COMS4045A Report: SurveillanceBot

Motilal, Suraksha 2108903
Mahlangu, Fezile 2089676
Raphiri, Mmasehume 2089198

12 June 2022

1 Installations and Running

Due to ROS working with Python2, we had to install older versions of the packages we needed that were compatible with Python2.

The following installation is required to run our code:

```
pip install scipy==0.16
```

In a new tab run "./startWorld". Then cd to the scripts folder in the motion package and change the permissions of all the python scripts by running "chmod +x *py". cd back to "robotics_assignment_ws" workspace and run "source devel/setup.bash". Then run "rosrun motion control.py".

Input the desired goal x and y coordinates.If path is not found, then the goal coordinates entered are an obstacle. Feasible goal coordinates to try are : (3.685,6.65) , (-0.754 , 10.302) or navigating back to (0,0).

2 Gmapping and Occupancy map

To generate the occupancy map we used gmapping and kobuki keyboard operations (see read me for code lines) to move through the world. We then edited the map that we created using OpenCV, skimage and NumPy packages (refer to the file called **imgEditing.ipynb**) by converting the image to binary, performing opening to clean the image and erosion to increase the sizes of the obstacles (padding):

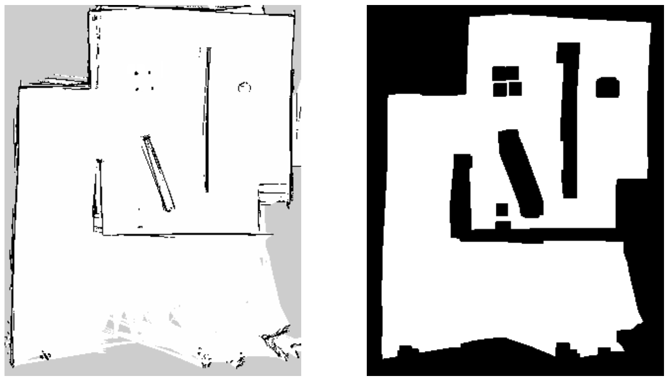

Figure 1: Map generated using gmapping and kobuki before and after image processing

We could not use some of the packages in Python2 (namely skimage), thus the editing was done outside of ROS. For us to work with the map in ROS, we then exported the image as a text file called **mapArray.txt** with 0's to indicate obstacles and 1's to indicate free spaces.

3 Path planning and PID

We used the Probabilistic Road Map with Dijkstra as our path planning algorithm.

3.1 Pre-processing of data for PID

The map array text file was loaded in and used to determine the coordinates of the obstacles in pixel coordinates. As the array coordinates were incompatible with the actual world coordinates, a conversion was done to change pixel coordinates to world coordinates. The start and goal coordinates were taken in as world coordinates and converted to pixel coordinates so that the PRM algorithm could work with them in the array. Once PRM returned a path, that path was then changed to world coordinates so that PID could be used to navigate to those locations.

3.2 Probabilistic Road Map

Our PRM algorithm takes in $start_x$, $start_y$, $goal_x$, $goal_y$, $obstacles_x$, $obstacles_y$ and $robotRadius$. It then randomly samples 500 points. Using these random samples, it generates a map that checks if the samples collide with any obstacles. To check for collision, it uses KDTree on the obstacles (nearest neighbour from scipy) and checks if the distance is less than or equal to the robot size/radius. If so then that point is said to be a collision. After generating the map it performs Dijkstra to get the shortest path. PRM then returns the coordinates of the path from *start* to *goal*. We then perform PID on said coordinates to navigate the turtle bot to the goal.

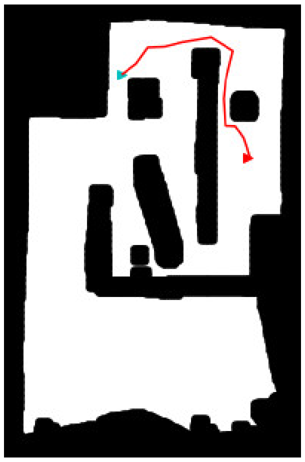

Figure 2: Map with path generated by PRM, from turtlebot's inital position to goal at (3.685, 6.7)

3.3 PID

While the turtlebot navigates through each point in the path, the linear velocity is controlled by a PID controller which tries to minimize the distance between the turtlebot's current position and the goal's position. Similarly for the angular velocity, the PID tries to minimize the discrepancy between the goal's target angle and the present state angle while ensuring that the velocity is not greater than the turtlebot's maximum velocity and not less than its minimum velocity. This makes the turtlebot to be steered at the right direction at each timestep.

4 Visual scan of the pixels to determine Utility Cart

The Utility cart is added directly in gazebo from the insert tab. To determine whether or not the Utility Cart was detected we enabled PointCloud and Camera, Image in rviz. In the robot.rviz file we changed the topic for image and camera to "/camera/rgb/image_raw". We then subscribed to the topic "/camera/rgb/image_raw" to get the image of what the turtle sees through its camera.

In two separate tabs we ran "roslaunch turtlebot_bringup 3dsensor.launch" to enable the camera and "roslaunch turtlebot_rviz_launchers view_robot.launch" to launch rviz, respectively.

A colour picker was used to get the utility cart bgr color which was then converted to hsv. Thereafter, we got the minimum and maximum thresholds of the colours. A cv2.inRange function was used to obtain the colours within the threshold that were present in the hsv image (i.e a mask).

Finally, we checked the mask for any non-zero values which meant that the image contained the shade of green of the utility cart. If the mask contained all zero then there is no cart. We then published the results to "witsdetector" topic.
For this to work, after running control.py we ran colorcontrol.py and in a separate tab we ran "rostopic echo witsdetector".