
#         dists = torch.square(torch.linalg.norm(src_vertices -␣
 ↪tgt_vertices[j], axis=1))
#         min_idx_lst.append(torch.argmin(dists))

#         chamfer_loss += torch.min(dists)

#     deltaP_tgt = torch.sum(torch.square(torch.linalg.norm(tgt_deltaP -␣
 ↪src_deltaP[min_idx_lst], axis=1)))

#     return chamfer_loss + deltaP_src + deltaP_tgt
```

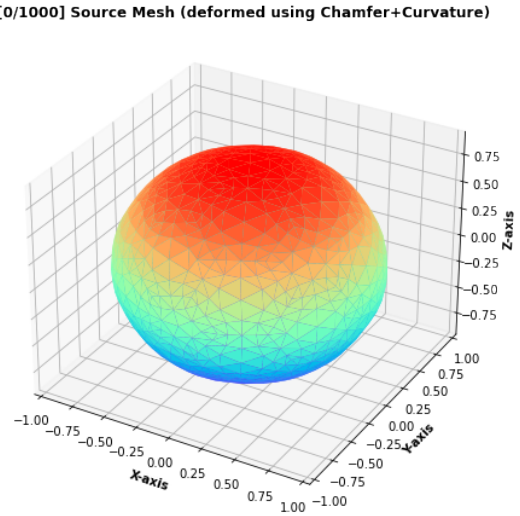
### 1.0.11 Below code iterates and deforms the source vertices towards target mesh using Curvature+Normal+Chamfer Loss

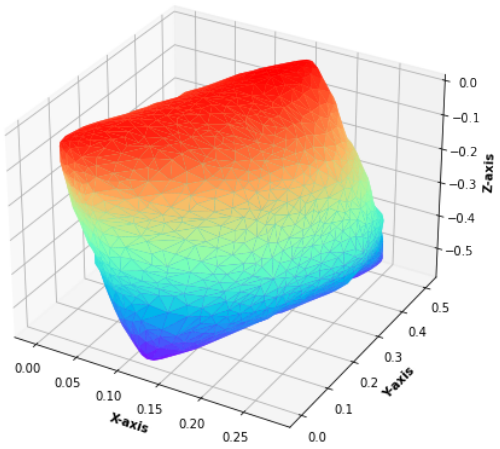
[21]:
```
curv_losses = []
num_itr = 1000

weights_curv = nn.Parameter(torch.clone(src_vtx_ten)).to(device)

# Instantiate optimizer
optimizer = torch.optim.Adam([weights_curv], lr=0.005)

disp_itr = [0, 10, 20, 40, 50, 80]
for itr in tqdm(range(num_itr)):
    loss = compute_curvature_loss(weights_curv, tgt_vtx_ten)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    curv_losses.append(loss.item())
```

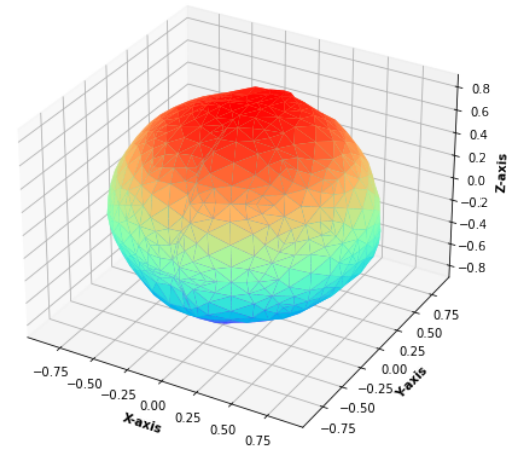
```
    if (itr % 100 == 0) or (itr in disp_itr):
        deformed_curv_vertices = weights_curv.cpu().detach().numpy()
        plot_meshes_3d(target_vertices,
                       target_faces,
                       deformed_curv_vertices,
                       source_faces,
                       "Target Mesh (original)",
                       "[{}/{}] Source Mesh (deformed using Chamfer+Curvature)".
↪format(itr, num_itr))

deformed_curv_vertices = weights_curv.cpu().detach().numpy()
```

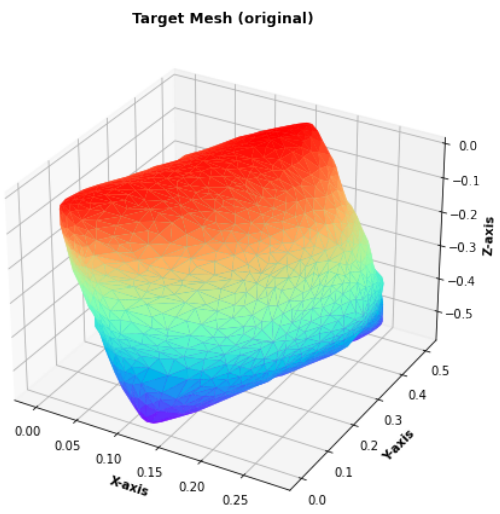
```
  0%|          | 0/1000 [00:00<?, ?it/s]
```

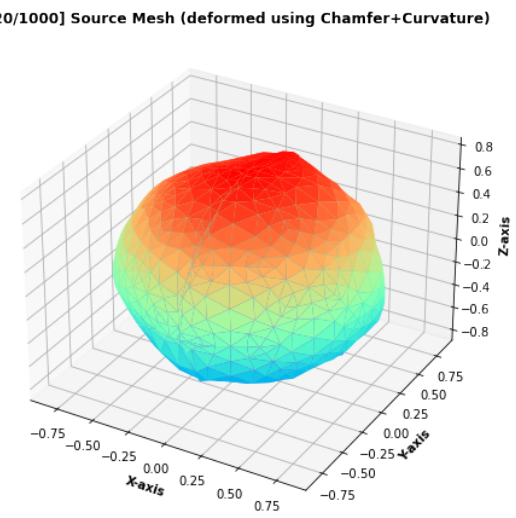


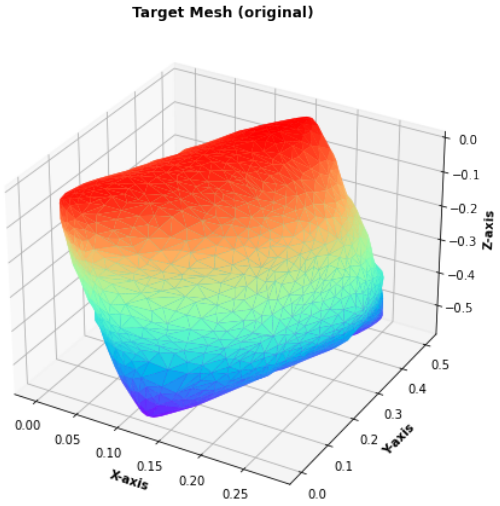



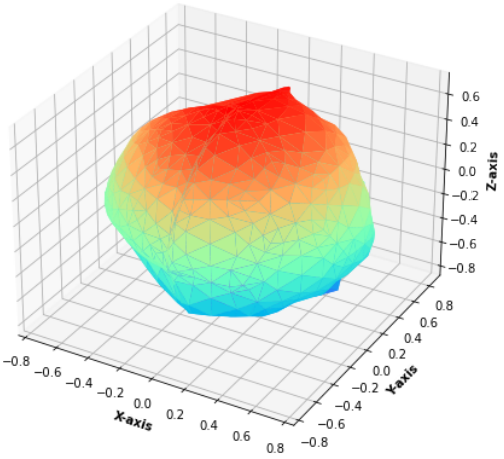



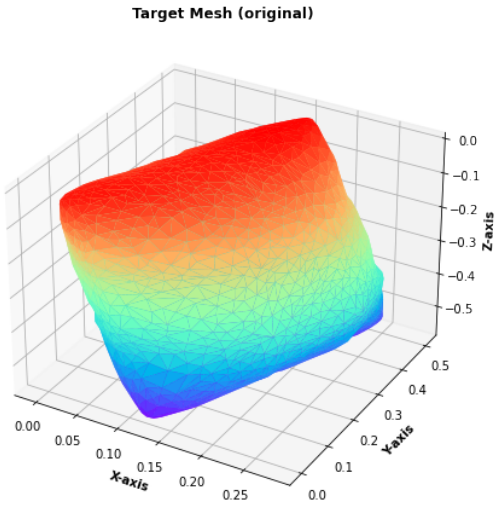



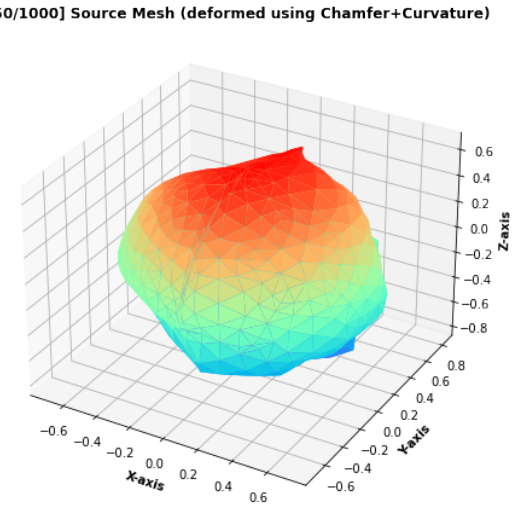

Target Mesh (original)
X-axis
Y-axis
Z-axis

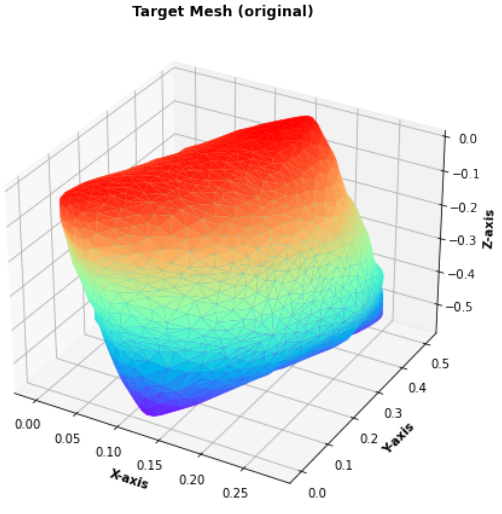

[20/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis

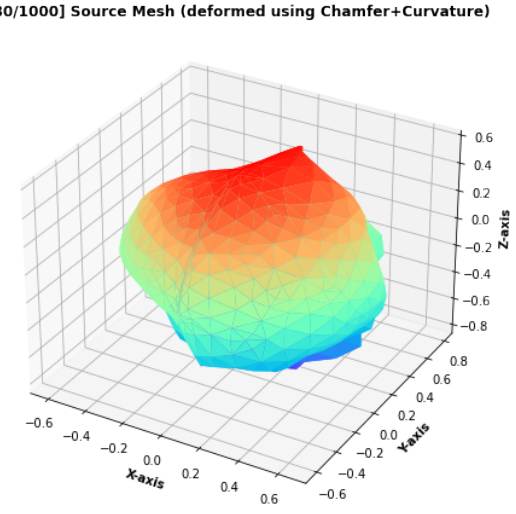

Target Mesh (original)
X-axis
Y-axis
Z-axis

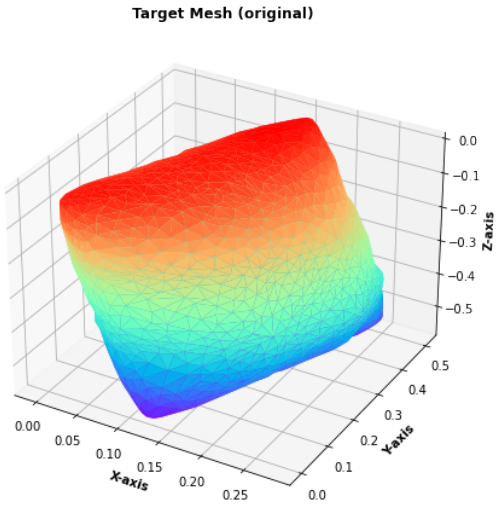

[40/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis

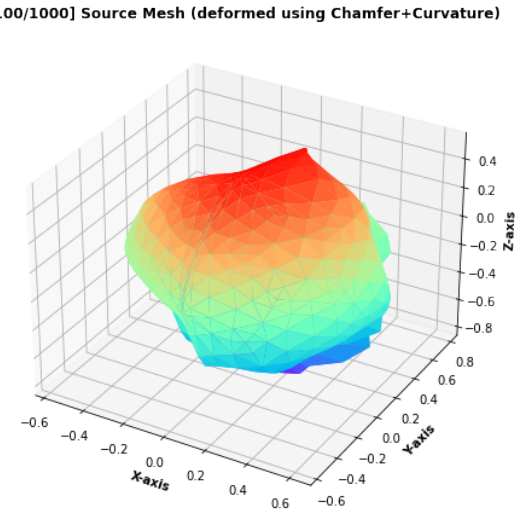

Target Mesh (original)
X-axis
Y-axis
Z-axis

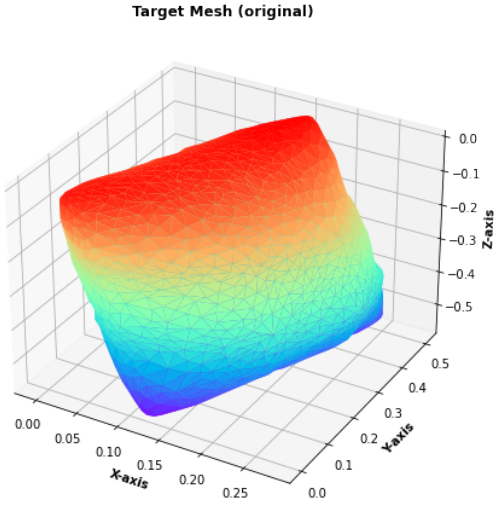

[50/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis

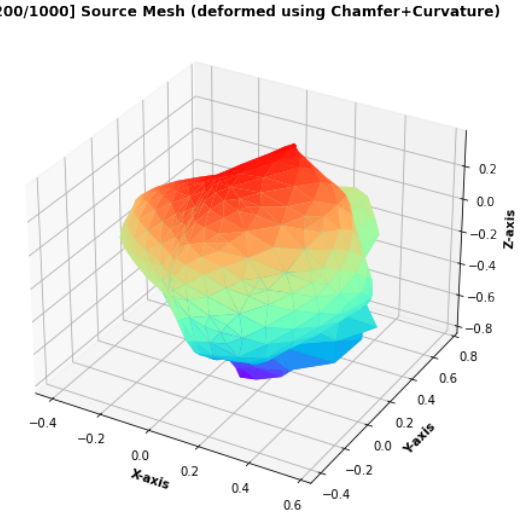

Target Mesh (original)
X-axis
Y-axis
Z-axis

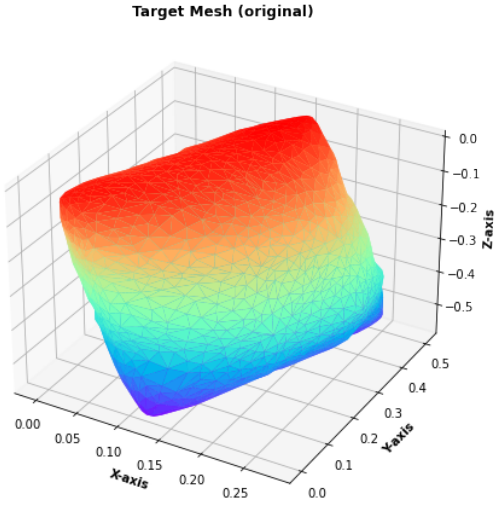

[80/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis

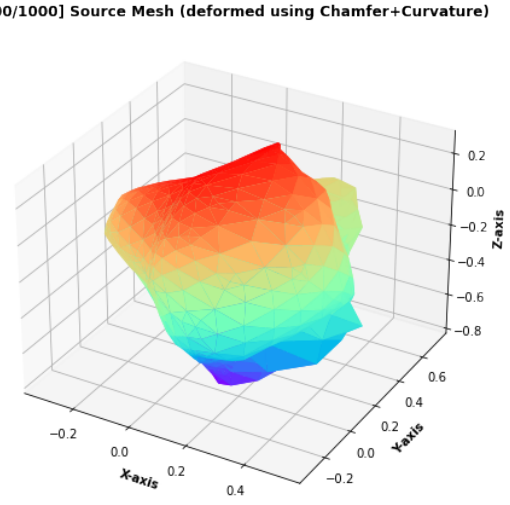

Target Mesh (original)
X-axis
Y-axis
Z-axis

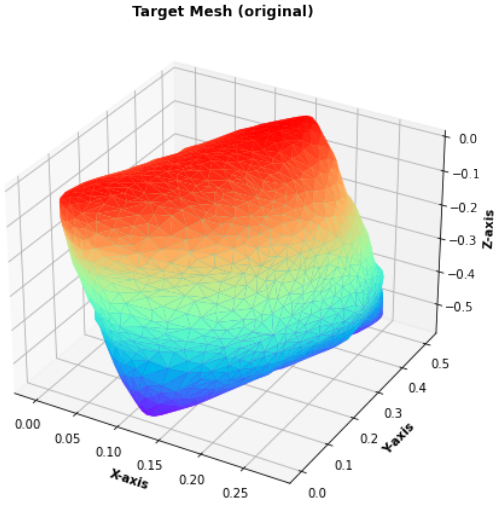

[100/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis

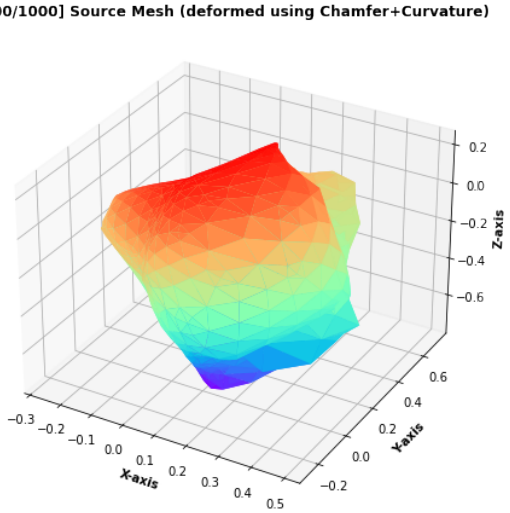

Target Mesh (original)
X-axis
Y-axis
Z-axis

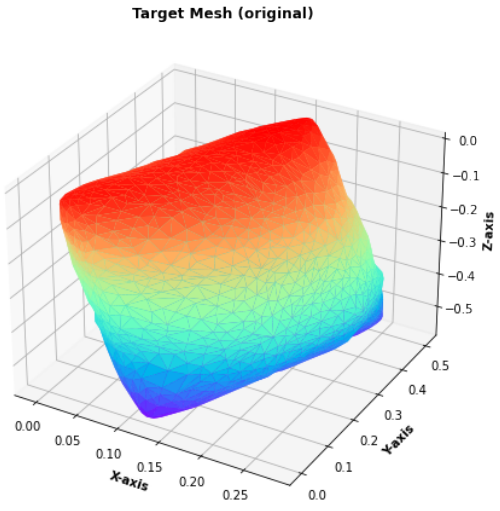

[200/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis

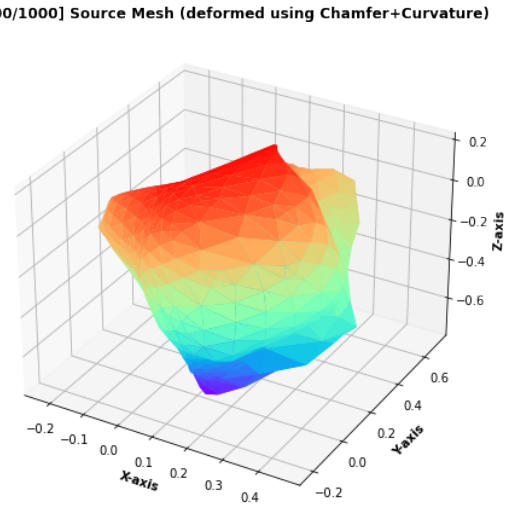

Target Mesh (original)
X-axis
Y-axis
Z-axis

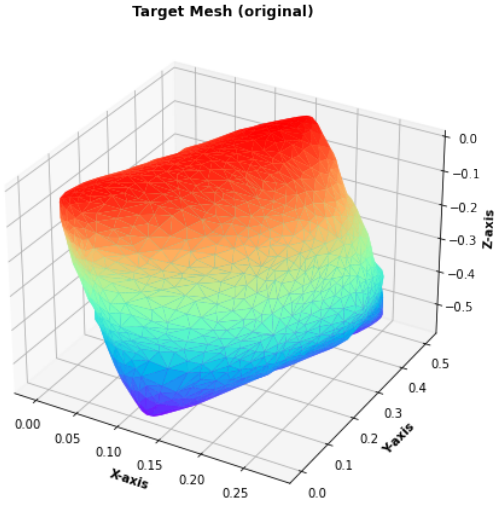

[300/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis

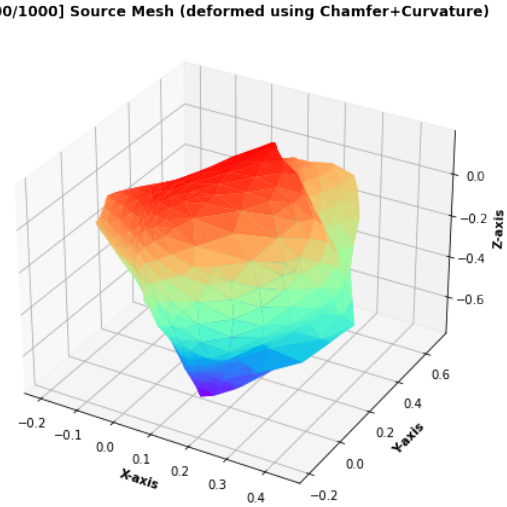

Target Mesh (original)
X-axis
Y-axis
Z-axis

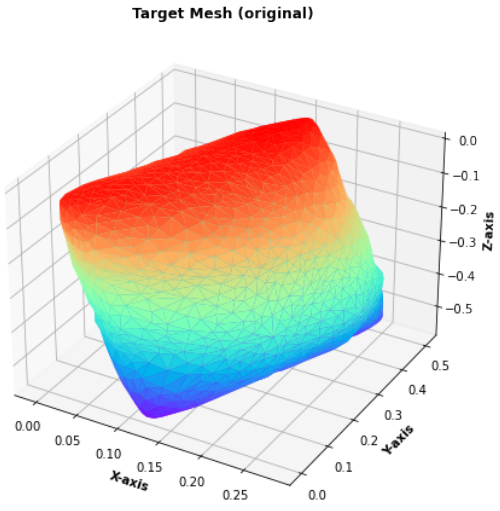

[400/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis

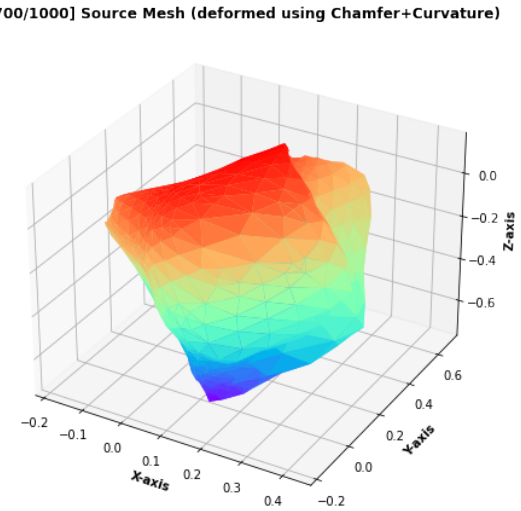

Target Mesh (original)
X-axis
Y-axis
Z-axis

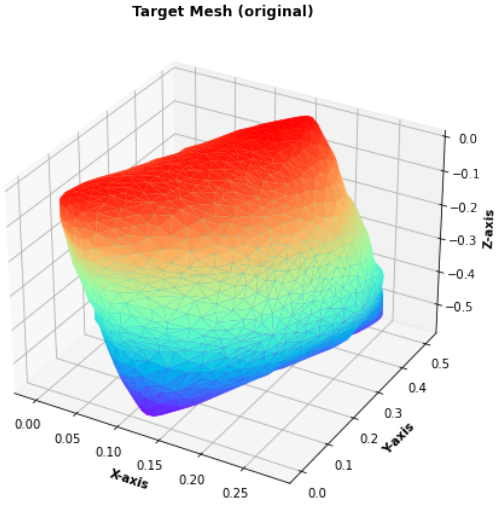

[500/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis

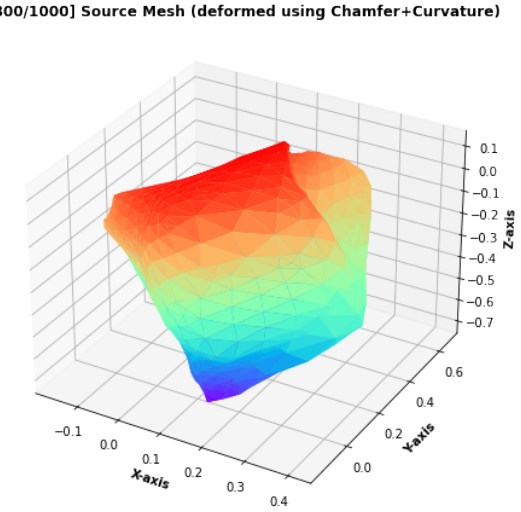

Target Mesh (original)
X-axis
Y-axis
Z-axis

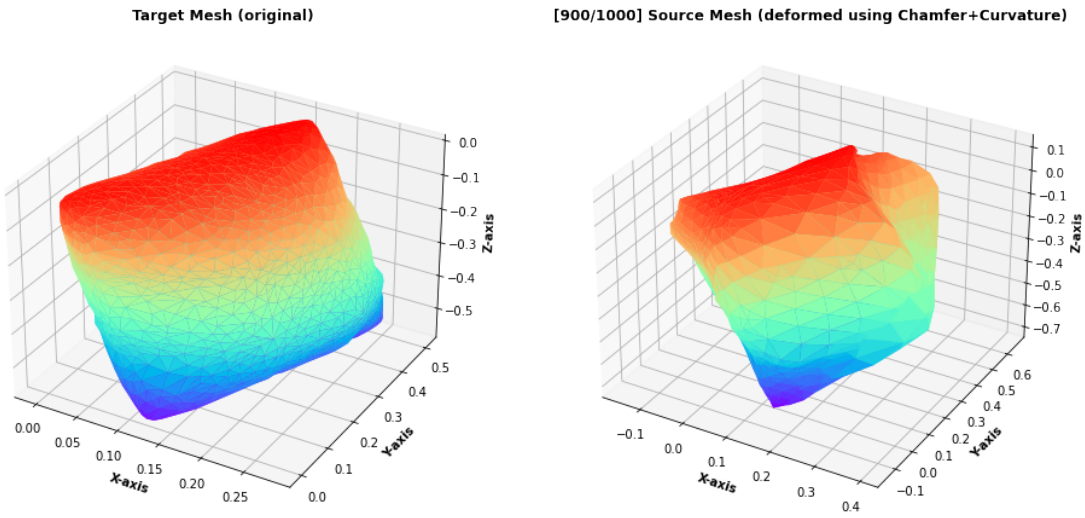

[600/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis


Target Mesh (original)
X-axis
Y-axis
Z-axis

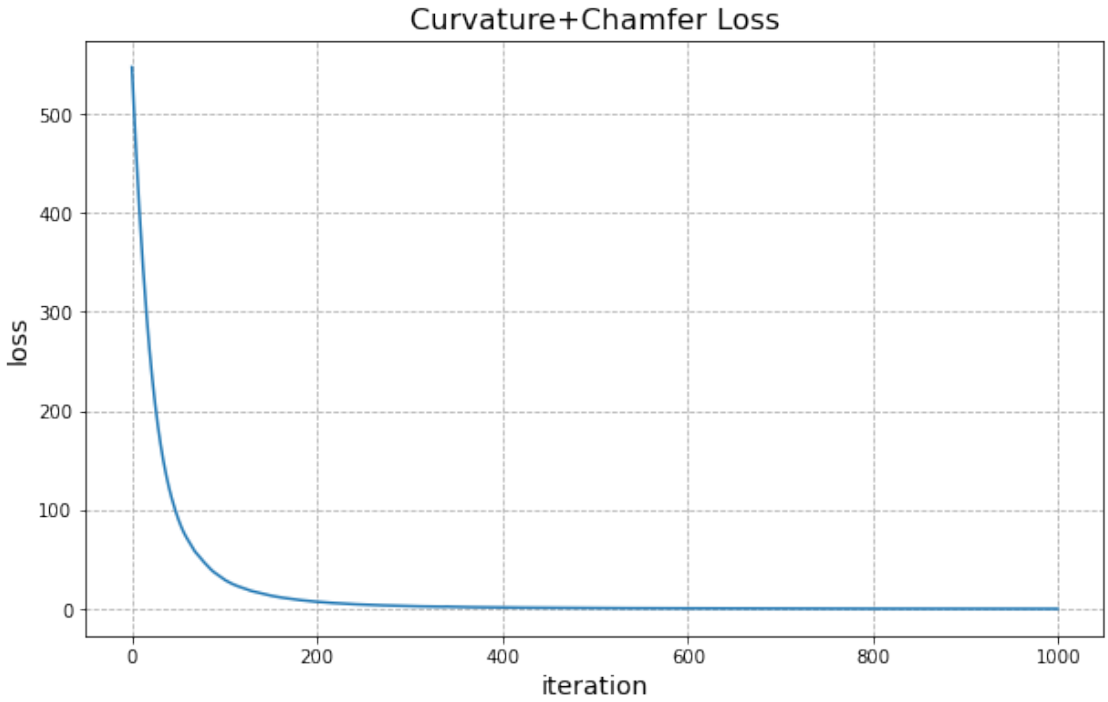

[700/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis


Target Mesh (original)
X-axis
Y-axis
Z-axis


[800/1000] Source Mesh (deformed using Chamfer+Curvature)
X-axis
Y-axis
Z-axis

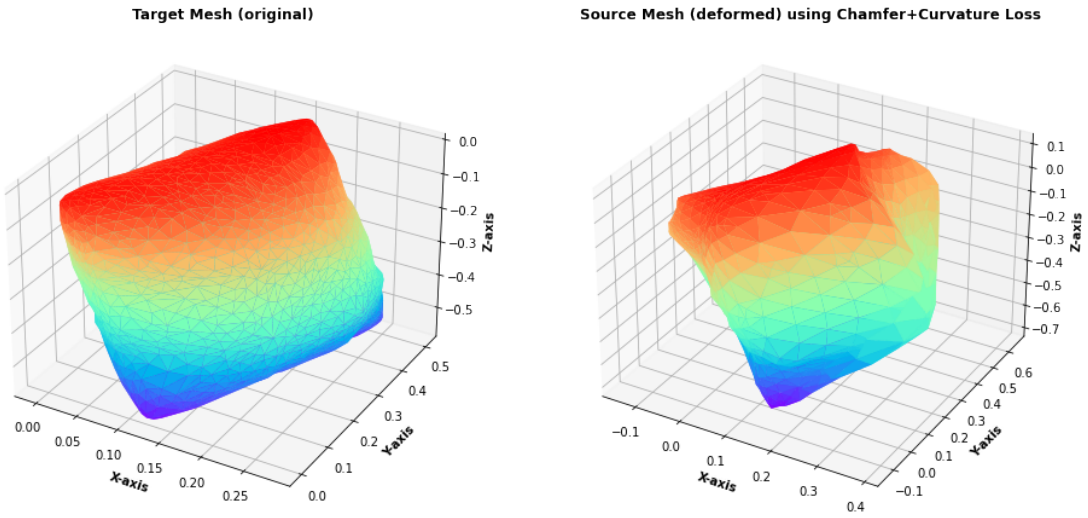



[70]:
```
fig = plt.figure(figsize=(10, 6))
plt.plot(curv_losses)
plt.grid(linestyle='--')
plt.title("Curvature+Chamfer Loss", fontsize=16)
plt.xlabel("iteration", fontsize=14)
plt.ylabel("loss", fontsize=14)
plt.show()
```



### 1.0.12 Final Result: Comparison between Target Mesh and Deformed Mesh using Curvature+Chamfer Loss

[71]:
```
plot_meshes_3d(target_vertices,
               target_faces,
               deformed_curv_vertices,
               source_faces,
               "Target Mesh (original)",
               "Source Mesh (deformed) using Chamfer+Curvature Loss")
```



# hw2_Q2

November 17, 2022

## 1 Problem 2: ICP

[1]:
```
import sys
import time
import numpy as np
from tqdm.notebook import tqdm

# You can use other visualization from previous homeworks, like Open3D
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import trimesh
print("Trimesh version:", trimesh.__version__)
```

```
Trimesh version: 3.16.2
```

[2]:
```
"""Visualization utilies."""
def show_points(points):
    fig = plt.figure(figsize=(14,8))
    ax = plt.axes(projection ='3d')
    ax.set_xlim3d([-2, 2])
    ax.set_ylim3d([-2, 2])
    ax.set_zlim3d([0, 4])
    ax.scatter(points[:, 0], points[:, 2], points[:, 1])


def compare_points(points1, points2, plt_title):
    fig = plt.figure(figsize=(14,8))
    ax = plt.axes(projection ='3d')

    ax.set_xlim3d([-2, 2])
    ax.set_ylim3d([-2, 2])
    ax.set_zlim3d([0, 4])

    ax.scatter(points1[:, 0], points1[:, 2], points1[:, 1])
    ax.scatter(points2[:, 0], points2[:, 2], points2[:, 1])

    ax.set_xlabel('X-axis', fontweight ='bold')
```

```
    ax.set_ylabel('Y-axis', fontweight ='bold')
    ax.set_zlabel('Z-axis', fontweight ='bold')
    ax.set_title(plt_title, fontweight ='bold')
```

### 1.0.1 Load source and target point cloud data

[3]:
```
"""Load data."""
source_pcd = trimesh.load("../data/banana_source.ply").vertices
target_pcd = trimesh.load("../data/banana_target.ply").vertices
gt_T = np.loadtxt("../data/banana_pose.txt")

assert(source_pcd.shape == target_pcd.shape)
print("Number of 3D points:", source_pcd.shape[0])  # =N
```

```
Number of 3D points: 16384
```

### 1.0.2 Function to compute correspondences between source and target pcd using nearest neighbor

[4]:
```
def get_correspondence(source_pcd, target_pcd):
    num_pts = source_pcd.shape[0]   # 16384
    correspondences = []

    for i in range(num_pts):
        pt = source_pcd[i,:]
        dists = np.linalg.norm(target_pcd - pt, axis=1)  # (16384,)

        min_dist_idx = np.argmin(dists)
        correspondences.append((i, min_dist_idx))

    correspondences = np.array(correspondences)   # (16384,2)
    return correspondences
```

[5]:
```
start_time = time.time()
correspondences = get_correspondence(source_pcd, target_pcd)  # (16384,2)
print("Time taken to find correspondences:", round(time.time()-start_time, 3),␣
 ↪'s')
```

```
Time taken to find correspondences: 7.684 s
```

[6]:
```
def compute_cross_covariance(source_pcd, target_pcd, correspondences):
    assert(source_pcd.shape == target_pcd.shape)

    source_corr_pcd = target_pcd[correspondences[:,1], :]
    cov_mat = np.matmul(source_corr_pcd.T, source_pcd)

#     cov = np.zeros((3, 3))
```

```
#     num_pts = source_pcd.shape[0]

#     for i in range(num_pts):
#         p_point = source_pcd[[i], :]  # (1,3)
#         q_point = target_pcd[[correspondences[i,1]], :]  # (1,3)

#         cov += np.matmul(q_point.T, p_point)

#     assert(np.sum(np.abs(cov - cov_mat)) < 1e-10)

    return cov_mat
```

[7]:
```
def verify_rot_mat(R):
    if np.abs(np.linalg.det(R) - 1.0) > 1e-10:
        print("Rotation matrix determinant error!!!:", np.linalg.det(R))
        return False

    if np.sum(np.abs(np.matmul(R, R.T) - np.eye(3))) > 1e-5 or np.sum(np.abs(np.
 ↪matmul(R.T, R) - np.eye(3))) > 1e-5:
        print("Rotation matrix RRT error!!!")
        return False

    return True
```

### 1.0.3 Function: ICP Implementation

[8]:
```
"""Implement your own ICP."""
def icp(source_pcd, target_pcd, num_itr=10):
    """Iterative closest point.

    Args:
        source_pcd (np.ndarray): [N1, 3]
        target_pcd (np.ndarray): [N2, 3]

    Returns:
        np.ndarray: [4, 4] rigid transformation to align source to target.
    """
    dists_list = []
    transform = np.eye(4)

    target_com = np.mean(target_pcd, axis=0)  # (3,)
    target_pcd_centrd = target_pcd - target_com  # (N,3)

    source_pcd_copy = np.copy(source_pcd)

    for itr in tqdm(range(num_itr)):
        # compute center of mass of the source and target point clouds
```

```
        source_com = np.mean(source_pcd_copy, axis=0)  # (3,)
        source_pcd_centrd = source_pcd_copy - source_com  # (N,3)

#         correspondences = get_correspondence(source_pcd_copy, target_pcd)
        correspondences = get_correspondence(source_pcd_centrd,␣
 ↪target_pcd_centrd)

        # compute distance between corresponding points in source and target pcd
        dist = np.sum(np.linalg.norm(source_pcd_copy -␣
 ↪target_pcd[correspondences[:,1], :], axis=1))
        dists_list.append(dist)

        cov_mat = compute_cross_covariance(source_pcd_centrd,␣
 ↪target_pcd_centrd, correspondences)  # (3,3)

        U, S, VT = np.linalg.svd(cov_mat)  # (3,3), (3,3), (3,3)
        R_est = np.matmul(U, VT)   # (3,3)
#         assert(verify_rot_mat(R_est) == True)

        t_est = target_com - np.matmul(R_est, source_com)    # (3,)

        source_pcd_copy = np.matmul(source_pcd_copy, R_est.T) + t_est   # (N,3)

        transform[:3,:3] = np.matmul(R_est, transform[:3,:3])
        transform[:3, 3] = np.matmul(R_est, transform[:3, 3]) + t_est

    return transform, dists_list
```

### 1.0.4 Functions to compute evaluation metrics - RRE and RTE

[9]:
```
"""Metric and visualization."""
def compute_rre(R_est: np.ndarray, R_gt: np.ndarray):
    """Compute the relative rotation error (geodesic distance of rotation)."""
    assert R_est.shape == (3, 3), 'R_est: expected shape (3, 3), received shape␣
 ↪{}.'.format(R_est.shape)
    assert R_gt.shape == (3, 3), 'R_gt: expected shape (3, 3), received shape␣
 ↪{}.'.format(R_gt.shape)
    # relative rotation error (RRE)
    rre = np.arccos(np.clip(0.5 * (np.trace(R_est.T @ R_gt) - 1), -1.0, 1.0))
    return rre


def compute_rte(t_est: np.ndarray, t_gt: np.ndarray):
    assert t_est.shape == (3,), 't_est: expected shape (3,), received shape {}.
 ↪'.format(t_est.shape)
```

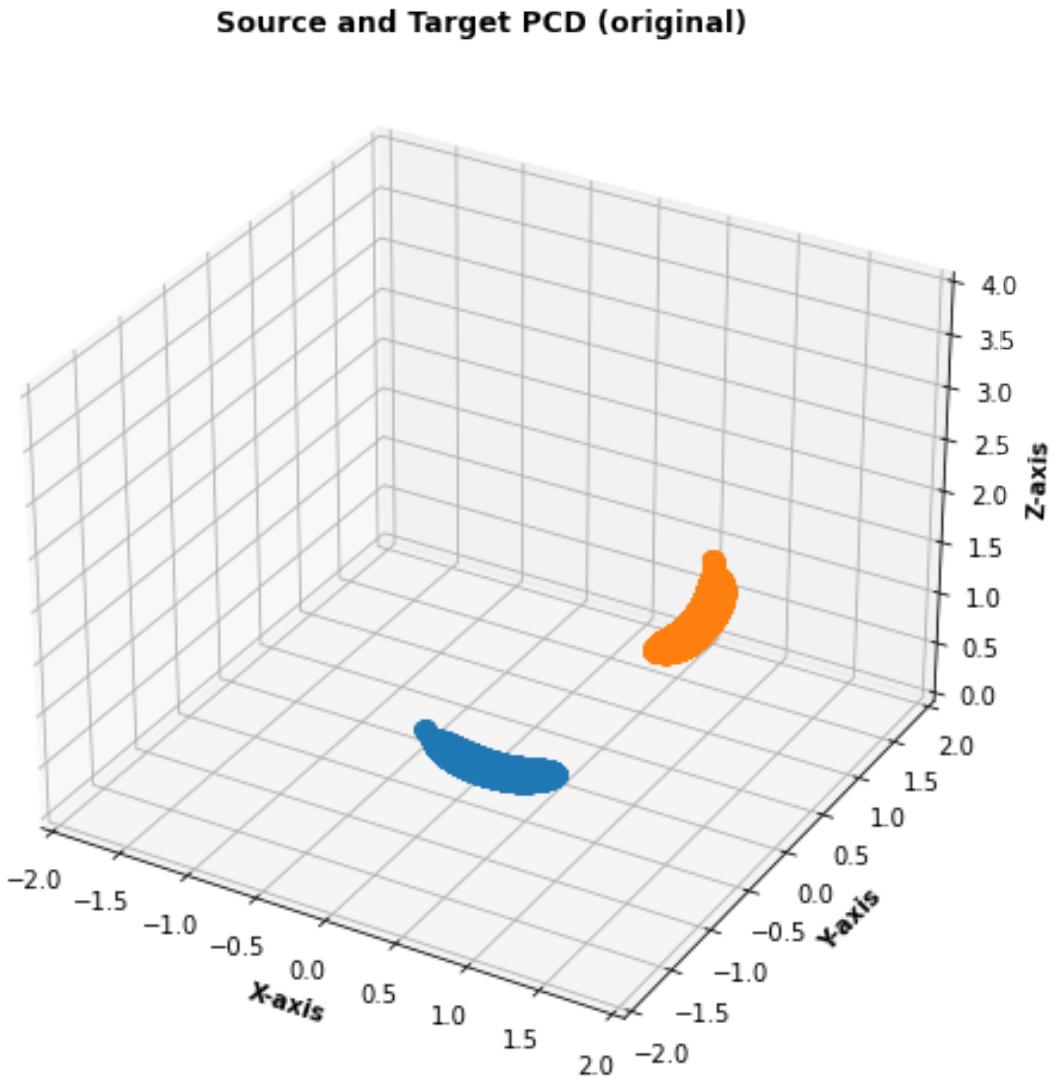
```
    assert t_gt.shape == (3,), 't_gt: expected shape (3,), received shape {}.'.
 ↪format(t_gt.shape)
    # relative translation error (RTE)
    rte = np.linalg.norm(t_est - t_gt)
    return rte
```

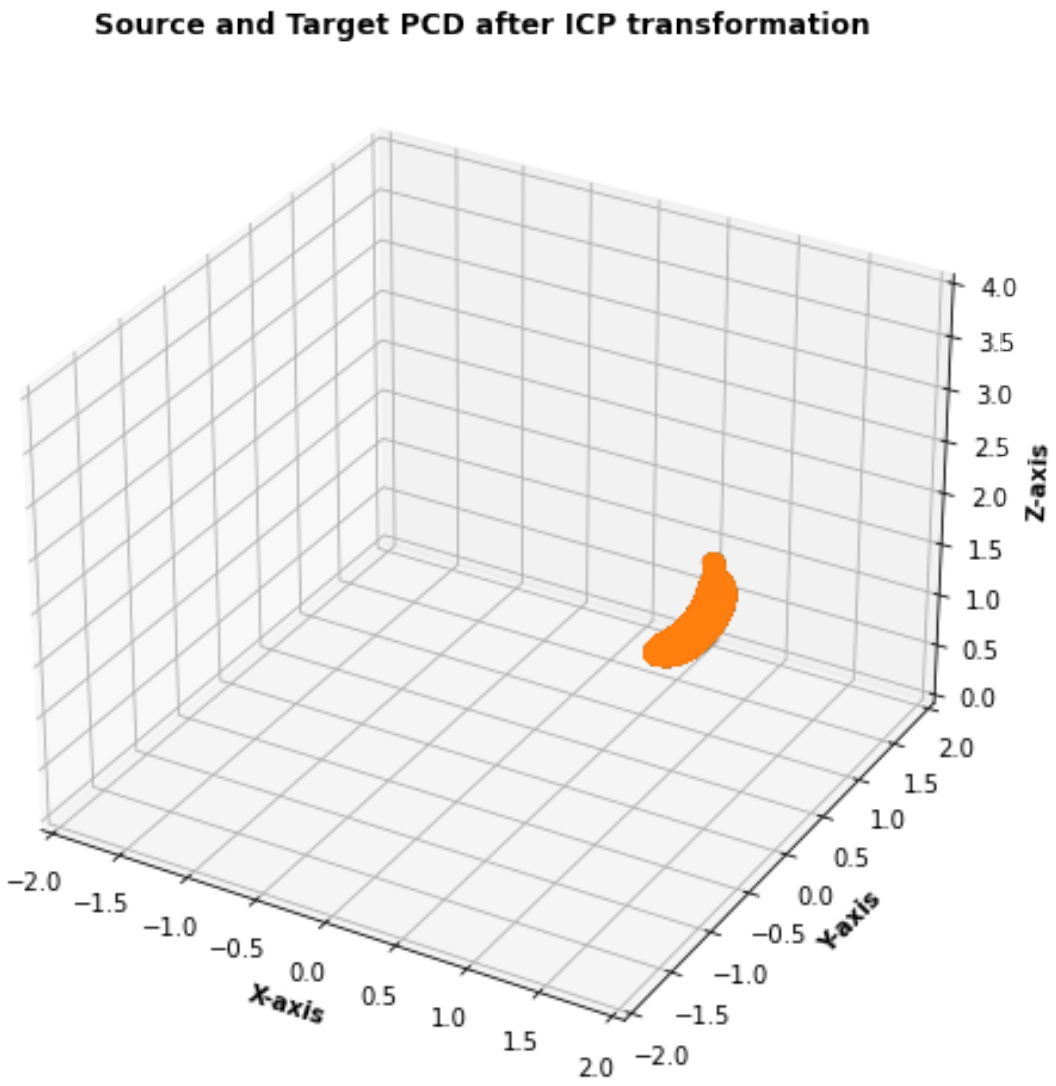
### 1.1 Visualize source and target pcd before and after ICP transformation

[10]:
```
# Visualization
transform_preICP = np.eye(4)
compare_points(source_pcd @ transform_preICP[:3, :3].T + transform_preICP[:3,␣
 ↪3],
               target_pcd,
               "Source and Target PCD (original)")

NUM_ICP_ITR = 20
transform_postICP, dists_list = icp(source_pcd, target_pcd, NUM_ICP_ITR)

rre = np.rad2deg(compute_rre(transform_postICP[:3, :3], gt_T[:3, :3]))
rte = compute_rte(transform_postICP[:3, 3], gt_T[:3, 3])
print(f"RRE={rre}, RTE={rte}")

compare_points(source_pcd @ transform_postICP[:3, :3].T + transform_postICP[:3,␣
 ↪3],
               target_pcd,
               "Source and Target PCD after ICP transformation")
```

```
  0%|          | 0/20 [00:00<?, ?it/s]
```

```
RRE=0.0, RTE=4.069774370551601e-10
```

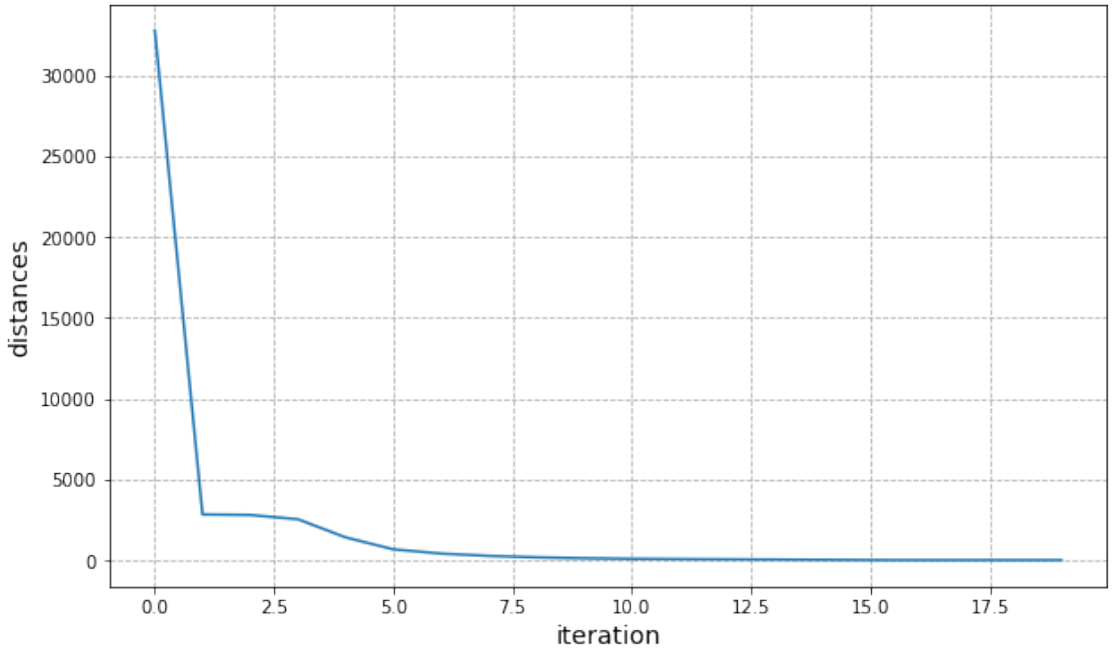

Source and Target PCD (original)
X-axis
Y-axis
Z-axis
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0




### 1.1.1 RRE = 0.0, RTE = 4.06977e-10

### 1.1.2 Plot showing variation of distance between source and target pcd with ICP iteration

[11]:
```
fig = plt.figure(figsize=(10,6))
plt.plot(dists_list)
plt.ylabel("distances", fontsize=14)
plt.xlabel("iteration", fontsize=14)
plt.grid(linestyle='--')
```




## 1.2 Problem 4 (Course Feedback)

**1. How many hours did you spend on this homework?**

- Approximately 75-100 hrs

**2. How many hours did you spend on the course each week?**

- In general, aside from the assignments, I spend roughly 7-8 hrs on this course/week (including lectures).

**3. Do you have any course related feedback?**

- No.

## 1.3 References


- https://nbviewer.org/github/niosus/notebooks/blob/master/icp.ipynb
- http://www.open3d.org/docs/release/tutorial/pipelines/icp_registration.html
- https://towardsdatascience.com/how-to-use-pytorch-as-a-general-optimizer-a91cbf72a7fb
- https://www.youtube.com/watch?v=djnd502836w


## 1.4 Acknowledgement

I discussed few questions and doubts related to this assignment with following of my classmates. I thank each of them for their valuable time and help. - Chinmay Talegoankar - Sambaran Ghosal - Xuan Tang (piazza post really helpful)

# CSE 291 Assignment-2 (Problem 3)

**Saqib Azim**
sazim@ucsd.edu
Benchmark Username: sirius1707

## 1 Method

The objective in this problem is to estimate the poses of objects in the test dataset using training data. My approach relies on the fact that we have objects with same ids as test data in the training data and we know the ground-truth pose information for the train dataset. The overall procedure can be described in following points.

- Extract all the file names in the training, test and validation data.
- Take the training data, iterate over all the scenes in the training data. For each instance of an object, extract the image coordinates in the pixel coordinate frame (PCF) using the label segmentation image.
- Using the intrinsic matrix and the depth image provided for each scene, convert each object's pixel coordinates from PCF to camera coordinate frame (CCF).
- Using the extrinsic matrix provided for each scene, transform these points in the CCF to world coordinate frame (WCF). The 3D points obtained in the WCF are then scaled by multiplying with the inverse of the scaling factor provided for each object in the scene. Now, the obtained 3D points in the WCF are rotated and translated version of their canonical form.
- We transform the points in the WCF to their canonical original form by multiplying with the inverse of the ground-truth pose provided for every object in the scene in training data.
- For each of the 79 objects present, we store their various canonical original point cloud from different scenes in a list. This acts as a dictionary later when we estimate the pose of an object in the test scene data.
- In order to speed up the training and test process, we stored the training, validation and test data in separate pickle files in a one-time computation. Later, we reload train and test pickle files during training extraction and testing and use these to load the data. This saves significant I/O time as only once we have to load images (depth, label and metadata) one-by-one directly from the memory.
- During test, we want to estimate the 6D poses of each object in all the 200 test scenes. For each object in a test scene, we follow similar process as for the training data and estimate their 3d coordinates in WCF from their 2d pixel coordinates using intrinsic, extrinsic matrix and scale factor.
- Now, the sad part is we do not have ground-truth pose information (transformation matrix from their canonical original form to rotated + translated form) for these point clouds. Here we use the list of partial point clouds for each object that we created during train process.
- For each object, there exists a list of partial point clouds created during training. We use Iterative Closest Point (ICP) algorithm to estimate the transformation between test point cloud (referred as target) and each of the training point clouds (referred to as source).
- For initial transformation provided to the ICP algorithm, we used the ground truth pose for the source point cloud in the training data. The threshold value provided to the ICP is a hyperparameter. We used thresh_value = 0.001, 0.008, 0.01, 0.02, etc.

Preprint. Under review.

- The test process is quite exhaustive as it requires performing ICP between every object's point cloud in a test scene and a list of train point clouds containing 1000's of point cloud for each object. We use ICP implementation from Open3D library as it is fast and much more efficient as compared to my own implementation.
- For time reduction and accuracy improvement (to some extent), we have uniformly downsampled the test point cloud and training point clouds and then performed ICP. The downsample factor is a hyperparameter and we tested with downsample = 4,8,16,32,64. As expected, with 64, speed is fast but accuracy is poor whereas with downsample=4, the accuracy is high but it's super slow on a CPU.
- We used the fitness parameter provided by the Open3D's ICP to find the best point cloud (maximum fitness) among the list of training point clouds.

## 2 Experiments

Results on Validation dataset:

With ICP thresh = 0.001, DOWNSAMPLE = 32, RRE = 0.7291051 and RTE = 0.00114901
With ICP thresh = 0.01, DOWNSAMPLE = 4, RRE = 0.521865 and RTE = 0.00100371
With ICP thresh = 0.01, DOWNSAMPLE = 16, RRE = 0.6926190 and RTE = 0.00253707