COLLEGE OF DESIGN AND ENGINEERING

Department: Mechanical Engineering

ME5413 Autonomous Mobile Robotics

The Report of Final Project

Group Number : 12

Team member:

A0263231R

A0263694R

A0263137H

A0263127J

Spring 2023

1 Introduction

In this project, the techniques including perception, localization, and planning of autonomous mobile robots used in previous assignments are applied to a real-world situation. In detail, the main task is performing SLAM and navigation in the specified built environment, which is to reach goals in sequence from a assembly line to packaging area, and finally to the delivery vehicle as well as avoiding collisions with obstacles along the way.

In the first stage of mapping, the task is to build a map representing the environment including the available paths and obstacles, for which the applied algorithm plays an import role in whether the obstacles are be detected properly, thus, a pipeline for building the map is illustrated in detail to gain a better understanding of the algorithm. Besides that, to highlight merit of the chosen technique, a simple baseline 2D Gmapping is experimented, and its performance is compared with our method, which is also quantitatively evaluated by absolute error with the ground truth. In the part of navigation, we explore possible path planning algorithms using the previously created map and use one of them as a planner for navigation, the principle for which is described later.

In the following content, the second part gives a detailed description of the algorithms used and explains the whole process of building maps and the conversion between different map files in order to prepare for navigation. Chapter 3 presents the implementation process of the navigation and an introduction to the algorithm applied. Beyond this, we list some of the difficulties encountered in the project and propose solutions to improve the performance of the algorithm based on possible causes. Finally, we conclude with a summary of the above and a discussion of the subsequent work.

2 Mapping

In this section of work, several kinds of techniques including 2D/3D LIDAR, vision or multi-sensor fusion can be applied to build the map of the environment. Here we start with a simple SLAM method with build-in Gmapping to see its performance in task and analysis its results to explore improvement with better approach like 3D mapping, which is illustrated in the follow content of this part.

2.1 2D Lidar SLAM by Gmapping

For this algorithm, it uses data from a laser range finder, which scans the environment and produces a 2D map of obstacles and other features, and then uses this data to build a probabilistic map of the environment, which represents the likelihood of the robot being in a particular location and the probability of obstacles being present in different areas of the map, and it benefits from the light-weighted computation and built-in availability in ROS.

To perform localization and map building, only a rosnode needs to be run after launching the world, the 2D laser scan nearby the robot that is visualized in rivz shows in left image of Figure 1, then the map can be obtained by driving the mobile robot to explore all available regions. After the work is completed, the resulting map is shown as the right image of Figure 1.

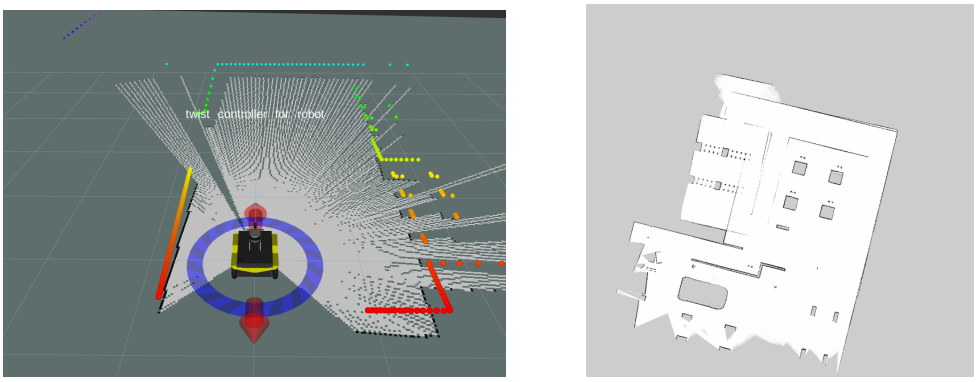

Figure 1: 2D laser scan in rviz (left) and Built map by 2D Lidar SLAM (right)

From the built map it can be seen that even with several loops of exploration, it has not converged to the desired result, which shows in the overlapped area between two rooms. More importantly, the 2D Lidar can not detect the height information of obstacles, for example, in this environment, it can only see the first level of the stair, and the higher steps are recognized as free space, and even the whole door frame cannot be detected, so the robot is likely to collide when passing these obstacles.

In order to intuitively evaluate the performance of the algorithm, we use evo to quantify its error from the ground truth. Figure 2 shows the deviation between the two trajectories and the absolute error is calculated to confirm our conclusion. The results show the localization is obviously distinguished from the ground truth and error is hardly acceptable, which lead to the unsatisfactory performance of this method, so we later adopted another 3D Lidar SLAM approach to improve the mapping performance.

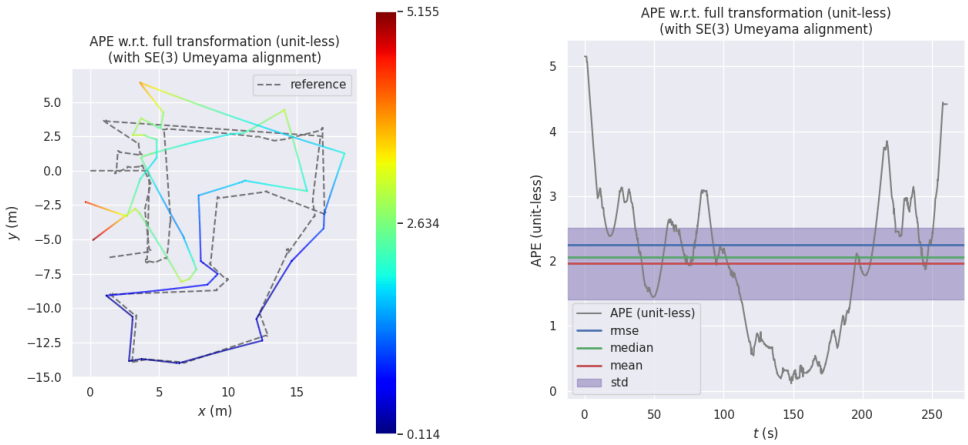

Figure 2: Evaluation on Gmapping by evo

2.2 3D Lidar SLAM by A-LOAM

In order to improve performance, we chose an efficient 3D Lidar algorithm, A-LOAM builds on the well-known LOAM (Lidar Odometry and Mapping) algorithm, which uses a laser rangefinder (lidar) sensor to create a map of the environment and estimate the motion of the sensor. However, it improves on LOAM by introducing several adaptive mechanisms that enable it to perform better in a wide range of environments and is a highly accurate and efficient algorithm that has been shown to outperform other state-of-the-art lidar-based localization and mapping algorithms in various bench-

mark datasets.

To fit the SLAM algorithm with the current environment, the topic is first matched in the configuration file. Specifically, the point cloud topic published in the environment is set to be consistent with the topic subscribed in the algorithm. By proper configuration, the point clouds detected by 3D Lidar is visualized in rviz, which contains the height information of obstacles that benefits map building.

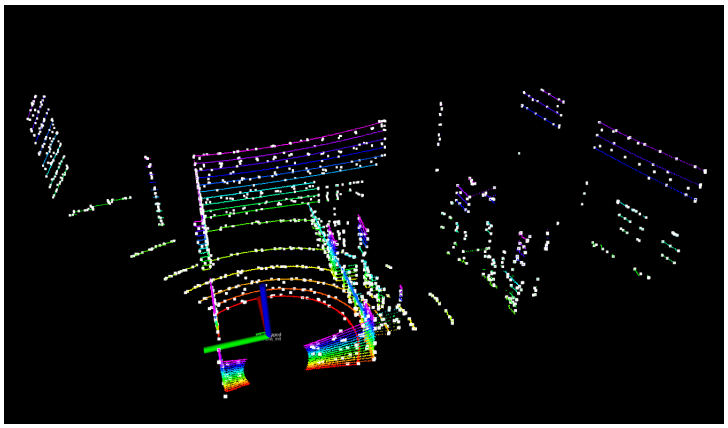

Figure 3: 3D pointclouds in rviz

After applying the same method to control the robot by keyboard to explore the available area in the environment, the obtained point cloud is saved as a pcl file as shown below Figure 4 through the **pcl_ros** function package. However, the map format required for subsequent navigation is a pgm file, for which it needs to be converted. Here, a custom package named **pcl2pgm** is used to complete this operation. After projecting the 3D point cloud to the ground plane, the global map can be obtained as shown in the Figure 4.

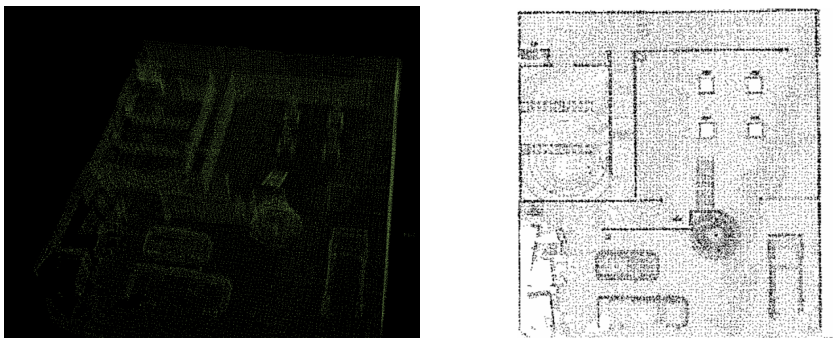

Figure 4: Pointclouds in pcl file (left) and Built map by 3D SLAM (right)

It can be seen from the results that the map constructed with 3D Lidar SLAM reflects more environmental details. First of all, it has better convergence on the whole, and the entire environment map is constructed through only one exploration; secondly, it can detect the whole picture of obstacles, that is, including height information, so there will be no issue that the obstacle cannot be detected because it is too high or too low.

However, it is worth noting that since all the point cloud information is projected onto the plane, these also include the point clouds of the ground, which can be seen from the large number of spots in the figure, and they will lead to a problem that the robot mistakes it for an obstacle and is unable

to pass normally. To address this, an effective method is to set a pass-through filter, and project the point cloud with a height within a certain range as an obstacle. Here we set the height threshold as follow:

$$(z_{min}, z_{max}) = (0.1, 5.0) \tag{1}$$

The correct map obtained after treating the point cloud information of this interval as obstacles is shown in the Figure 5.

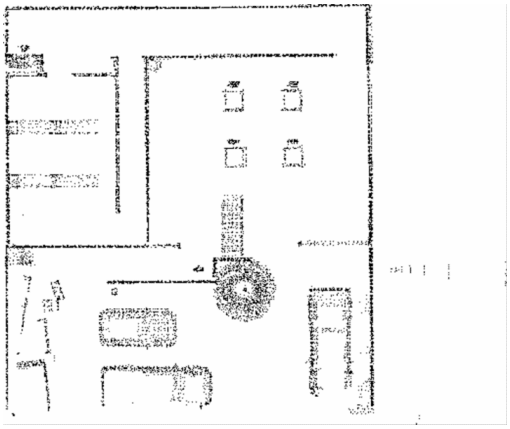

Figure 5: Improved map by filter

The above results show that the map obtained after applying the filter retains the original information while removing the influence of ground noise, thus can be used for subsequent navigation work. In order to quantify the mapping performance of this algorithm, similarly, evo is used to evaluate its localization performance, and the results obtained are shown in the Figure 6 below. It can be seen that compared with the 2D mapping method used before, this technique is increasing the detailed information, at the same time, the error between the trajectory and the ground truth is greatly reduced. Although there is still a certain deviation between the starting point and the ending point, we think this is acceptable, which will be confirmed in the follow-up navigation.

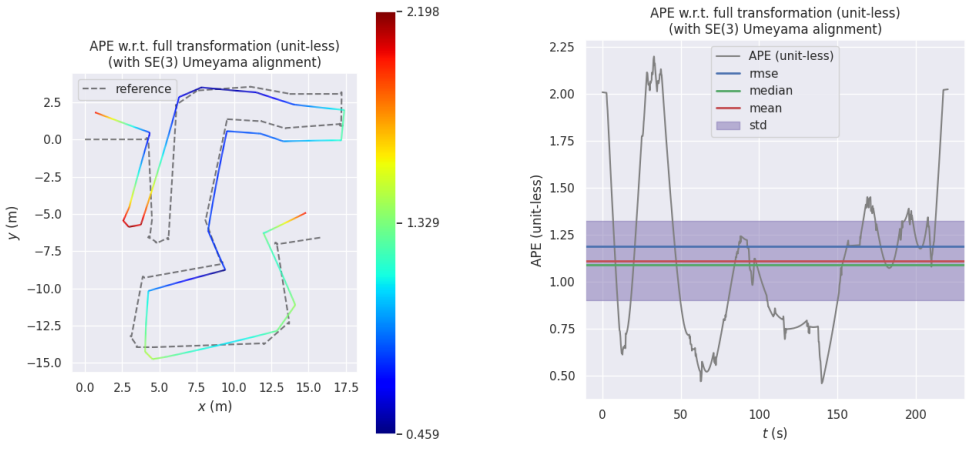

Figure 6: Evaluation on 3D Lidar SLAM

3 Navigation

In this section, we navigate the robot at rviz. For the global planning, after comparing Dijkstra and A*, we just use the Dijkstra method which is the default algorithm, the costmap is gener-

ated according to the map which is produced by A-LOAM. For the local planning, we compared three methods (**base_local_planner**, **dwa_local_planner** as well as **teb_loca_planner**), then we select the **teb_local_planner** as our algorithm. In addition to performing project-specified obstacle avoidance, we also added obstacles to the simulation environment to test the efficiency of the local planning algorithm. Lastly, we discuss the difficult we faced when we doing project and some possible solutions.

3.1 Global Planner

There are three global planners in ROS: **carrot_planner**, **navfn** and **global_planner**. For **navfn**, the algorithm used is Dijkstra. We prefer the **global_planner**, which contains two algorithms, Dijkstra, and A*, and more options. For better performance, we compare the effect of the two algorithms, discuss their shortages and advantages.

3.1.1 Comparison of Two Methods

Firstly, we add **global_planner_params.yaml** file in params folder and set **lethal_cost** = 253, **neutral_cost** = 66 and **cost_factor** = 0.55. Then we set **use_dijkstra** to true or false to use Dijkstra or A*. The results are shown in Figure 7, we note that when using Dijkstra, the visited area is large, but the path is basically in the middle. However, for A*, although the visited area is small, the path is so close to the wall looks very dangerous. Therefore, we finally choose the Dijkstra as global planning algorithm.

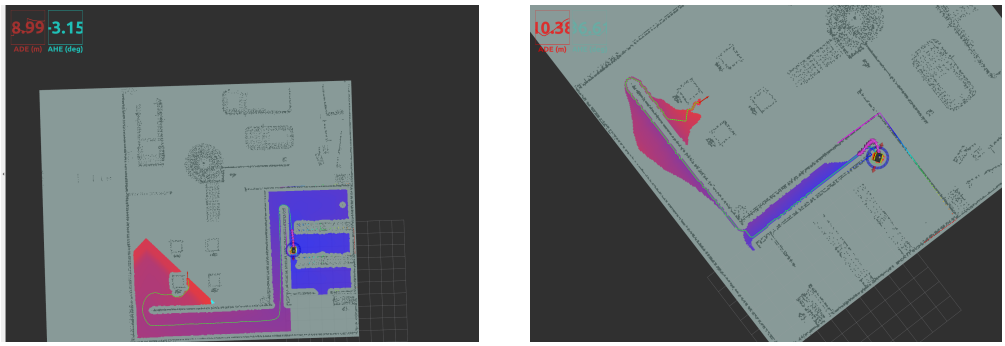

Figure 7: Comparison of Dijkstra algorithm (left) and A* algorithm (right)

3.1.2 Prohibited area

The control room is inaccessible, so when doing global planning, this area is not considered. Then we use the expansion pack, which is called **costmap_prohibition_layer**, then we just need to add a **.yaml** file which defines the prohibited area, then add this node to **move_base** and **global_costmap** parameter file. Results as shown below:

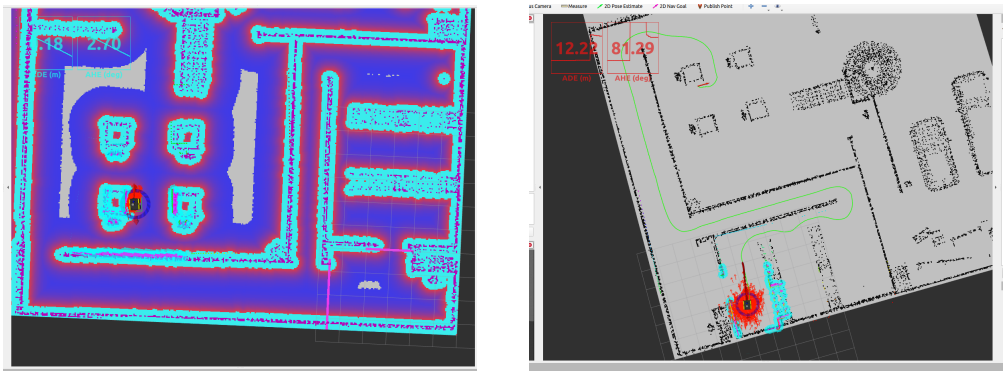

Figure 8: Map with prohibited area (left) and Global planning path (right)

Figure 8 left shows the control room is forbidden (lower right corner), and we find the global planning path is avoided this room as shown at right of Figure 8.

3.2 Local Planner

The default plug-in is **base_local_planner**, another plug-in is called **dwa_local_planner**, the execution ideas of these two algorithms are similar, but we find that the use effect of **base_local_planner** is not very good, the local path planning is very short, the car moves slowly and often deviates from the track. Another possible selection is **teb_local_planner**. After comparison and data research we find that among them, the forward-looking algorithm of **dwa_local_planner** is shorter, and the car will basically keep running along the global path, while the forward-looking algorithm of **teb_local_planner** is relatively long so when encountering obstacles, the local path will automatically plan to avoid obstacles, but if the car is in a more complex environment, the local path will be more prone to confusion.

3.3 Obstacle Avoidance Effect

For the **teb_local_planner**, we tune the parameter file, some main parameters like **dt_ref** and **dt_hysteresis** which determine the forward-looking distance, we set these two parameters as 0.25 and 0.05, because if **dt_ref** and **dt_hysteresis** are too large, then the length of forward-looking is not enough to avoid a big obstacle. A set of obvious comparisons is shown in Figure 9:

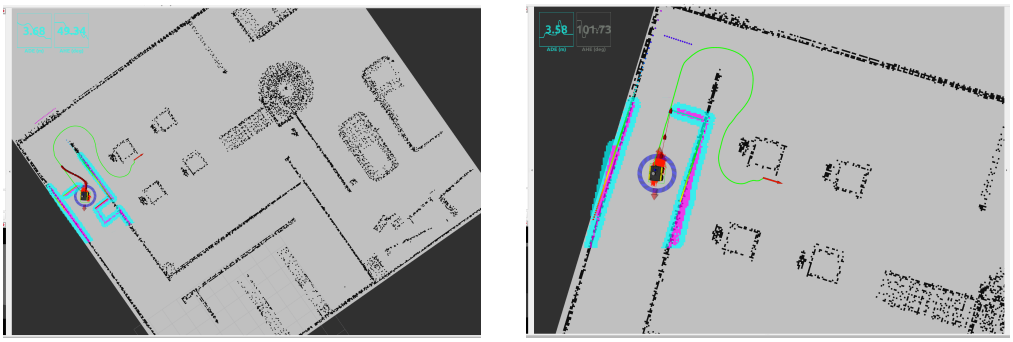

Figure 9: Comparison with different parameters (0.25 and 0.05 for left, 0.8 and 0.1 for right)

From the above images, we find that the number of arrows and forward-looking length is shorter than before if the value of these two parameters is set larger. Other important parameters are listed follow:

- **footprint_model**: set as “polygon”;
- **vertices**: [[-0.21, -0.165], [-0.21, 0.165], [0.21, 0.165], [0.21, -0.165]], respect to the size of robot;
- **min_obstacle_dist**: this parameter determines the minimum distance between our robot and obstacle, the robot is possible come to the dead zone if this value is small (0.1 for our);
- **include_costmap_obstacles**: set as “True”, if set “false”, the robot will not avoid obstacle.

After deciding the parameters, we put obstacles at gazebo randomly and observe the performance of navigation algorithm, some avoiding obstacle effects are shown as follows:

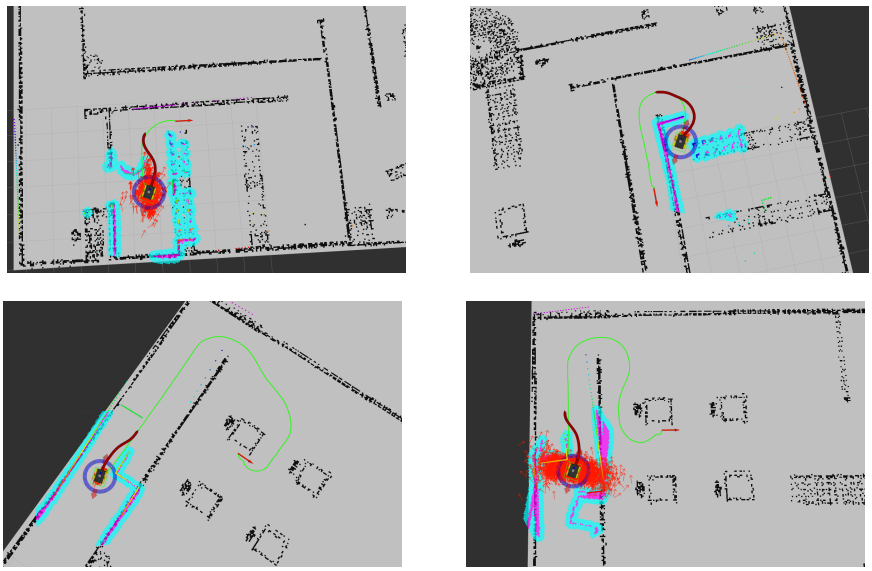

Figure 10: Avoiding obstacle effect

From the above figures, we can find that the avoiding obstacle effect is good, and the local path planning is very good even though there exist some neighbor obstacles just like the last two images.

Note: because these obstacles are added later, so we need to change the obstacle layer parameter at **costmap_common_parameter** file, set the **sensor_frame** as “tim551” to get the real-time pointcloud information.

3.4 Challenges and Proposed Solutions

There are also some problems we meet when we doing navigation, some problems are solved but some are not.

3.4.1 Hitting the wall

When we navigate the robot at the start, we find that the robot does not follows the global planning path, and it will hit the wall when facing the 180-degree corner. The basic reason is due to the parameter of the local planner algorithm, so we change the default algorithm and tune the parameter of the new algorithm, another important element is the width and height of the local cost-map, the value does not need to be large, finally, we determine using width and height equal 5. Figure 11 left shows the hitting wall, and Figure 11 right shows the result after tuning the parameter.

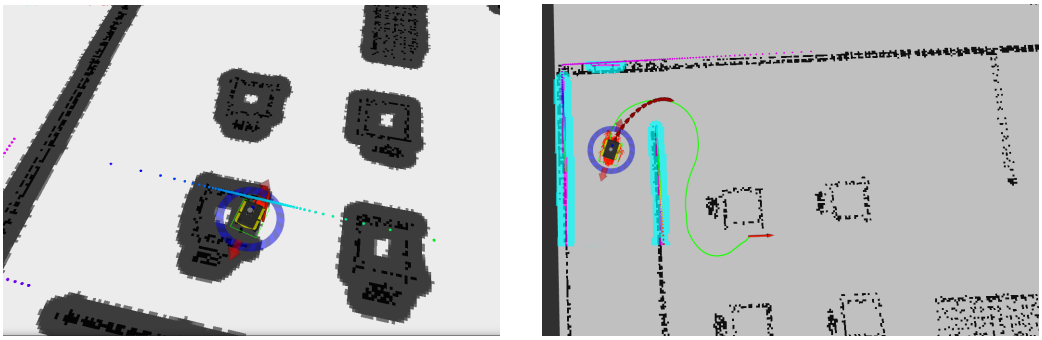

Figure 11: Hitting the wall (escape the planning path) and Follow the planning path (after tuning)

So, one of feasible methods to solve this problem is:

- Replace some better local algorithm for the default;
- Decrease the size of local cost-map;
- Give large value of inflation parameters to global cost-map and on the contrary, give small value to local cost-map.

3.4.2 Point cloud offset

We find that the point cloud will drift if the robot does not move. Although the ROS will re-match the point cloud position when the robot starts moving, but it will result in some problems such as unable to plan the route because the endpoint at the obstacle area is due to the offset. Figure 12 left shows the point cloud drift circumstance when we not tuning the parameters, Figure 12 right shows the results after we optimise the parameters (both stay at one location around long time).

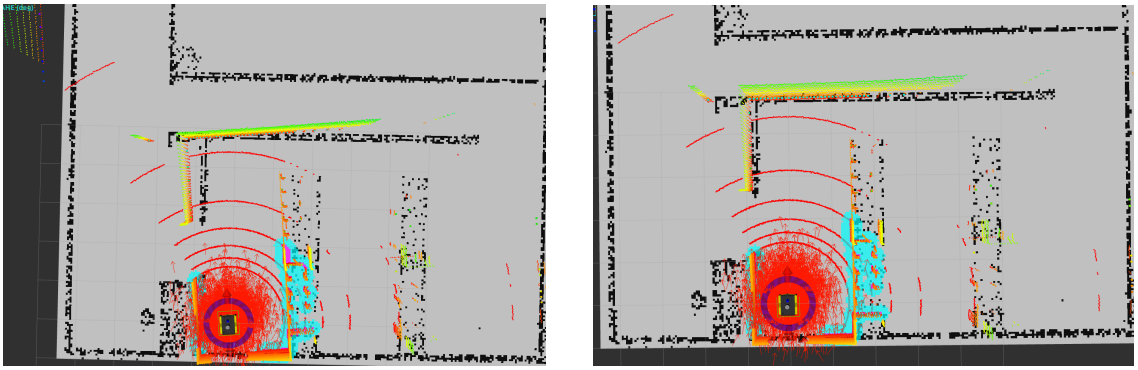

Figure 12: Point cloud drift obviously and Result after optimization

The left image above with a small value of global **inflation_radius** (0.5) and gives a large value of **cost_scaling_factor** (10.0), as well as an empty cost-map converter, the right image above with opposite value of global inflation parameters (1.5 and 3.0) and specifies cost-map converter. We can draw the conclusion that tuning these parameters has an evident effect to optimize the point cloud offset problem, but it can not eliminate this issue completely.

So, for this problem, some solutions are listed below:

- inflation parameters: **inflation_radius** and **cost_scaling_factor** for global cost-map;

- costmap_converter parameters: The default is no parameter for **costmap_converter_plugin**, there are four choices can be selected, after trying, we find "**CostmapToLinesDBSRANSAC**" has the best contribution to solve this issue;
- localization algorithm: The default location algorithm is amcl (Adaptive Monte Carlo Localization), there are some other better algorithm like multi-sensor fusion positioning algorithm.

3.4.3 Running close to wall

The global planning algorithm will calculate the best choice of path which means that the cost of the path is the smallest, but under this circumstance, it is very easy to occur that the robot hit the wall or turn takes many times and some similar problems. In this project, we do not need the best path, so we try to find the path which is always at the road mid. The inflation layer has contribution to this issue, so one of the solutions is increasing the inflation parameter of the global cost-map to make the allowing pass area close to the middle of the road. Figure 13 left shows the path which is near the wall, and Figure 13 right shows the result after increasing the inflation parameter.

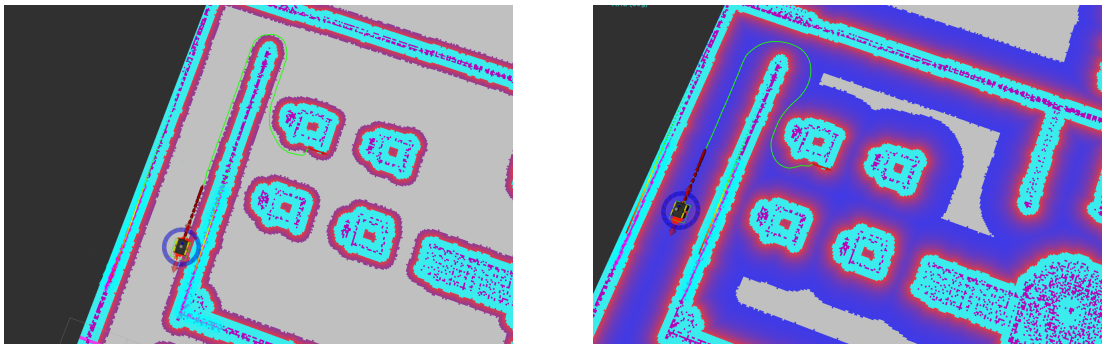

Figure 13: Planning path near the wall and Planning path at the middle of road

3.5 Overall Effect

For the navigation performance test, we score our algorithm from the following aspects: error of the final pose and the goal pose, navigation with randomly selected start and end pose. The given sequence of goal poses of our group is "**Assembly Line 1**", "**Packaging Area 1**" and "**Vehicle 1**", following images show the error between the final pose and the goal pose.

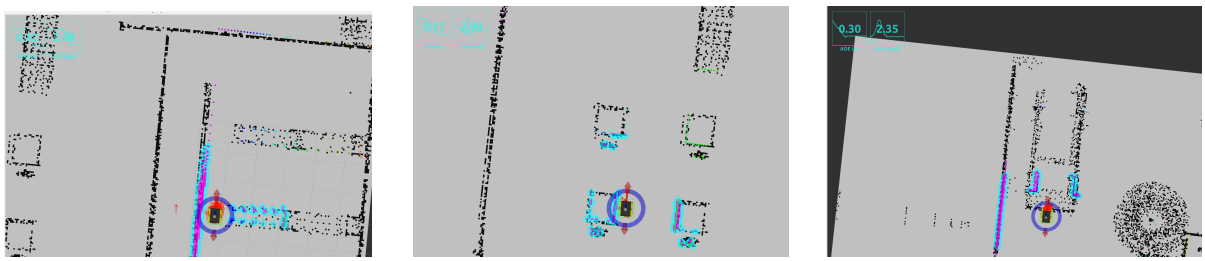

Figure 14: Assembly Line 1, Packaging Area 1 and Vehicle 1 (from left to right)

As the above table shows, the error is acceptable, some reasons like the global map not matching the true map excellently will result in the error. Then we can select a better mapping algorithm or tune the local planner algorithm's parameters like **yaw_goal_tolerance** and **xy_goal_tolerance**, these two parameters can limit the robot reach the goal pose as close as possible, but note that

Pose	Assembly Line 1	Packaging Area 1	Vehicle 1
Position Error(m)	0.12	0.17	0.30
Heading Error(deg)	2.48	2.99	2.35

Table 1: Position Error and Heading Error of three poses

the local planner algorithm will calculate many local paths when the robot is close to the goal position and the robot will oscillating near the target point if the value of these parameters is too small.

For randomly selected start and end poses, if we want to change the initial position of the robot, we should change the value of **urdf_spawner** at the **spawn_jackal.launch** file and the value of **initial_pose_x** and **initial_pose_y** at the **amcl.launch** file. We change the goal position value at **config.yaml** file easy to do error analysis. One performance of randomly selected start and end pose is shown as follows, we can find the performance is good because the error is 0.19m and 2.34deg:

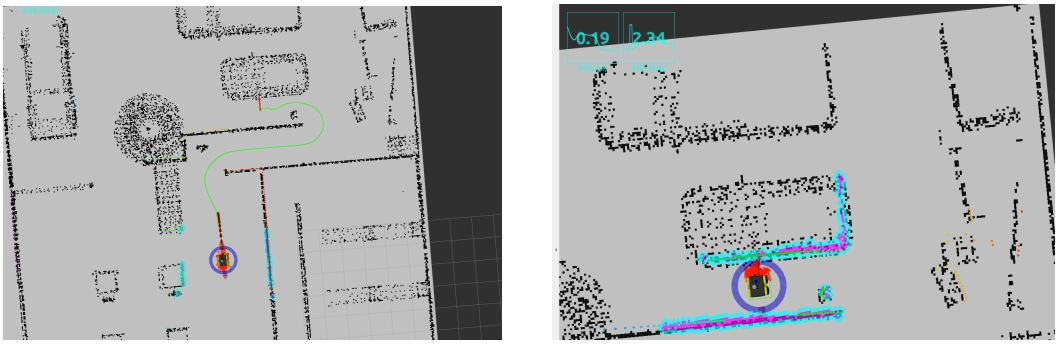

Figure 15: Navigation of randomly selected start and end pose

4 Conclusion

In this project, we first build the map by using the A-LOAM algorithm, then for navigation, we compare the Dijkstra and A* and determine the default algorithm is enough and better than A*, the local planner is **teb_local_planner**. Then we give some solutions regarding the problems when we running, these solutions have obvious effects, but they can not solve completely.

For the avoiding obstacles, we put additional obstacles at gazebo, and the local planner can avoid them perfectly. Lastly, we select the start and end point randomly to test the performance, the effect is satisfactory.

Github repo link: `https://github.com/zzf-zzf/mobile_final_group12.git`