TP1 ZHANG kai

Q1

Filter algorithm

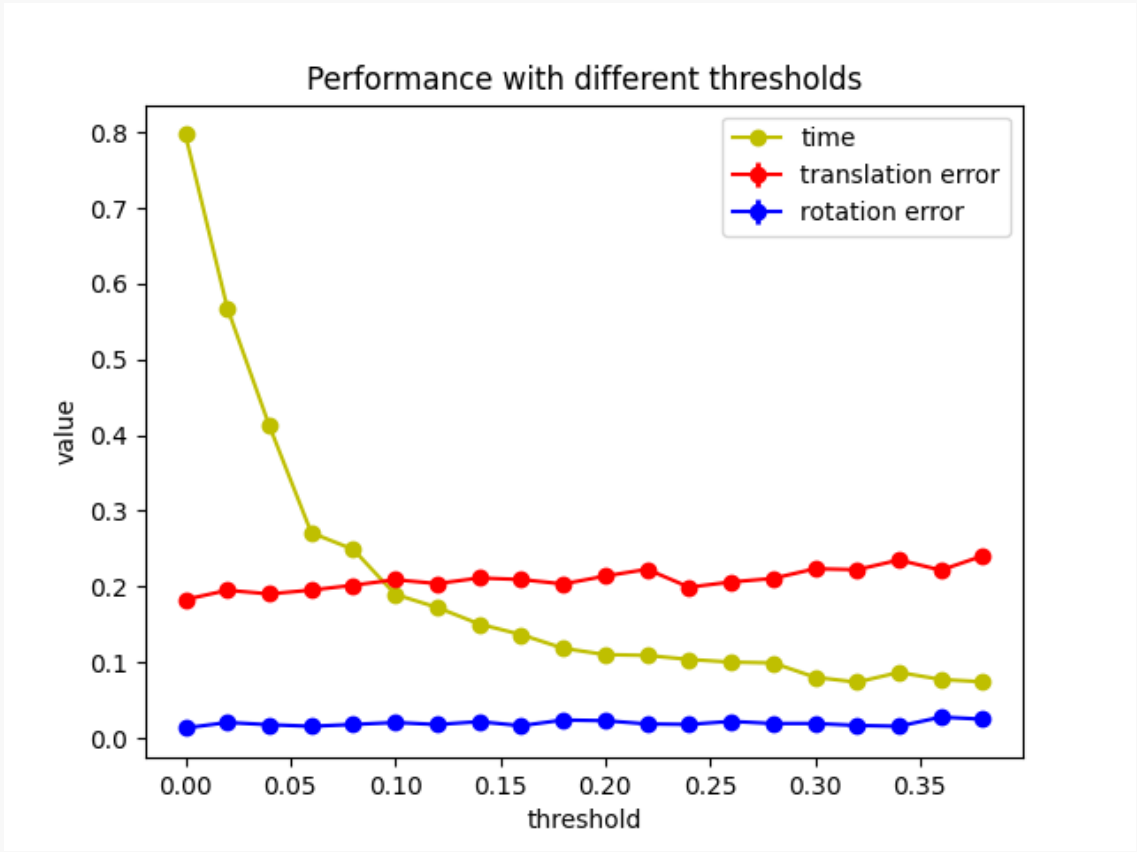


Figure 1

We can see from Figure 1 that the most appropriate threshold is 0.30, when the algorithm runs efficiently and achieves excellent performance (with less average error). The corresponding matching results is demonstrated in Figure 2.

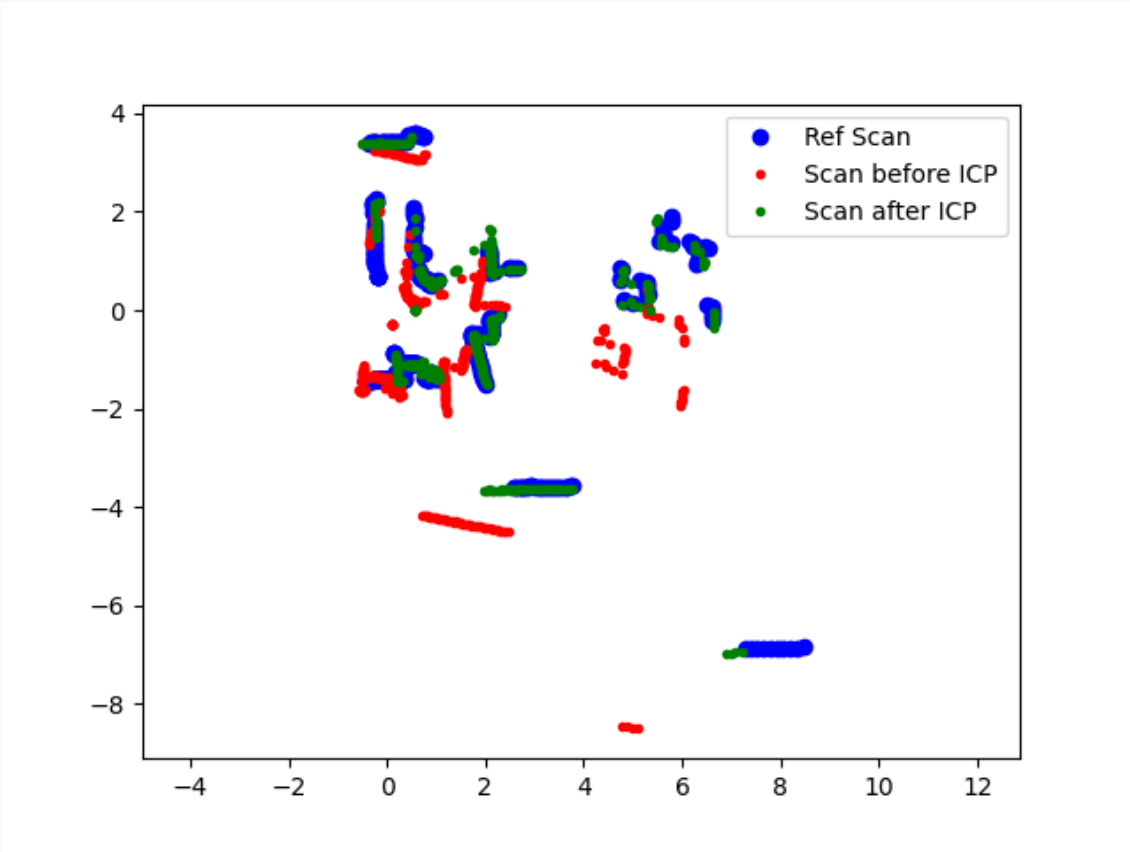


Figure 2

```
dist_thres = 0.30
num_points = dat.shape[1]
is_key_point=[False for i in range(num_points)]
print("Before filter, total {} points".format(num_points))
dat_t = np.transpose(dat)
dist = cdist(dat_t, dat_t, 'euclidean')
i = 0
for it in range(dat_t.shape[0]):
    len_next = list(np.where(dist[i, i:] > dist_thres))[0]
    is_key_point[i] = True
    if len_next.shape[0] > 0:
        i = len_next[0] + i
    else:
        break
dat_filt = dat[:, is_key_point]
print("After filter, total {} points".format(dat_filt.shape[1]))
```

Q2

xx% best matchings

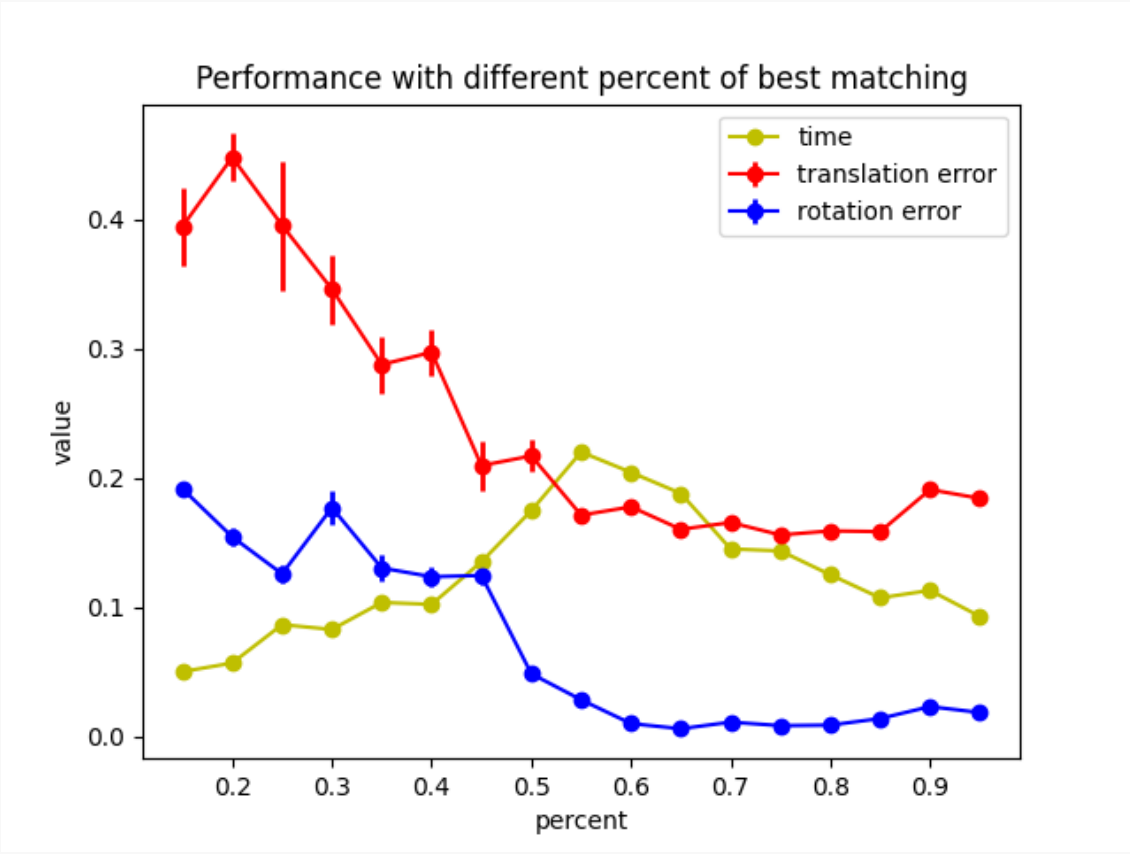


Figure 3. All the experiments are based on the condition where filter threshold equals 0.30.

The relation between the performance of ICP algorithm and the percent of best matching points is shown in Figure 3. When we pick more points for matching, the mean error decreases but the running time increases. To balance the time and error, we conclude that when the percent number is 85%, it can achieve the most best performance.

```
dat_matched = []
new_index=[]
percent_match=0.85
temp_distance=distance.copy()
temp_distance.sort()
# find top percent matchings
num_best_match=int(len(distance)*percent_match)
distance_threshold=temp_distance[num_best_match]
for i in range(len(distance)):
    if distance[i]<distance_threshold:
        dat_matched.append(dat_filt[:,i])
        new_index.append(index[i])

dat_matched=np.array(dat_matched).T
index=np.array(new_index)
```

Q3

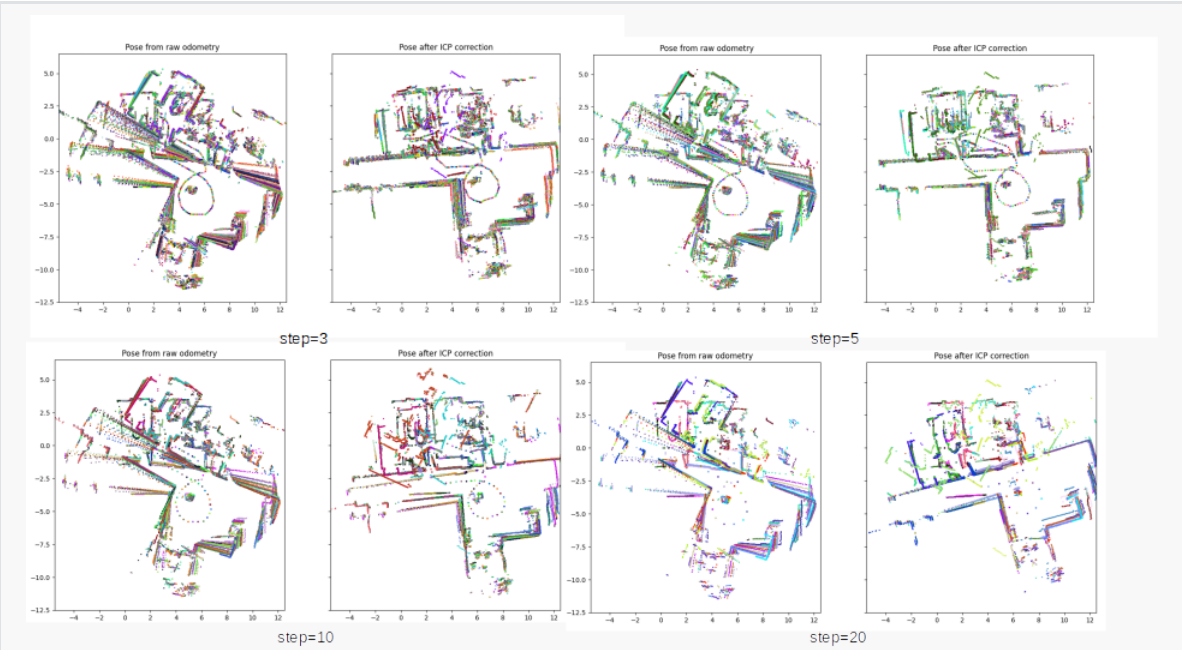
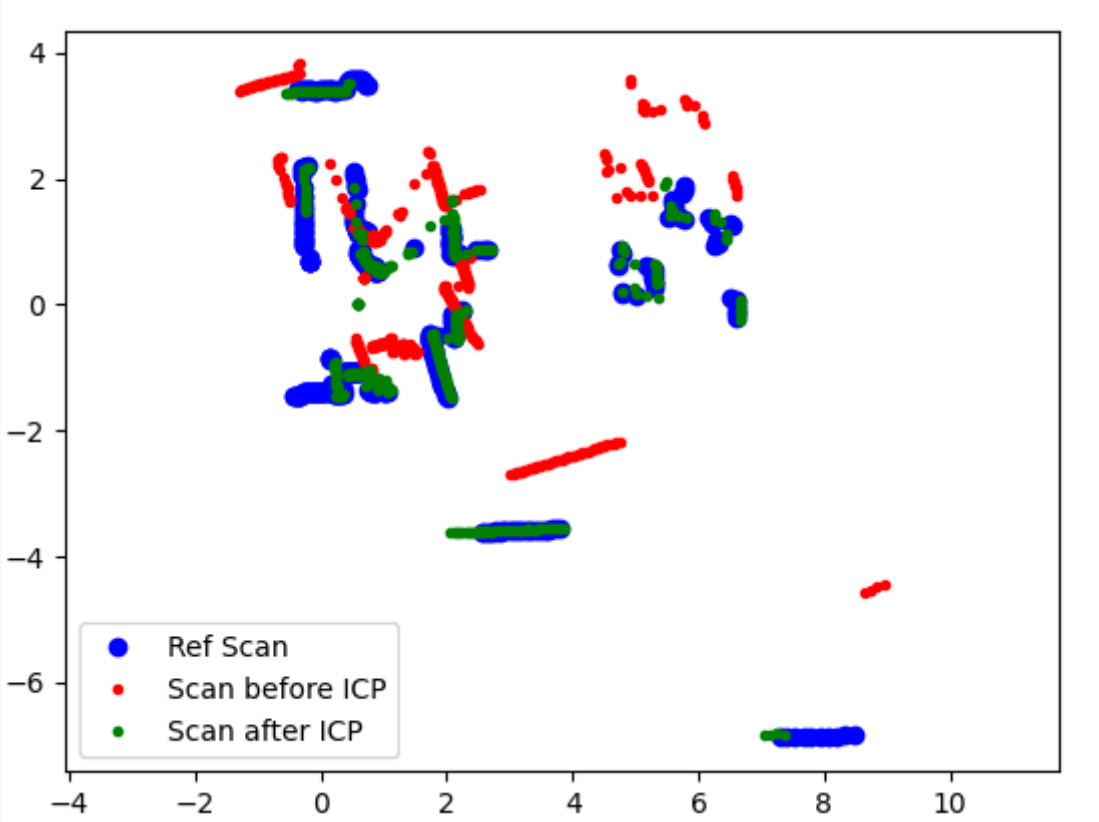


Figure 4. Performance with different steps

Figure 4 illustrates the impact of *step* to the point matching performance. With fewer steps, the points cloud is updated frequently, which introduces more details but with noise and takes more time for computation. When the step is big, the time interval between updates becomes longer, which has more computation efficiency and suffers from less noise. However, some details are lost, which may produce the isolated segments. In conclusion, an appropriate value of *step* is essential for final performance.

Q4

The closest matching

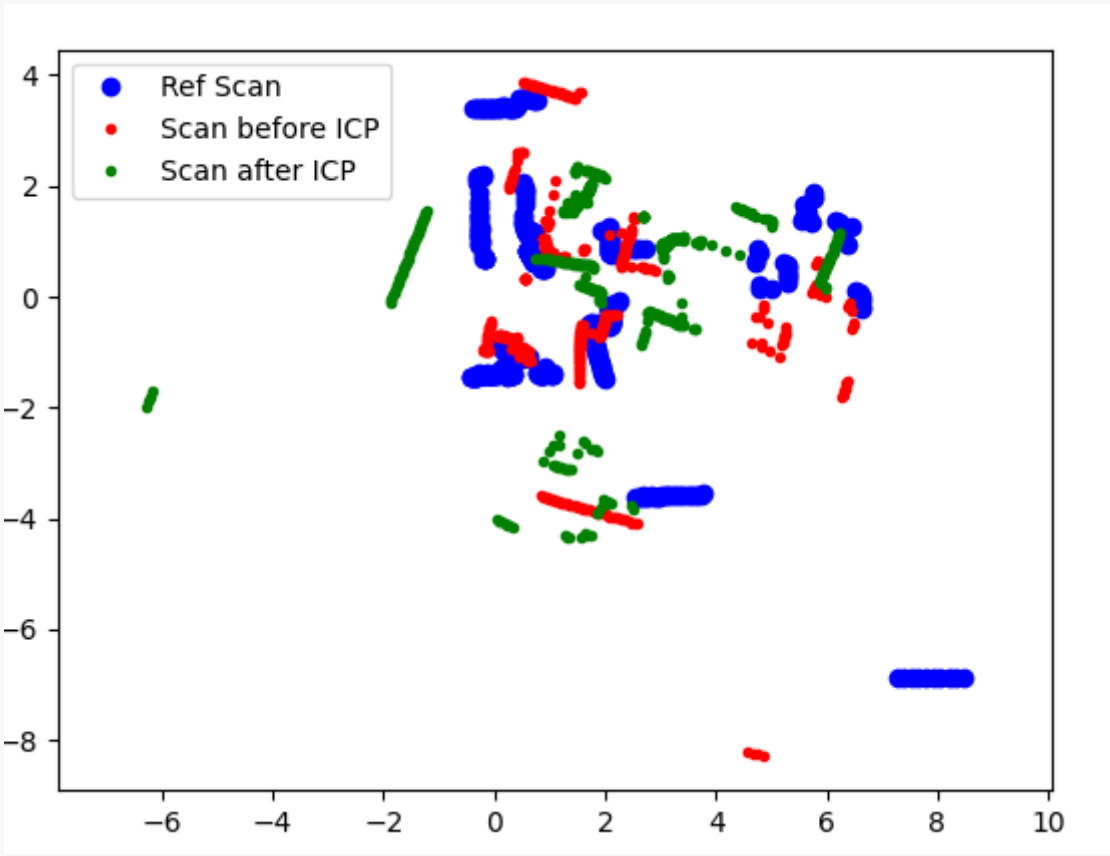


Figure

```
tree = KDTree(ref.T)
distance, index = tree.query(dat_filt.T)
mean_distance=np.mean(distance)
dat_matched = []
new_index=[]
index_set=set(index)
for i in index_set:
    ind=np.where(index==i)[0]
    min_dis_ind=np.argmin(distance[ind])
    temp_dat=dat_filt[:,ind[min_dis_ind]]
    dat_matched.append(temp_dat)
    new_index.append(i)

dat_matched=np.array(dat_matched).T
index=np.array(new_index)
```

Normal shooting matching



Figure

```
tree = KDTree(ref.T)
distance, index = tree.query(ref.T, k=2) #find the closed points to
calculate norm vector

dat_matched = []
new_index = []
min_dis=1000
vector = ref[:, index[:, 1]] - ref[:, index[:, 0]]  # 2xn
for i in range(dat_filt.shape[1]):
    dat_i = dat_filt[:, i].reshape(2, 1)
    product = np.abs(np.sum((ref - dat_i) * vector, axis=0))  # calculate
the cosine
    ind = np.argmin(product)
    if product[ind] < min_dis:
        min_dis=product[ind]
        new_index=[ind]
        dat_matched=[dat_filt[:, i]]
index = np.array(new_index)
dat_matched = np.array(dat_matched).T
x = ref[:, index] - dat_matched
meandist = np.squeeze(x.T@x)
```

Comparison

Compared with KNN matching, the normal shooting matching achieves unstable performance. Sometimes it works well but sometimes it obtains bad results. I think it is due to the isolated point cloud sets but not the structured point cloud sets. Its normal vectors hit unrelated points which caused wrong matching.