Laser scan matching through ICP Correction

David FILLIAT - ENSTA Paris

January 5, 2021

1 Question 1

Question 1 : Implement points filtering (removing points too close to each other in the 'data' scan) and show the consequences (on error, variance, computation time,...) as a function of the parameter values.

The following code removes the next points in the scan that are too close from the current one, then switch the current point to the next valid one. Note that it assumes there is not too much noise as this might lead to leave points that are too close to each other if a point between them in the scan is farther than the threshold. This is however quite rare in practice and this approximation performs well.

```python
    prevPt = dat[:, 0]
    dat_filtered = []
    for a in range(dat.shape[1]):
        pt = dat[:, a]
        if np.linalg.norm(prevPt - pt) > minResolution:
            dat_filtered.append(pt)
            prevPt = pt
    dat_filt = np.stack(dat_filtered, axis=1)
```

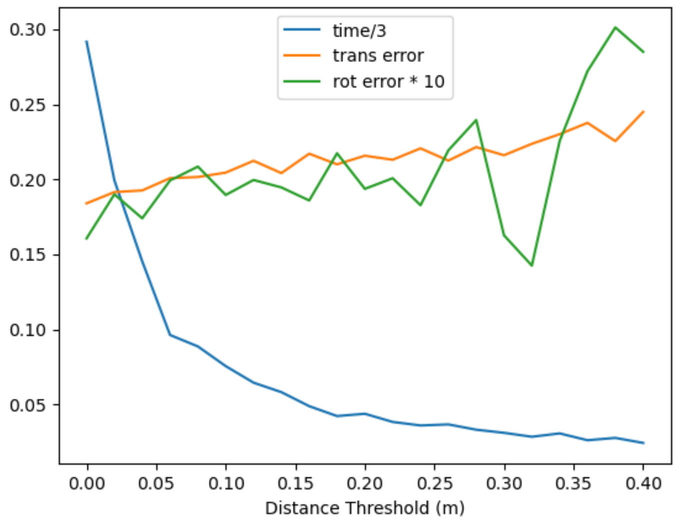

Figure 1: Effect of distance threshold on point filtering.

Testing this code with a parameter between 0 and 0.4 lead to the results reported in Figure 1. We can see that computation time steadily decrease, which is simply linked to the fact that there are less points to process afterwards. The performance almost steadily decrease, as less points are present and the discretization linked to the minimum spacing brings more and mode noise. Note also that the erratic behavior of the rotation error above 0.25 is linked to a higher variance in the results, which is apparent in the mean because we always use the same scans here for which some specific threshold are more adapted than other. Overall, we are then facing a quite common computation/precision tradeoff. A threshold around

0.05 seems to be a good compromise, which can be increased or decreased depending on the application requirements.

2 Question 2

Question 2 : Implement match filtering by keeping the XX% best matching. Try different values for XX and show the consequences (error, variance, computation time,...).

The following code sorts the distance vector and takes the distance threshold corresponding to the best_matching % values. Then points with distances below this value are selected.

```
sorted_dist = np.sort(distance)
valid = distance <= sorted_dist[int(best_matching*(len(sorted_dist)-1))] # only best
    matching
dat_matched = dat_filt[:, valid]
index = index[valid]
```

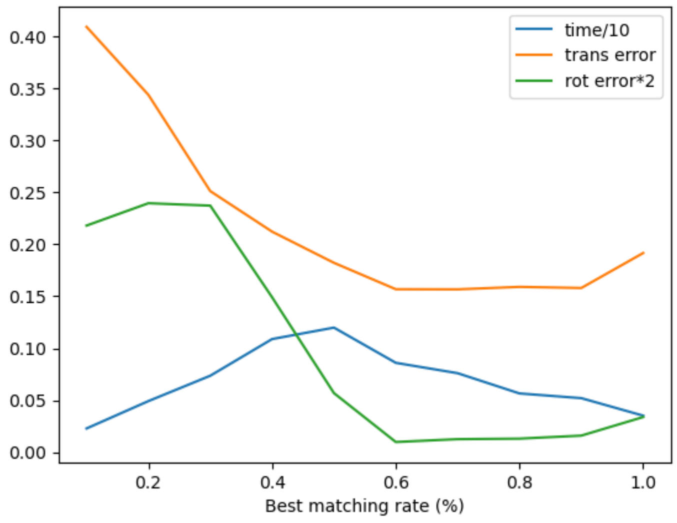

Figure 2: Effect of threshold on matching filtering.

Testing this code with a parameter between 10% and 100% lead to the results reported in Figure 2. We can observe that the computation times first grow below 50% and then decreases again. The initial growth is linked to the fact that more and more matching are used. The decrease after 50% is linked to the fact that with more matching, the updates are better and the algorithm converges with less iterations. The performance is quite bad with a low number of points and steadily increases up to 90%, but increases again at 100%. This is linked to the fact that there are almost always a few bad matching that are removed when the 10% of worst matching are discarded. A reasonable compromise is therefore to keep between 80% and 90% of the matchings.

3 Question 3

Question 3 : Use the best setting you found with the icpLocalization function. Analyse visually the quality of the result as a function of the step parameter and of the parameters of the previous filtering. Propose a reasonable compromise between the quality of the localization and the computation time.

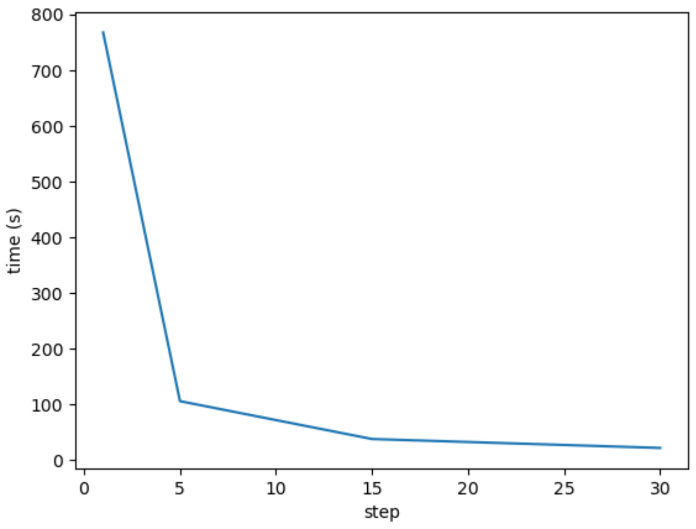

Figure 3: Effect of `step` on the computation time.

Figure 4 shows the effect of the `step` variable on the map quality and Figure 3 shows the corresponding computation time. We can see that the best map is obtained for `step=5`, with a reasonable computation time and a good coverage of the environment features.

For `step=1`, the map quality is quite low because, while this configuration with very close scans is more favorable for ICP, applying ICP for each scan leads to accumulation of small errors at each iteration that lead to a worse map than for `step=5`. The computation time is also very high, because ICP is used more often, and the map contains much more scan which also add computation time for finding and matching to the nearest neighbor. We can also see that there are much more scans than needed to represent correctly the obstacles.

For `step=15` and `step=30`, the computation time continue to decrease, but the map quality degrades because ICP is performed on scans that are far from each other and therefore do not overlap correctly in some situations. It is for example the case when living the two rooms at the top and entering the corridor: matching scan taken inside and outside the room leads to a poor overlap and bad position estimate.

4 Question 4

Question 4 : Implement matching using 'normal shooting' instead of nearest neighbor matching and compare with the previous approach.

Normal shooting is in fact not well adapted to the problems we are facing. It is mostly interesting where you have a continuous surface representation, while we have a discrete set of points. There are several problems that you need to take into account to reach correct performances:

- Normals are badly defined in areas with lot of noise and around the obstacles borders. To filter this, you can compute tangents with the previous and next point and only consider the point if the tangents are mostly aligned (Figure 5, left: point with green tangents will be kept, points with red tangents will be discarded. Code line 20).

- If the scan is mostly aligned with the reference (Figure 5, right), there will often be points without any correct correspondance on their normal because all the neighbor points are aligned. To avoid this, you can use normal shooting only when the closest point is far enough (Figure 5, center, Code line 21), and use the nearest neighbor otherwise.

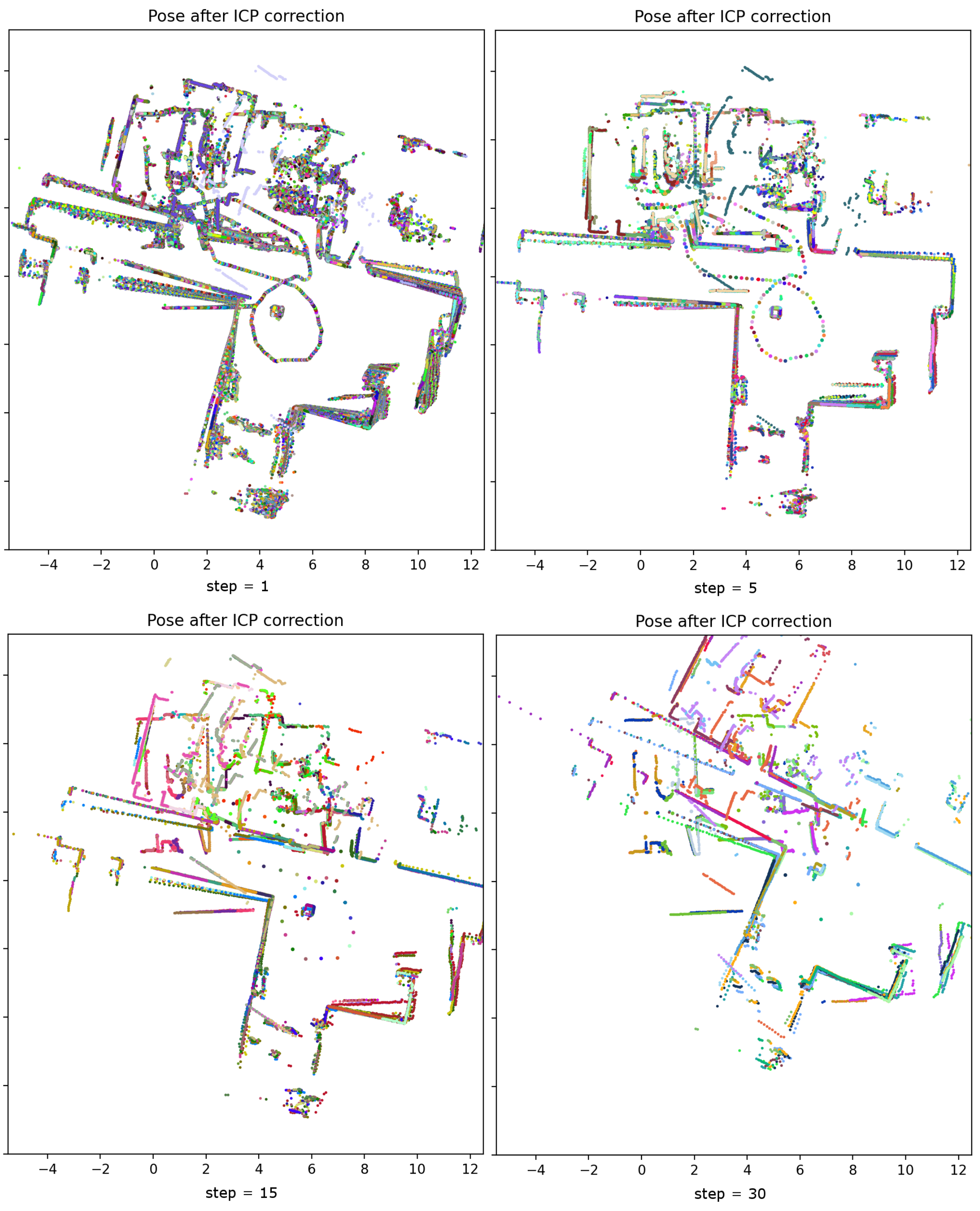

Figure 4: Effect of `step` on map quality.

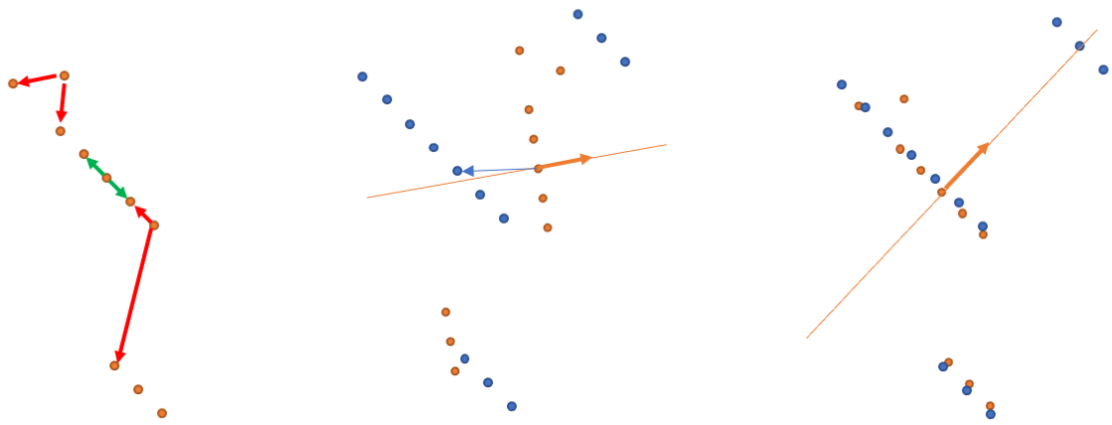
Figure 5: Normal shooting limitations with discrete scans.

- The point that is the best aligned on the normal can be in very different area of the scan (Figure 5, right). To avoid this, you can restrain search around the nearest neighbor (Code line 23) and validate points only if they are not much farther than the closest point (Code line 25).

Implementing all these tricks lead to the code below:

```
 1 # ----- Find nearest Neighbors for each point, using kd-trees for speed
 2 tree = KDTree(ref.T)
 3 distance, index = tree.query(dat_filt.T)
 4 meandist = np.mean(distance)
 5 
 6 # ----- Find association using normal shooting (reusing nearest neighbor for matching filtering)
 7 normal_idx = np.zeros(dat_filt.shape[1], dtype=int)
 8 normal_cos = np.ones(dat_filt.shape[1])
 9 NS = 0
10 NN = 0
11 for i in range(1,dat_filt.shape[1] - 1):
12     # computes tangent with points before and after to filter regular areas
13     vect_tan_suiv = dat_filt[:, i + 1] - dat_filt[:, i]
14     vect_tan_prec = dat_filt[:, i - 1] - dat_filt[:, i]
15 
16     vect_tan = vect_tan_suiv - vect_tan_prec # mean tangent vector
17     cos_tan = abs(np.dot(vect_tan_suiv , vect_tan_prec) /(np.linalg.norm(vect_tan_suiv) * np.linalg.norm(vect_tan_prec))) # angle between the two tangents
18 
19 
20     if cos_tan > 0.9: # if we are in a regular area
21         if distance[i] > 3 * minResolution: # if we are far enough to have a good normal shooting
22             cos = np.ones(ref.shape[1])
23             for j in range(max(0,index[i]-20), min(index[i]+20, ref.shape[1])): # look for normal shooting around nearest neighbor
24                 vect_norm = ref[:, j] - dat_filt[:, i]
25                 if np.linalg.norm(vect_norm) < 1.3 * distance[i]:
26                     cos[j] = abs(np.dot(vect_tan , vect_norm) /(np.linalg.norm(vect_tan) * np.linalg.norm(vect_norm))) # angle between tangent and normal
27                     #print(cos[j])
28 
29             normal_idx[i] = np.argmin(cos)
30             normal_cos[i] = np.min(cos)
31             if normal_cos[i] <= 0.1:
32                 NS = NS + 1
33 
34         else: # if we are close to the other scan, keep nearest neighbor
35             NN=NN+1
```

```
                normal_idx[i] = index[i]
                normal_cos[i] = 0

        if distance[i] > 2 * meandist or cos_tan<= 0.9:
            # remove matchings with far nearest neighbor and in irregular areas
            normal_idx[i] = 0
            normal_cos[i] = 1.0


    valid = normal_cos <= 0.1 # keep only matchings close to normal
    dat_matched = np.array(dat_filt[:, valid])
    index = normal_idx[valid]
    print('Matched with normal shooting : ', NS)
    print('Matched with NN : ', NN)
    print('Valid matchings : ',np.sum(valid))
```

In the same configuration, this procedure will improve a bit on the performances compared to nearest neighbor (Mean (var) translation error : 1.55e-01 (5.49e-06), Mean (var) rotation error : 4.84e-03 (7.49e-06) for normal shooting VS Mean (var) translation error : 1.92e-01 (1.49e-04), Mean (var) rotation error : 1.88e-02 (4.26e-05) for nearest neighbor). However, the computation time is much larger as this approach computes the nearest neighbor and add a lot of processing (Mean computation time : 2.27 VS : 0.68).

For the lab report, I am not expecting all these features, a correct basic implementation of normal shooting and a quick analysis of the limitations in our scenario will get the maximum points.