SC627 assignment-1
Bug-1 Implementation in Gazebo

V ADARSH (21307R001)

1 Introduction

In this assignment, the simulation of Bug-1 algorithm was implemented in ROS+gazebo using turtlebot.
Bug-1 is a motion planning algorithm that helps a robot navigate in an environment filled with obstacles so that it can reach the "goal" location from a starting point. The location of goal is always known to the robot and it only has local knowledge of it's environment, hence it needs some kind of sensor to detect obstacles when it comes close to one. In this implementaion, turtlebot's 360^o LIDAR was used.

2 Implementaion Details

The basic tasks to be carried out for bug-1 until goal is reached are:

1. Move towards goal in a staright line.
2. If an obstacle is encountered, circumnavigate the obstacle boundary fully and go to the location on the boundary that is closest to goal and repeat step(1).

In this implementation, the algorithm node in ROS subscribed to laser and odometry topic. These are needed because to detect obstacles and follow boundary distance data of obstacles is needed and to find direction of goal and distance from it, we need "pose" of robot at each instant. These information are provided by the above mentioned topics.
Before reaching goal the robot is doing either of the following tasks:

1. Move towards goal in straight line. (MOVE_TO_GOAL)
2. Circumnavigate the obstacle (CIRCUM)
3. Move to closest point on boundary (LEAVE_POINT)

Now, a further explanation is provided on the above 3 tasks:

2.1 MOVE_TO_GOAL

Here the main job is to make the robot head towards goal in a straight line. Using the odom topic we can get the (x,y) position and yaw of robot at each instant. Since the goal coordinates are known already, we can use robot's location to find the direction of goal with respect to current location. Also robot's current heading is known (yaw). Using this, a simple "PROPORTIONAL" controller was implemented, where at each instant robot compares it's heading with desired heading. If it is within a desired threshold it moves straight, else it stops to adjust it's heading and then move straight.

2.2 CIRCUM

The robot has a laser sensor mounted on it that rotates in 360^o and gives 360 readings each instant. The readings were obtained by subscribing to laser topic. The readings in front and right side of robot were used. While robot is doing the MOVE_TO_GOAL task, if it comes close to an object, we switch to CIRCUM task.
Now the robot has to follow the obstacle boundary to fully circumnavigate it. Robot has to do 2 main things : follow_wall AND find_corner.

2.2.1 follow_wall

Here the robot follows the boundary by trying to maintain a fixed distance "ob_dist" from itself and the obstacle wall. To employ this, the readings from "RIGHT" side laser is taken to measure robot's distance from wall, so the robot follows the wall with it's right side aligned to wall.
A simple PD controller was used for the robot to maintain the desired distance. Further a error band was assumed (ob_dist_error), to decrease oscillations.

- error= (ob_dist)-(right sensor reading)
- if $\|error\| \geq$ ob_dist_error, an appropriate angular velocity along z-axis was given using PD controller.
- else, move straight

2.2.2 find_corner

Further we need to detect corners. Corners can be of 2 types: (1) The boundary wall turns left in which case robot has to also turn left (2) The boundary wall turns right and robot also has to turn right. In case of (1) we can detect this using sensor readings in front of robot. If they are below a certain threshold it means the wall ahead is turning left. In such a case robot is stopped and a fixed angular velocity is given that makes the robot rotate left until the front sensor readings are more than a certain value. This ensures the way ahead is clear and there us no upcoming turn ahead.

In case of (2) to detect it we can use the readings in the "front-right" region of robot. If such a corner approaches, those readings will be increasing than a certain threshold, and we can stop the robot and make it turn right with a fixed angular velocity.

2.3 LEAVE_POINT

When the robot comes close to an object in MOVE_TO_GOAL and switches to CIRCUM, that location is stored as "hit_point". Now as the robot is circumnavigating, if it comes close to "hit_point", i.e within a distance of "step_size", then it means it has fully circumnavigated. While circumnavigating it stores it's (x,y) positions in an array. After finishinh cirumnavigation, it finds the point in that array closest to goal "leave_point" and then switches to CIRCUM to go towards it. After reaching that point it switches to MOVE_TO_GOAL.

3 Pseudocode

Algorithm 1 BUG-1

```
1: while ||loc − goal|| ≥ step_size do
2:     if close to obstacle then
3:         CIRCUM
4:         if close to hit_point then
5:             LEAVE_POINT
6:             if close to leave_point then
7:                 MOVE_TO_GOAL
8:             else
9:                 MOVE_TO_GOAL
10:
```

4 Simulation results

The below figure shows the plot of odometer data of turtlebot when a simulation was run from a start point of (1.5,2.5) to a goal point (1.5,-2).

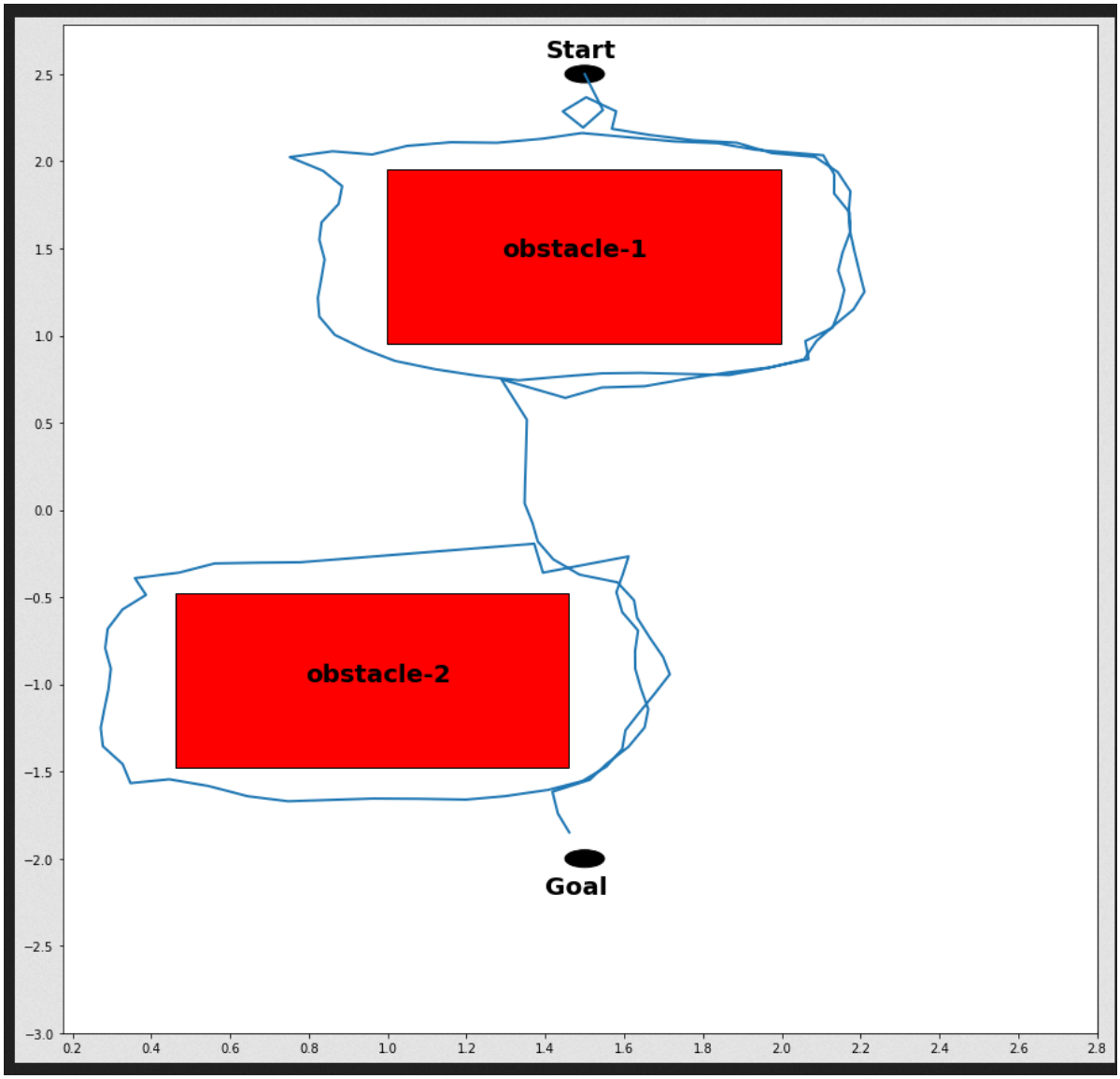
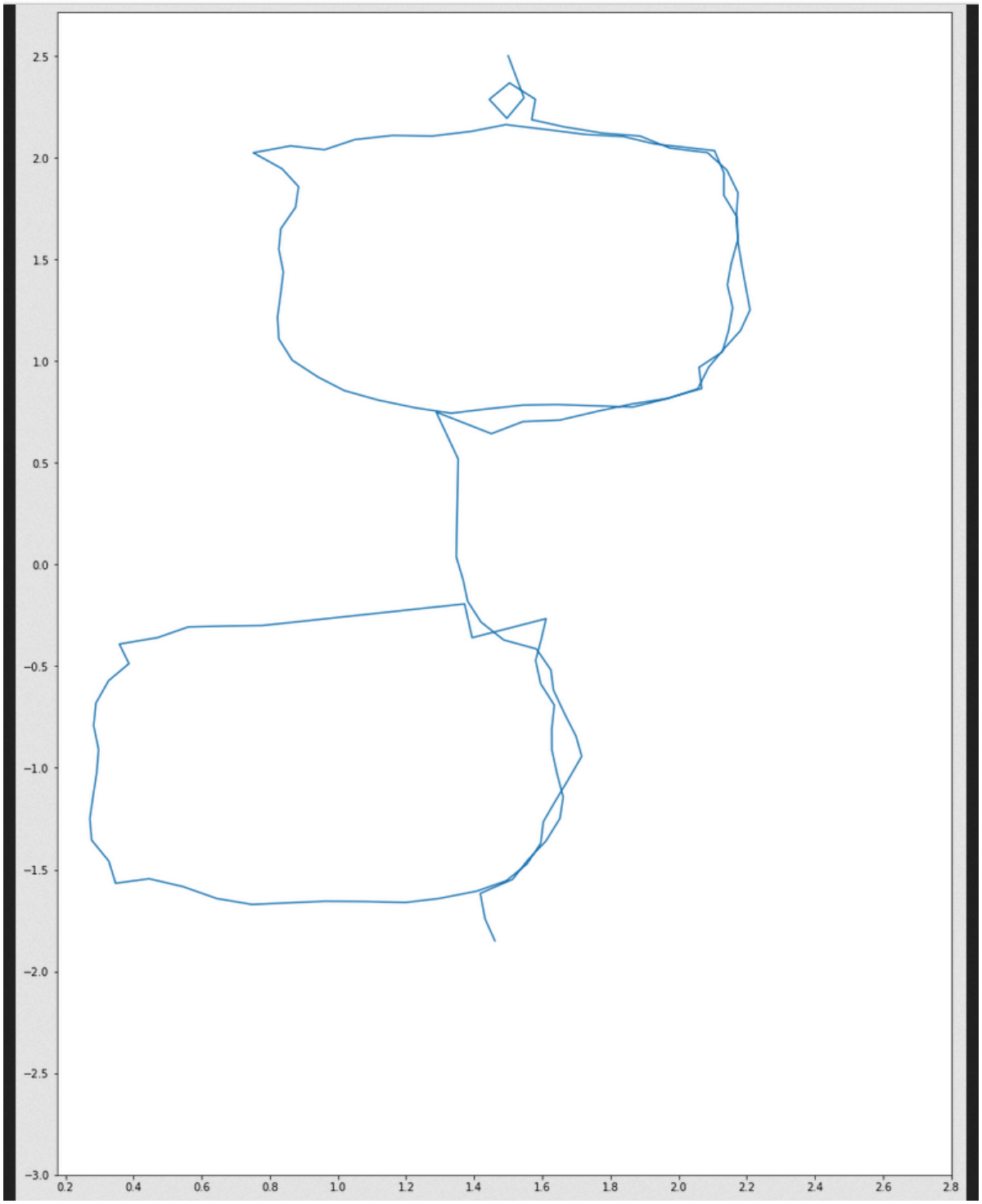

Start
obstacle-1
obstacle-2
Goal
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
2.6
2.8

Figure 1:


2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
2.6
2.8

The following images show snapshots from simulation where robot navigates through 2 obstacles in a gazebo environment to reach goal.

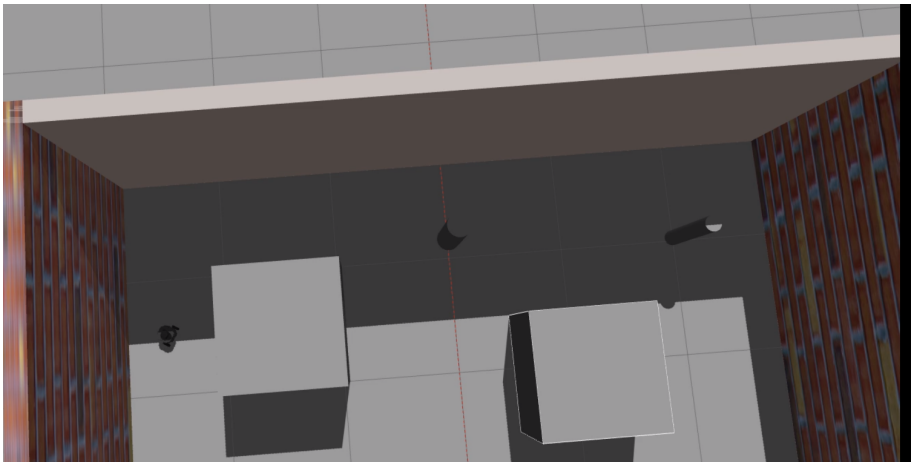


Figure 2:

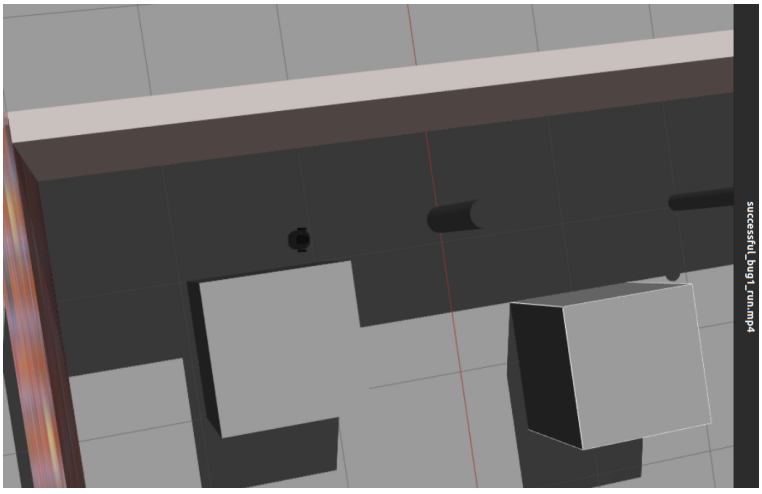


Figure 3:

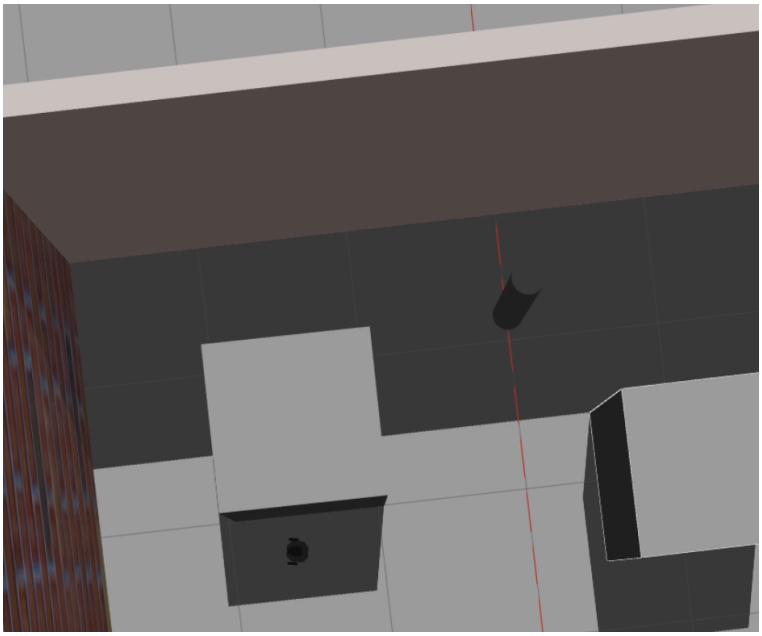


Figure 4:

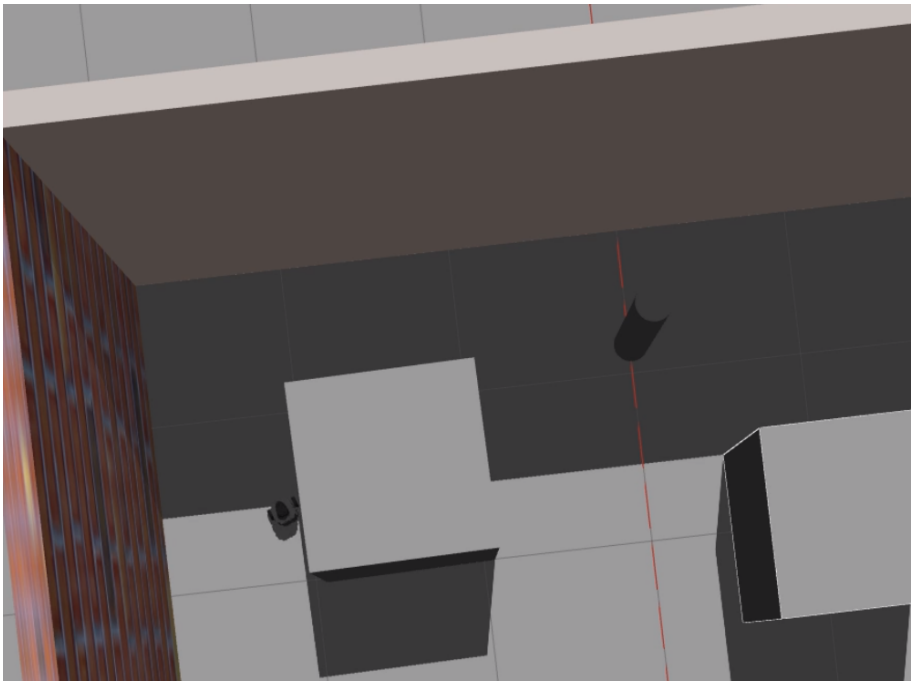


Figure 5:

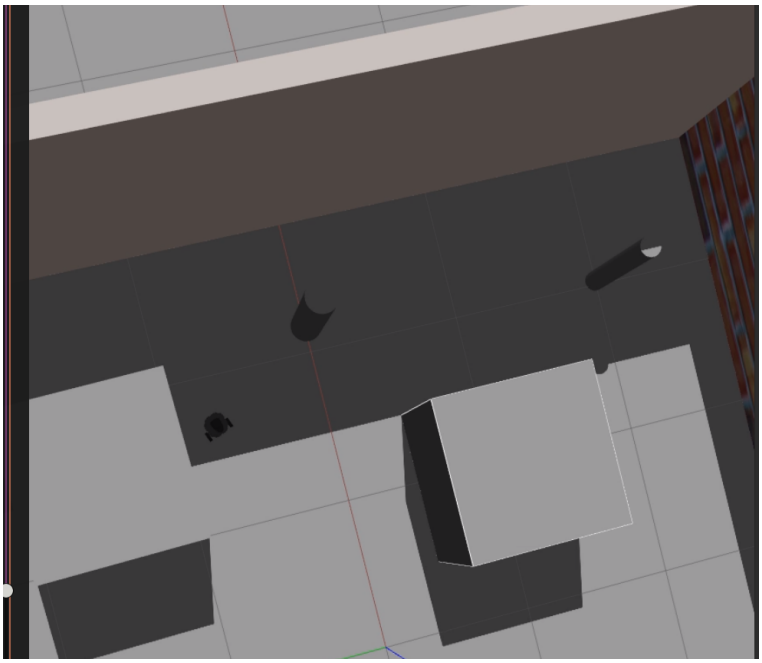


Figure 6:

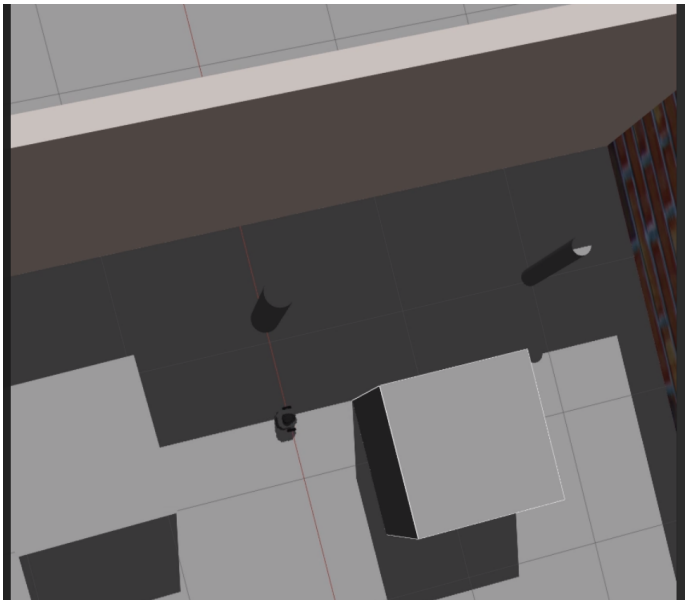


Figure 7:

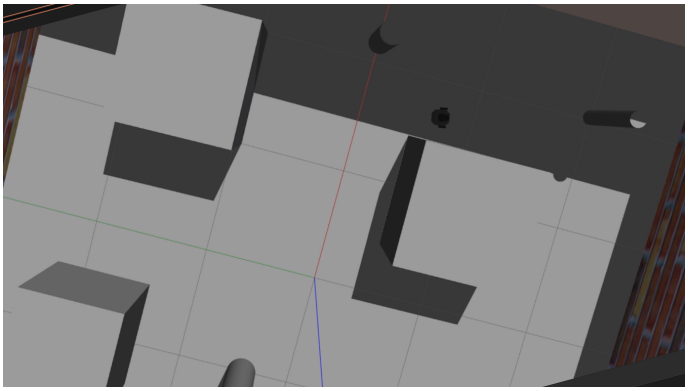


Figure 8:

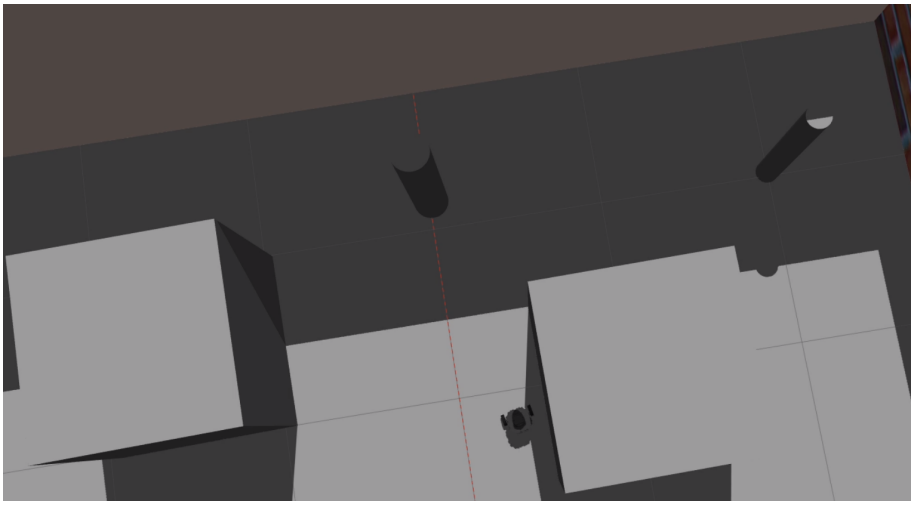


Figure 9:

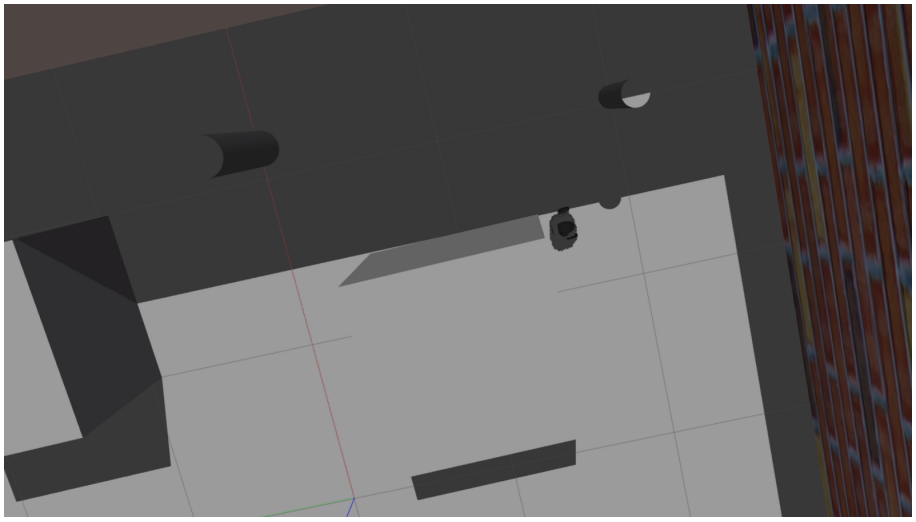


Figure 10:

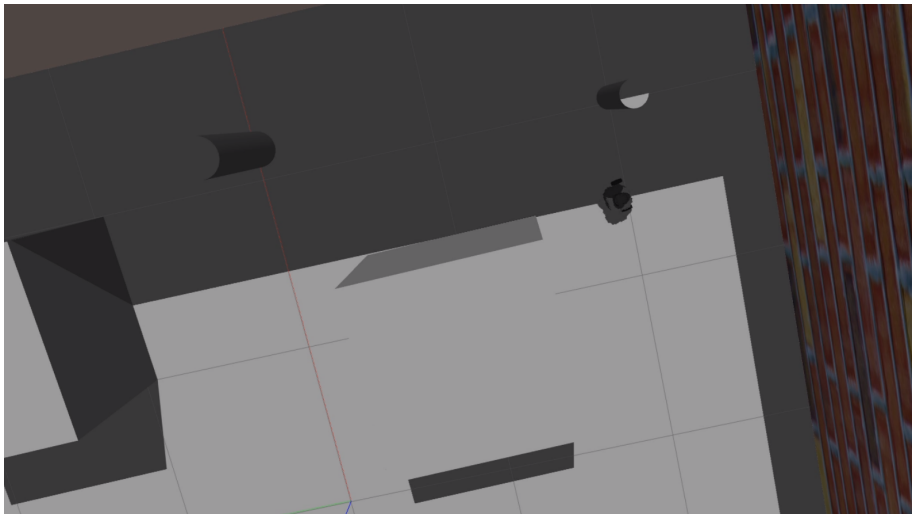


Figure 11: