EE346 Capstone project

Name：王孜昊，刘江舢　　SID：11811718,11811716

Introduction:

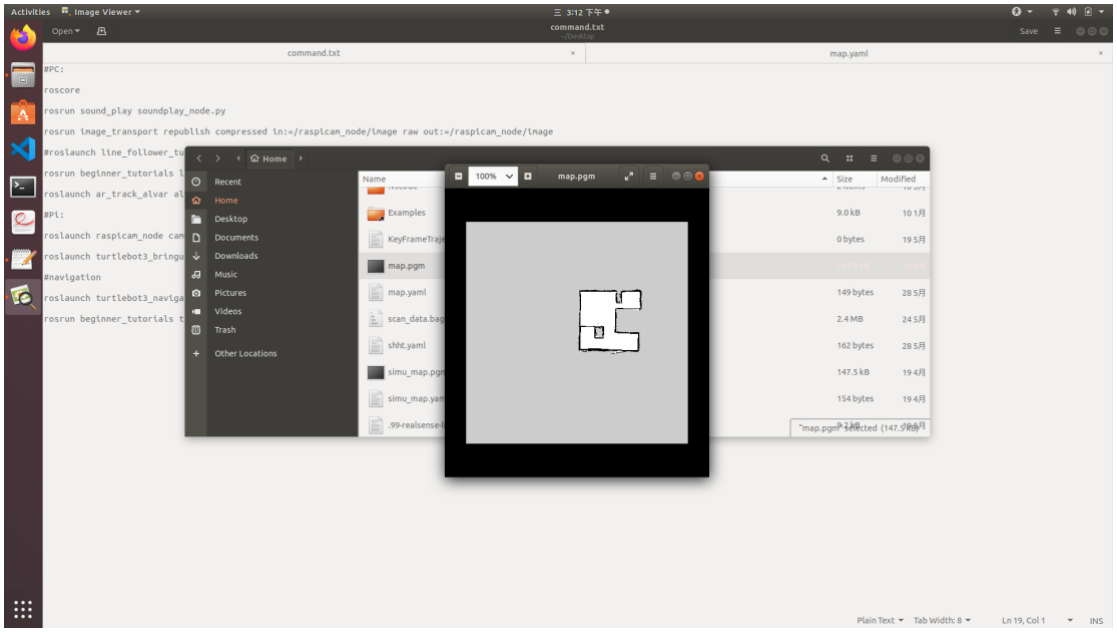
In this experiment, we use the method applied in previous labs to achieve the aims demanded in the requirement file. For round one, we should go over each target in order with accuracy. For round two, we need to design route to achieve as many jobs as possible. Here is the lab procedure from the pre-lab preparation to the final result.

Process and Result:

Pre-Lab preparation:

1. Constructing the map:

   Recall the operation we done in the previous lab. Using RVIZ, we manage to build a map using keyboard manipulator. The robot will go over the whole field and using SLAM to draw a map. Note that due to the propogation delay of network and radar, when we manipulate the turtlebot robot, we should move the turtlebot as smooth as possible. Turning and moving simotaneously will lead to glitches on the edges of the map. If the glitches are too big or in the important tunnel, it will block the tunnel. For example, if the glitch happens to locate at the tunnel area, the turtlebot may recognize the empty space as obstacle. Overlapping with the radar real-time sensing result, the available path maght be too narrow or even blocked somehow. This will lead to waste of time or even failure in both round.



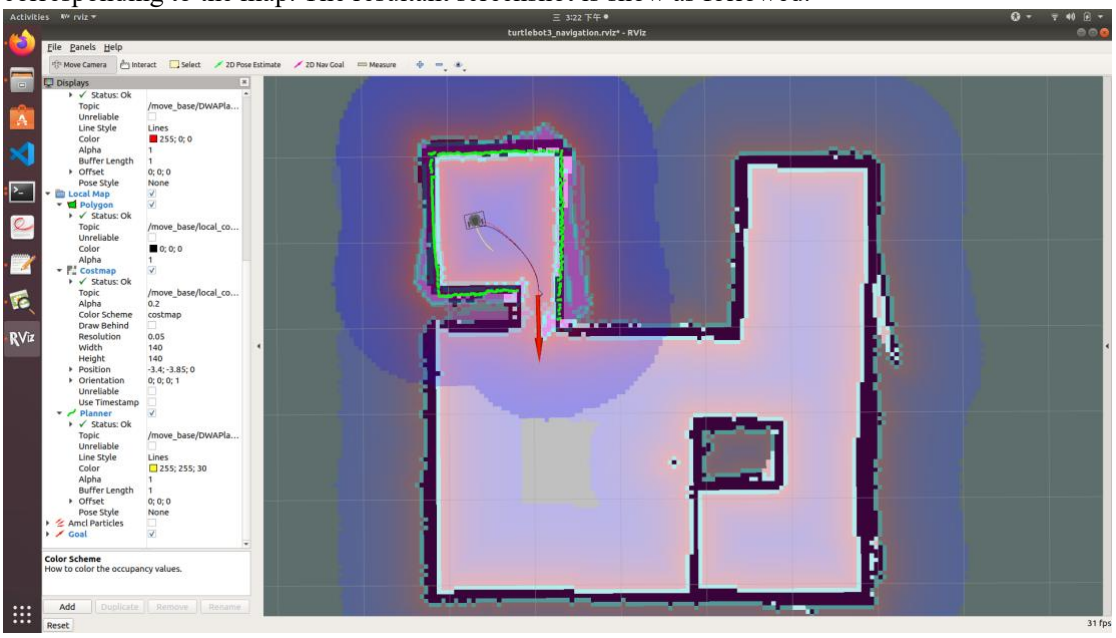
2. Test the AR tag recognation and performance of sound

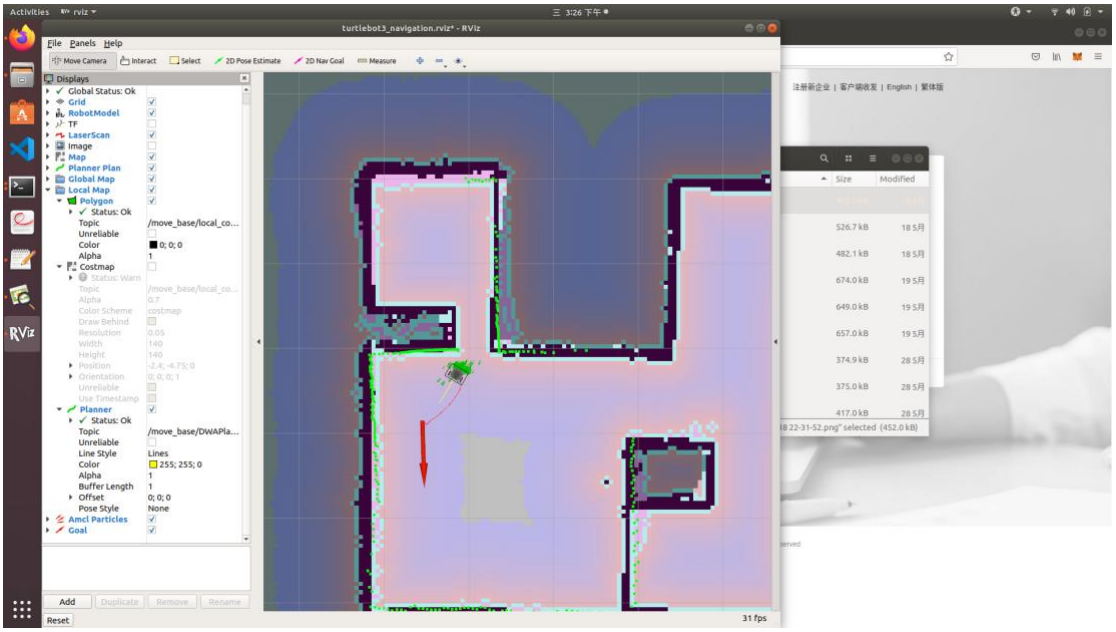
   As we test the turtlebot camera is of 30 frame per second, which the high refreshing rate enable fast recognition of AR tags. If the camera catch the AR tag, the remote PC will sound for certain time which should be the ID of the AR tag. As we tested in previous lab, the maximum distance of recognition is about 1.2m with little delay. This data certainly helps in the final round.

3. Simulation using navigation package

   Navigation launch file is an embeded file to execute the auto-navigation function with given

type of robot and map file. Using RVIZ, we need to first calibrate the position of robot corresponding to the map. The resultant screenshot is show as followed.





Executive plan and route design:

Based on the move_base message package, we use the ros listener operation to catch the message publish by the move_base message. Using the following command, we can move the robot to certain position.

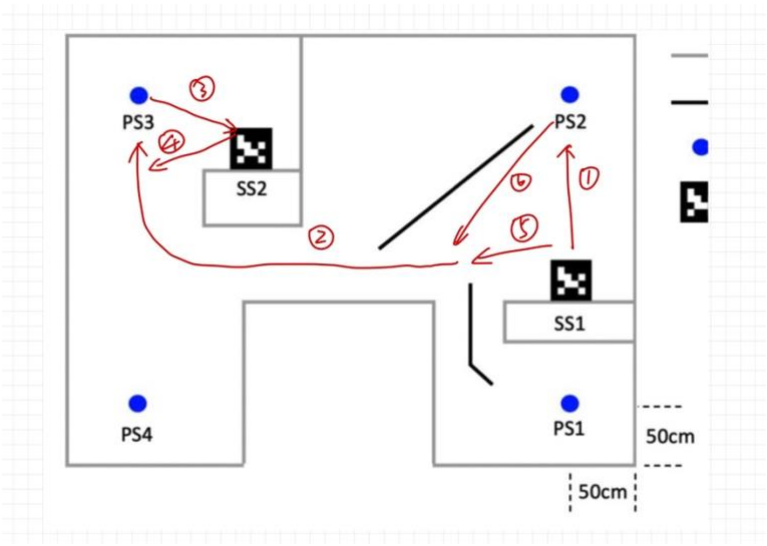
```
def send_goal(self, locate):
    self.goal = MoveBaseGoal()
    self.goal.target_pose.pose = self.locations[locate]
    self.goal.target_pose.header.frame_id = 'map'
    self.goal.target_pose.header.stamp = rospy.Time.now()
    self.move_base.send_goal(self.goal) #send goal to move_base
```

Therefore, we can send the robot to certain position, and the path follows the navigation result. All we have to provide is each internal and final destination position. Using rostopic command to echo the resultant position of each needed point, we can collect all points. The following points are

necessary points for the whole path. Note that, if the points are more than enough, due to each stop at each and every points, more time will be wasted. However, if the points are not sufficient, chances are that the turtlebot might lost its way due to unability to located its position. The sequence of this internal destination can be set inside the code. The Point parameters describe the absolute position of turtlebot. The Quaternion parameters describe the orientation of the robot. For the seven parameters, only four of them will make a difference. More details of code is shown in the attached file.

```
self.locations = dict()
self.locations['one']   = Pose(Point(3.48478,  1.91937, 0.000), Quaternion(0.000, 0.000,1, 0.00))# check
self.locations['two']   = Pose(Point(0.4459,  2.6991, 0.000), Quaternion(0.000, 0.000, -1, 0.0536))# check
self.locations['three'] = Pose(Point(2.1629,  2.5802, 0.000), Quaternion(0.000, 0.000, -0.05, 0.98))# check
self.locations['four']  = Pose(Point(0.3174, -1.44786, 0.000), Quaternion(0.000, 0.000, 0.9999, 0.00001))# check
self.locations['five']  = Pose(Point(0.29322, -0.58135, 0.000), Quaternion(0.000, 0.000, 0.016, 0.99974))# check
self.locations['six']   = Pose(Point(4.3621, -1.4576, 0.000), Quaternion(0.000, 0.000, -0.3147, 0.949))# check
```

The design of route in round 2 is more complicated. We need to go over all the points and find the most efficient points. According to the map shown below, in the requirement file, we need to switch sides for every two times. We can remark the map and put route in to segment as followed. There



are 6 segments included. The reason why PS4 is excluded in the whole path is that PS4 takes too much time to reach over SS2 and PS4. If we want to achieve 6 goals we need to go over the following order: PS1-SS1-PS2-PS3-SS2-PS2-SS1. However, if we need to achieve more points, the following path demand a long path of segment 2 which will take more than 20 to get PS3. And no matter how we manage the final route, the segment 2 will always be included for 3 times. As mentioned before, the AR tag recognition is delayless and we can recognize the AR tag with a conservative distance 1.0m distance. So, if we use the following path: PS1-SS1-PS2-PS3-SS2-SS1-PS3-SS2, the path shown above can enable us to achieve 70 points. This way we will decline the time we stop at the SS1 at the second time because the AR tag recognition demands little time.

Final result:

The final result is shown in the video attached as followed.