EE346 Final Lab Report

11813118 徐衡
11911039 田宇琼

Introduction

In the final 5-minute competition, there are three separate tasks that our turtlebot can attempt: autonomous navigation to P1, P2, P3 or P4, lane following, and aruco marker detection. The robot will begin its operation at P1, and score ranking is based on a points system:

Each successful visit to a Pn is worth 10 points;
Success traversal of the racetrack is worth 20 points;
Each detection of an aruco marker is worth 5 points;
The robot must return to P1 (worth 10 points) before a next lap;
Each "cut corner" in the racetrack will result in a deduction of 1 point;
Deviation to Pn will result in a deduction of 1 point per 2cm;
Lift and re-place the robot will result in a penalty of 5 points;

According to the above rules, our robot can repeat tasks to obtain higher points, and we choose the lane following and P2 navigation as our designed route.

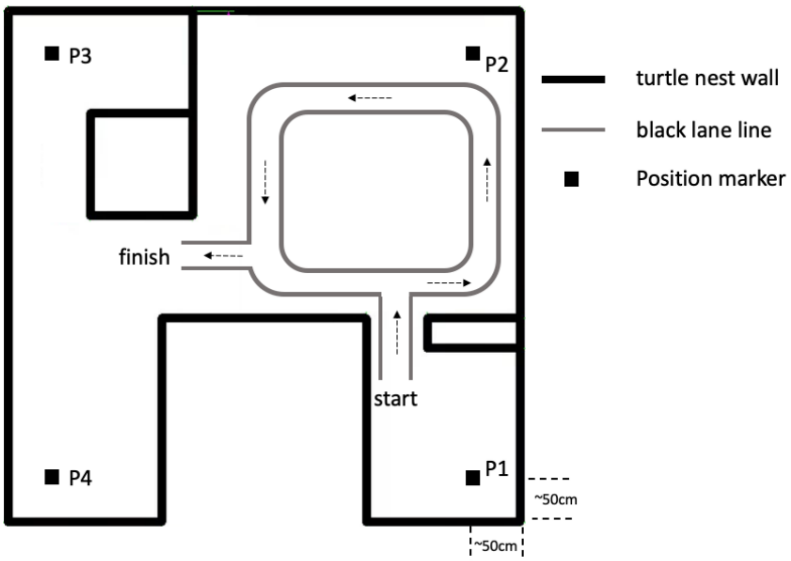

Figure 1: Robot environment with the racetrack to traverse and positions to visit

Experimental platform

i. Turtlebot3 Burger

TurtleBot3 is a small, affordable, programmable, ROS-based mobile robot for use in education, research, hobby, and product prototyping. It is combined with chassis, motor, wheels, OpenCR board, computer(Raspberry Pi), sensor(LiDAR, camera), battery. For more details, see turtlebot3 burger overview.

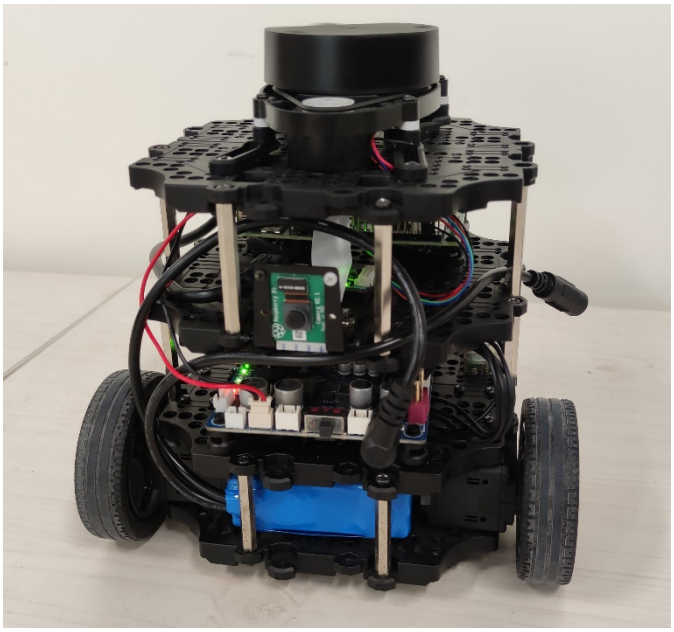

Figure 2: Turtlebot3 Burger

ii. Sensors

This project mainly uses Laser Distance Sensor LDS-01 and Raspberry Pi Camera on the turtlebot. LDS can navigate the robot to P2 and P1, while camera is able to capture images and guide the robot for lane following. See LDS and Raspberry Pi Camera.

iii. Remote PC

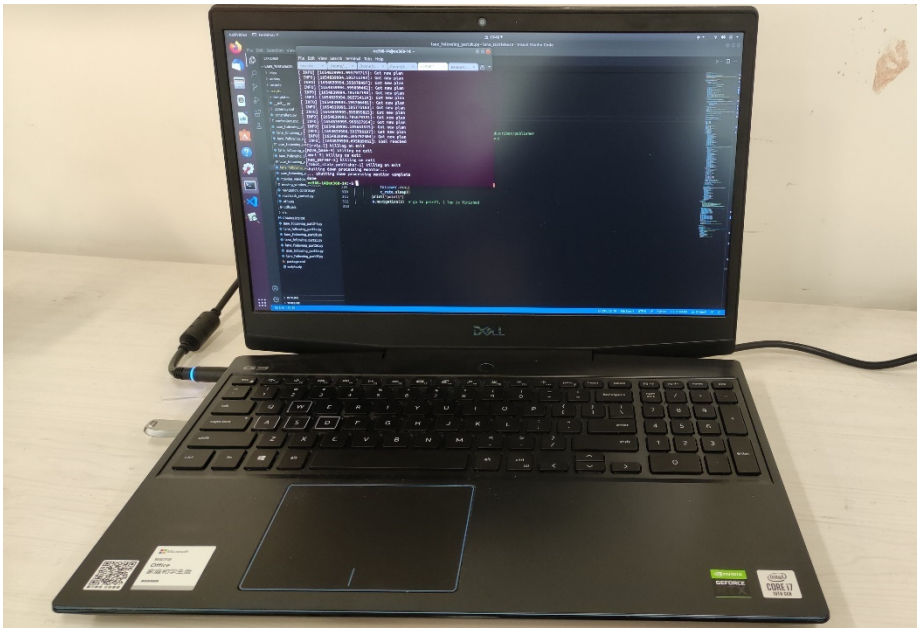

Figure 3: Dell G3-3500 2020

The remote PC (or host PC) used in this project is Dell G3-3500 2020, with i7-10750H CPU, RTX2060 GPU and 16GB RAM. To accomplish the task, the

computer should have Ubuntu 18.04 operating system and be installed with ROS Melodic.

iv. ROS

ROS, or Robot Operating System, is an open-source robotics middleware suite. Although ROS is not an operating system (OS) but a set of software frameworks for robot software development, it provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. For more details, see ROS wiki.

Approach

Mapping

By following the turtlebot3 slam tutorial, we can build the racing environment map for the future navigation task.

Launch mapping program:
roslaunch turtlebot3_slam turtlebot3_slam.launch
Use keyboard to control Turtlebot3:
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch

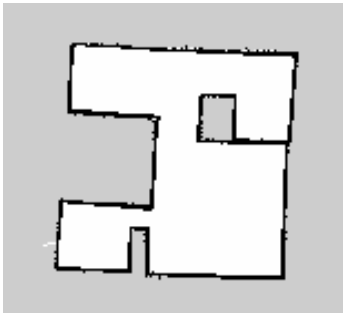

Figure 4: Saved map

As shown in Figure 4, the result mapping is not perfectly precise, which is due to the error data matching between two steering engines and LiDAR. When the robot is instructed to move forward at full speed, it will deviate to the left, hence we discover that the revolution speed of two wheels is not accurate, which results in the mapping deviation.

Lane Following

We should first repeatedly subscribe images from the camera:

```
def image_callback(self, msg):
    # receive image
    self.image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
```

Next, do the homography transformation to obtain the bird's-eye view of the lane. It is worth noticing that the threshold segmentation for black lines should be placed before homography process for a better performance.

In order to enter the racetrack and pass the first corner, we design a series command of moving forward, turning right, moving forward and turning left. However, this type of design requires a quite precise starting pose, while the navigation from P1 to the entrance can hardly provide this accurate position and orientation without oscillation. Therefore, we replace the first “moving forward” command by the double-line following to adjust robot pose and reduce the starting error.

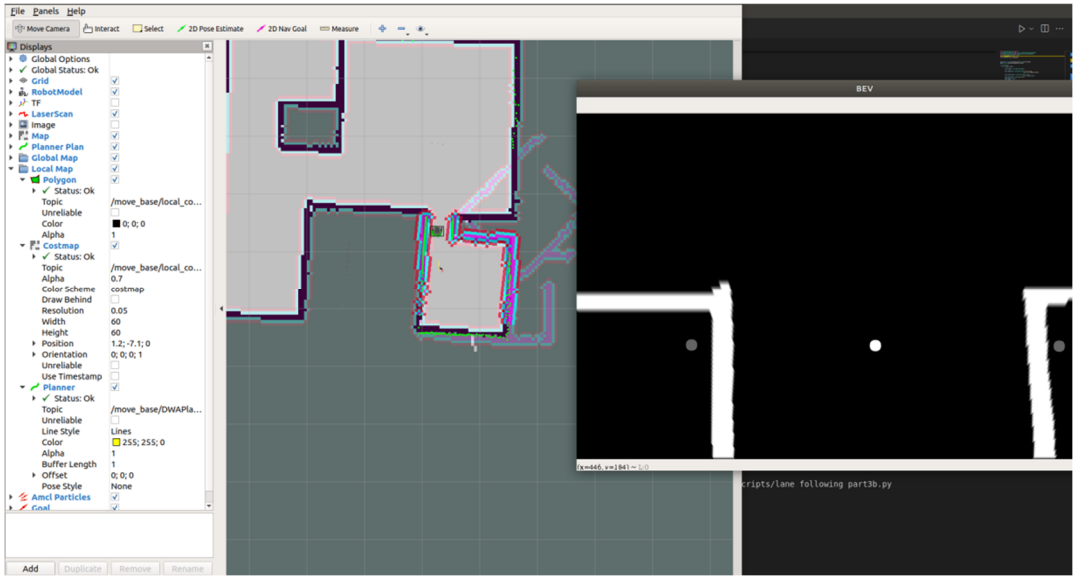

Figure 5: Entering the racetrack (including the 1st corner turning)

When going to the corner near P2, the program is switched from lane following to navigation. After the visit to P2, the program will be switched back to lane following when the navigation orientation is towards the lane.

For the basic double-line following, we use linear error control of angular speed, while the linear speed is set to be maximum. When either line is missing, the program completes a centroid on that side and continue lane following:

```
if (M1['m00'] > 0 or M2['m00'] > 0):
    if M1['m00'] == 0:
        cx1 = 200
        cy1 = 300  # complete left line (circle)
        cx2 = int(M2['m10'] / M2['m00'])
        cy2 = int(M2['m01'] / M2['m00'])
        fpt_x = (cx1 + cx2) / 2
        fpt_y = (cy1 + cy2) / 2
    elif M2['m00'] == 0:
        cx1 = int(M1['m10'] / M1['m00'])
```

```
        cy1 = int(M1['m01'] / M1['m00'])
        cx2 = 800
        cy2 = 300  # complete right line (circle)
        fpt_x = (cx1 + cx2) / 2
        fpt_y = (cy1 + cy2) / 2
    else:
        cx1 = int(M1['m10'] / M1['m00'])
        cy1 = int(M1['m01'] / M1['m00'])
        cx2 = int(M2['m10'] / M2['m00'])
        cy2 = int(M2['m01'] / M2['m00'])
        fpt_x = (cx1 + cx2) / 2
        fpt_y = (cy1 + cy2) / 2

    cv2.circle(BEV, (cx1, cy1), 10, (100, 255, 255), -1)
    cv2.circle(BEV, (cx2, cy2), 10, (100, 255, 255), -1)
    cv2.circle(BEV, (fpt_x, fpt_y), 10, (255, 100, 100), -1)

    err = 10 + w / 2 - fpt_x  # linear control
    self.twist.linear.x = 0.22
    self.twist.angular.z = err * 0.01
    self.cmd_vel_pub.publish(self.twist)
```

For corner turning, we use the similar series-command design: moving forward for a while and turning left. The turning trigger is based on the mask judgement: when the upper left of the image detects no black lines, turn left; when the upper right of the image detects no black lines, turn right to exit the racetrack; when the top half of the image has no black content, start the racetrack entering command. Mask demonstrations are in the below:

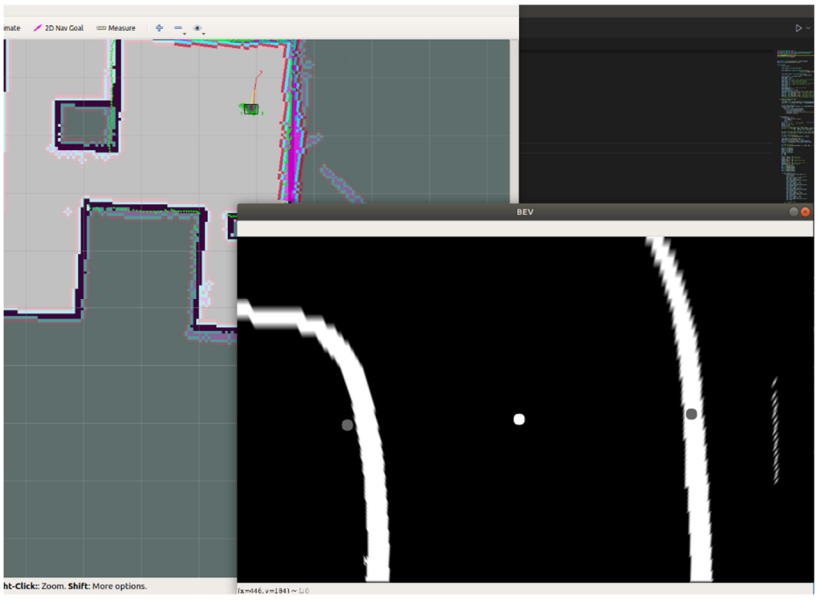

Figure 6: The 2nd corner detecting for P2 navigation

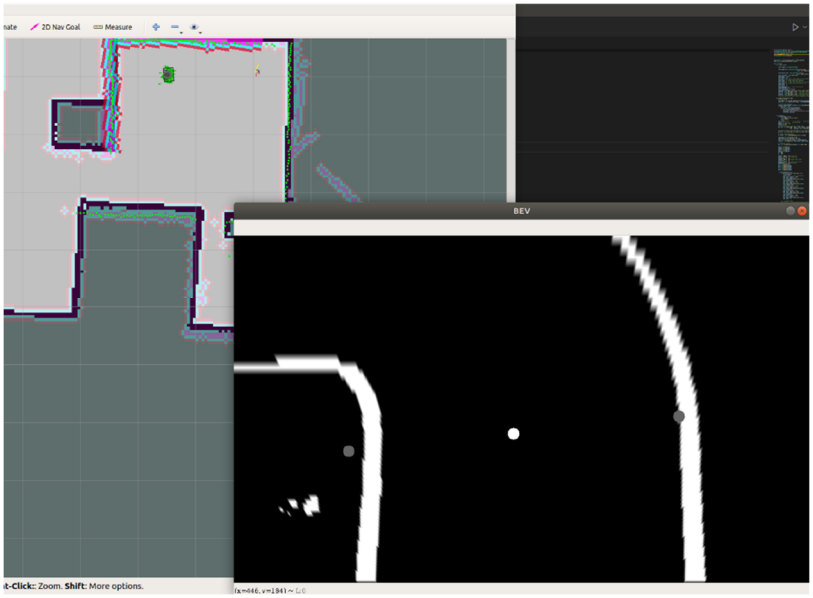

Figure 7: The 3rd corner turning

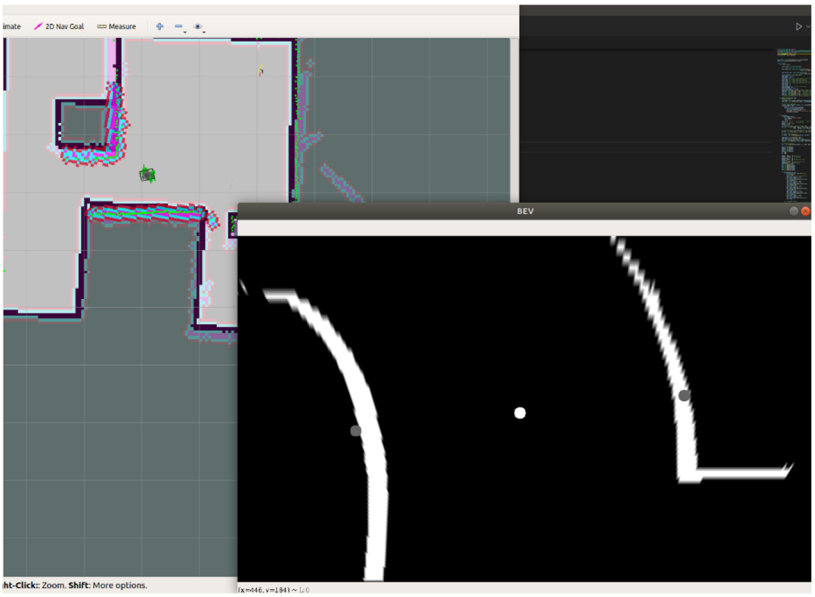

Figure 8: The 4th corner turning

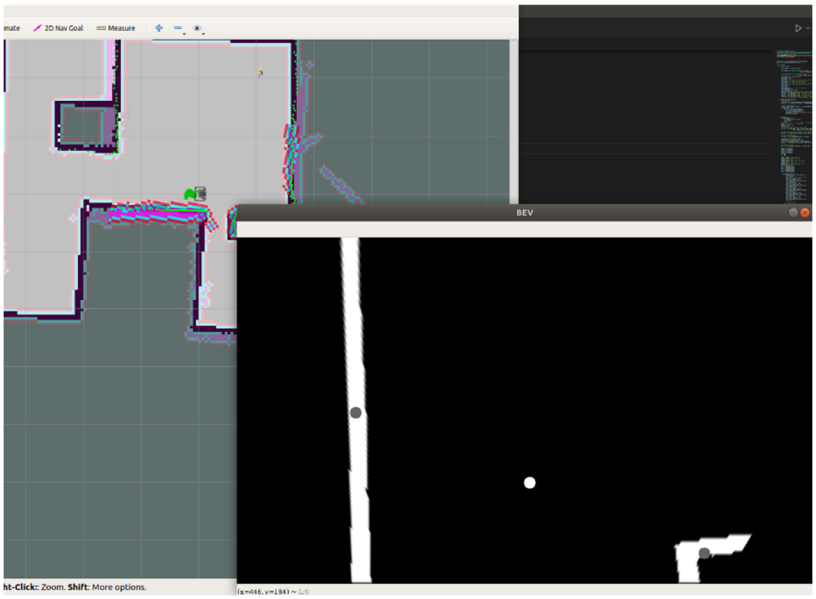

Figure 9: Exiting the racetrack

After command turning, however, images cannot be updated immediately, hence continuous turning happens based on the not-updated image. To avoid this situation, we introduce several timers, and corresponding judging conditions are listed below:

```
if M3['m00'] == 0 and self.finish == 0 and t2 - self.s0 > 15:
```

```
elif self.turn2 == 1 and t2 - self.s1 > 7:
```

```
elif self.turn3 == 1 and t2 - self.s2 > 7:
```

```
if thistime - self.s3 >= 3 and self.right==0 and M4['m00']==0 and self.finish == 0:
```

In the original design, timers are initialized at the start of the program. However, the robot will turn left or right once the lane following fails, since the starting process, including navigating to the starting point, entering the racetrack and turning the first corner, consumes much time. Therefore, timers are added by 120s at initialization to solve this problem, and they are updated when required.

Navigation

Through command “rostopic echo /amcl_pose”, we can get the current pose of the robot in our saved map, and we save these required points data:

```
self.goalPoints = [
    # position(x,y,z) and pose(quaternion)
    # In this lab, x is from bottom to top and y is from right to left
    [(4.125, -0.05, 0.0), (0.0, 0.0, 0.737849902397, 0.67)], # p2
    [(4.4, 3.8, 0.0), (0.0, 0.0, 0.025369587372, 0.999678140221)], # p3
    [(0.26703, 4.1, 0.0), (0.0, 0.0, 0.99998940623, 0.00460298028122)], # p4
    [(0.030, 0.00, 0.0), (0.0, 0.0, 0.73358, 0.719)], # p1
    [(0.735, 0.7, 0.0), (0.0, 0.0, 0.0456504303085, 1)], # lane_follow starting point]
```

After placing the robot at P1, the program will send the initial pose of it in the map, and then navigate it to the starting point of the racetrack.

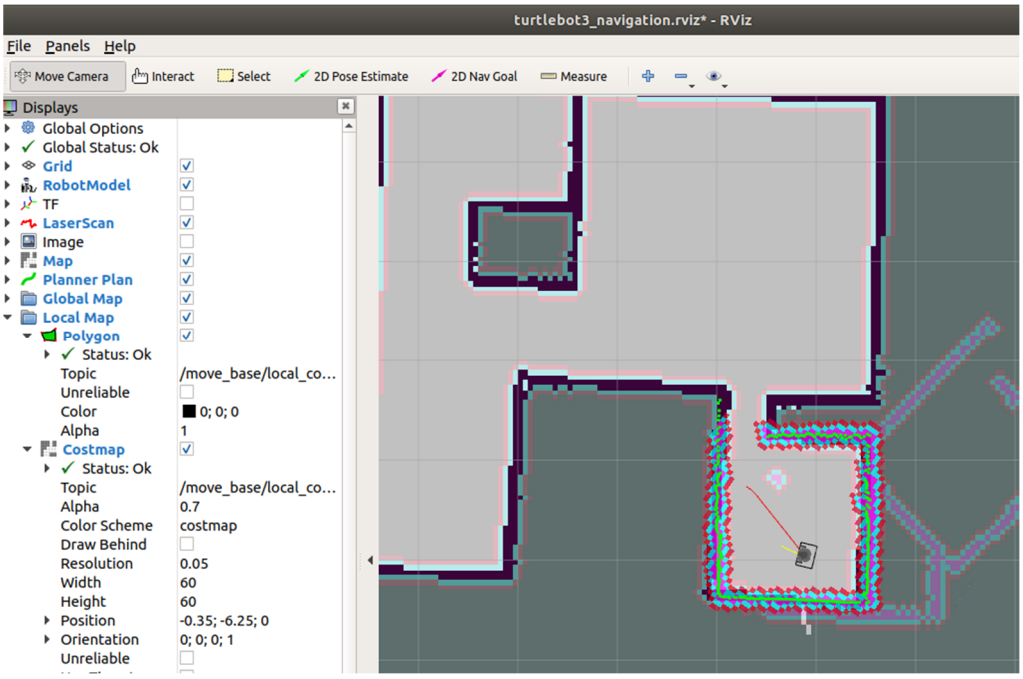

Figure 10: Navigating to the starting point of the racetrack

After detecting the second corner, the program should navigate the robot to P2.

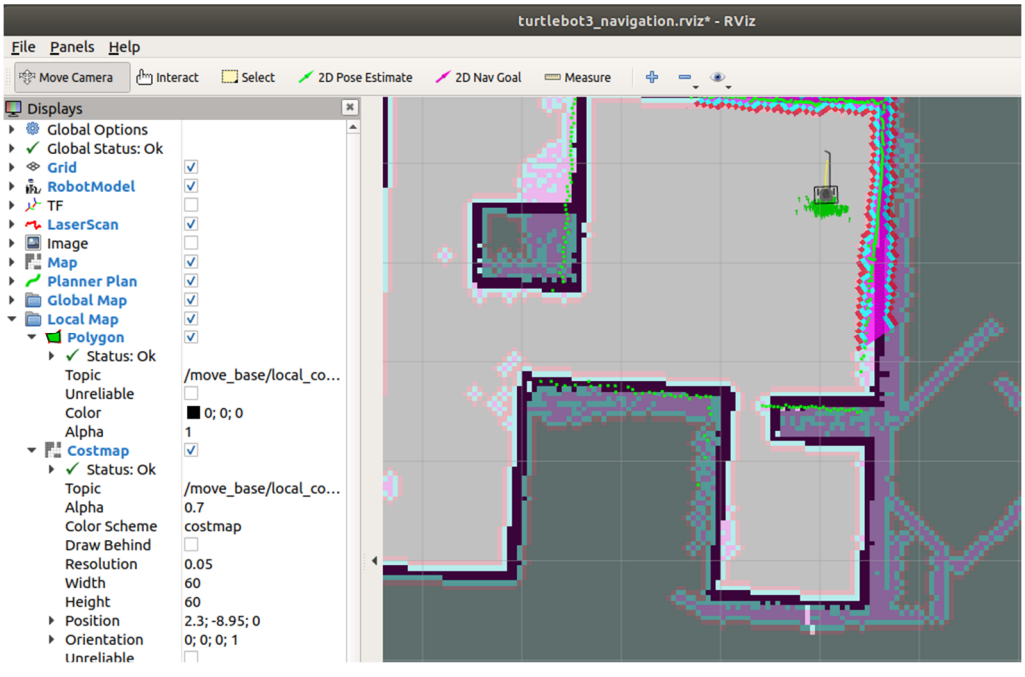

Figure 11: Navigating to P2

After exiting the racetrack, the program should navigate the robot back to P1.

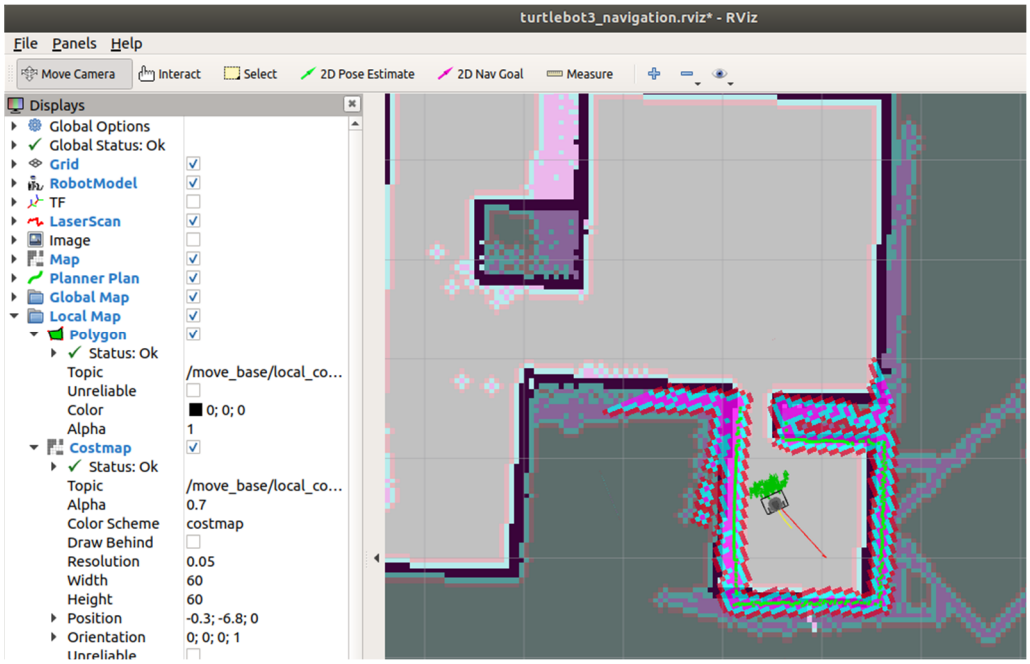

Figure 12: Navigating to P1

After returning to P1, we should not appoint the initial pose again, because the navigation to P1 is not absolutely precise in reality, which deviates from the initial pose (or P1 point data). Thus, we can directly enter the next lap of the competition.

Parameter adjustment:
In dwa_local_planner_params_burger.yaml, min_vel_theta is changed from 1.37 to 0.75, and acc_lim_theta is reduced from 3.2 to 2.0, which aims to avoid oscillation when the robot is adjusting its orientation. In addition, yaw_goal_tolerance is adjusted from 0.17 to 0.125 for a more accurate orientation when navigating.

In costmap_common_params_burger.yaml, inflation_radius is reduced from 1.0 to 0.1, which can help reduce obstacle blocking when leaving the racetrack at the T-Junction.

Aruco marker detection

Play audio once an aruco-marker is detected, and report its ID.

```
corners, self.markerID = aruco.detectMarkers(self.gray, aruco_dict, parameters=param)
if len(corners) > 0:
    aruco_ID = self.markerID.squeeze()
    if not self.aruco_found[aruco_ID]:
        self.aruco_found[aruco_ID] = True
        playsound("/home/ee368-14/catkin_ws/src/lane_turtlebot3/audios/"+self.audio_
filename[aruco_ID],block=False)
        print(self.markerID)
```

Competition Result

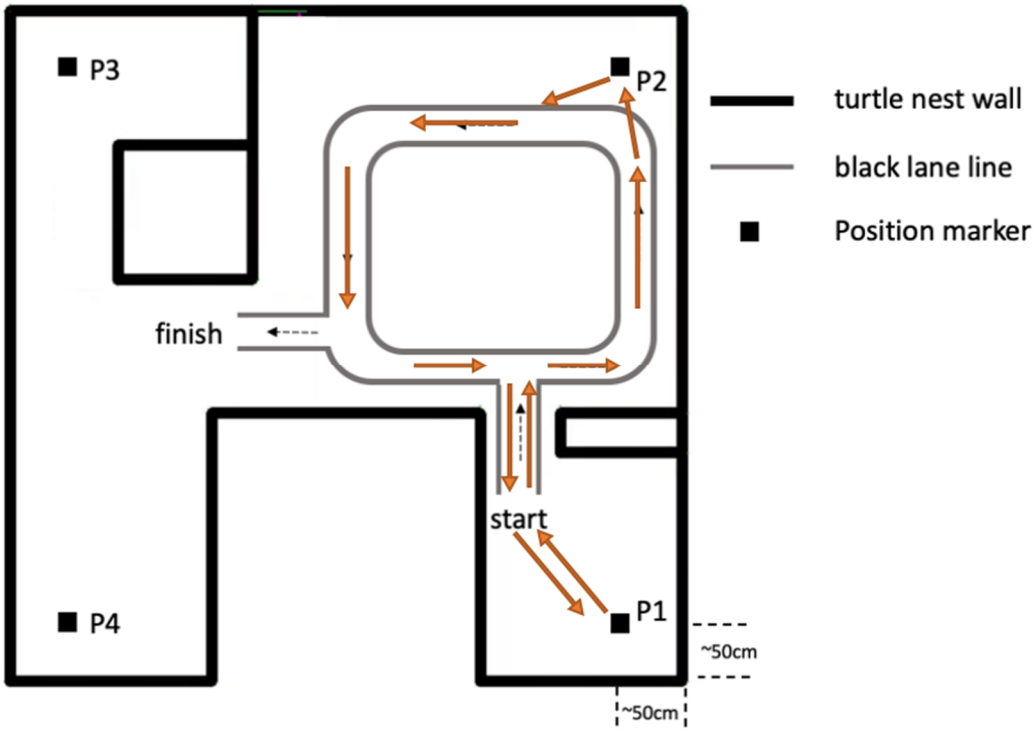

Figure 13: Designed route for the competition

In order to gain higher points in 5 minutes, we design the above route: P1 → racetrack → P2 → racetrack → P1, which is shown in the orange arrows. Ideally, our turtlebot can run up to 4 laps in 5 minutes.

i. Round 1

At the first lap, unfortunately, the scanned map rotated severely at the T-Junction exit, therefore the navigation stuck into the wall for a while on the way back to P1. When facing this problem, DWA local planner will do the rotation recovery for the robot to re-calibrate the map. As a result, this stuck situation and robot rotation spent about 20 seconds to return to normal, and our robot could only achieve 3.5 laps in the end.

Scores in round 1:

$$7P + 3.5R + 3M - 3 = 152$$

where P is navigation point Pn, R is racetrack lap, M is aruco marker detection and point deduction is due to 2 “cut corners” (or covering the inner lines) and 1 2cm-deviation from Pn.

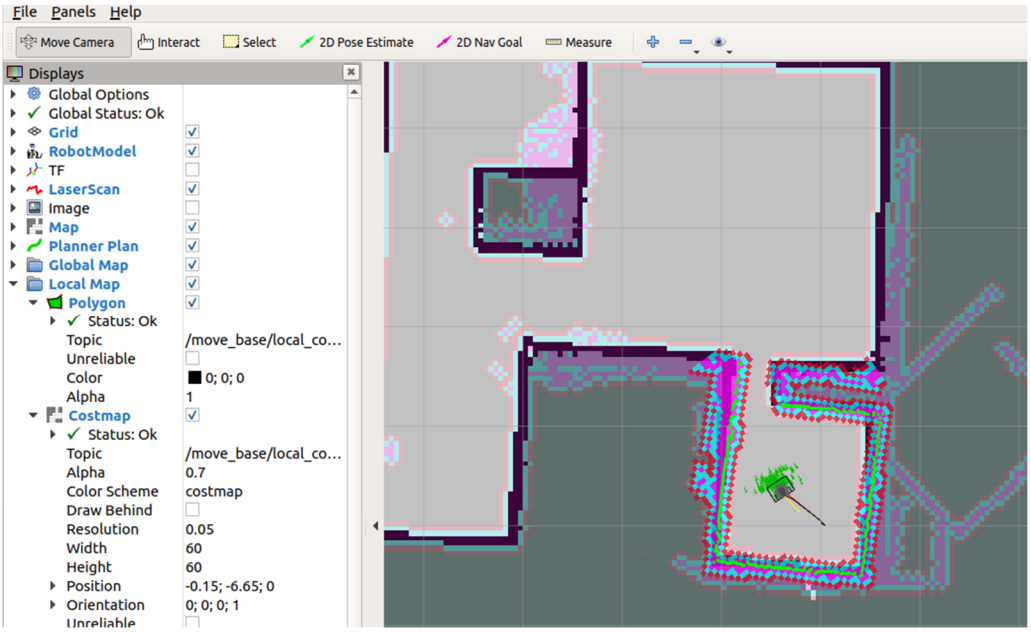

Figure 14: Rotatory deviation in the map

Actually, this map-stuck accident can hardly reappear, because the navigation program can dynamically calibrate the map when the robot is moving. Figure 12 is the demonstration of this rotatory deviation. When the scanned wall blocks the T-junction, our navigation will fail and require rotation recovery, but this situation is quite rare.

ii. Round 2

In the second round, our robot performed perfectly, with only a small problem of one "cut corner", while all the aruco-markers were detected correctly and P1&P2 were visited precisely.

Scores in round 2:

$$7P + 4R + 4M - 1 = 169$$

Conclusion

This competition is an educational one and we learned a lot from it. When dealing with problems like network, robot hardware problems and lane-following strategies, the main thing is to understand why and find out the proper method of solution. It is a pleasure that our robot finally accomplished all the tasks perfectly, although Lab 6 and Lab 7 (this project) consumes over 60 extra hours in the lab.

Video Demo: https://www.bilibili.com/video/BV1zF411F77z/

GitHub Site: https://github.com/ZeonH/EE346-Final-Lab

Contribution:

Heng Xu: Method design, code writing, parameter adjustment (70%)

Yuqiong Tian: Environment configuration, report writing (30%)