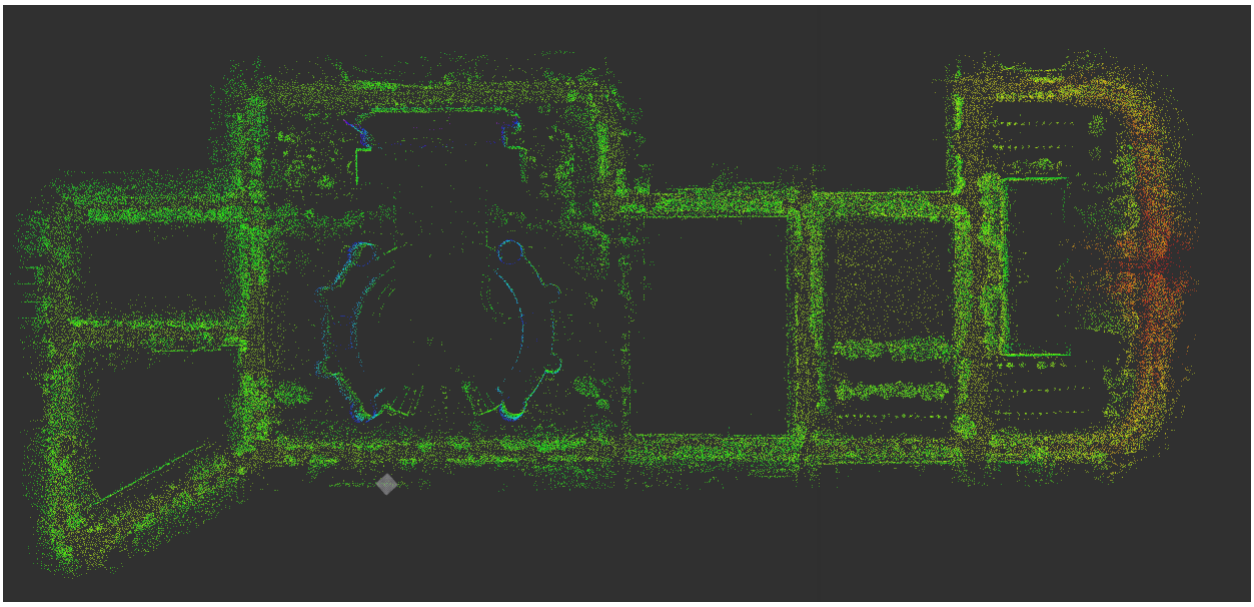
HW4

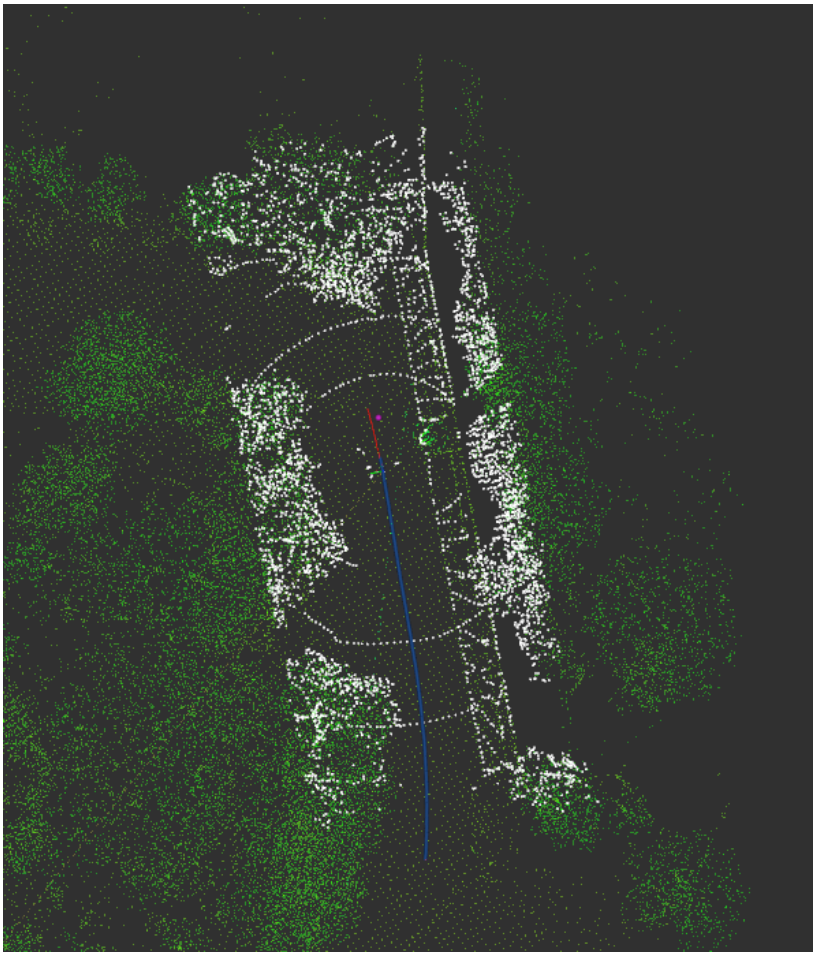
309512074電控碩一 黃柏叡

- Map point cloud(change to world frame for visualization)

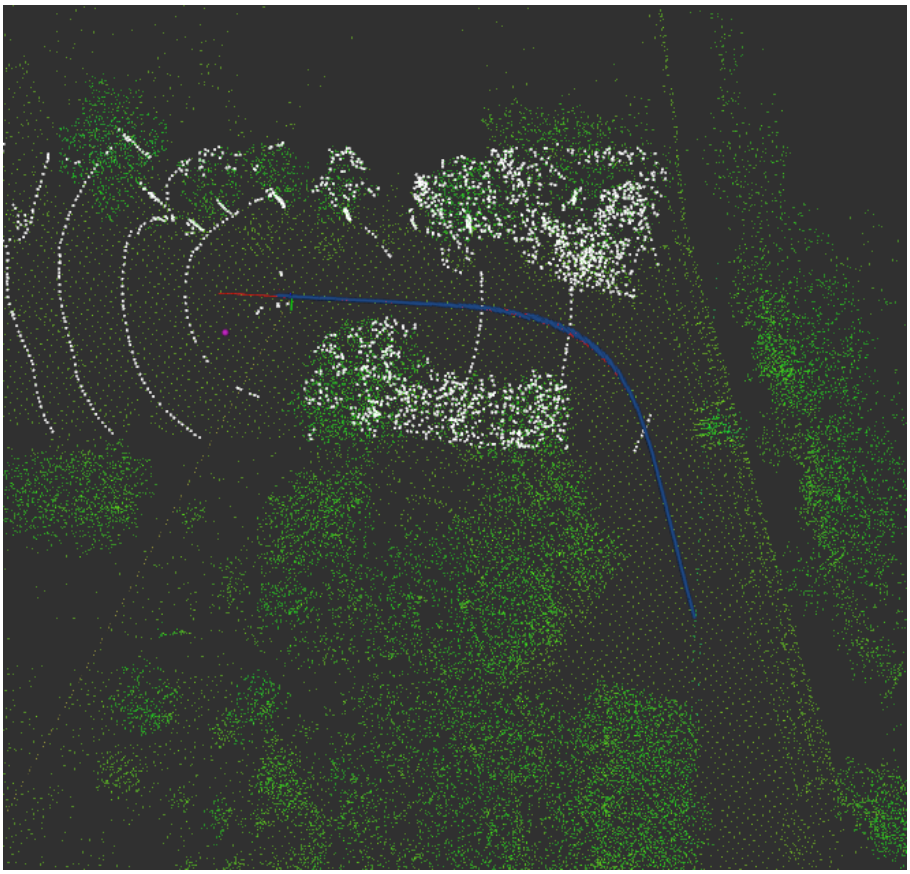


- ICP Localization
  - Red Arrows: ground truth odometry
  - Blue Arrows: my localization result
  - Pink Dots: GPS measurements
  - White Points: ICP Result
  - Color Points: point cloud map

Result1:



Result2:



```
#include <iostream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <ros/package.h>
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <sensor_msgs/PointCloud2.h>
#include <tf2/LinearMath/Matrix3x3.h>
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/conversions.h>
#include <pcl/registration/icp.h>
#include <tf/transform_broadcaster.h>
#include <pcl_ros/transforms.h>
#include <tf/transform_listener.h>
#include <nav_msgs/Odometry.h>
#include "math.h"

// using namespace ros;
using namespace std;

class icp_localization
{
private:
  ros::Subscriber sub_lidar_pc;
  ros::Publisher  pub_icp_pc, pub_odom, pub_map;
  ros::NodeHandle nh;

  sensor_msgs::PointCloud2 map_pc, icp_pc;
  pcl::PointCloud<pcl::PointXYZI>::Ptr load_map;
  pcl::PointCloud<pcl::PointXYZI>::Ptr map_voxel;
  pcl::PointCloud<pcl::PointXYZI>::Ptr pc_input;
  pcl::PointCloud<pcl::PointXYZI>::Ptr pc_input_voxel;
  Eigen::Matrix4f initial_guess;
  tf::TransformListener listener;
```

```
public:
  icp_localization();
  void cb_lidar_pc(const sensor_msgs::PointCloud2 &msg);
  void cb_gps(const geometry_msgs::PoseStamped &msg);
  Eigen::Matrix4f get_initial_guess();
  Eigen::Matrix4f get_transfrom(std::string link);
};

icp_localization::icp_localization(){
  load_map.reset(new pcl::PointCloud<pcl::PointXYZI>());
  map_voxel.reset(new pcl::PointCloud<pcl::PointXYZI>());
  pc_input.reset(new pcl::PointCloud<pcl::PointXYZI>());
  pc_input_voxel.reset(new pcl::PointCloud<pcl::PointXYZI>());

  if (pcl::io::loadPCDFile<pcl::PointXYZI> ("/home/ray/self-driving-car-2021/catkin_ws/src/309512074_hw4/map/map.pcd", *load_map) == -1)
  {
    PCL_ERROR ("Couldn't read file map_downsampled.pcd \n");
    exit(0);
  }

  //=======voxel grid filter=====================
  cout << "PointCloud before filtering: " << load_map->points.size() << endl;
  pcl::VoxelGrid<pcl::PointXYZI> sor_map;
  sor_map.setInputCloud (load_map);
  sor_map.setLeafSize (0.5f, 0.5f, 0.5f);
  sor_map.filter (*map_voxel);
  cout << "PointCloud after filtering: " << map_voxel->points.size() << endl;

  pcl::toROSMsg(*map_voxel, map_pc);

  sub_lidar_pc = nh.subscribe("lidar_points", 10, &icp_localization::cb_lidar_pc, this);
  pub_icp_pc = nh.advertise<sensor_msgs::PointCloud2>("ipc_pc", 10);
  pub_odom = nh.advertise<nav_msgs::Odometry>("odom_result", 10);
  pub_map = nh.advertise<sensor_msgs::PointCloud2>("map", 10);

  //wait for gps
  cout << "waiting for gps message" << endl;
  initial_guess = get_initial_guess();
  cout << "initial guess:" << endl;
  cout << initial_guess << endl;
}

Eigen::Matrix4f icp_localization::get_initial_guess(){

  tf2::Quaternion q;
  double yaw=-2.2370340344819 ;//rad
  q.setRPY(0,0,yaw);
  tf2::Matrix3x3 rotation;
  rotation.setRotation(q);
  Eigen::Matrix4f trans = Eigen::Matrix4f::Zero();
  geometry_msgs::PointStampedConstPtr gps;
  gps = ros::topic::waitForMessage<geometry_msgs::PointStamped>("/gps", nh);

  trans << rotation[0][0], rotation[0][1], rotation[0][2],  (*gps).point.x,
           rotation[1][0], rotation[1][1], rotation[1][2],  (*gps).point.y,
           rotation[2][0], rotation[2][1], rotation[2][2],  (*gps).point.z,
                0,              0,              0,                      1;

  return trans;
}

Eigen::Matrix4f icp_localization::get_transfrom(std::string link){

  tf::StampedTransform transform;
  Eigen::Matrix4f trans;

  try{
      ros::Duration five_seconds(5.0);
      listener.waitForTransform("base_link", link, ros::Time(0), five_seconds);
      listener.lookupTransform("base_link", link, ros::Time(0), transform);
  }
  catch (tf::TransformException ex){
      ROS_ERROR("%s",ex.what());
      return trans;
  }
  Eigen::Quaternionf q(transform.getRotation().getW(), \
    transform.getRotation().getX(), transform.getRotation().getY(), transform.getRotation().getZ());
  Eigen::Matrix3f mat = q.toRotationMatrix();
```

```
  trans << mat(0,0), mat(0,1), mat(0,2), transform.getOrigin().getX(),
           mat(1,0), mat(1,1), mat(1,2), transform.getOrigin().getY(),
           mat(2,0), mat(2,1), mat(2,2), transform.getOrigin().getZ(),
           0, 0, 0, 1;
  return trans;
}

void icp_localization::cb_lidar_pc(const sensor_msgs::PointCloud2 &msg){

  pcl::fromROSMsg(msg, *pc_input);
  Eigen::Matrix4f trans = get_transfrom("velodyne");
  transformPointCloud(*pc_input, *pc_input, trans);
  ROS_INFO("transformed to base_link");


  cout<<"original: "<< pc_input->points.size()<<endl;
  //=========filter============
  pcl::PassThrough<pcl::PointXYZI> pass;
  pass.setInputCloud(pc_input);
  pass.setFilterFieldName ("y");
  pass.setFilterLimits (-10, 10);
  pass.filter (*pc_input);
  cout<<"filter_y: "<<pc_input->points.size()<<endl;

  pcl::PassThrough<pcl::PointXYZI> pass1;
  pass1.setInputCloud(pc_input);
  pass1.setFilterFieldName ("x");
  pass1.setFilterLimits (-25, 25);
  pass1.filter (*pc_input);
  cout<<"filter_x: "<<pc_input->points.size()<<endl;

  // //=======voxel grid filter=====================
  pcl::VoxelGrid<pcl::PointXYZI> sor;
  sor.setInputCloud (pc_input);
  sor.setLeafSize (0.5f, 0.5f, 0.5f);
  sor.filter (*pc_input_voxel);
  cout<<"voxel grid filter: "<<pc_input_voxel->points.size()<<endl;


  //=======icp==========================
  pcl::IterativeClosestPoint<pcl::PointXYZI, pcl::PointXYZI> icp;
  icp.setInputSource(pc_input_voxel);
  icp.setInputTarget(map_voxel);
  icp.setMaximumIterations (50);
  // icp.setTransformationEpsilon (1e-12);
  icp.setMaxCorrespondenceDistance (2);
  // icp.setEuclideanFitnessEpsilon (0.01);
  // icp.setRANSACOutlierRejectionThreshold (0.06);
  pcl::PointCloud<pcl::PointXYZI> Final;
  icp.align(Final, initial_guess);

  cout << "has converged:" << icp.hasConverged() << " score: " << icp.getFitnessScore() << endl;
  cout << icp.getFinalTransformation() << endl;
  initial_guess = icp.getFinalTransformation();

  tf::Matrix3x3 tf3d;
  tf3d.setValue((initial_guess(0,0)), (initial_guess(0,1)), (initial_guess(0,2)),
        (initial_guess(1,0)), (initial_guess(1,1)), (initial_guess(1,2)),
        (initial_guess(2,0)), (initial_guess(2,1)), (initial_guess(2,2)));
  tf::Quaternion tfqt;
  tf3d.getRotation(tfqt);
  tf::Transform transform;
  transform.setOrigin(tf::Vector3(initial_guess(0,3),initial_guess(1,3),initial_guess(2,3)));
  transform.setRotation(tfqt);

  static tf::TransformBroadcaster br;
  br.sendTransform(tf::StampedTransform(transform, ros::Time::now(),"world","base_link"));

  // Publish my lidar pointcloud after doing ICP.
  pcl::toROSMsg(Final, icp_pc);
  icp_pc.header=msg.header;
  icp_pc.header.frame_id = "world";
  pub_icp_pc.publish(icp_pc);

  //=======show map====================
  map_pc.header.frame_id = "world";
  map_pc.header.stamp = ros::Time::now();
  pub_map.publish(map_pc);
```

```
  //====publish localization result====
  nav_msgs::Odometry odom;
  odom.header.frame_id = "world";
  odom.child_frame_id = "base_link";
  odom.pose.pose.position.x = initial_guess(0,3);
  odom.pose.pose.position.y = initial_guess(1,3);
  odom.pose.pose.position.z = initial_guess(2,3);
  tf2::Matrix3x3 tfm;
  tfm.setValue(initial_guess(0,0) ,initial_guess(0,1) ,initial_guess(0,2) ,
              initial_guess(1,0) ,initial_guess(1,1) ,initial_guess(1,2) ,
              initial_guess(2,0) ,initial_guess(2,1) ,initial_guess(2,2));
  tf2::Quaternion tfq2;
  tfm.getRotation(tfq2);
  odom.pose.pose.orientation.x = tfq2[0];
  odom.pose.pose.orientation.y = tfq2[1];
  odom.pose.pose.orientation.z = tfq2[2];
  odom.pose.pose.orientation.w = tfq2[3];
  pub_odom.publish(odom);
}
int main (int argc, char** argv)
{
  ros::init(argc, argv, "icp_localization");
  icp_localization icp;
  ros::spin();
}
```