Midterm Competition Report

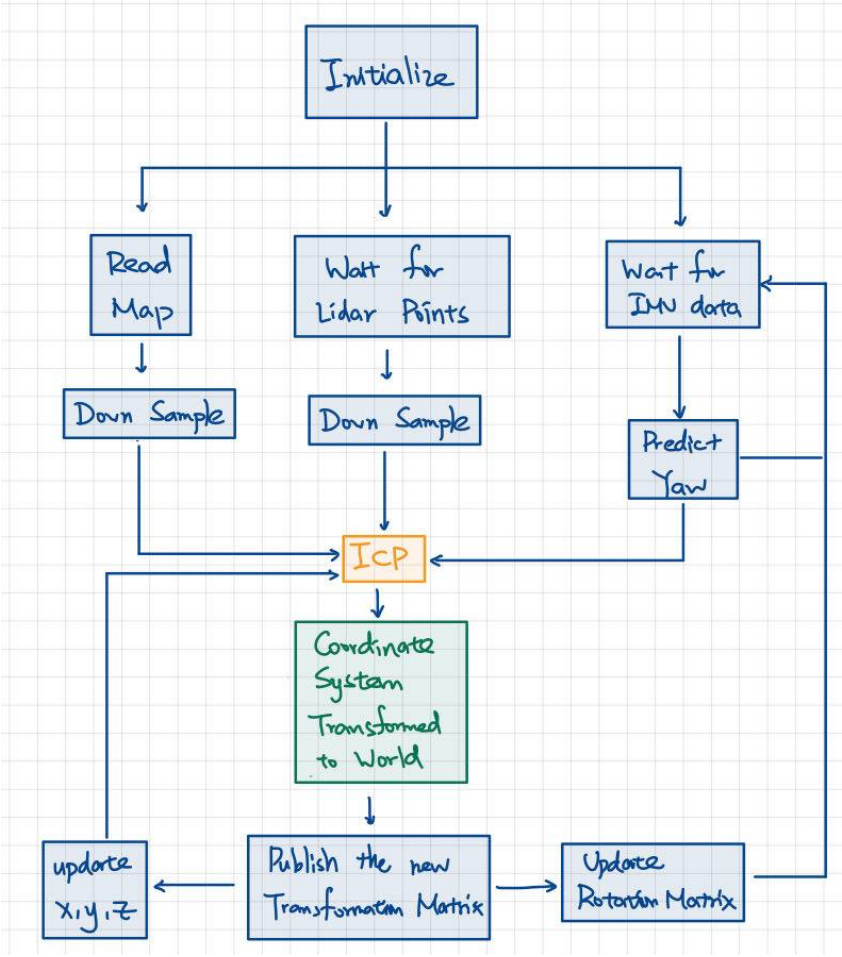
[EECN30168] Self-Driving Cars 2022

Student ID: 311605004 **Name:** 劉子齊 **Date:** 2022.11.24

1. Introduction

In the midterm competition, our goal is to develop a localization module for the estimation of the pose of the self-driving car in the corresponding dataset, which are the ITRI dataset, and the nuScenes dataset. In this project, I mainly applied the iterative closest point as my localization algorithm, and I will explain my implementation in the following session.

2. Pipeline



3. Implementation Details

In the very beginning of this project, I setup the parameters and the location of the storage file, and declare the nodes that would be subscribes and published, and set the initial guess of the car as an identity matrix, which is shown in the following figures.

```
float mapLeafSize = 1., scanLeafSize = 1.;
std::vector<float> d_max_list, n_iter_list;

ros::NodeHandle _nh;
ros::Subscriber sub_map, sub_points, sub_gps, sub_imu;
ros::Publisher pub_points, pub_pose, pub_test;
tf::TransformBroadcaster br;

pcl::PointCloud<pcl::PointXYZI>::Ptr map_points;
pcl::PointXYZ gps_point;
bool gps_ready = false, map_ready = false, initialied = false;
Eigen::Matrix4f init_guess;
Eigen::Matrix3f eigen_C_now;
int cnt = 0;

pcl::IterativeClosestPoint<pcl::PointXYZI, pcl::PointXYZI> icp;
pcl::VoxelGrid<pcl::PointXYZI> voxel_filter;

std::string result_save_path;
std::ofstream outfile;
geometry_msgs::Transform car2Lidar;
std::string mapFrame, lidarFrame;
```

```
_nh.param<std::vector<float>>("baselink2lidar_trans", trans, std::vector<float>());
_nh.param<std::vector<float>>("baselink2lidar_rot", rot, std::vector<float>());
_nh.param<std::string>("result_save_path", result_save_path, "result.csv");
_nh.param<float>("scanLeafSize", scanLeafSize, 1.0);
_nh.param<float>("mapLeafSize", mapLeafSize, 1.0);
_nh.param<std::string>("mapFrame", mapFrame, "world");
_nh.param<std::string>("lidarFrame", lidarFrame, "nuscenes_lidar");

sub_map = _nh.subscribe("/map", 1, &Localizer::map_callback, this);
sub_points = _nh.subscribe("/lidar_points", 400, &Localizer::pc_callback, this);
sub_gps = _nh.subscribe("/gps", 1, &Localizer::gps_callback, this);
sub_imu = _nh.subscribe("/imu/data", 1, &Localizer::imu_callback, this);
pub_points = _nh.advertise<sensor_msgs::PointCloud2>("/transformed_points", 1);
pub_pose = _nh.advertise<geometry_msgs::PoseStamped>("/lidar_pose", 1);
pub_test = _nh.advertise<sensor_msgs::PointCloud2>("/test_points", 1);
init_guess.setIdentity();
ROS_INFO("%s initialized", ros::this_node::getName().c_str());
```

Furthermore, we can find the pc_callback function, which is responsible of conducting Iterative Closest Points on the scanning point cloud obtained by the lidar on the car and the map point cloud, and publish the final result back through ROS. Besides, we can also find the map_callback, and the gps_callback functions, which are responsible of handling the map and the GPS data received from rosbag.

```
void map_callback(const sensor_msgs::PointCloud2::ConstPtr& msg){
void pc_callback(const sensor_msgs::PointCloud2::ConstPtr& msg){
void gps_callback(const geometry_msgs::PointStamped::ConstPtr& msg){
```

Let us we move onto the align_map function, this function is the main part of this project, where we conduct the Iterative Closest Point algorithm to find the best result. First of all, we have to down sample the point clouds and we will have to remove some points that are unnecessary, and might even mislead the result in some

circumstances, when conducting the Iterative Closest Point algorithm. The implementations of the down sampling and the removal of points are shown in the following figure.

```
pcl::PassThrough<pcl::PointXYZI> pass;
pass.setInputCloud(scan_points);
pass.setFilterFieldName("z");
      pass.setFilterLimits(-5, 7);
pass.filter(*filtered_scan_ptr);

//*filtered_scan_ptr = *scan_points;
voxel_filter.setInputCloud(filtered_scan_ptr);
voxel_filter.setLeafSize(0.2f, 0.2f, 0.2f);
voxel_filter.filter(*filtered_scan_ptr);
std::cout << "Filtered Scan Size: " << filtered_scan_ptr->points.size() << std::endl;

std::cout << "Original Map Size: " << map_points->points.size() << std::endl;
voxel_filter.setInputCloud(map_points);
voxel_filter.setLeafSize(0.2f, 0.2f, 0.2f);
voxel_filter.filter(*filtered_map_ptr);
std::cout << "filtered_map_ptr: " << filtered_map_ptr->points.size() << std::endl;
```

```
pcl::PassThrough<pcl::PointXYZI> pass;
pass.setInputCloud(scan_points);
pass.setFilterFieldName("z");
      pass.setFilterLimits(-5, 7);
pass.filter(*filtered_scan_ptr);


pcl::PassThrough<pcl::PointXYZI> pass2;
pass2.setInputCloud(filtered_scan_ptr);
    pass2.setFilterFieldName("y");
pass2.setFilterLimits(-20, 20);
pass2.setFilterLimitsNegative(true);
pass2.filter(*filtered_scan_ptr);
```

Then it comes to the finding of the initial orientation for the first scan, which implementation is shown in the following figure. At first, I tried to set the initial pose as a constant. However, I found the results didn't went to well by the convergence fitness score of the initial pose. Hence, I attempt to find the best initial pose by trying out different yaw degrees and eventually make use of the yaw degree with the best fitness score after conducting the Iterative Closest Points algorithm.

```
if(!initialied){
  pcl::IterativeClosestPoint<pcl::PointXYZI, pcl::PointXYZI> first_icp;
  float yaw, min_yaw, min_score = std::numeric_limits<float>::max();
  Eigen::Matrix4f min_pose(Eigen::Matrix4f::Identity());
      /* [Part 3] you can perform ICP several times to find a good initial guess */

  // yaw = -M_PI * 1.23;

  // min_pose << cos(yaw), -sin(yaw), 0.0, gps_point.x,
  //             sin(yaw),  cos(yaw), 0.0, gps_point.y,
  //                  0.0,       0.0, 1.0, gps_point.z,
  //                  0.0,       0.0, 0.0,         1.0;

  for(yaw = 0; yaw < (M_PI * 2); yaw += 0.6){
    Eigen::Translation3f init_translation(gps_point.x, gps_point.y, gps_point.z);
    Eigen::AngleAxisf init_rotation_z(yaw, Eigen::Vector3f::UnitZ());
    init_guess = (init_translation * init_rotation_z).matrix();

    first_icp.setInputSource(filtered_scan_ptr);
    first_icp.setInputTarget(filtered_map_ptr);

    first_icp.setMaximumIterations(3000);
    first_icp.setMaxCorrespondenceDistance(1);
    first_icp.setTransformationEpsilon(1e-10);
    first_icp.setEuclideanFitnessEpsilon(1e-10);
    //icp.setRANSACOutlierRejectionThreshold(0.02);
    first_icp.align(*transformed_scan_ptr, init_guess);

    double score = first_icp.getFitnessScore(0.5);
    if (score < min_score) {
        min_score = score;
        min_pose = first_icp.getFinalTransformation();
        ROS_INFO("Update best pose");
    }
  }
```

After obtaining the best initial orientation, we start to conduct the Iterative Closest Points algorithm on the dataset. After each iteration, we update the initial guess matrix through the transformation matrix obtained after the alignment, so the starting point of the next Iterative Closest Point algorithm can converge faster. Besides, to make better observation of the result of each of the iterations, I will print the fitness score of the result in each of the iterations. The implementation is shown in the following figure.

```
icp.setInputSource(filtered_scan_ptr);
icp.setInputTarget(filtered_map_ptr);

icp.setMaximumIterations(3000);
icp.setMaxCorrespondenceDistance(4);
icp.setTransformationEpsilon(1e-10);
icp.setEuclideanFitnessEpsilon(1e-10);
//icp.setRANSACOutlierRejectionThreshold(0.02);
icp.align(*transformed_scan_ptr, init_guess);

std::cout << std::endl
          << "========================================" << std::endl
          <<"    Converged: " << icp.hasConverged() << "    Score: " << icp.getFitnessScore() << std::endl
          << "========================================" << std::endl << std::endl << std::endl;

// if(icp.hasConverged() == 1 && icp.getFitnessScore() > 3){
//   initialied = false;
// }

result = icp.getFinalTransformation();

// std::cout << "Initial Guess: " << std::endl << init_guess << std::endl << std::endl;
// std::cout << "ICP Guess: " << std::endl << icp.getFinalTransformation() << std::endl << std::endl;
// std::cout << "Result: " << std::endl << result << std::endl << std::endl;

pcl::toROSMsg(*transformed_scan_ptr, *test_msg);
test_msg->header.frame_id = "world";
pub_test.publish(test_msg);

// eigen_C_now = result.topLeftCorner<3,3>();

    /* Use result as next initial guess */
init_guess = result;

return result;
```

In addition, since I found the performance of the pose estimation drops whenever the car is turning, I implemented the imu_callback function to solve this problem. The imu_callback function obtains the angular velocity wx, wy, wz of the body frame from the IMU, and calculate σ and B. With the σ and B, we can calculate the C matrix, then we replace the upper left corner of the transformation matrix by matrix C to fix the error mentioned above, which implementation is shown in the following figures.

```
void imu_callback(const sensor_msgs::Imu::ConstPtr& imu_msg){
  ROS_INFO("Got imu message");
  float dt = 0.1;
  float wx = imu_msg->angular_velocity.x;
  float wy = imu_msg->angular_velocity.y;
  float wz = imu_msg->angular_velocity.z;
  float sigma = sqrt(wx*wx + wy*wy + wz*wz)*dt;
  Eigen::Matrix3f B;
  B <<      0, -wz*dt,  wy*dt,
        wz*dt,      0, -wx*dt,
       -wy*dt,  wx*dt,      0;

  tf::Matrix3x3 C_now;
  Eigen::Vector3f ea;
  Eigen::Matrix3f eigen_C_next;

  double roll, pitch, yaw;
  eigen_C_next = eigen_C_now*(Eigen::Matrix3f::Identity(3,3) + sin(sigma)/sigma*B + (1-cos(sigma))/sigma/sigma*B*B);
  // imu_t_old = imu_t_now;
  init_guess.topLeftCorner<3,3>() = eigen_C_next;
}
```

$$\sigma = |\omega \cdot dt|$$

$$B = \begin{bmatrix} 0 & -wz & wy \\ wz & 0 & -wx \\ -wy & wx & 0 \end{bmatrix}$$

$$C(t+dt) = C(t)\left(I + \frac{\sin(\sigma)}{\sigma} \cdot B + \frac{1-\cos(\sigma)}{\sigma^2} \cdot B^2\right)$$

4. Contribution

On the second and third competitions, since I found the performance of the pose estimation drops whenever the car is turning, to improve the accuracy of the results, I especially made use of the angular velocity of the IMU estimate the initial state value of the next step on the nuScenes datasets.

In the first competition, I thought that filtering out some edge data would help reduce the noise in the data. Therefore, I initially filtered out the points on the floor, which was I the shape of many circles, and the points far away from the X-axis and Y-axis.

However, after various tests and tweaks, I found that the dots on the floor actually help with the z-axis positioning. And I mentioned earlier that I worried that points farther from the X-axis and Y-axis would affect the prediction results, such problems can actually be avoided through parameter optimization.

Interestingly, when the ICP parameters are properly adjusted, points farther from the X-axis and Y-axis can actually increase the stability of our prediction and even bring better results.

In the second competition, if we use a method applied in competition one and directly give it a small Max Correspondence Distance to let it execute, it is not feasible. The reason for this is that in the second competition, the vehicle initially reverses up a few steps before starting to move forward. If our Max Correspondence Distance is set to be very small at this time, it will cause the ICP to be unable to pull through because of too much offset from the original point, and eventually lead to poor results.

After constant adjustments and comparisons, I found that when I set Max Correspondence Distance to 4, I not only solved the problem that the vehicle would reverse first, but also got the best results.

In addition, before finding the best initial position, in order to avoid the extra points on the Z axis from affecting the prediction of the initial attitude, I have filtered out the points between −5 and 7 on the Z axis in advance. After finding the best initial position, in order to prevent the ICP result from being disturbed by the vehicle dynamics, I filter out the points between −20 and 20 on the Y axis before running the ICP.

For the third competition, the main implementation method is similar to the first competition. After multiple tests, I only do down sampling on the data, instead of filtering out some points on each axis. The main reason for doing this is because there is a huge difference in the data that needs to be filtered out when the vehicle is going straight and when the vehicle is turning.

Therefore, in order to avoid mistakenly filtering out some data that should not be filtered out, I decided not to do data filtering on each axis separately, but to use down sampling to reduce dynamic points, and then use the left and right sides and distant buildings to conduct positioning.

5. Problem Encountered & Solutions

5-1. Accuracy drops when the car is turning

As I mentioned above, on competition 2&3, I found the performance of the pose estimation drops whenever the car is turning. To solve this problem, I made use of the angular velocity of the IMU estimate the initial state value of the next step, which solved this problem and brought better results.

5-2. Without any iconic static objects (Competition 2)

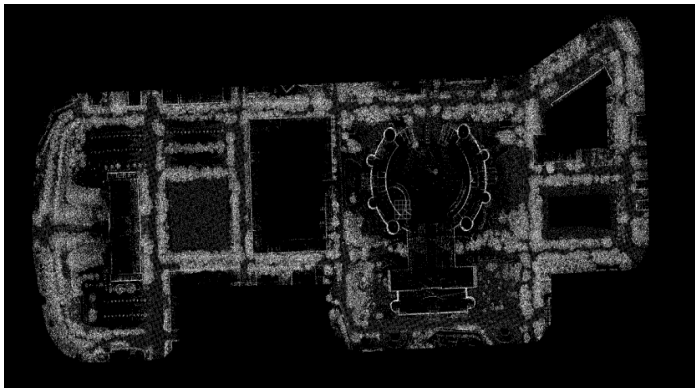
In the second competition, there were nearly no iconic static objects around the car, what we could depend on was the surrounding which was relatively moving for the car. This struggled me a lot, since without the parameters of the ICP set properly, it was easy to get stuck in place until the end.

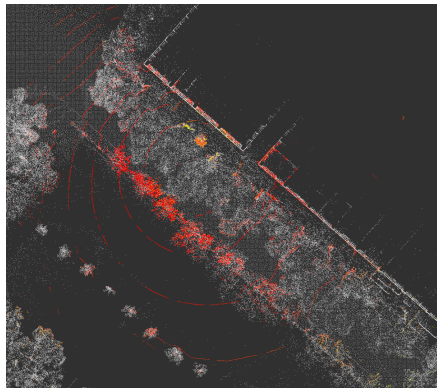
5-3. Couldn't see the map (Competition 2 & 3)

In the beginning of implementing the second and the third competition, I got some trouble of displaying the map's point cloud on the rviz display. However, I figure out that I should modify the topic of the "nuscenes.rviz" from /world to /map. By the modification, the map will appear at the top right of the scene correctly and will follow the view aspect of the result of localization.

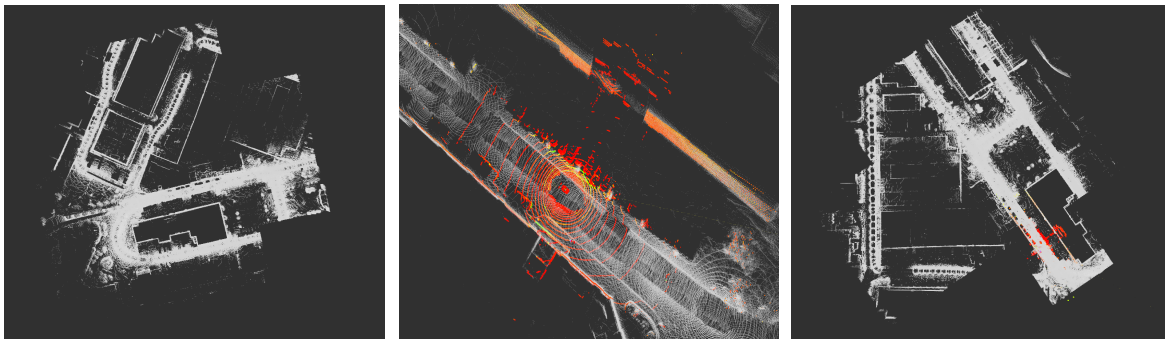
6. Result

6-1. ITRI (Competition I) Public Score: 0.02222





6-2. nuScenes-1 (Competition II) Public Score: 0.10269



6-3. nuScenes-2 (Competition III) Public Score: 0.14378