Aniket Sanjay Gujarathi
1007629979

Assignment 2

AER 1515
November 9, 2022

Feature Point Detection and Correspondences

Q 1.1 Feature Detection

For feature detection we are using the SIFT detector [1]. SIFT (Scale Invariant Feature Transform) addresses the critical issue in most descriptors of scale invariance. As image scale can vary quite often, most hand-tuned feature detectors fail to find reliable and robust features which leads to wrong feature matching across images with scale difference. The steps taken to detect SIFT features are (reference class notes):

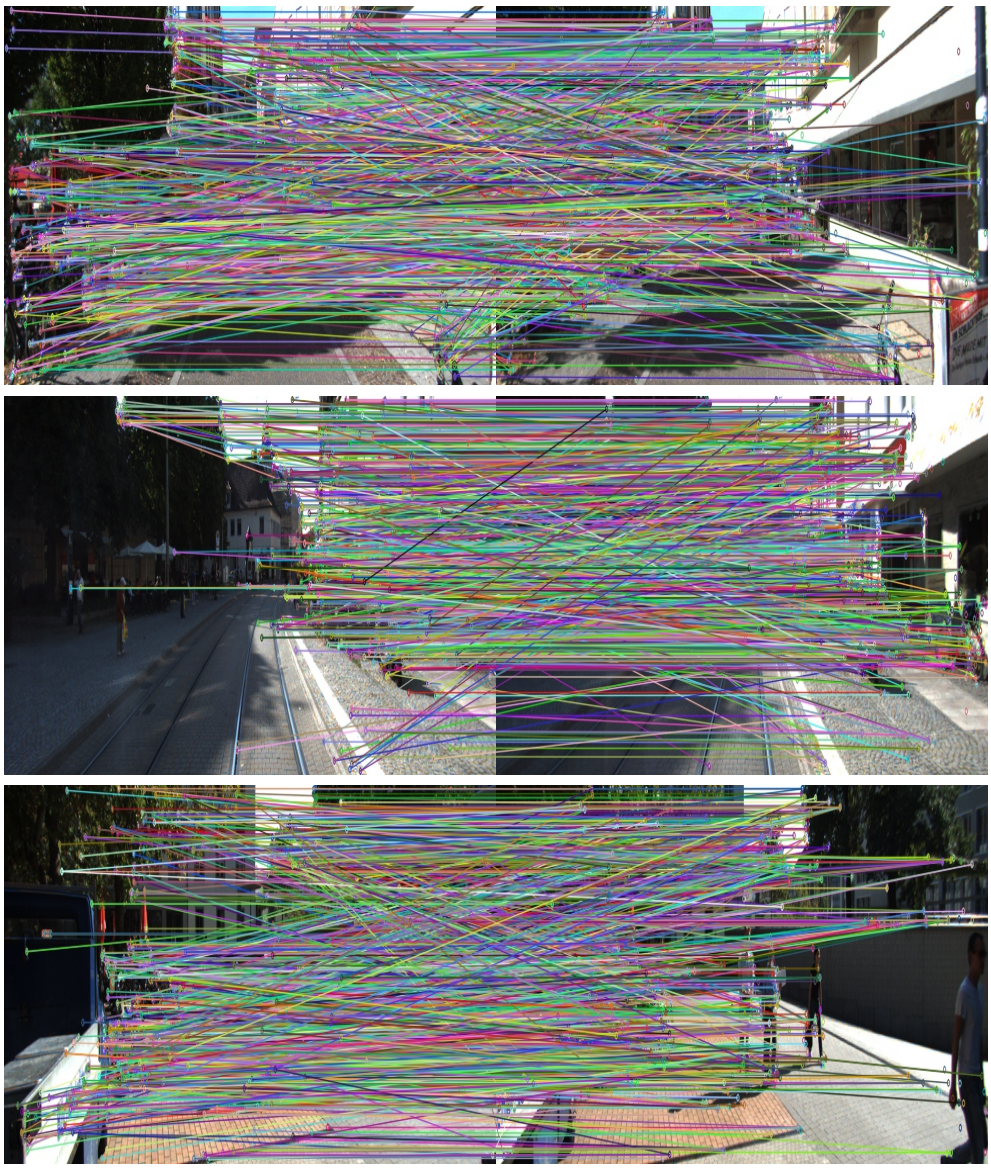
1. Construct a scale space representation of image
2. Find keypoints with Difference-of-Gaussian (DoG)
3. Reject bad / poor keypoints (similar to Harris approach)
4. Assign orientation to keypoints
5. Assemble full descriptor

OpenCV has an API to detect these SIFT features and compute the descriptors for the features which we are using in this assignment.

As seen in the images above, we can see that there are quite a few keypoints (1000) per image. However, many of these keypoints are unreliable and would not work well for feature matching. For a feature to be considered a good feature it must be salient, local, repeatable, and compact. Of the 1000 keypoints in each image, we can see that some of the keypoints are either located on the ground (Image 2), wall (Image 5), or tree (Image 4) which may lead

to wrong matches due to perceptual aliasing. Further, in Image 4 we can see many keypoints on the shadows of trees, which are not repeateable, as the shadows might change as per the sun's location when the photos are taken.

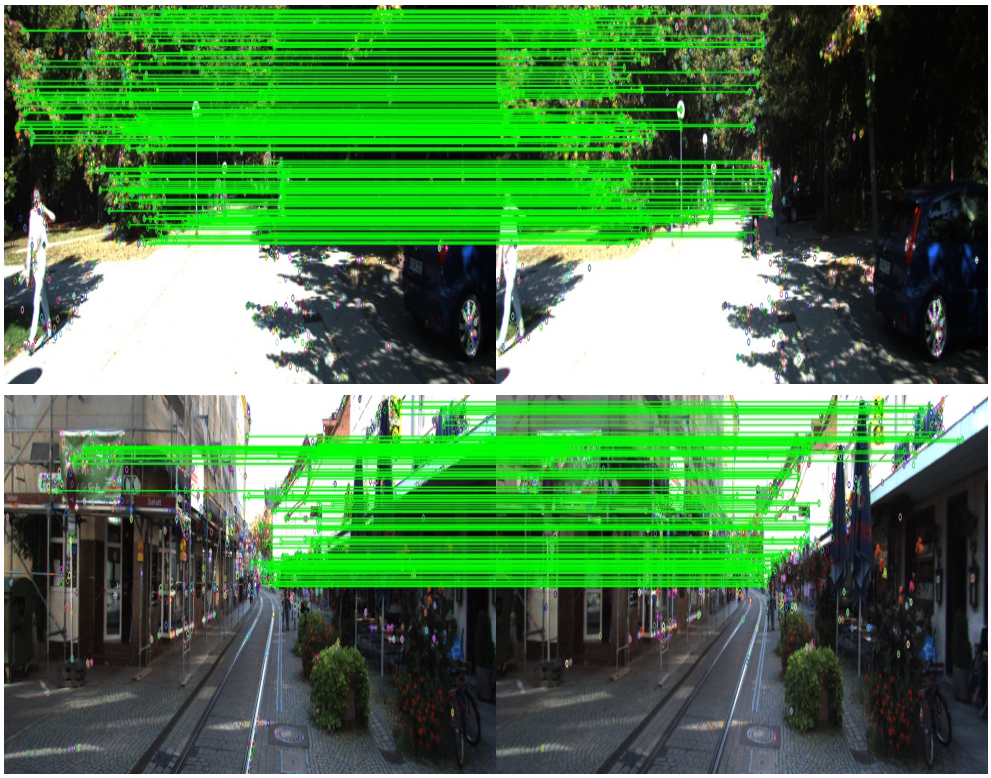
Q 1.2 Feature Matching



For Feature Matching, we are using a brute-force matcher (BF-Matcher). BF-Matcher uses the descriptors calculated in the previous section to perform the matches. It takes the descriptor of one feature in the first image and is matched with all the features in the corresponding image. In this implementation, we return k (2) nearest neighbours. Finally, we find the good matches using Lowe's ratio test. As this is a brute-force approach of feature matching, it is computationally expensive and takes a lot of time depending on the number of descriptors. As an alternative, we can also use FLANN (Fast library for approximate nearest neighbours) for better speed improvements at the cost of relatively lower accuracy.

As it is given that the images follow epipolar constraints, ideally we should be seeing horizontal match lines. But as we can see in the images above, the matches do not seem to follow epipolar constraints. This is due to the unstable keypoints and their corresponding descriptors which the matcher is unable to distinguish from the actual matching descriptor. These wrong matches are of no use and may severely affect the performance of any downstream perception application like object detection, place recognition, slam, etc. Hence, we need to filter these keypoints to provide reliable matches. This can be done through outlier rejection algorithms like Ransac, Prosac, etc.

Q 1.3 Outlier Rejection



For outlier rejection, we are using Ransac (Random Sample Consensus) [2]. Ransac tries to estimate a model for a given set of data containing outliers. Basically the algorithm is as follows (Reference ROB 501 notes):

1. Determine smallest number of data points required to fit the model of choice.
2. Draw smallest subset randomly from available data
3. Check size of subset of all points that are within threshold of model
4. If subset size is above threshold, terminate (and then improve model)
5. Otherwise, repeat 2—5 above until threshold is met or maximum iterations have been reached.

In this part, we use OpenCV's ransac implementation. We tune the parameters for ransac as follows to get the best results:

1. Maximum allowed reprojection error threshold to treat a point pair as an inlier = 8.0
2. Confidence level = 0.99
3. Maximum number of ransac iterations = 10000

We can clearly see in the images above that after applying Ransac, the matches follow epipolar constraints and give horizontal match lines as expected. For each test image, we get approximately 120 matches from the 1000 keypoints that we detected earlier. However, these 120 matches are stable and can be used in perception algorithms reliably.

3D Point Cloud Registration

Q 2.1 ICP Loss

Rabbit

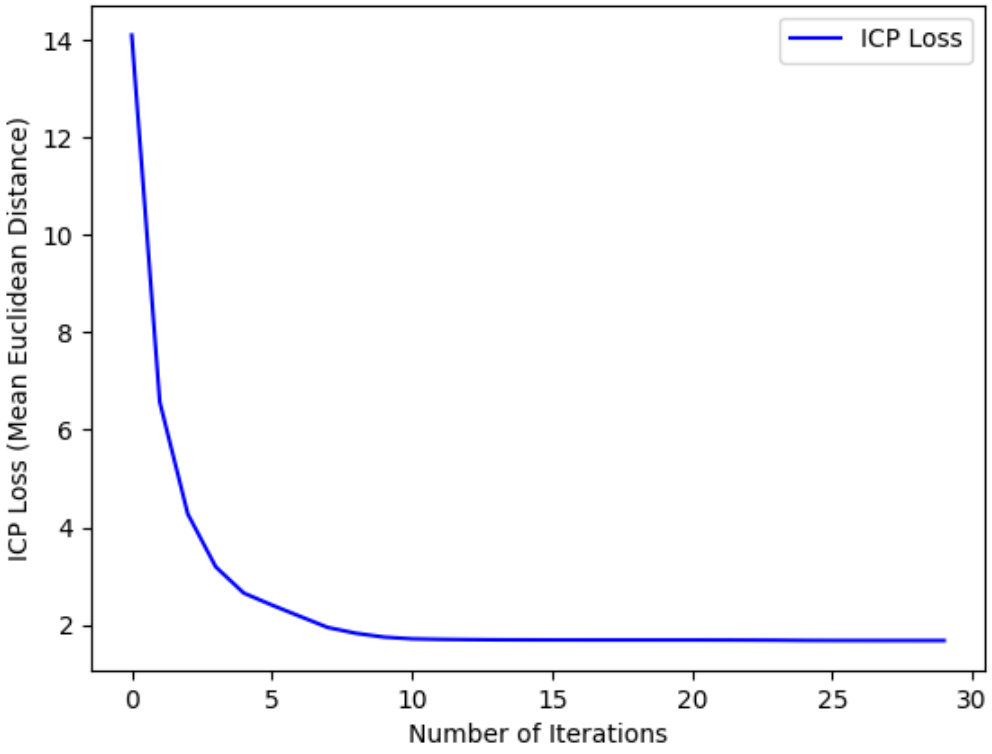


Figure 1: ICP Loss for the Rabbit

Dragon

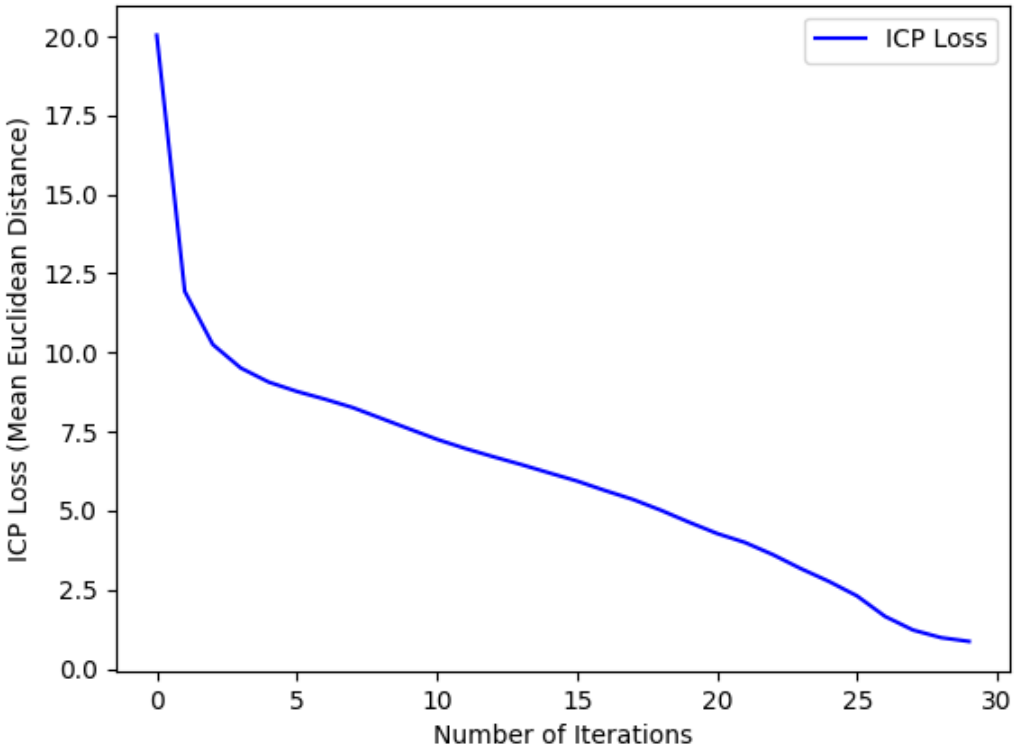


Figure 2: ICP Loss for the Dragon

Armadillo

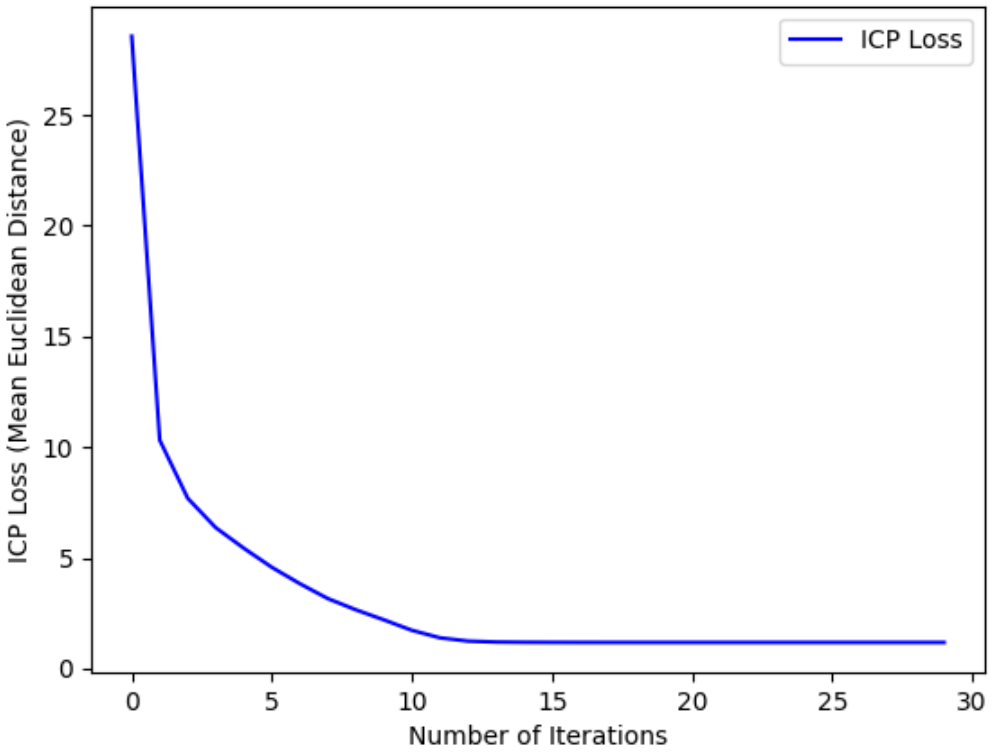


Figure 3: ICP Loss for the Armadillo

To get the ICP loss, we calculate the mean euclidean distance between the points in the source pointcloud and the target pointcloud at each ICP iteration. As we apply the transformation to the source pointcloud at each ICP iteration based on the estimated pose, we expect the source pointcloud to slowly merge with the target pointcloud as we are trying to reduce the distance between the closest points. This can be clearly seen in the Figures 1, 2, 3, the ICP loss seems to go down monotonically, thus proving that the ICP solution is converging properly. This depends on the number of iterations we run the ICP for. For example, if we run ICP just for a few iterations, the estimation may not be accurate and we might get wrong registration due to higher ICP loss. Also, if we have too many iterations, the solution may not improve much after some iterations. Thus having too many iterations might just put additional load on the computation and take more time.

We can also see that this convergence depends on a case-to-case basis. For example, for the rabbit registration we can see in Fig. 1 that the solution converges in fewer iterations as compared to the dragon (Fig. 2) and the armadillo (Fig. 3) examples. Hence, it is not

guaranteed that the ICP solution will always work, but rather depends on the complexity of the registration and the initial guess.

Q 2.2 Pose Estimation from the Correspondence

Rabbit

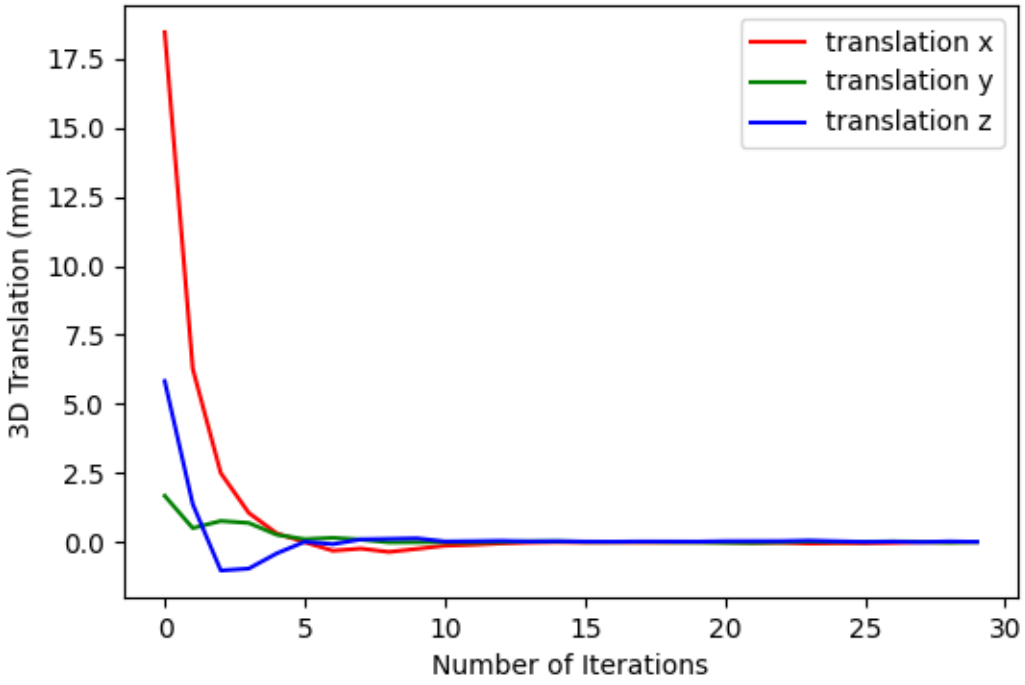


Figure 4: Estimated 3D translation for the Rabbit

Dragon

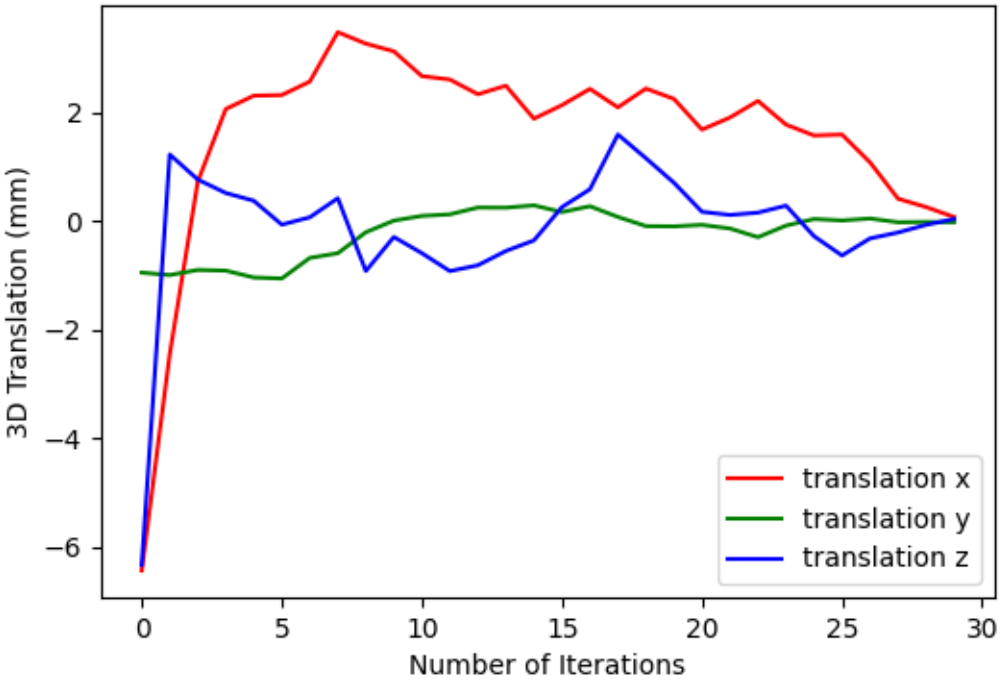


Figure 5: Estimated 3D translation for the Dragon

Armadillo

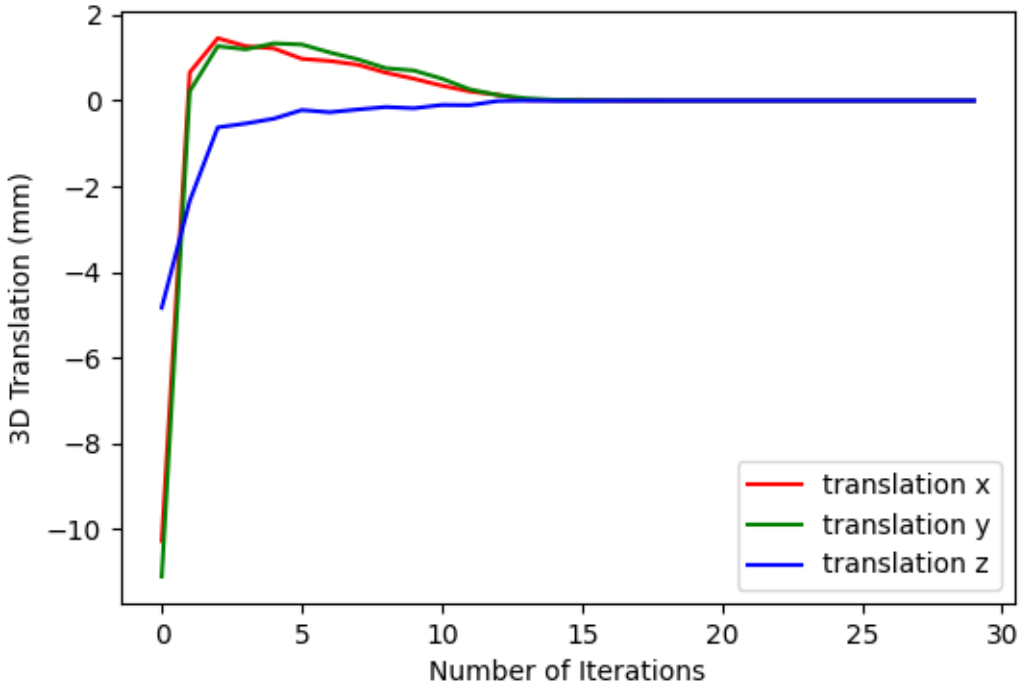


Figure 6: Estimated 3D translation for the Armadillo

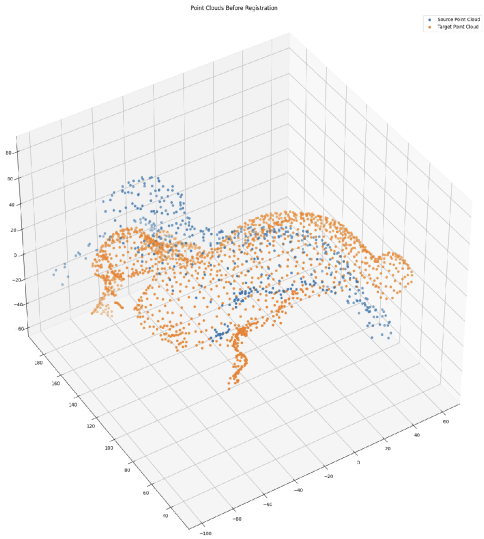
The 6D pose after ICP is a 4×4 matrix of the form

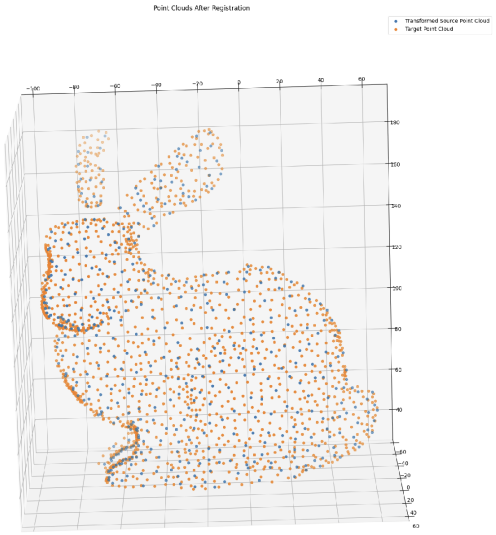
$$\begin{bmatrix} C_{DM} & -C_{DM} t_m^{DM} \\ 0 & 1 \end{bmatrix} \tag{1}$$

where the C_{DM} is the 3×3 rotation matrix and $-C_{DM} t_m^{DM}$ is the 3×1 translation vector. The plots 4, 5, and 6 show the translation plots for each ICP iteration for the rabbit, dragon and armadillo respectively. As, we can see in all the three plots, the translation seems to move towards zero as the number of iterations increase. As discussed in the previous section, as we are transforming the source pointcloud with the estimated pose after every ICP iteration, once the source pointcloud is registered perfectly, it stops any translation and rotation. These plots show this exact behavior where initially for the first few iterations we can see considerable translations in x, y, and z. But once the solution approaches towards convergence, the translations go to zero. Similarly, for the rotation matrix too, we can see there is not much changes once we reach a convergence point and the source pointcloud does not rotate as much when rotated by the estimated rotation matrix.

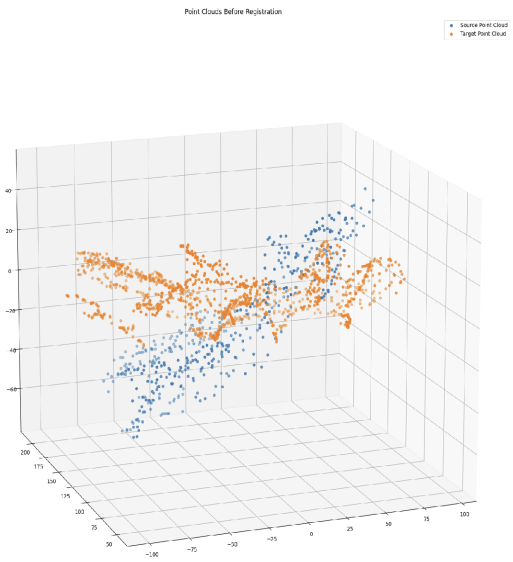
Q 2.3 Iterative Closet Point Registration

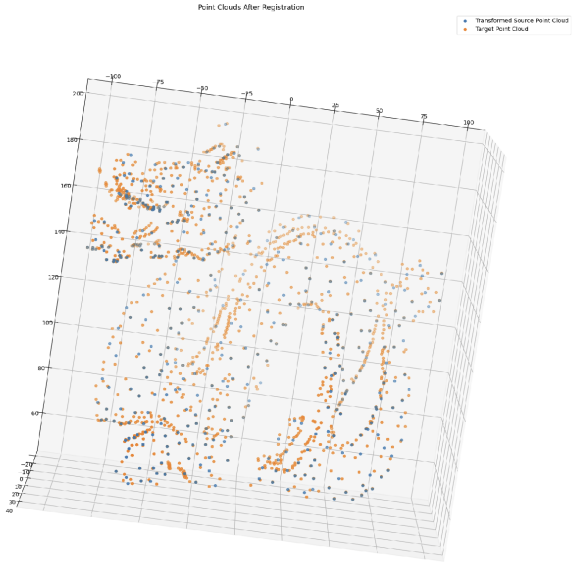
Rabbit



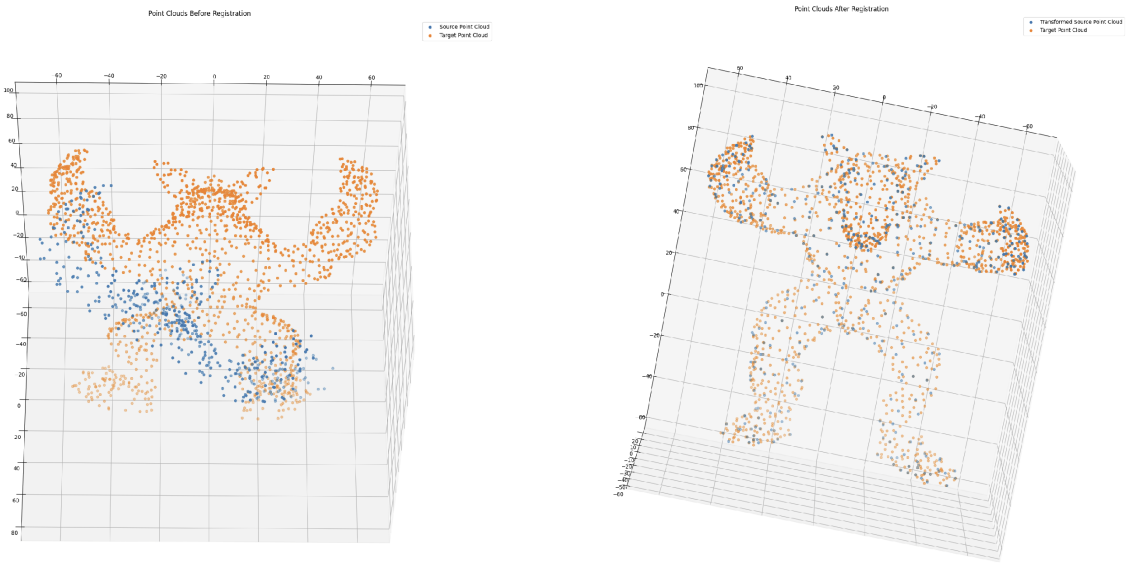


Dragon





Armadillo



The registration results are visualized in the images above. We can clearly see the offset between the objects before ICP and the proper registration after ICP.

We have calculated the rotation and translation error between the estimated pose after 30 iterations of ICP and the provided ground truth poses for the training set. For calculating the rotational error, we first calculate the roll-pitch-yaw angles from the rotation matrix and get the error for each Euler angles. To calculate the translational error, we get the difference between the estimated translation vector and the ground truth translation vector. The 6D pose error on the training set is:

1. Rabbit:
   (a) Rotation Error: $\begin{bmatrix} 3.91407395e^{-5} & -7.58936102e^{-5} & -5.77728400e^{-5} \end{bmatrix}$
   (b) Translation Error: $\begin{bmatrix} 1.57170371e^{-5} & 4.17231147e^{-5} & -4.16190054e^{-5} \end{bmatrix}$
2. Dragon
   (a) Rotation Error: $\begin{bmatrix} 3.20041963e^{-6} & -8.74959122e^{-6} & -6.70153531e^{-5} \end{bmatrix}$
   (b) Translation Error: $\begin{bmatrix} -3.13273922e^{-5} & -2.68158350e^{-5} & -2.81697660e^{-5} \end{bmatrix}$

The 6D pose output on the test (Armadillo) data set is:

$$\begin{bmatrix} 0.77754146 & -0.06607265 & 0.62535085 & -2.68972169 \\ -0.38404892 & 0.73755698 & 0.55544227 & -1.85834297 \\ -0.49793143 & -0.67204472 & 0.548106 & -5.96192544 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$