CSE291E-FA22

Problem 1: Deform a shape

In this problem, we will practice part of what we learned in the image-to-3D lecture for shape deformation.

1. Laplacian

Given a mesh $M = (V, E, F)$, we assume that the adjacency matrix is $A \in \mathbb{R}^{n \times n}$, $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix where $D[i, i]$ is the degree of the i-th vertex. The Laplacian matrix is defined as $L = D - A$.

Prove that:

(a) $\sum_{(i,j) \in E} \|x_i - x_j\|^2 = x^T L x$ for $x \in \mathbb{R}^n$. [1pt]

(b) $L \in \mathbb{S}_+^n$, i.e., L is a symmetric and positive semi-definite matrix. [1pt]

(c) For the data matrix $P \in \mathbb{R}^{n \times 3}$ where each row corresponds to a point in $\mathbb{R}^3$, denote the columns of P as $P = [x, y, z]$ and rows of P as $P = [p_1^T; p_2^T; \dots; p_n^T]$, show that $\sum_{(i,j) \in E} \|p_i - p_j\|^2 = x^T L x + y^T L y + z^T L z$. (hint: Use the conclusion from 1(a)) [1pt]

a)

Let us consider in the graph. Then $\sum_{i,j \in E} ||x_i - x_j|^2$ will contain the term x_i^2 $D[i, i]$ times, x_j^2 $D[j, j]$ terms and the term $x_i x_j$ will apear once with weight $-A[i, j]$ and once with weight $-A[j, i]$ but since A is symmetric, this will just occur with weight -2. So the portion of $\sum_{i,j \in E} ||x_i - x_j|^2$ containing x_i, x_j will be $D[i, i]x_i^2 + D[j, j]x_j^2 - 2A[i, j]x_i x_j$ which can be rearraged as $\begin{bmatrix} x_i & x_j \end{bmatrix} \begin{bmatrix} D[i, i] & -A[i, j] \\ -A[j, i] & D[j, j] \end{bmatrix} \begin{bmatrix} x_i \\ x_j \end{bmatrix}$. So if we consider all points $x_1, x_2, x_3, \dots, x_n$, using the above pattern, we get

$$\sum_{i,j \in E} ||x_i - x_j||^2 = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix}^T (D - A) \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix} = x^T L x$$

b)

Consider the LHS of the above property, the LHS of the equation is always positive and 0 only if the graph is empty. Hence $\sum_{i,j \in E} ||x_i - x_j||^2 \geq 0$. Hence now we have $x^T L x \geq 0$ and 0 only if $x = 0$. Hence $L \in \mathcal{S}_+^n$ i.e. L is a positive-semidefinite matrix.

c)

Let $p_i^T = \begin{bmatrix} x_i & y_i & z_i \end{bmatrix}$ and P be the collection of all points i.e. $P = \begin{bmatrix} p_1^T \\ p_2^T \\ . \\ . \\ p_n^T \end{bmatrix}$. Now

$\sum_{i,j \in E} ||p_i - p_j||^2 = \sum_{i,j \in E}(||x_i - x_j||^2 + ||y_i - y_j||^2 + ||z_i - z_j||^2) = \sum_{i,j \in E}(|x_i - x_j||^2) + \sum_{i,j \in E}(||y_i - y_j||^2) + \sum_{i,j \in E}(||z_i - z$
$\implies \sum_{i,j \in E} ||p_i - p_j||^2 = x^T L x + y^T L y + z^T L z$
using the part a result along each coordinate.

2. Normalized Laplacian

Normalized Laplacian is defined as the normalized version of the Laplacian matrix above:

$$L_{norm} = D^{-1} L$$

(a) Prove that the sum of each row of L_{norm} is 0. [1pt]

a)

Observe that $[A]_i$ is the connectivity of the i^{th} vertex to all other vertices, so sum of all elements in the i^{th} row of A defines the total number of connections of the i^{th} vertex which is the degree of the i^{th} vertex and is equal to $D[i, i]$. Hence we have $D[i, i] = \sum_{j=1}^n A_{ij}$, and since D is a diagonal matrix, we can write $D[i, i] = \sum_{j=1}^n D_{ij}$, hence we get
$\sum_{j=1}^n D_{ij} = \sum_{j=1}^n A_{ij} \implies \sum_{j=1}^n (D_{ij} - A_{ij}) = 0 \implies \sum_{j=1}^n [D^{-1} L]_i = 0 \ \forall i$

(b) The difference between a vertex x and the average position of its 1-ring neighborhood is a quantity that provides interesting geometric insight of the shape. It can be shown that,

$$x - \frac{1}{|N(x)|}\sum_{y_i \in N(x)} y_i \approx H\vec{n}\Delta A$$

for a good mesh, where $N(x)$ is the 1-ring neighborhood vertices of x by the mesh topology, $H = \frac{1}{2}(\kappa_{min} + \kappa_{\max})$ is the mean curvature at x (in the sense of the underlying continuous surface being approximated), $\vec{n}$ is the surface normal vector at x, and ΔA is a quantity proportional to the total area of the 1-ring fan (triangles formed by x and vertices along the 1-ring).

Define $\Delta p_i := p_i - \frac{1}{|N(p_i)|}\sum_{p_j \in N(p_i)} p_j$. Prove that $\Delta p_i = [L_{norm}P]_i$, where P and p_i are defined as in 1(c), and $[X]_i$ is to access the i-th row of X. [1pt]

b)

Consider a graph (V, E, F). Consider a vertex p_i. The 1-ring neighbourhood of p_i is simply the edges origination from p_i. The adjacency matrix's i^{th} row indicates which vertices are connected to p_i and hence $|N(p_i)| = D[i, i]$ and $\sum_{p_j \in N(p_i)} p_j = [A]_i P = [AP]_i$. Now also D is a diagonal matrix, hence $\frac{1}{D[i,i]}[AP]_i = [D^{-1}AP]_i$. Now $p_i = [P]_i = [IP]_i = [D^{-1}DP]_i$. Therefore we can finally write $\Delta p_i := p_i - \frac{1}{|N(p_i)|}\sum_{p_j \in N(p_i)} p_j = [D^{-1}DP]_i - [D^{-1}AP]_i = [D^{-1}DP - D^{-1}AP]_i = [D^{-1}(D-A)P]_i = [D^{-1}LP]_i = [L_{norm}P]_i$

3. Shape Deformation (extra credit)

Please load the source.obj and target.obj files using the trimesh library of Python, and optimize to deform the vertices of the source.obj to match target.obj. Plot the source object, target object, and deformed object. [5pt]

In [1]:
```
import trimesh
import numpy as np
import torch
import torch.nn as nn
from torch.optim import Adam
from chamfer import Chamfer_distance_torch
import matplotlib.pyplot as plt
import open3d
from tqdm import tqdm
source = trimesh.load('source.obj')
target = trimesh.load('target.obj')
```

```
Jupyter environment detected. Enabling Open3D WebVisualizer.
[Open3D INFO] WebRTC GUI backend enabled.
[Open3D INFO] WebRTCWindowSystem: HTTP handshake server disabled.
```

In [2]:
```
source_vertex = source.vertices
```

In [3]:
```
source_tensor = torch.tensor(np.array(source_vertex)[None,:,:], requires_grad = True, dtype = torch.float32)
source_tensor.dim()
```

Out[3]:
```
3
```

In [4]:
```
target_tensor = torch.tensor(np.array(target.vertices)[None,:,:], dtype = torch.float32)
target_tensor.shape
```

Out[4]:
```
torch.Size([1, 1502, 3])
```

In [5]:
```
target_vertex = target.vertices
```

In [6]:
```
def plot_mesh(mesh,color, title):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_trisurf(mesh.vertices[:, 0], mesh.vertices[:,1], triangles=mesh.faces, Z=mesh.vertices[:,2], cma
    plt.title(title)
    plt.show()
```

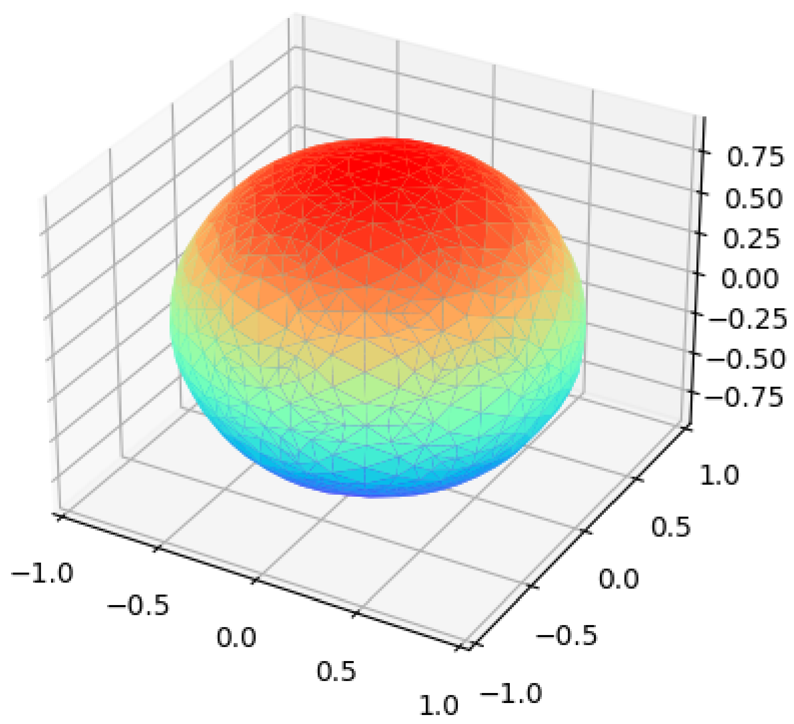
In [7]:
```
from matplotlib.cm import get_cmap
NUM_OBJECTS = 79
cmap = get_cmap('rainbow', NUM_OBJECTS)
COLOR_PALETTE = np.array([cmap(i)[:3] for i in range(NUM_OBJECTS + 3)])
COLOR_PALETTE = np.array(COLOR_PALETTE * 255, dtype=np.uint8)
COLOR_PALETTE[-3] = [119, 135, 150]
COLOR_PALETTE[-2] = [176, 194, 216]
COLOR_PALETTE[-1] = [255, 255, 225]
```

Wthout using neural network (Directly optimising the source vertex)

In [8]:
```
chamfer_losses = []
iterations = 1000

# Instantiate optimizer
weights_deform = nn.Parameter(source_tensor)
optimizer = torch.optim.Adam([weights_deform], lr=1e-2)

for it in tqdm(range(iterations)):
    dist1, idx1, dist2, idx2 = Chamfer_distance_torch(weights_deform, target_tensor)
    loss = torch.mean(dist1) + torch.mean(dist2)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    chamfer_losses.append(loss.item())
    if it %100 == 0:
        weights = torch.clone(weights_deform)
        weights =weights.detach().numpy()
        deformed = trimesh.Trimesh(vertices = weights.reshape(-1,3), faces = source.faces)
        plot_mesh(deformed,COLOR_PALETTE[2]/255, f'Achieved mesh using chamfer loss only after {it} iteratio

deformed_chamfer_vertices = weights_deform
```

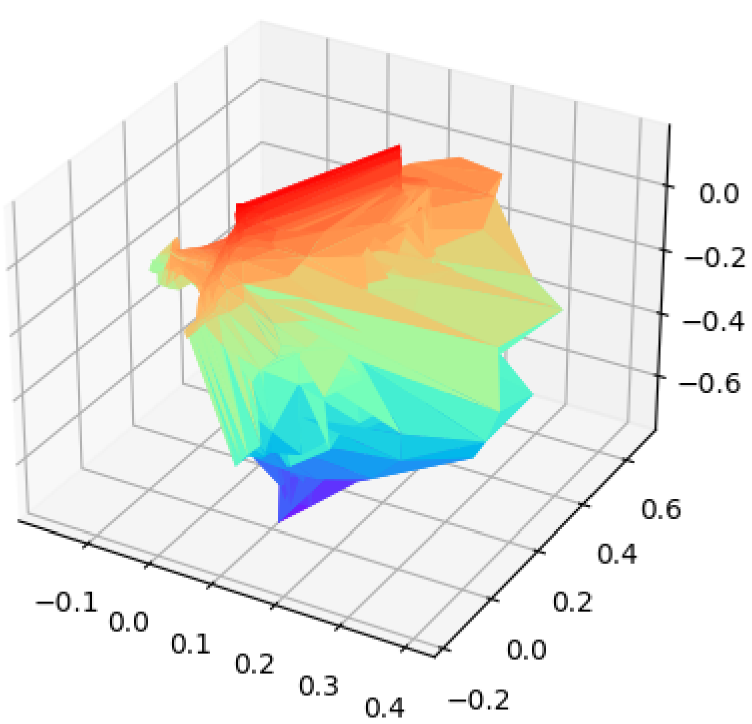
```
  0%|                                                    | 0/1000 [00:00<?, ?it/s]
```

Achieved mesh using chamfer loss only after 0 iterations



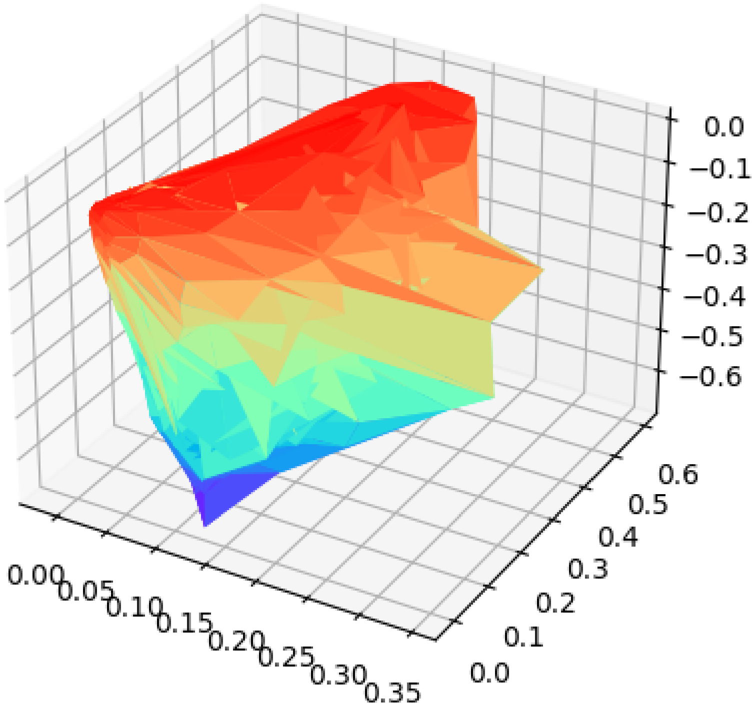

```
 10%|                                        | 98/1000 [00:02<00:17, 52.04it/s]
```

Achieved mesh using chamfer loss only after 100 iterations



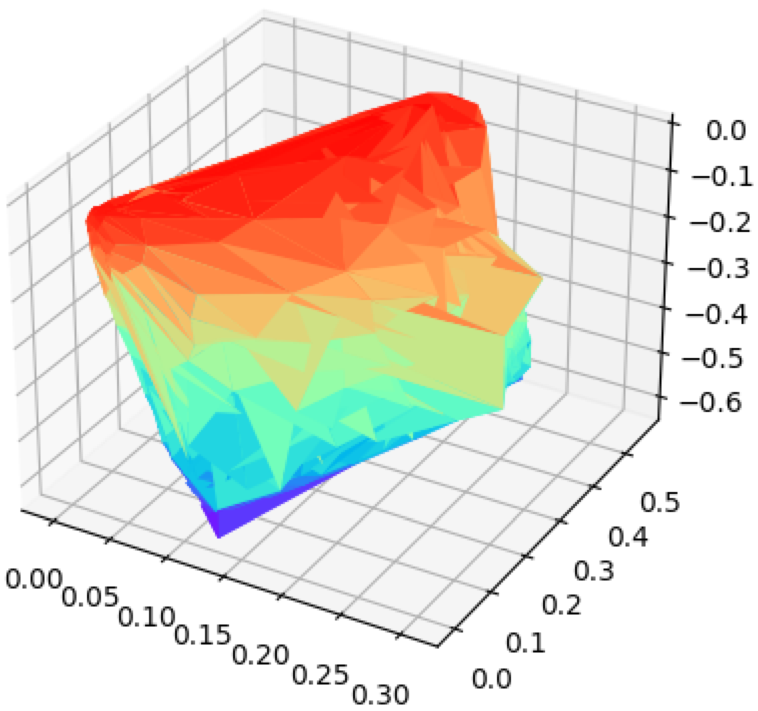

```
 20%|                                       | 199/1000 [00:04<00:15, 52.21it/s]
```

Achieved mesh using chamfer loss only after 200 iterations



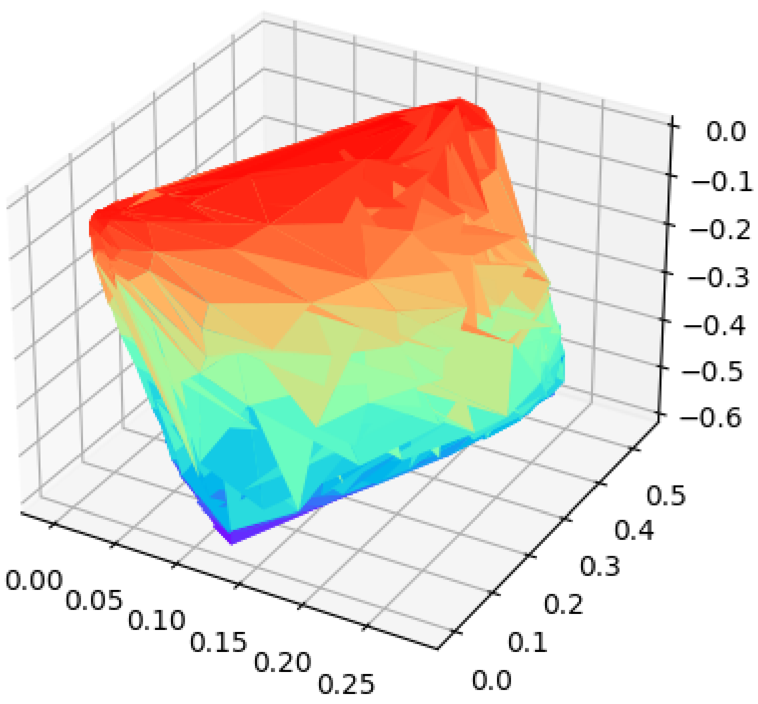

```
30%|                    | 297/1000 [00:06<00:15, 45.93it/s]
```

Achieved mesh using chamfer loss only after 300 iterations



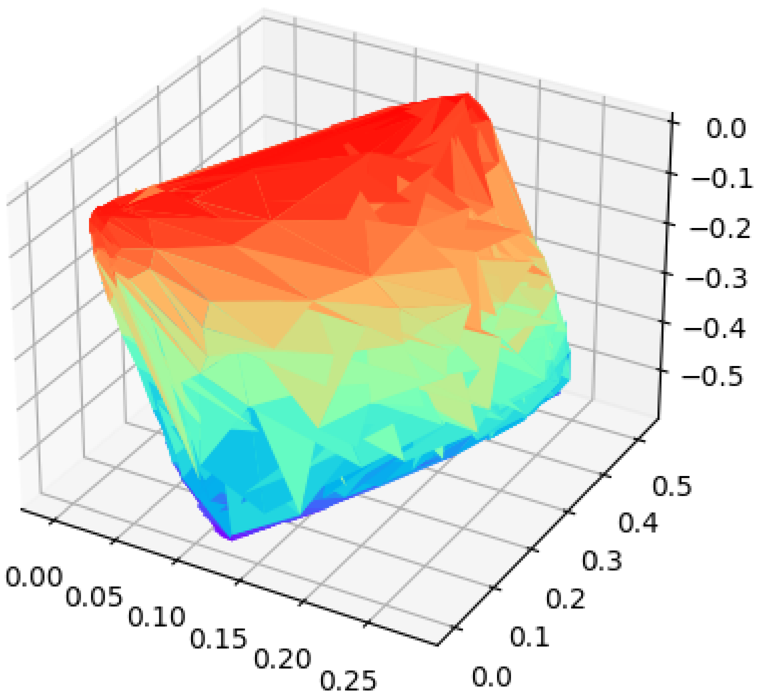

```
40%|                    | 395/1000 [00:08<00:11, 51.69it/s]
```

Achieved mesh using chamfer loss only after 400 iterations



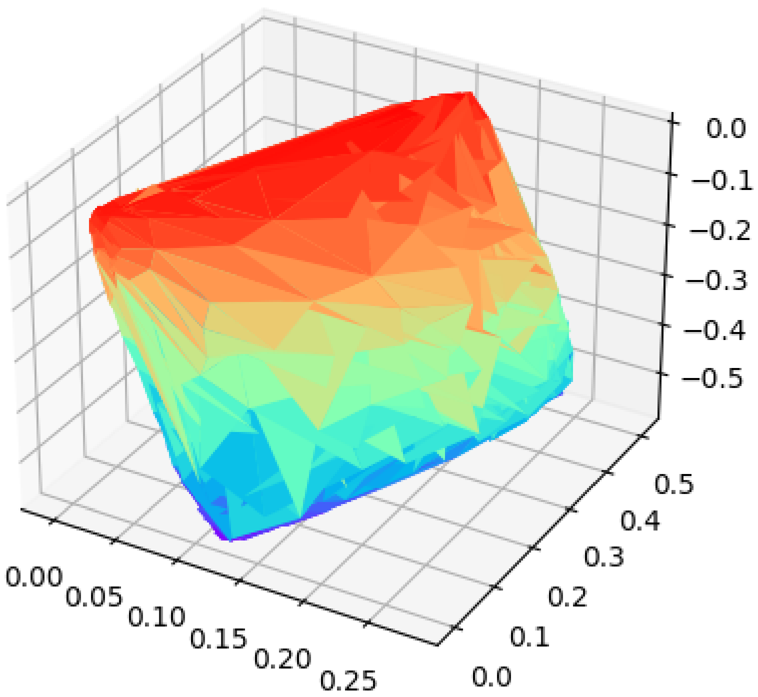
```
50%|                              | 496/1000 [00:11<00:09, 50.56it/s]
```

Achieved mesh using chamfer loss only after 500 iterations



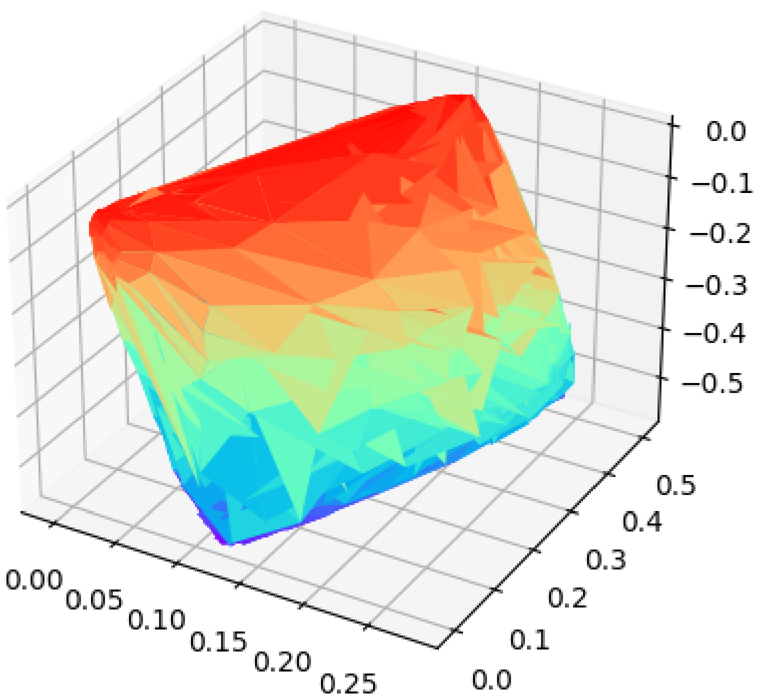

```
60%|                              | 600/1000 [00:13<00:07, 51.74it/s]
```

Achieved mesh using chamfer loss only after 600 iterations



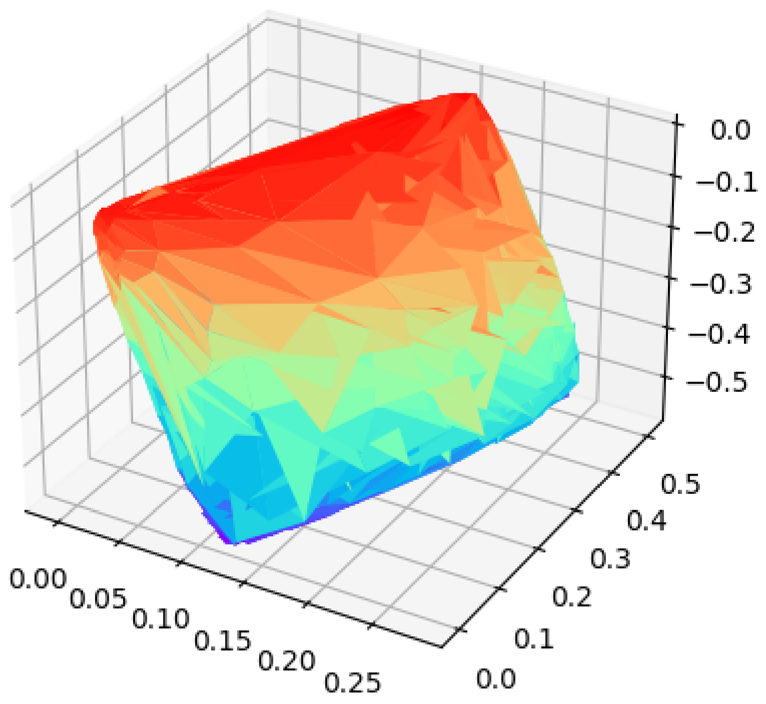

```
70%|                              | 697/1000 [00:15<00:06, 46.18it/s]
```

Achieved mesh using chamfer loss only after 700 iterations



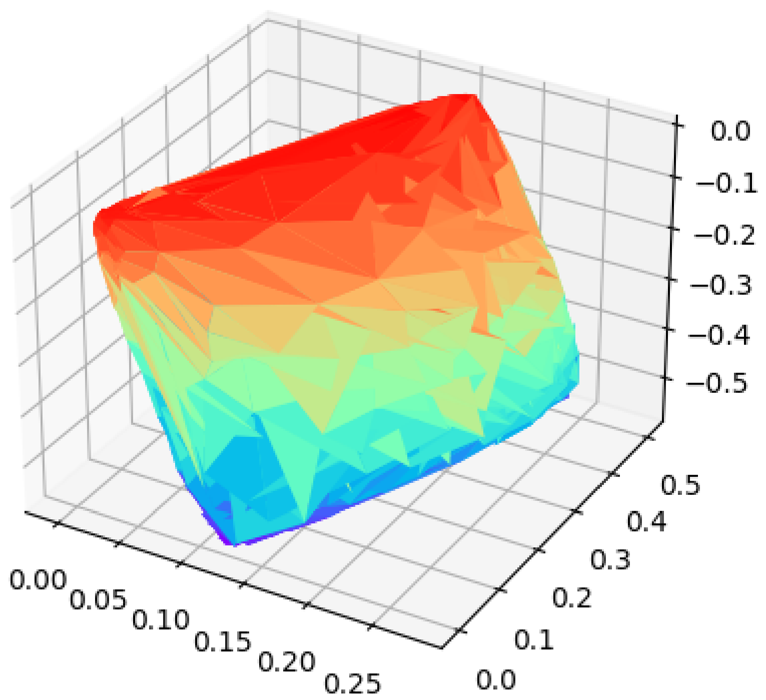
```
80%|████████████████████████        | 799/1000 [00:18<00:04, 48.90it/s]
```

Achieved mesh using chamfer loss only after 800 iterations




```
90%|███████████████████████████     | 898/1000 [00:20<00:01, 51.34it/s]
```

Achieved mesh using chamfer loss only after 900 iterations




```
100%|██████████████████████████████| 1000/1000 [00:22<00:00, 44.53it/s]
```

Using the chamfer loss, the source vertices move closer to the corresponding nearest target vertex. We can see that the corners are not so smooth, are slightly curved whereas the target mesh has sharp corners and overall smooth edges.

(b) Curvature and normal-based loss

Using chamfer + curvature normal based loss

using the formula $\Delta p_i = [L_{norm} P]_i$, we nee dto create the adjacency matrix A, degree matrix D, $L = D - A$ and $L_{norm} = D^{-1} L$

In [9]:
```
def adjacency_matrix(faces, num_pts):
    num_faces = faces.shape[0]
    A = torch.zeros((num_pts, num_pts))

    for i in range(num_faces):
        vertex = faces[i]
        A[vertex[0], vertex[1]] = 1
        A[vertex[1], vertex[0]] = 1

        A[vertex[0], vertex[2]] = 1
        A[vertex[2], vertex[0]] = 1

        A[vertex[1], vertex[2]] = 1
        A[vertex[2], vertex[1]] = 1

    return A

def Lnorm(A):
    num_pts = A.shape[0]
    D = torch.diag(torch.sum(A, axis=1))
    L = D - A
    Lnorm = torch.inverse(D) @ L

    return Lnorm
```

In [10]:
```
source_A, target_A = adjacency_matrix(source.faces, source_vertex.shape[0]), \
                        adjacency_matrix(target.faces, target_vertex.shape[0])
```

In [11]:
```
Lnorm_source, Lnorm_target = Lnorm(source_A), Lnorm(target_A)
```

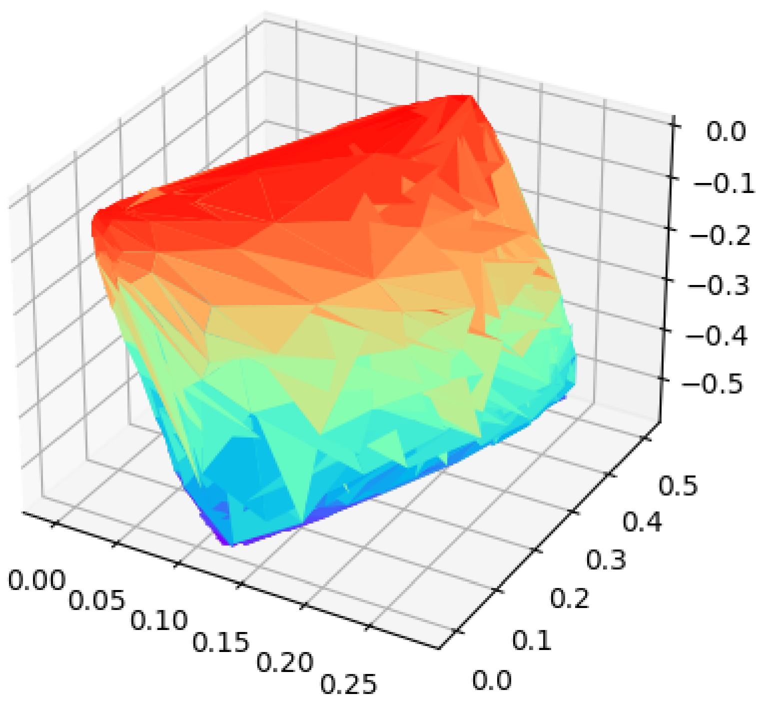
The Δp_i matrix does not change for target since it is not the one undergoing deformation. So it can be computed once and thats it. We need to compute Δp_i for source because it is undergoing deformation. But L_{norm} will not change since connectivity does not change.

In [12]:
```
delta_p_target = Lnorm_target @ target_tensor.reshape(-1,3)
```

In [13]:
```
iterations = 1000
combined_loss = []

weights_deform = nn.Parameter(source_tensor)
optimizer = torch.optim.Adam([weights_deform], lr=1e-3)

for it in tqdm(range(iterations)):
    delta_p_source = Lnorm_source @ weights_deform.detach().numpy().reshape(-1,3)
    dist1, idx1, dist2, idx2 = Chamfer_distance_torch(weights_deform, target_tensor)
    loss = torch.mean(dist1) + torch.mean(dist2)
    loss += torch.mean(torch.norm(delta_p_source - delta_p_target[idx1], dim = 1))
    loss += torch.mean(torch.norm(delta_p_target - delta_p_source[idx2], dim = 1))
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    combined_loss.append(loss.item())
    if it %100 == 0:
        weights = torch.clone(weights_deform)
        weights =weights.detach().numpy()
        deformed = trimesh.Trimesh(vertices = weights.reshape(-1,3), faces = source.faces)
        plot_mesh(deformed,COLOR_PALETTE[2]/255, f'Achieved mesh using chamfer + curvature loss after {it} i
```

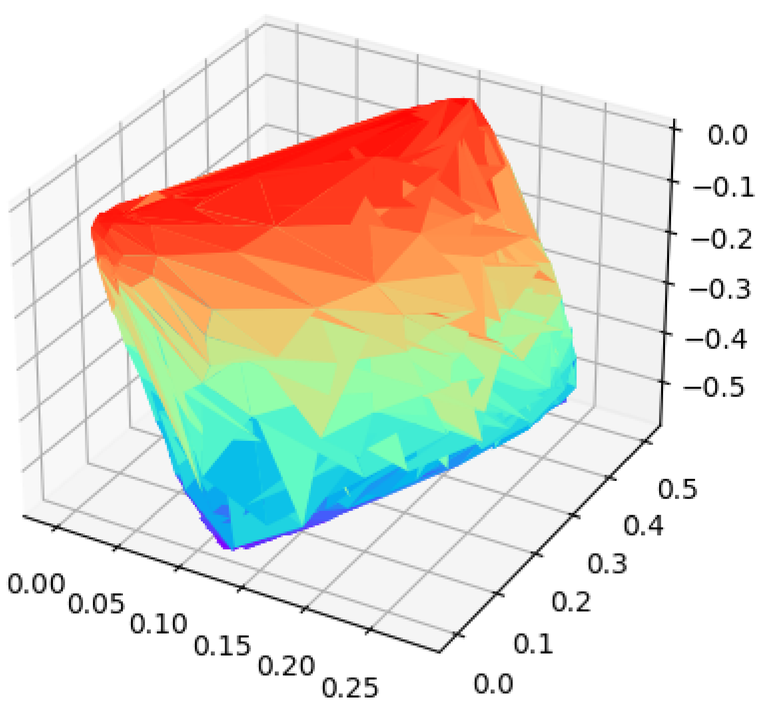
```
  0%|                                                   | 0/1000 [00:00<?, ?it/s]
```

Achieved mesh using chamfer + curvature loss after 0 iterations



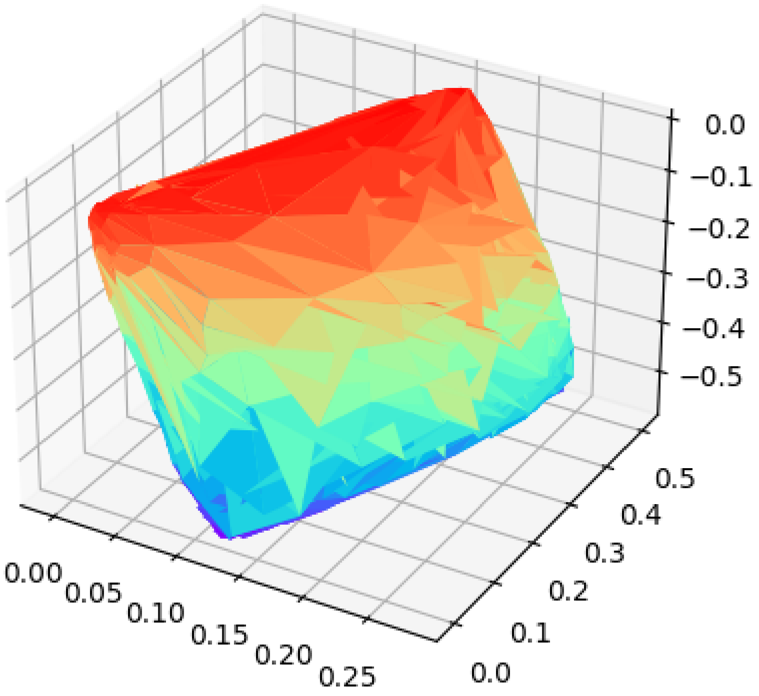

```
 10%|                                   | 100/1000 [00:06<00:55, 16.28it/s]
```

Achieved mesh using chamfer + curvature loss after 100 iterations



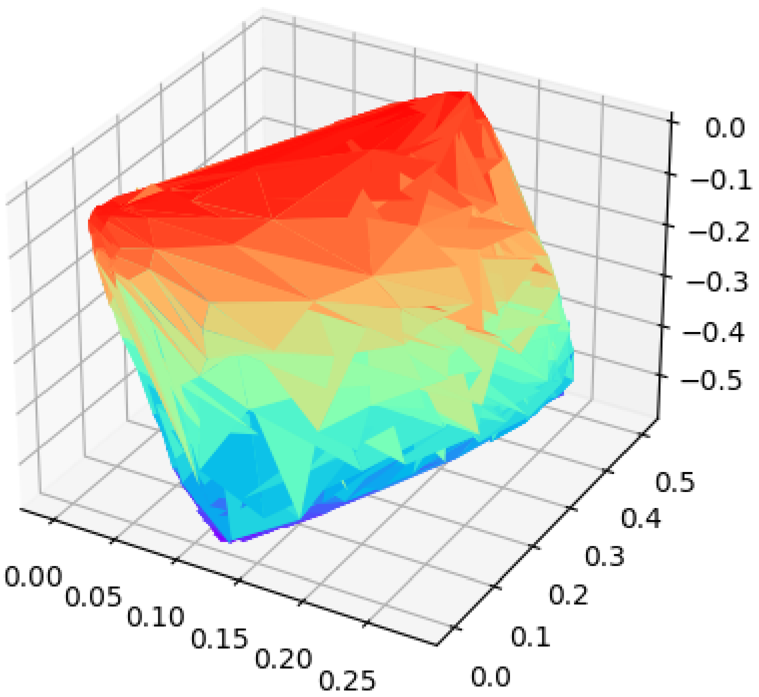

```
 20%|                                   | 200/1000 [00:12<00:53, 14.91it/s]
```

Achieved mesh using chamfer + curvature loss after 200 iterations



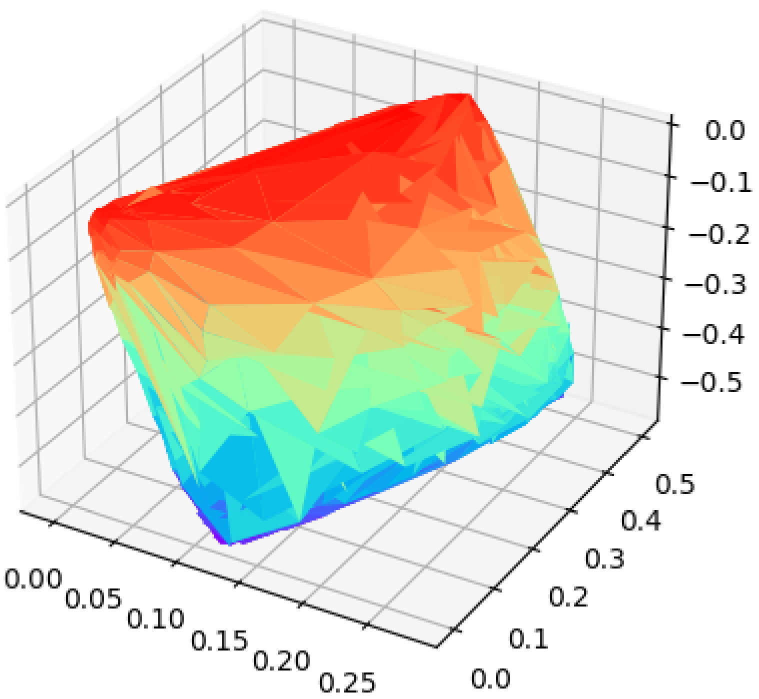

```
30%|                    | 300/1000 [00:17<00:38, 18.18it/s]
```

Achieved mesh using chamfer + curvature loss after 300 iterations



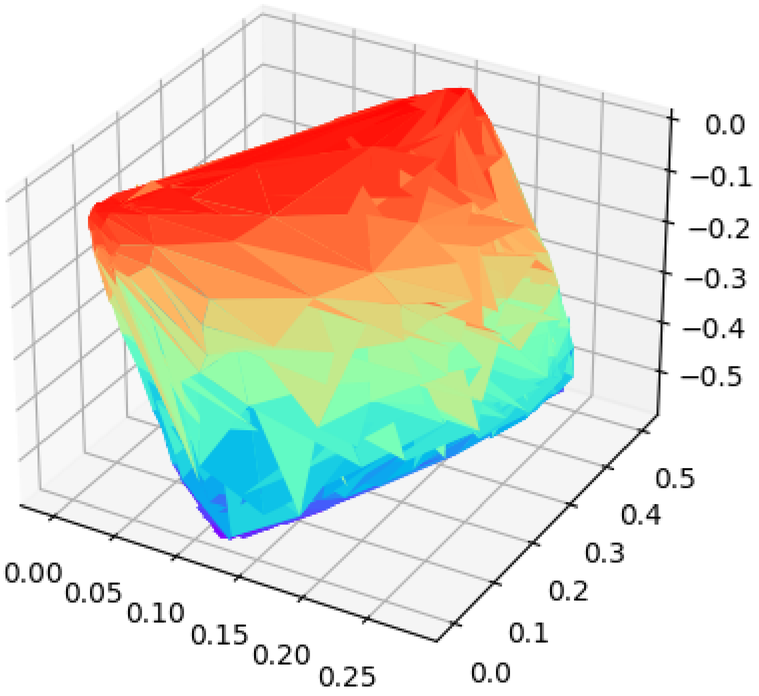

```
40%|                    | 400/1000 [00:23<00:35, 17.14it/s]
```

Achieved mesh using chamfer + curvature loss after 400 iterations



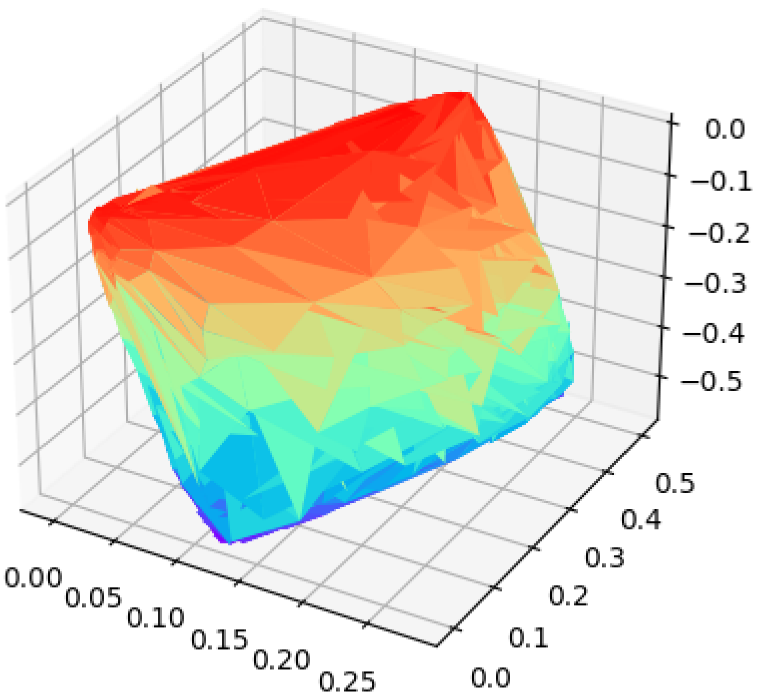

```
50%|                    | 498/1000 [00:29<00:25, 19.33it/s]
```

Achieved mesh using chamfer + curvature loss after 500 iterations



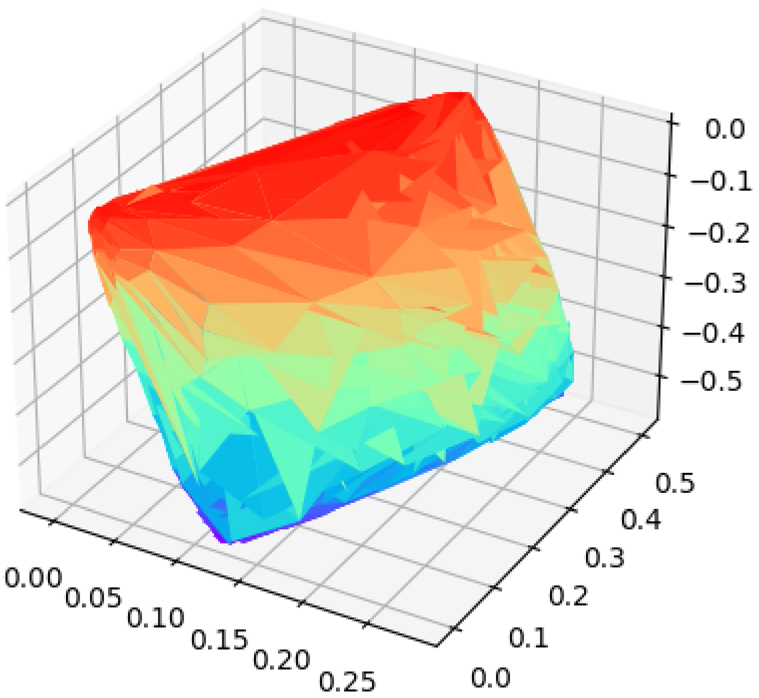

```
60%|                                         | 600/1000 [00:36<00:23, 16.74it/s]
```

Achieved mesh using chamfer + curvature loss after 600 iterations



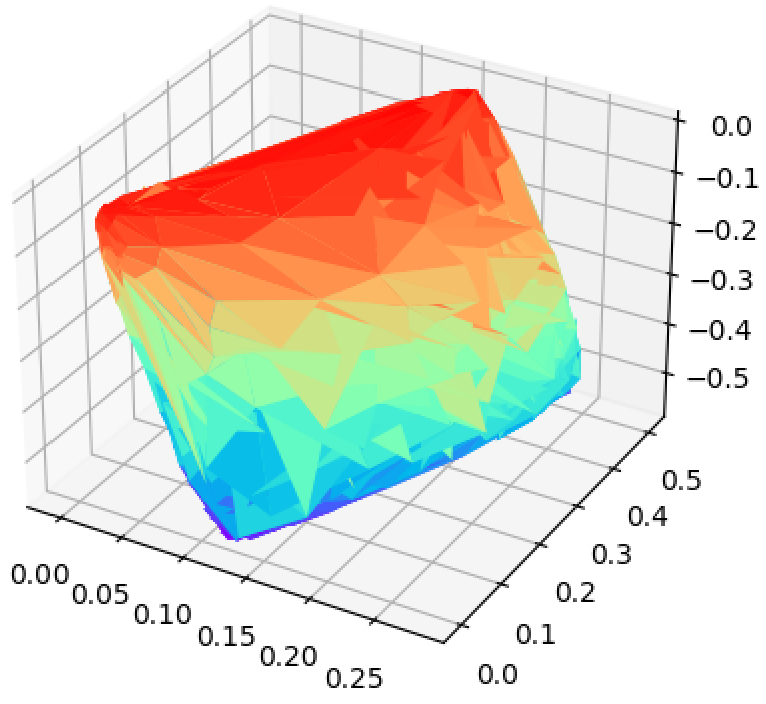

```
70%|                                         | 699/1000 [00:42<00:16, 18.31it/s]
```

Achieved mesh using chamfer + curvature loss after 700 iterations



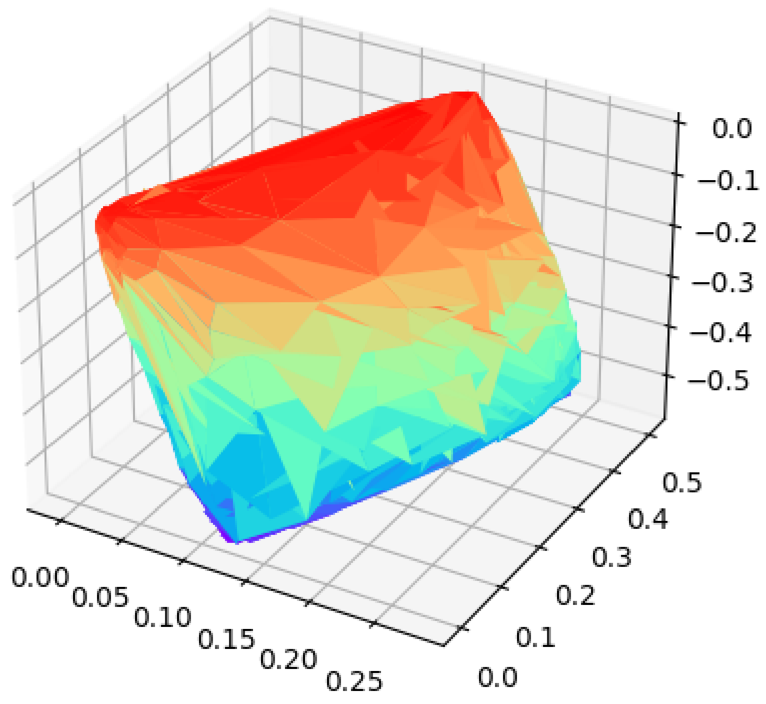
```
 80%|████████████████████████████████        | 800/1000 [00:48<00:09, 20.27it/s]
```

Achieved mesh using chamfer + curvature loss after 800 iterations




```
 90%|████████████████████████████████████    | 899/1000 [00:53<00:05, 18.61it/s]
```

Achieved mesh using chamfer + curvature loss after 900 iterations




```
100%|████████████████████████████████████████| 1000/1000 [01:00<00:00, 16.66it/s]
```

Using the combined loss, we observe that the deformation is a lot smoother than the one using only chamfer loss. Also we observe that the deformed mesh corners are very similar to the target mesh corners whereas the one using only chamfer loss had kind of round corners.

Problem 2: ICP

In [14]:
```
"""Visualization utilies."""

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

def show_points(points):
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1,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()
    ax = fig.add_subplot(1,1,1,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])
```

In [15]:
```
"""Load data."""

import trimesh
import numpy as np

source_pcd = trimesh.load("banana.source.ply").vertices
target_pcd = trimesh.load("banana.target.ply").vertices
gt_T = np.loadtxt("banana.pose.txt")
```

In [16]:
```
source_pcd
```

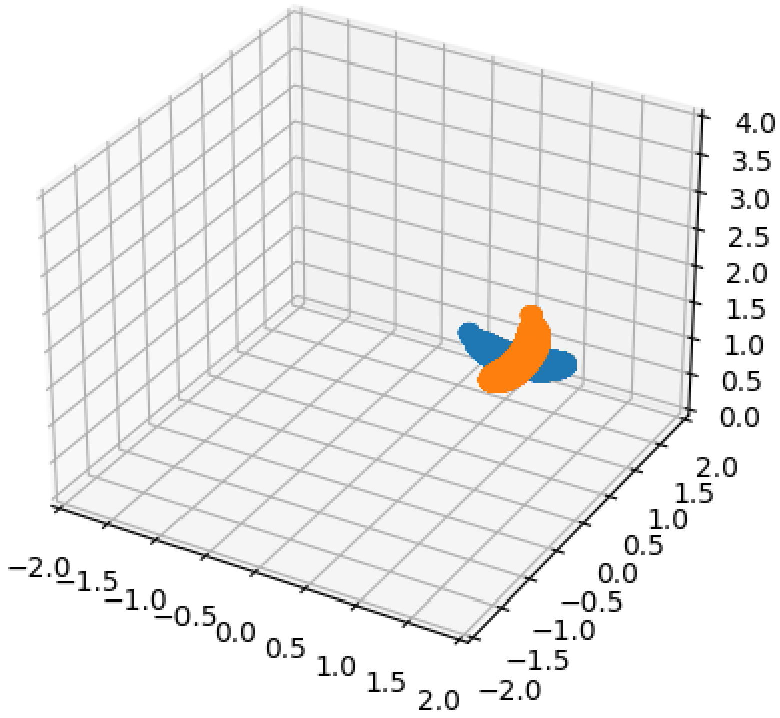
Out[16]:
```
TrackedArray([[-0.23091938, -0.10104883, -0.07753404],
              [ 0.27073169, -0.00146855, -0.0353344 ],
              [-0.39152163, -0.00135914, -0.07481582],
              ...,
              [-0.29234454, -0.14529917, -0.09030698],
              [ 0.19018735, -0.19979134, -0.05973255],
              [-0.09481647, -0.07675853, -0.01570301]])
```

In [17]:
```
target_pcd
```

Out[17]:
```
TrackedArray([[1.07478809, 1.47837925, 0.68953747],
              [1.1249311 , 1.503407  , 0.17942953],
              [1.0583688 , 1.60131693, 0.83220923],
              ...,
              [1.0651176 , 1.44293368, 0.7569496 ],
              [1.12221241, 1.31854594, 0.29003513],
              [1.13867629, 1.48905003, 0.55263776]])
```

In [18]:
```
from sklearn.neighbors import NearestNeighbors

def nearest_neighbor(src, dst):
    '''
    Find the nearest (Euclidean) neighbor in dst for each point in src
    Input:
        src: Nxm array of points
        dst: Nxm array of points
    Output:
        distances: Euclidean distances of the nearest neighbor
        indices: dst indices of the nearest neighbor
    '''

    assert src.shape == dst.shape

    neigh = NearestNeighbors(n_neighbors=1)
    neigh.fit(dst)
    distances, indices = neigh.kneighbors(src, return_distance=True)
#     print("indices from sklearn : {}".format(indices))
    return indices.ravel()
```

In [19]:
```
source = np.copy(source_pcd) + np.mean(target_pcd, axis = 0) - np.mean(source_pcd, axis = 0)
target = np.copy(target_pcd)
compare_points(source, target)
```

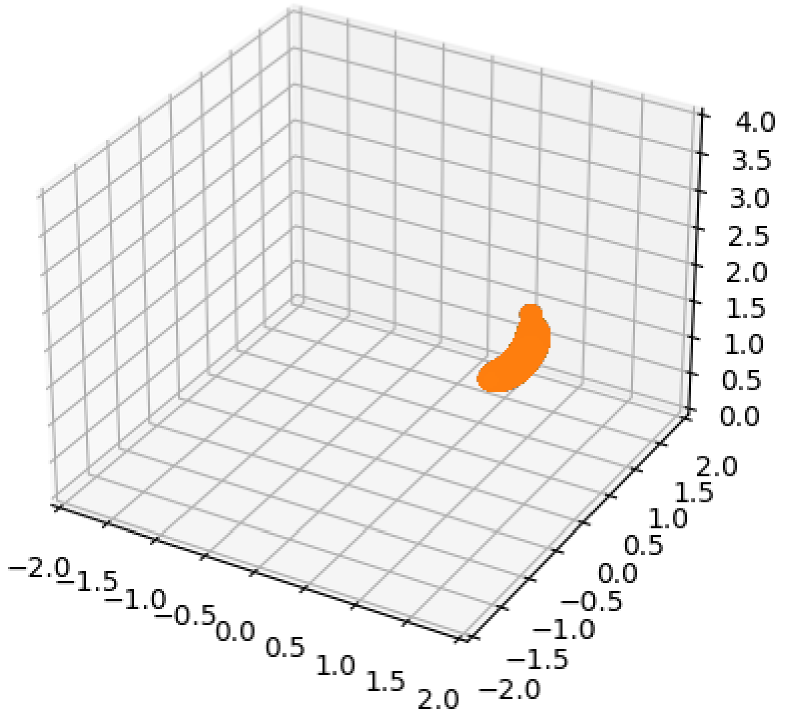


In [20]:
```
"""Implement your own ICP."""

def icp(source_pcd, target_pcd):
    """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.
    """
    T = np.eye(4)
    target_mean = np.mean(target_pcd, axis = 0)
    R = T[:3, :3]
    T[:3,3] = target_mean[:] - np.mean(source_pcd, axis = 0)
    source = np.copy(source_pcd) + target_mean - np.mean(source_pcd, axis = 0)
    target = np.copy(target_pcd)

    # Implement your own algorithm here.
    for it in tqdm(range(50)):
        corr_idx = nearest_neighbor(source, target)
        p = source.T
        q = target_pcd[corr_idx, :].T
        M = (q - np.mean(q,axis = 1, keepdims=True))@ (p - np.mean(p, axis = 1, keepdims=True)).T
        U,S,V = np.linalg.svd(M)
        R = U@V
        if np.linalg.det(R) < 0:
            V[:,-1] = -V[:,-1]
        R = U@V
        t = np.mean(q,axis = 1, keepdims=True) - R@np.mean(p,axis = 1, keepdims=True)
#         print(f"relative translation : {t}")
        source_new = (R@p + t).T
        source = source_new
        T_temp = np.block([[R,t], [np.zeros((1,3)), 1]])
        T = T_temp @ T
        rot_error = np.rad2deg(compute_rre(T[:3, :3], gt_T[:3, :3]))
        rte = compute_rte(T[:3, 3], gt_T[:3, 3])
        print(f"rot error : {rot_error} || translation error : {rte}")
    return T
```

In [21]:
```python
"""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


# Visualization
T = icp(source_pcd, target_pcd)
print(T)
rre = np.rad2deg(compute_rre(T[:3, :3], gt_T[:3, :3]))
rte = compute_rte(T[:3, 3], gt_T[:3, 3])
print(f"rre={rre}, rte={rte}")
compare_points(source_pcd @ T[:3, :3].T + T[:3, 3], target_pcd)
```

```
  8%|██                                         | 4/50 [00:00<00:03, 14.49it/s]

rot error : 86.9389956369664 || translation error : 0.06805395849299617
rot error : 81.75630675910436 || translation error : 0.07271131712948033
rot error : 66.68250314519055 || translation error : 0.07132642738497441
rot error : 37.83088208887614 || translation error : 0.06730364391722787

 16%|████                                       | 8/50 [00:00<00:02, 16.67it/s]

rot error : 19.15749343913435 || translation error : 0.05435682431670539
rot error : 12.192294583035894 || translation error : 0.041270809782583136
rot error : 8.502858371087228 || translation error : 0.030706188300612954
rot error : 6.144945107571997 || translation error : 0.022948807405673392

 30%|███████                                   | 15/50 [00:00<00:01, 24.56it/s]

rot error : 4.603721323573633 || translation error : 0.017597647737449172
rot error : 3.601915811625485 || translation error : 0.01411664220014118
rot error : 2.926201185042271 || translation error : 0.011923650600620181
rot error : 2.4556035790776765 || translation error : 0.010546637888584977
rot error : 2.087853515398311 || translation error : 0.009634729553363301
rot error : 1.8040245479665649 || translation error : 0.008962465217761758
rot error : 1.5624026077999607 || translation error : 0.008417617450711004

 38%|█████████                                 | 19/50 [00:00<00:01, 27.45it/s]

rot error : 1.3582308483377696 || translation error : 0.00795194333044214
rot error : 1.197291747633845 || translation error : 0.007562006552250469
rot error : 1.0703514222654023 || translation error : 0.007223724460592022
rot error : 0.9724970879253186 || translation error : 0.006911819587216404
rot error : 0.8944459312073222 || translation error : 0.006603292087126549
rot error : 0.82957449358498 || translation error : 0.006324198855981207
rot error : 0.7730870919130947 || translation error : 0.006062577240403191

 54%|█████████████                             | 27/50 [00:01<00:00, 30.53it/s]

rot error : 0.7232375279934821 || translation error : 0.00581022359959737
rot error : 0.6794266686736165 || translation error : 0.005556780191458625
rot error : 0.6386900793785545 || translation error : 0.005300023128476780 5
rot error : 0.6022671549025616 || translation error : 0.005037167007824063
rot error : 0.5680862289699448 || translation error : 0.004751109659483922
rot error : 0.5334276490730013 || translation error : 0.004457785228096964
rot error : 0.4974802049341708 || translation error : 0.004124112375002523

 70%|█████████████████                         | 35/50 [00:01<00:00, 32.60it/s]

rot error : 0.4549059484657815 || translation error : 0.0037319325573458855
rot error : 0.40253700264621073 || translation error : 0.0032507857036372855
rot error : 0.32783487706955144 || translation error : 0.0025978771697115887
rot error : 0.22194712252748913 || translation error : 0.0017044469292163457
rot error : 0.09189078018094955 || translation error : 0.0006621359364264844
rot error : 0.007129130602820963 || translation error : 4.858286288210888e-05
rot error : 6.3890569402603996e-06 || translation error : 4.503890859503922e-08
rot error : 0.0 || translation error : 4.069790255192151e-10

 86%|█████████████████████                     | 43/50 [00:01<00:00, 35.22it/s]

rot error : 0.0 || translation error : 4.0697916949491695e-10
rot error : 0.0 || translation error : 4.069787321199127e-10
rot error : 0.0 || translation error : 4.069792089503863e-10
rot error : 0.0 || translation error : 4.069789522071813e-10
rot error : 0.0 || translation error : 4.0697895360695704e-10
rot error : 0.0 || translation error : 4.0697881243084497e-10
```

```
rot error : 0.0 || translation error : 4.069790311183021e-10
rot error : 0.0 || translation error : 4.069789944622533e-10

100%|██████████████████████████████████████████| 50/50 [00:01<00:00, 28.92it/s]

rot error : 0.0 || translation error : 4.069789944622533e-10
rot error : 0.0 || translation error : 4.069789958620517e-10
rot error : 0.0 || translation error : 4.0697874051864017e-10
rot error : 0.0 || translation error : 4.069790353176969e-10
rot error : 0.0 || translation error : 4.069788180300561e-10
[[ 0.04139069 -0.12505187  0.99128646  1.14856815]
 [-0.15543338  0.9792519   0.13002374  1.55152014]
 [-0.98697886 -0.15946078  0.02109467  0.44714717]
 [ 0.          0.          0.          1.        ]]
rre=0.0, rte=4.069788180300561e-10
```



4. Course Feedback

1. I spent around 35-40hrs on this Homework (Most of the time was spent on tuning parameters for Q3)
2. I spent around 10hrs on the course last week because of other assignment deadlines and mid terms.
3. I feel like if given the conda file for the environment that has all packages installed and correct versions, it will help the students a lot.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

6D Pose Estimation using ICP

Sambaran Ghosal, USERNAME : 007, Attempt : Ezra Bridger >>>>>> Luke Skywalker
Department of Electrical and Computer Engineering
UC San Diego
La Jolla, USA
sghosal@ucsd.edu

I. Introduction

An important problem in Robotics and Computer Vision is to determine the pose of objects from a given image. This is useful in many applications such as robotic manipulation where the manipulator needs to know the position of some object to pick up, or in autonomous driving to keep track of where different agents in the scene are and determine a safe trajectory keeping in mind these agents.

In this work, we will implement a method to extract the 6D poses of objects in a given image using Point Clouds and Iterative Closest Point Algorithm.

II. Problem Formulation

Given a set of target point clouds of a given object extracted from training data as $\mathcal{P}train = \{x_i, y_i, z_i\}_{i=1}^{N}$ in the world canonical frame, we want to extract the pose of the same object in a given test set image. Let the test set point clouds of this object be $\mathcal{P}^{test} = \{x_{ti}, y_{ti}, z_{ti}\}_{i=1}^{N_o}$, then objective of the problem is to find $T \in SE(3)$ such that

$$T = \underset{T}{argmin} ||T\mathcal{P}^{train} - \mathcal{P}^{test}||_F^2 \tag{1}$$

This is to be done for all objects in a given test image and for all test images provided to us. In the next section, we will discuss details of how the training point clouds are extracted, and how the ICP is performed.

III. Technical Approach

A. *Extracting point clouds from training data*

Given the training data set, the meta data for each scene contains the objects present in the scene and the corresponding segmented and depth map images. The depth map image is first converted to point clouds using the inverse pinhole camera model as follows

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = K_{cam}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} Z \tag{2}$$

where Z is the depth value at pixel (u, v). This point clouds is in the camera frame of reference, we need to convert this point cloud to the world canonical frame. We are given the camera extrinsic matrix which is pose of camera wrt world and the object world poses which is the pose of the point cloud in the canonical frame. First to get the point cloud in world frame, we multiply the point cloud with the inverse of the camera extrinsic matrix as follows

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = {}^{world} T_{camera}^{-1} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3}$$

and finally, multiply this with the world pose inverse of the given object which is available to us in the meta data

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = {}^{world} T_{pose}^{-1} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{4}$$

This stores the point cloud in the canonical frame in non-rotated non-translated pose which will be easier to use for the ICP registration with the test object point cloud.

These point clouds are stored for each object from each scene in the training data. We also save the object world pose in the different training images.

B. *ICP registration on test set*

Now that we have the partial point clouds of objects from each training image, we visit each test image and for each object in the test image, first we extract the test object point cloud using Eq(2) and Eq(3) which transforms the test object point cloud to the world frame but not in the canonical state. We now iterate through all partial point clouds of this object from the training data that we saved, use the **Open3d** ICP registration function to do the alignment. For the initial transformation guess, we use the world pose of the object in the training data scene that we saved. Then for each training point cloud, the ICP will match it to the target point cloud. We save the pose of the test object as the pose corresponding to the alignment that yields maximum fitness in the Open3D registration.

C. *Hyperparameters*

While experimenting with the validation set, it was observed that most important hyperparameters affecting the performance of the ICP registration were as follows

- Downsampling : It was observed that downsampling both the training and testing partial point clouds affected the performance significantly. On one hand, downsample helps ICP registration avoid local minima by finding non-redundant correspondences between the training and test point clouds and hence positively affects the point cloud. On the other hand, downsampling by a lot negatively affects the performance becasue both point clouds become very sparse and can lead to easy but inaccurate matching. Downsampling also helps reducing the time complexity of the algorithm.

- Threshold : The ICP registration function of Open3D takes in as parameter the threshold. Threshold defines the radius within which the algorithm will look for correspondence. A smaller value means that points closer to each other will be considered as correspondences and if points are further away, there will be no correspondences. If the threshold is large, further away points also are considered for the correspondence. Larger threshold values lead to larger computation time and may also lead to redundant correspondences which negatively affect the performance. Whereas smaller threshold may lead to smaller correspondences, lesser time complexity and avoid local minima, but too less a threshold will again lead to very less correspondences and hence easier but inaccurate matching.
- Initial transformation : The ICP takes in a guess initial transformation to start with. If this guess is good, then we obtain better correspondence between the source and target point clouds which leads to better matching.
- Iterations : ICP max iterations slightly affects the performance. If the initial transformation and correspondences are good, ICP converges very quickly in about 50-100 iterations. For safer limit, we almost kept iterations at around 200.

IV. Results

A. *Validation Set*

We experimented on only 47 images of the validation set because of time constraints, and we varied hyperparameters downsample, threshold. A summary of results obtained using **benchmark.py** for the pose5deg_1cm criteria is shown in Table (I), (II) and (III).

Threshold	Accuracy
0.05	56.17%
0.02	67.23%
0.01	80.85%
0.008	84.26%

TABLE I: Effect of varying threshold for downsample 4

Threshold	Accuracy
0.05	54.47%
0.02	65.63%
0.01	81.28%
0.008	83.40%

TABLE II: Effect of varying threshold for downsample 8

Threshold	Accuracy
0.05	50.21%
0.02	60.00%
0.01	75.74%
0.008	81.28%

TABLE III: Effect of varying threshold for downsample 16

B. *Test Set*

From the above table, we see that we have the highest accuracy using downsample 4 and threshold 0.008.Hence we do the testing using these parameters. As submitted to the internal benchmark (**USERNAME : 007, Attempt : Ezra Bridger** >>>>>> **Luke Skywalker**), we have a test accuracy of 86.67% on the test set and pose5deg_1cm benchmark. Figures(1) - (3) show some succesfull alignments whereas Figures(4) - (5) show some unsuccesfull alignments for the training and testing point clouds.

V. Acknowledgement

I would like to acknowledge Mr Albert Liao for a very productive discussion over last Sunday Dinner where we went through our ideas, exchanged crucial ideas about making testing fast and accurate. Really enjoyed his company. 10/10 the vibe.

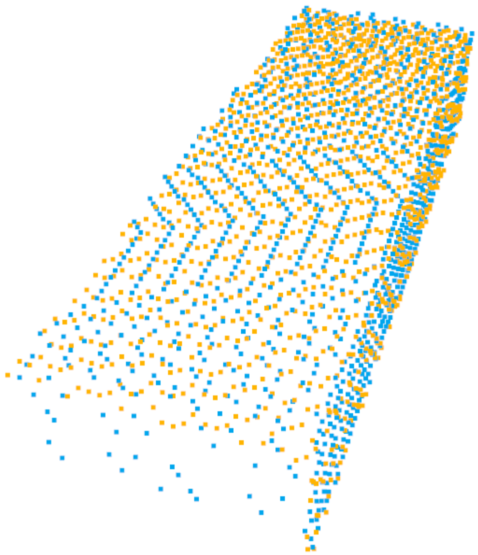


Fig. 1: Success alignment 1

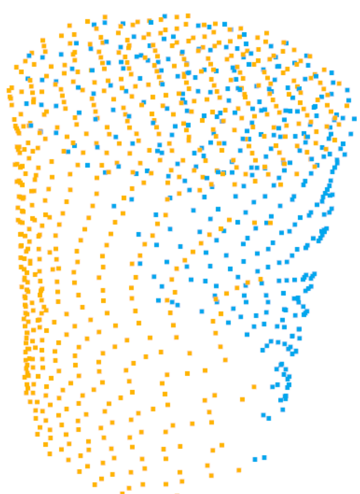


Fig. 2: Success alignment 2

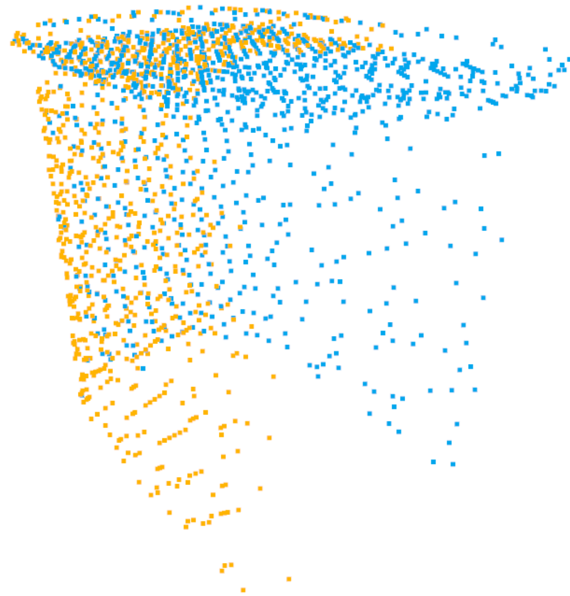


Fig. 3: Success alignment 3

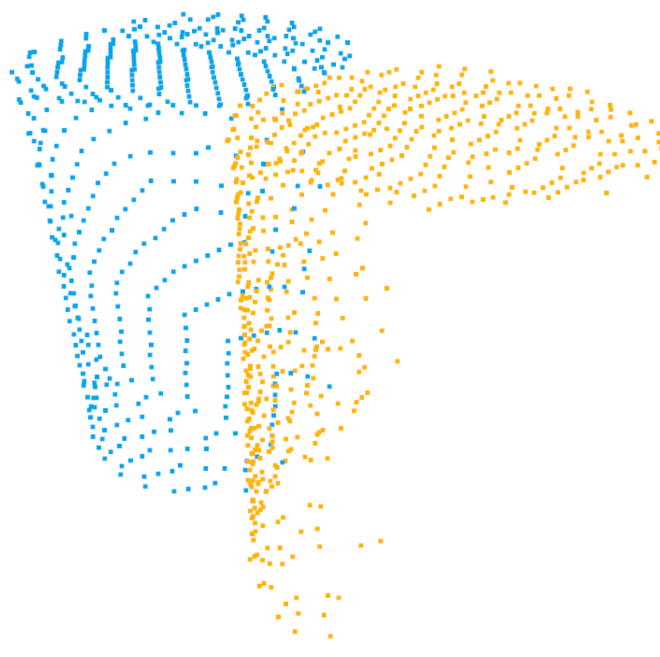


Fig. 4: Unsuccessful alignment 1

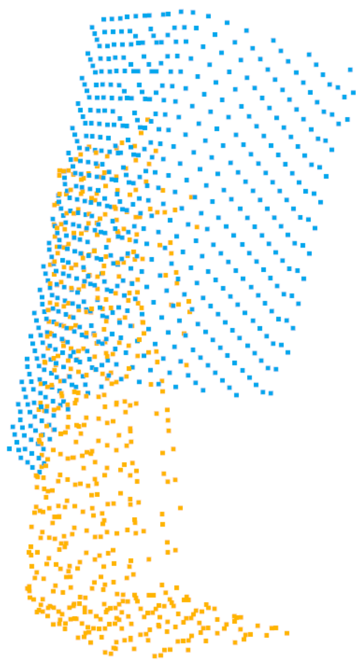


Fig. 5: Unsuccessful alignment 2