Electronics Design Lab Report #3

Introduction

In the following two weeks I studied about mapping and localization. Although I was well prepared theoretically, it was a big challenge for me to complete the coding part because I encountered a lot of strange bags and other mistakes related to the path in the launch and mapping files, PointCloud class and sensor_msgs types, publishing new messages and transferring the information from the simulation to other parts. This assignment made me learn a lot about the pcl library and its performance (I used this as reference https://dbloisi.github.io/corsi/lezioni/pcl.pdf), how to handle it and utilize in the algorithm. I learned about how to set up different packages, where to read the information about them as it is also very hard to understand many manuals in the ros website; and consequently, how to debug numerous errors during the launching. Another thing is to use rviz as I did not understand initially that we should subscribe to the package. Additionally, by analyzing the final program, I learned about the ICP and NDT_OMP algorithm behaviors, how their performance differs from each other, which one better and which one harder to implement. Finally, by filling this code, I was forced to analyze its logic very deeply which gave me the experience how to create mapping and localization package from scratch by myself which will be very beneficial in our f1 tenth project where the camera is used in the localization.

Explanation

The following code may seem confusing as I was very confused when wrote it. And I still don't understand the performance of some parts as I used a lot of references.

TODO1

```
        pcl::PointXYZI pointBuf;
        //TODO:#1
        // feel the code here
//msg->info.width
        float x = (msg -> ranges[i])*cos(msg->angle_increment*i + msg->angle_min);
        float y = (msg -> ranges[i])*sin(msg->angle_increment*i + msg->angle_min);
        float z = 0;
        pointBuf.x = x;
        pointBuf.y = y;
        pointBuf.z = z;
        cloud_in_ptr->points.push_back(pointBuf);
    }
```

In this part I was very confused, but you helped me a lot with your suggestions through email. I read the documentation about the messages and understood that messages work in the polar coordinates. So we take the message, gets its angle in the unit time and use sin/cos function and radius to get x and y coordinates. Then we create the point out of this and upload to the cloud.

TODO2

```
// Implement below two algorithm and compare the result.
// You can switch the registration algorithm between ICP and NDT_OMP
// TODO:#2
// 1. ICP(No multi-thread)
//ICPMatching(cloud_in_ptr, m_StaticMap_ptr); //Too slow
// 2. NDT_OMP(multi-thread)
NDTMatching(cloud_in_ptr, m_StaticMap_ptr);
```

This one is simple, we just decomment the algorithm that we want to use and then another one to compare then in the future.

TODO3

```
    //TODO:#3
    // feel the code here.
    //Convert occupied grid to the x-y coordinates to put the target(pcl::PointCloud<pcl::PointXY
//float x = width * msg->info.resolution + msg->info.resolution/2;
//float y = height*msg->info.resolution + width*msg->info.resolution / 2;
float x = width * msg->info.resolution + msg->info.origin.position.x + msg->info.resolution/2;
//float y = height*msg->info.resolution + width*msg->info.resolution / 2;
float y = height * msg->info.resolution + msg->info.resolution/2 + msg->info.origin.position.y;
float z = 0;
//m_StaticMap_ptr->points[width*height].x = x;
//m_StaticMap_ptr->points[width*height].y = y;
pcl::PointXYZI pointBuf;
pointBuf.x = x;
pointBuf.y = y;
pointBuf.z = z;
m_StaticMap_ptr->points.push_back(pointBuf);
}
```

Here we need make an array of data that stores the message “picture” which has properties as height and width. As it is 2d and array is 1d we store it in the index of [width*height], as previously we create a point, give info to it and then upload to the cloud. To get the x and y values from message, we take its resolution and position information. Resolution plays a role of angle here.

TODO4

```
    // TODO:#4
    // feel the code here
    icp.setMaximumIterations (80);
    icp.setInputSource(InputData);
    icp.setInputTarget(TargetData);
    // Run registration algorithm, and put the transformation matrix of previous step.
    pcl::PointCloud<pcl::PointXYZI>::Ptr result(new pcl::PointCloud<pcl::PointXYZI>());
    icp.align(*result, init_guess);
```

The input source should be the data that is taken from the messages, the result is given to the Target Data as this is cloud created for this. The algorithm terminates when it reaches the maximum number of iterations which I set to 80 here.

TODO5

```
    //TODO:#5
    //feel the code here --> convert pcl pointcloud to the ros msg.
    pcl::transformPointCloud(*InputData, *final_cloud, init_guess);
/*   float x;
    float y;
    sensor_msgs::PointCloud2::Ptr msg(new sensor_msgs::PointCloud2());

    for(int i = 0; i < final_cloud->points.size(); i++)
    {
    x = final_cloud->points[i].x;
    y = final_cloud->points[i].y;
    msg -> data[i] = sqrt(x*x+y*y);
//(msg -> ranges[i])*sin(msg->angle_increment*i + msg->angle_min);
//(msg -> ranges[i])*cos(msg->angle_increment*i + msg->angle_min);
    } */
```

Initially I wanted to implement the process from scratch, but then found that there is a function in pcl library that makes this all automatically.

TODO6

```
    //publish Odometry
    //TODO:#6
    //feel the code here --> Publish the result using nav_msgs/Odometry topic.
    tf::StampedTransform tf_transform;
    geometry_msgs::TransformStamped odom_trans; // msg
    ros::Time time = ros::Time::now(); // get current time
    tf::TransformBroadcaster odom_broadcaster;

    Eigen::Quaterniond eigen_quat(init_guess.block<3, 3>(0, 0).cast<double>());
    Eigen::Vector3d eigen_trans(init_guess.block<3, 1>(0, 3).cast<double>());

    tf::Quaternion tf_quat;
    tf::Vector3 tf_trans;
    tf::quaternionEigenToTF(eigen_quat, tf_quat);

    tf::vectorEigenToTF(eigen_trans, tf_trans);

    geometry_msgs::Quaternion odom_quat;
    tf::quaternionTFToMsg(tf_quat, odom_quat);
```

```
odom_trans.header.stamp = time;
    odom_trans.child_frame_id = "base_link";
odom_trans.header.frame_id = "map";
odom_trans.transform.translation.x = tf_trans.getX();
odom_trans.transform.translation.y = tf_trans.getY();
odom_trans.transform.translation.z = tf_trans.getZ();
odom_trans.transform.rotation = odom_quat;
odom_broadcaster.sendTransform(odom_trans);

m_odomScan.header.stamp = time;

m_odomScan.header.frame_id = "map";
// position
m_odomScan.pose.pose.position.x = tf_trans.getX();
m_odomScan.pose.pose.position.y = tf_trans.getY();
m_odomScan.pose.pose.position.z = tf_trans.getZ();
m_odomScan.pose.pose.orientation = odom_quat;
m_odomScan.child_frame_id = "base_link";
// velocity
double v = 0.02
m_odomScan.twist.twist.linear.x = v;
m_odomScan.twist.twist.linear.y = v;
m_odomScan.twist.twist.angular.z = v;
pubOdometry.publish(m_odomScan); //topic name: "/odom"
```

This code indeed was made with the help of ros tutorial about how to publish odom messages and with the code of Hyundai. Mostly it is even copy pasted. However I understood the ongoing process about registering time, taking position at that moment, finding quaternions and transformation matrix and use them to publish the position and orientation of the object. Quaternion is basically the structure that saves in itself both the position and rotational component of the object and there are operations with which you can extract all of these again (computer graphics class thought me it).

TODO7

```
//TODO:#7
//Feel the code here
pcl::PointCloud<pcl::PointXYZI>::Ptr result(new pcl::PointCloud<pcl::PointXYZI>());
ndt->align(*result, init_guess);
if (ndt->hasConverged())
{
    std::cout << "-----" << std::endl
            << "Score: " << ndt->getFitnessScore() << std::endl;
    std::cout << "Transformation:" << std::endl;
    std::cout << ndt->getFinalTransformation() << std::endl;
}
else
{
    return;
}
init_guess.block<3, 3>(0, 0) = ndt->getFinalTransformation().block<3, 3>(0, 0);
init_guess.block<3, 1>(0, 3) = ndt->getFinalTransformation().block<3, 1>(0, 3);

pcl::PointCloud<pcl::PointXYZI>::Ptr final_cloud(new pcl::PointCloud<pcl::PointXYZI>());
pcl::transformPointCloud(*InputData, *final_cloud, init_guess);
sensor_msgs::PointCloud2 FinalCloudToPointCloud2Msg;
pcl::toROSMsg(*result, FinalCloudToPointCloud2Msg); //!!!
```

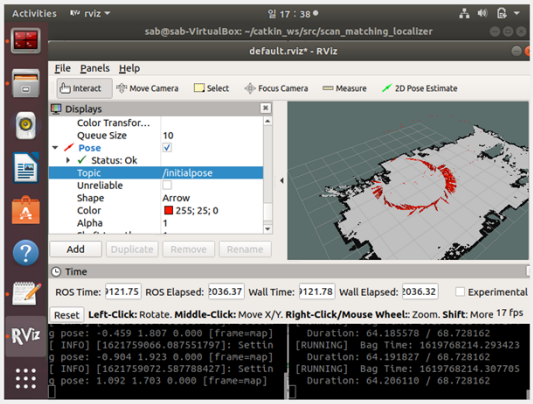
```
FinalCloudToPointCloud2Msg.header.frame_id = "map";
FinalCloudToPointCloud2Msg.header.stamp = ros::Time::now();
pubTransformedCloud.publish(FinalCloudToPointCloud2Msg);

tf::StampedTransform tf_transform;
geometry_msgs::TransformStamped odom_trans;
ros::Time time = ros::Time::now();
tf::TransformBroadcaster odom_broadcaster;
//tf::TransformBroadcaster odom_broadcaster();
Eigen::Quaterniond eigen_quat(init_guess.block<3, 3>(0, 0).cast<double>());
Eigen::Vector3d eigen_trans(init_guess.block<3, 1>(0, 3).cast<double>());
tf::Quaternion tf_quat;  // tf_q?
tf::Vector3 tf_trans;

tf::quaternionEigenToTF(eigen_quat, tf_quat);

tf::vectorEigenToTF(eigen_trans, tf_trans);

geometry_msgs::Quaternion odom_quat;
tf::quaternionTFToMsg(tf_quat, odom_quat);

odom_trans.header.stamp = time;
odom_trans.header.frame_id = "map";
odom_trans.child_frame_id = "base_link";
```

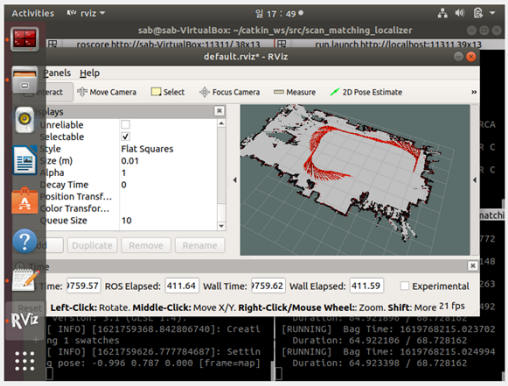
```
m_odomScan.header.frame_id = "map";
// position
m_odomScan.pose.pose.position.x = tf_trans.getX();
m_odomScan.pose.pose.position.y = tf_trans.getY();
m_odomScan.pose.pose.position.z = tf_trans.getZ();
m_odomScan.pose.pose.orientation = odom_quat;
m_odomScan.child_frame_id = "base_link";
// velocity
double v = 0.02
m_odomScan.twist.twist.linear.x = v;
m_odomScan.twist.twist.linear.y = v;
m_odomScan.twist.twist.angular.z = v;
pubOdometry.publish(m_odomScan); //topic name: "/odom"

prev_guess = init_guess;
```

This one is repetition of the all previous todo parts, so I copied and pasted them in place.

Discussion

The resultant path picture is seen in the pictures below:





ICP function NDT_OMP function

First, I would like to mention that I subscribed to all nodes needed as was shown in the video. However, I still could not achieve good visualization of points. They exist, but they are too small amount and vary light to see from the picture, I don't know why. But, it still can be recognizable that multi-thread algorithm is much faster and its performance in terms of mapping is better as seen by its path. The left picture path is very small and not into huge range as it recognizes the border where is does not exist. While multithread could see the border in a small distance and uses whole map for its motion. It even went to the small path where there is a hole in the middle. This concludes that the algorithm works better with multithread.

References used during coding

https://pointclouds.org/

https://answers.ros.org/question/201172/get-xy-coordinates-of-an-obstacle-from-a-map/

hyundai_laser_collision_checker.zip (dropbox.com)

https://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom