```
In [ ]: import numpy as np
        import open3d as o3d
```

2(d) MSE

```
In [ ]: # helper fun: parse pc, convert to arr
        def load_point_cloud_with_open3d(file_path):
            # Load the point cloud using Open3D
            pcd = o3d.io.read_point_cloud(file_path)

            # Convert Open3D.o3d.geometry.PointCloud to numpy array
            points = np.asarray(pcd.points)

            return points
```

```
In [ ]: # get point cloud (from /HW4_data_files/xx.ply)
        ri = load_point_cloud_with_open3d('HW4_data_files/ground_truth.ply') # ground truth
        ri_hat = load_point_cloud_with_open3d('HW4_data_files/transformed_associated.ply') # transformed
```

```
In [ ]: def kabsch_pc_alignment(A, B):
            assert A.shape == B.shape
            # n = A.shape[0]

            # Calculate centroids of the point clouds
            mu_A = np.mean(A, axis=0)
            mu_B = np.mean(B, axis=0)

            # Calculate cross-covariance matrix H
            H = ((B - mu_B).T @ (A - mu_A)) # normalize if needed?
            U, _, Vt = np.linalg.svd(H)

            # # Check if the determinant is negative (special case: reflection instead of rotation)
            # if np.linalg.det(Vt.T @ U.T) < 0:
            #     Vt.T[:, -1] *= -1  # Flip the sign

            # Obtain rotation matrix R from H
            R = (Vt.T @ U.T)

            # Obtain the translation vector t from H
            t = mu_A - R @ mu_B

            return R, t

        R, t = kabsch_pc_alignment(ri, ri_hat)
        optimal_R = R
        # R,t
```

```
In [ ]: # Calculate loss J(R,t)
        def calculate_mse(pc1, pc2):
            """Calculate the Mean Square Error (MSE) between two point clouds."""
            # method 1:
            # squared_diffs = np.square(pc1 - pc2)
            # mse = np.mean(squared_diffs)

            # # method 2
            # sums these squared differences for each point across all dimensions (X, Y, Z)
            # before averaging these summed squared differences across all points.
            mse = np.mean(np.sum((pc1-pc2)**2,axis=1))
            return mse
```

```
In [ ]: # Apply the alignment to ri_hat to align with ri
        R, t = kabsch_pc_alignment(ri, ri_hat)
        aligned_ri_hat = np.dot(ri_hat, R.T) + t

        # Calculate the MSE (Loss J) between the ground truth ri and the aligned ri_hat at optimal R*, t*
        loss_J = calculate_mse(aligned_ri_hat, ri)
        print(f"Loss J (MSE): {loss_J}")
```

```
Loss J (MSE): 0.007494166321482209
```

2(e) SE(3)

```
In [ ]: import pandas as pd

        def load_poses(file_path):
            # Load the CSV file containing the poses
            data = pd.read_csv(file_path)
            # CSV columns are ordered as x,y,z,w, px, py, pz
            quaternions = data[['x', 'y', 'z', 'w']].values
            translations = data[['px', 'py', 'pz']].values
            return quaternions, translations

        gt_q, gt_t = load_poses('HW4_data_files/gt_pose.csv') # groundtruth
```

```
In [ ]: from scipy.spatial.transform import Rotation

        def calculate_rotational_error(optimal_R_matrix, gt_quaternion):
            """
            Calculate the rotational error between an optimal rotation matrix and a ground truth quaternion.

            Parameters:
            optimal_R_matrix (numpy.ndarray): A 3x3 numpy array representing the optimal rotation matrix.
            gt_quaternion (numpy.ndarray): An array representing the ground truth quaternion.

            Returns:
            float: Rotational error in degrees.
            """
            # Create rotation objects from np array
            optimal_R = Rotation.from_matrix(optimal_R_matrix)
            gt_R = Rotation.from_quat(gt_quaternion)

            # Calculate the relative rotation matrix
            relative_R_matrix = optimal_R.as_matrix() @ gt_R.as_matrix().T.reshape(3,3)
            relative_R = Rotation.from_matrix(relative_R_matrix)

            # Calculate the rotational error (deg)
            rotational_error = np.linalg.norm(relative_R.as_rotvec(degrees=True))

            return rotational_error
```

```
In [ ]: # 1. Rotation error in SE(3): theta in axis-angle form
        rotational_error = calculate_rotational_error(optimal_R, gt_q)

        # 2. Translation error: Euclidean distance between the translation components of the two poses.
        translational_error = np.linalg.norm(t - gt_t, axis=1).reshape(1,)

        print("Rotational Error (deg):", rotational_error)
        print("Translational Error:", translational_error)
```

```
Rotational Error (deg): 1.1428636718937023e-12
Translational Error: [1.68941365e-13]
```