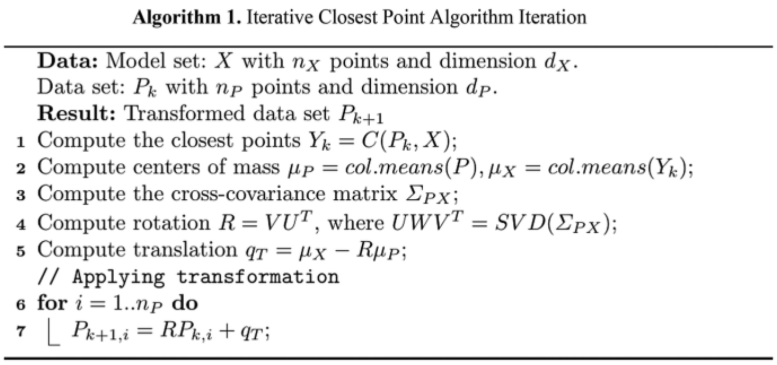
Project 1: ICP Odometry
By: Young, James Yang (20740589)

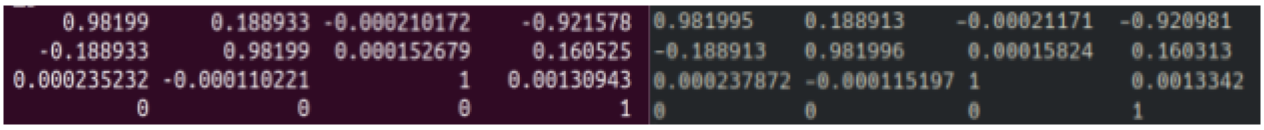
The main task of question 1 is to implement our own ICP algorithm different from the one provided. The implementation, steps, and equations needed are shown in the image below. For getting the nearest neighbor, I used the TA's advise to use KDTree and used PCL's KDTree library to obtain the nearest neighbor. For the rest of the steps, I mainly used the Eigen C++ library for performing matrix operations such as computing the SVD for the rotation matrix.



Algorithm 1. Iterative Closest Point Algorithm Iteration

```
  Data: Model set: $X$ with $n_X$ points and dimension $d_X$.
  Data set: $P_k$ with $n_P$ points and dimension $d_P$.
  Result: Transformed data set $P_{k+1}$
1 Compute the closest points $Y_k = C(P_k, X)$;
2 Compute centers of mass $\mu_P = col.means(P), \mu_X = col.means(Y_k)$;
3 Compute the cross-covariance matrix $\Sigma_{PX}$;
4 Compute rotation $R = VU^T$, where $UWV^T = SVD(\Sigma_{PX})$;
5 Compute translation $q_T = \mu_X - R\mu_P$;
  // Applying transformation
6 for $i = 1..n_P$ do
7     $P_{k+1,i} = RP_{k,i} + q_T$;
```

My implementation was really simple as it only used point-to-point ICP without weightings. For a more robust algorithm, in the future I could implement point-to-plane and try different sampling instead of just uniform sampling. Nevertheless, I feel the results of my ICP transformation matrix were satisfactory, ending up quite close to the ground truth given in "true_tf.txt".



```
     0.98199     0.188933 -0.000210172    -0.921578
   -0.188933      0.98199  0.000152679     0.160525
0.000235232 -0.000110221            1   0.00130943
           0            0            0            1
```

```
0.981995    0.188913     -0.00021171  -0.920981
-0.188913   0.981996     0.00015824   0.160313
0.000237872 -0.000115197 1            0.0013342
0           0            0            1
```

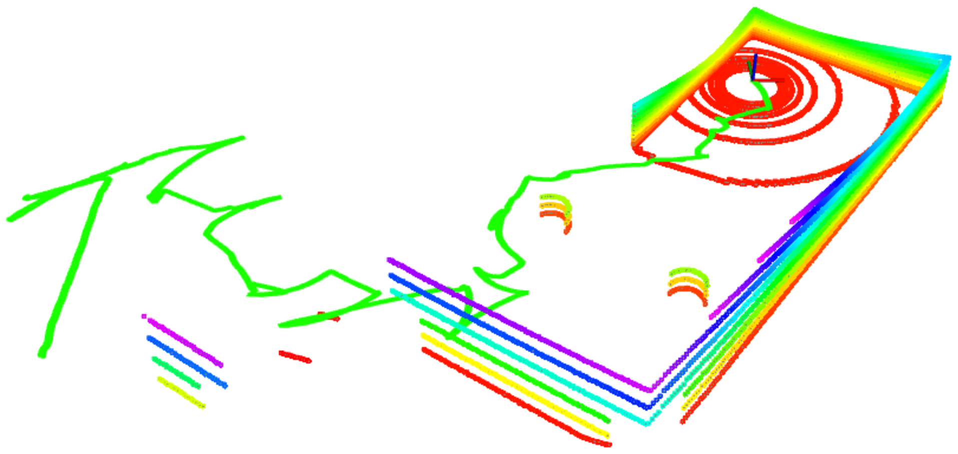
Own Implementation Result (left) vs. Ground Truth (right)

The next 3 parts required us to use the ICP implemented above to estimate the odometry of a robot with different pipelines given. For all 3 parts, I first downsampled the frame and reference point clouds using the VoxelGrid library given by PCL. Then, I calculated the predicted transformation from previous to current by taking the inverse of the initial pose and multiplying by the initial pose. This prediction is fed into the ICP algorithm as part of the initial guess for all 3 pipelines.

For pipeline 2, alongside the prediction, we had to pass the previous LiDar frame to get the transformation. Since this pipeline is constantly feeding frames into the ICP, with a small error it will accumulate and cause drift.

Therefore, pipeline 3 can alleviate this by using keyframes. Instead of constantly updating the ICP algorithm with a new frame, the keyframe pipeline allows us to choose when to update the keyframe to avoid causing drift. For my implementation, I only tested timestamps using the C++ std::chronos library.

For both pipeline 2 and 3, I was able to implement a solution that travels around most of the map. However, the trajectory is off as seen in the image below. This is most likely due to my transformation from body to world matrix not properly set, which is why the robot does not move in a correct way.



Pipeline 4 uses the keyframe from pipeline 3 to create a local map to feed into the ICP instead. Unfortunately, my implementation for this didn't work well most likely due to the fact that I was adding keyframes to the local map without properly filtering and downsampling it. This is why the further my robot moved the slower and more inaccurate it became until the robot eventually just stopped halfway through the simulation.

Resources Used:

- https://pcl.readthedocs.io/projects/tutorials/en/latest
- https://github.com/zjudmd1015/icp/blob/master/icp.cpp
- https://eigen.tuxfamily.org/dox/modules.html
- Lecture 5 Slides
- ChatGPT