SDC_HW4

terminal command

shell A

`$ roscore`

shell B

`$ cd /self-driving-cars-course/SDC_HW4/catkin_ws`

`$ catkin_make`

`$ source devel/setup.bash`

`$ rosrun pkg icp_locolization`

shell C

`$ cd /self-driving-cars-course/SDC_HW4/bag`

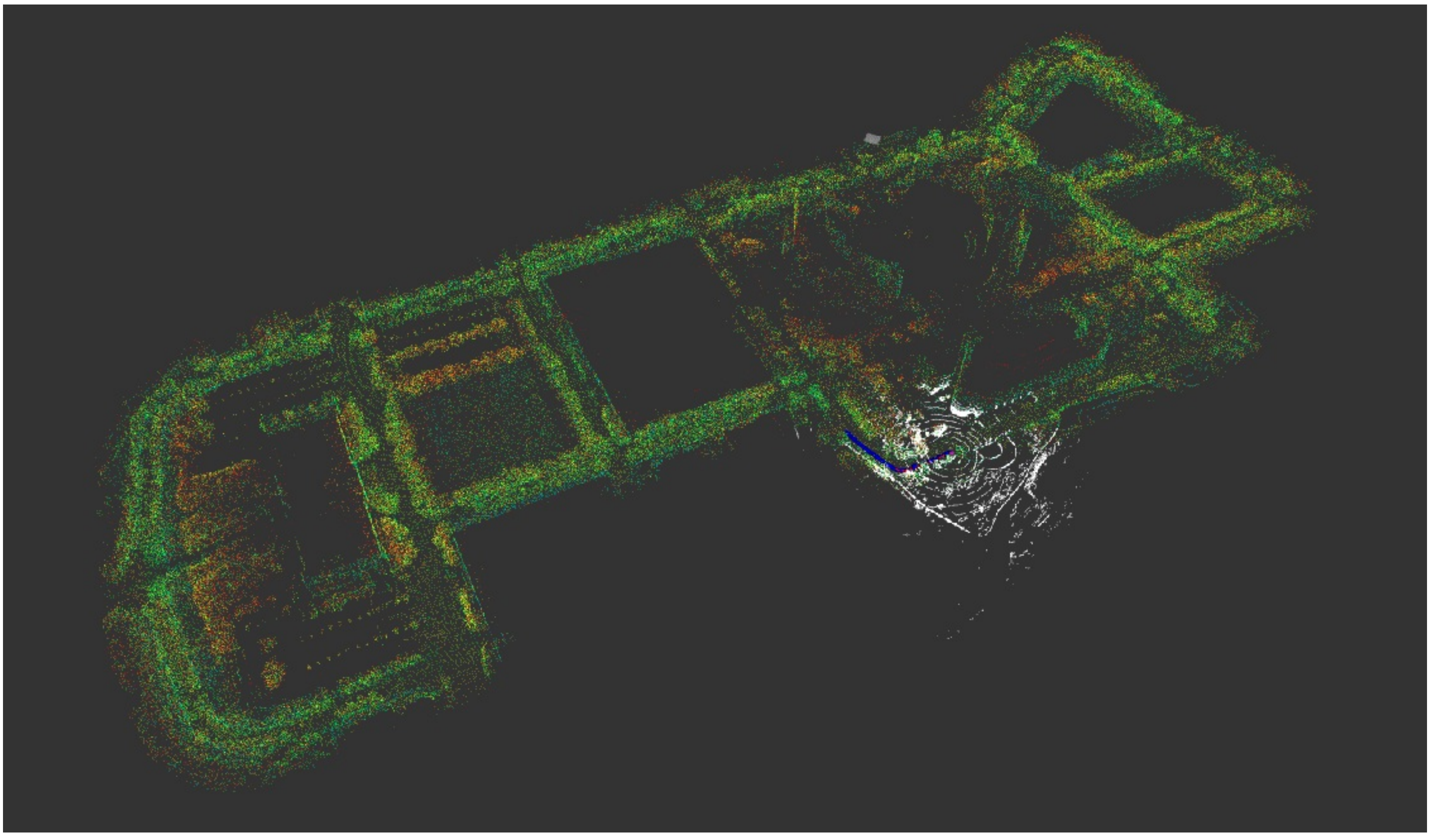
`$ rosbag play sdc_hw4.bag -r 0.1`

shell D

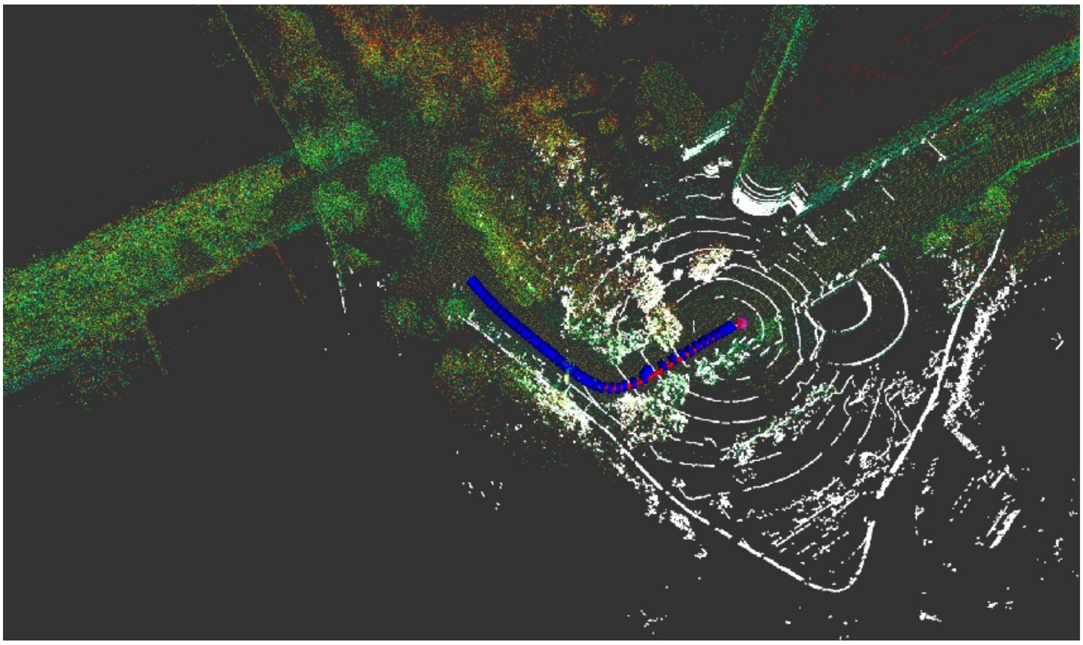
`$ cd /self-driving-cars-course/SDC_HW4`

`$ rosrun rviz rviz -d hw4.rviz`

Assignment - Publish and visualize the Lidar map



Assignment - ICP localization Resulting



parameters for ICP Algorithm

I have endeavour to choose the better parameters to prevent the odometry from deviating at the turning point.

The value for `setMaximumIterations()` should be larger if the initial alignment is poor and small if the initial alignment is already quite good.

The value for `setRANSACOutlierRejectionThreshold()` should typically be just below the resolution of your scanner.

The value for `setMaxCorrespondenceDistance()` should be set to approx. 100 times the value of `setRANSACOutlierRejectionThreshold()` (This depends on how good your initial guess is and can also be fine-tuned).

The value for `setTransformationEpsilon()` is your convergence criterion. If the sum of differences between current and last transformation is smaller than this threshold, the registration succeeded and will terminated.

The value `setEuclideanFitnessEpsilon()` is an divergence threshold. Here you can define the delta between two consecutive steps in the ICP loop for which the algorithm stops. That means that you can set an euclidean distance error barrier if the ICP diverges at some point, so that it does not become even worse over the remaining number of iterations.

```
 icp.setMaximumIterations (1000);
icp.setTransformationEpsilon (1e-12);
icp.setMaxCorrespondenceDistance (4);
icp.setEuclideanFitnessEpsilon (0.01);
icp.setRANSACOutlierRejectionThreshold (0.06);
```

code

/self-driving-cars-course/SDC_HW4/catkin_ws/src/pkg/src/icp_locolization.cpp

```
 #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_locolization
{
private:
  Subscriber sub_map,sub_lidar_scan;
  Publisher pub_pc_after_icp,pub_result_odom,pub_map;
  ros::NodeHandle nh;

  sensor_msgs::PointCloud2 map_cloud, fin_cloud;
  pcl::PointCloud<pcl::PointXYZI>::Ptr load_map;
  Eigen::Matrix4f initial_guess;
```

```
  tf::TransformListener listener;
  tf::TransformBroadcaster broadcaster;

public:
  icp_locolization();
  void cb_lidar_scan(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_name);
};

icp_locolization::icp_locolization(){
  // Load Lidar map.
  load_map = (new pcl::PointCloud<pcl::PointXYZI>)->makeShared();
  if (pcl::io::loadPCDFile<pcl::PointXYZI> ("/home/yellow/self-driving-cars-course/SDC_HW4/bag/map_downsample.pcd", *load_map) == -1)
  {
    PCL_ERROR ("Couldn't read file map.pcd \n");
    exit(0);
  }
  //=======voxel grid filter=====================
  pcl::VoxelGrid<pcl::PCLPointCloud2> sor_map;
  pcl::PCLPointCloud2::Ptr map_cloud2 (new pcl::PCLPointCloud2 ());
  pcl::toPCLPointCloud2(*load_map, *map_cloud2);
  sor_map.setInputCloud (map_cloud2);
  sor_map.setLeafSize (0.5f, 0.5f, 0.5f);
  sor_map.filter (*map_cloud2);
  pcl::fromPCLPointCloud2(*map_cloud2, *load_map);
  pcl::toROSMsg(*load_map, map_cloud);


  sub_lidar_scan = nh.subscribe("lidar_points", 10, &icp_locolization::cb_lidar_scan, this);
  pub_pc_after_icp = nh.advertise<sensor_msgs::PointCloud2>("pc_after_icp", 10);
  pub_result_odom = nh.advertise<nav_msgs::Odometry>("result_odom", 10);
  pub_map = nh.advertise<sensor_msgs::PointCloud2>("load_map", 10);


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

  printf("init done \n");
}


Eigen::Matrix4f icp_locolization::get_initial_guess(){

  Eigen::Matrix4f trans;
  geometry_msgs::PointStampedConstPtr gps_point;
  gps_point = ros::topic::waitForMessage<geometry_msgs::PointStamped>("/gps", nh);

  double yaw=-2.2370340344819 ;//rad
  trans << cos(yaw), -sin(yaw), 0,  (*gps_point).point.x,
           sin(yaw), cos(yaw),  0,  (*gps_point).point.y,
                 0,        0,         1,  (*gps_point).point.z,
                 0,        0,         0,  1;

  return trans;
}


Eigen::Matrix4f icp_locolization::get_transfrom(std::string link_name){

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

    try{
        ros::Duration five_seconds(5.0);
        listener.waitForTransform("base_link", link_name, ros::Time(0), five_seconds);
        listener.lookupTransform("base_link", link_name, 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_locolization::cb_lidar_scan(const sensor_msgs::PointCloud2 &msg)
{
  // Downsample the scan of the pointcloud using voxel grid filter to reduce the computation time.
  pcl::PointCloud<pcl::PointXYZI>::Ptr bag_cloud(new pcl::PointCloud<pcl::PointXYZI>);
  pcl::fromROSMsg(msg, *bag_cloud);

  Eigen::Matrix4f trans = get_transfrom("velodyne");
    transformPointCloud (*bag_cloud, *bag_cloud, trans);
  ROS_INFO("transformed to base_link");


  cout<<"original: "<<bag_cloud->points.size()<<endl;
  //=======voxel grid filter=====================
  pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
  pcl::PCLPointCloud2::Ptr bag_cloud2 (new pcl::PCLPointCloud2 ());
  pcl::toPCLPointCloud2(*bag_cloud, *bag_cloud2);
  sor.setInputCloud (bag_cloud2);
  sor.setLeafSize (0.1f, 0.1f, 0.1f);
  sor.filter (*bag_cloud2);
  pcl::fromPCLPointCloud2(*bag_cloud2, *bag_cloud);
  cout<<"voxel grid filter: "<<bag_cloud->points.size()<<endl;



  // Do ICP matching of the map and lidar scan pointcloud.
  //=======icp=====================
  pcl::IterativeClosestPoint<pcl::PointXYZI, pcl::PointXYZI> icp;
  icp.setInputSource(bag_cloud);
  icp.setInputTarget(load_map);
  icp.setMaximumIterations (1000);
  icp.setTransformationEpsilon (1e-12);
  icp.setMaxCorrespondenceDistance (4);
  icp.setEuclideanFitnessEpsilon (0.01);
  icp.setRANSACOutlierRejectionThreshold (0.06);
  pcl::PointCloud<pcl::PointXYZI> Final;
  icp.align(Final, initial_guess);

  std::cout << "has converged:" << icp.hasConverged() << " score: " <<
  icp.getFitnessScore() << std::endl;
  std::cout << icp.getFinalTransformation() << std::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"));
  // StampedTransform (const tf::Transform &input, const ros::Time &timestamp, const std::string &frame_id, const std::string &child_frame_id)



  // Publish your lidar scan pointcloud after doing ICP.
  sensor_msgs::PointCloud2 fin_cloud;
  pcl::toROSMsg(Final, fin_cloud);
  fin_cloud.header=msg.header;
  fin_cloud.header.frame_id = "/world";
  pub_pc_after_icp.publish(fin_cloud);


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


  // Publish your localization result as nav_msgs/Odometry.msg message type.
  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.y = initial_guess(1,3);
  odom.pose.pose.position.z = initial_guess(2,3);
  tf2::Matrix3x3 m;
  m.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;
  m.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_result_odom.publish(odom);
}



int main (int argc, char** argv)
{
  ros::init(argc, argv, "icp_locolization");
  icp_locolization iiiiiiiiiiiiiiiiiiiiiicp;
  ros::spin();
}
```