Robotics Project Report

EE50237 Robotic Software

University of Bath

Department of Machine Learning and Autonomous Systems

Student ID: 219540793

17 December 2021

Contents

1 Introduction

This report aims to design and implement a simulation of husarion ROSbot 2.0 trying to get into specific destination while avoiding several obstacles. ROSbot is an autonomous robot platform that is based on ROS. The simulation used for the testing in this case is using the Gazebo model environment. Running in an operating system of Ubuntu.

2 Problem Analysis

In this section, The task that the ROSbot should be able to do and the design approach for the ROSbot to be able to accomplish the task will be discussed.

2.1 The Task

In a Gazebo simulation, the ROSbot should be able to navigate an obstacle course going from one point to another. The ROSbot should be able to navigate itself from start to finish, meaning that the path should be decided by the ROSbot itself.

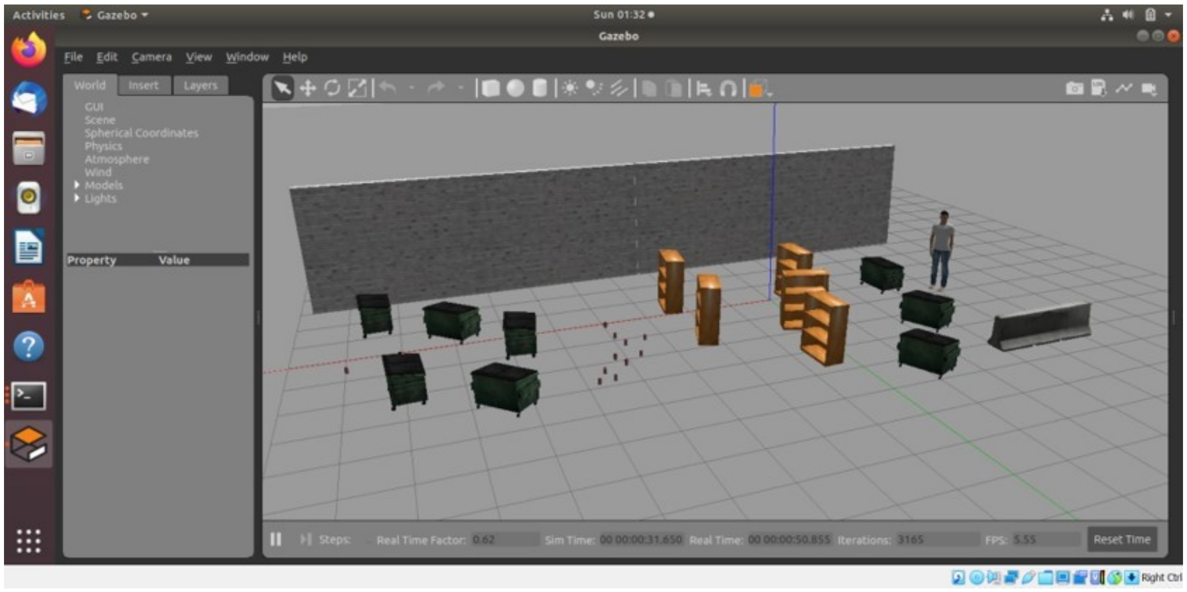


Figure 1: Gazebo simulation map (2021)

The ROSbot will start from the coke can, at the left-end of the map, and try to navigate itself by avoiding obstacles in-order to reach the person on the right-end of the map. The obstacles that the ROSbot need to avoid are rubbish bins, coke cans, book shelves, wall, and a barrier (as could be seen from figure 1).

2.2 Design Approach

2.2.1 Localization

The technique of establishing where a mobile robot is in relation to its surroundings is known as robot localization. Localization is one of the most basic skills required of an autonomous robot since knowing where the robot is is a prerequisite for making judgments about future actions. A map of the environment is accessible in a typical robot localization situation, and the robot is outfitted with sensors that view the surroundings as well as track its own movements. Using the information received from these sensors, the localization challenge becomes one of estimating the robot's location and orientation inside the map [1]. Determining the pose of a robot in the environment is Pose = position + orientation. In order to use the position and orientation of the robot, odometry and laser data will be used.

The measurement of change in location over time using data from moving sensors is known as odometry. Odometry is used by certain robots, whether they are legged or wheeled, to determine their location in relation to a starting point. This technique is prone to errors due to the integration of velocity readings across time to obtain location estimations. Odometry cannot be used successfully in most situations without rapid and exact data collection, device calibration, and processing [3].

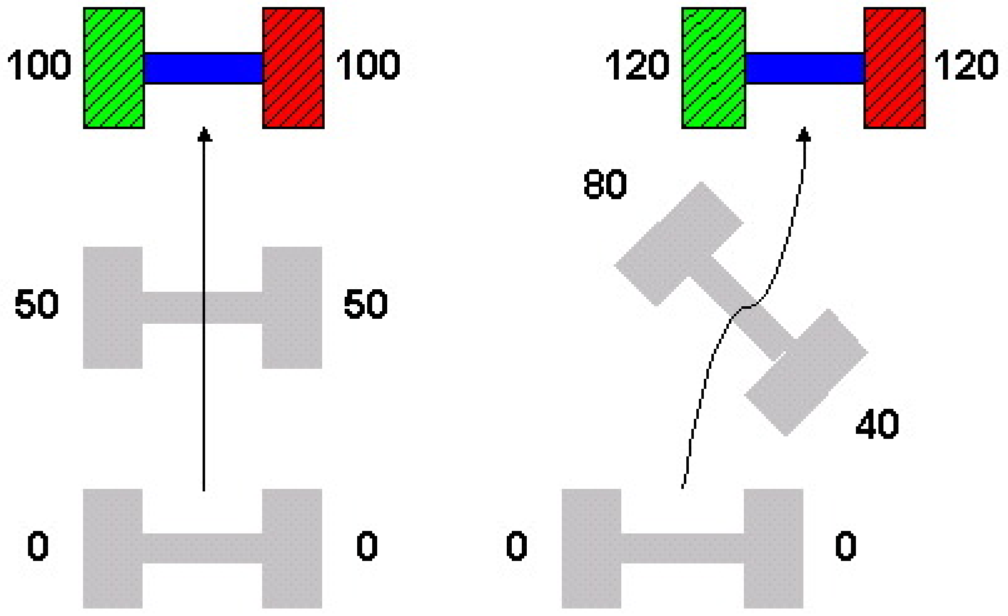


Figure 2: two different robot paths can have different values

In the diagram above, two possible motion patterns for a differential drive robot are presented. The distance values are shown next to the wheels. Because the robot in the left diagram moves in a straight line, the distance values are always the same. On the other hand, the robot on the right takes a serpentine path. At half-speed, the left wheel distance values are 80, while the right wheel distance values showing 40, suggesting that the robot was turning right. The distance value shows the same value at the end position as they did on the left, but they're higher since the robot took a longer route by sketching an s-shaped curve. If your control program just looked at the distance at the completion point, it would think the robot followed a straight path. As a result of the delayed distance processing, errors may arise. Analysing the odometry signals should be done as soon as possible unless the robot's travel in a straight path is physically secured (as certain locomotion systems are) [2].

2.2.2 Moving towards The Destination

For ROSbot to be able to move towards the destination wanted, the direction the ROSbot need to move forward to will be needed. In order to find the direction that the ROSbot should go, finding the correct orientation of the ROSbot is crucial. By using odometry, it is posible to get the orientation position of x, y, w, and z of the ROSbot. By using this information we can transform this data in to Euler angles.

3 Software Architecture

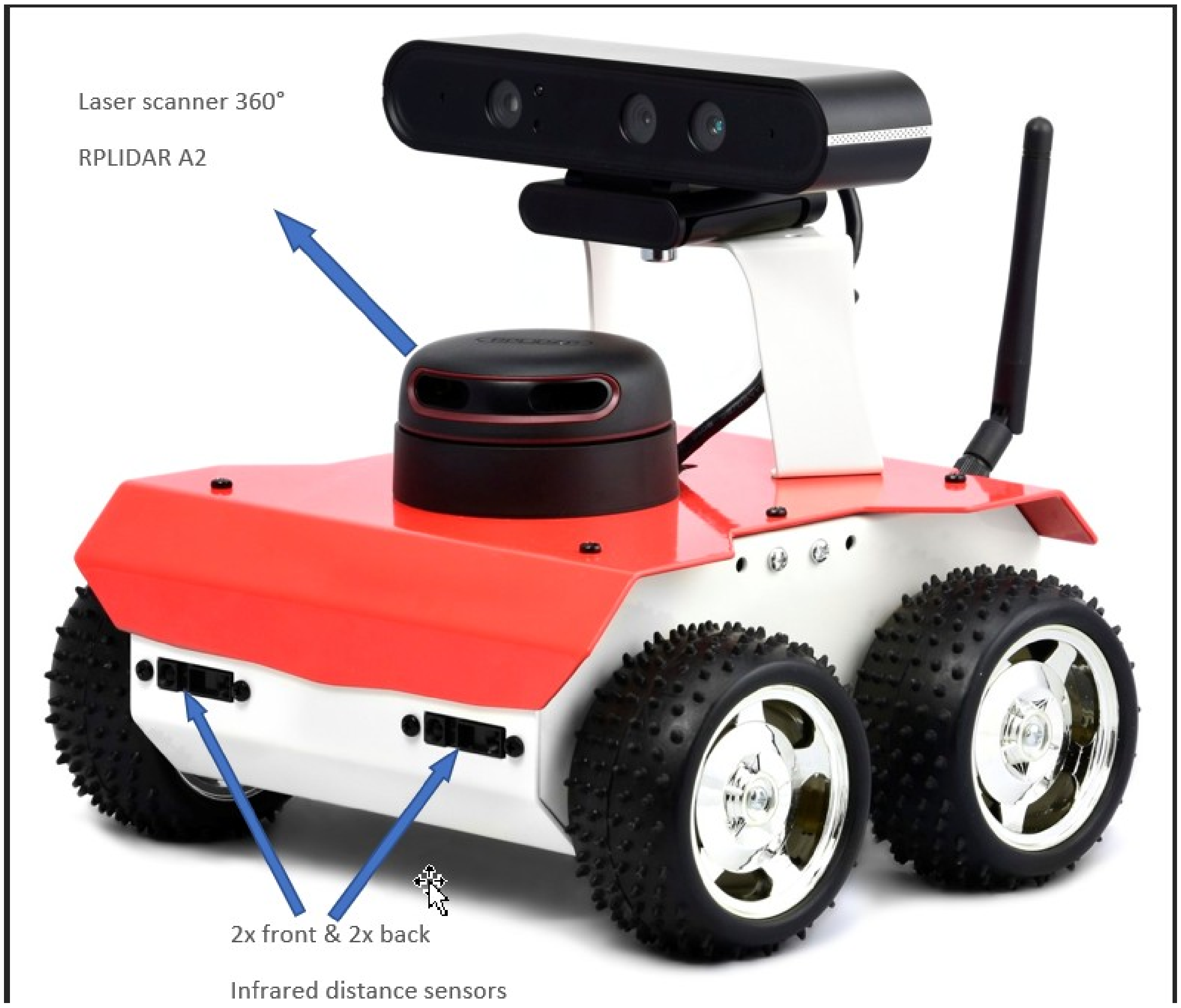


Figure 3: ROSbot 2.0

As could be seen on figure 3, the robot we are using for the simulation is husarion ROSbot 2.0. It consists of 4-wheeled motor, A2 RPLIDAR Laser scanner, and 4 infrared distance sensors. For this project, the sensors that are used is the RPlidar laser scanner, and the two infrared distance sensors that are located in the front.

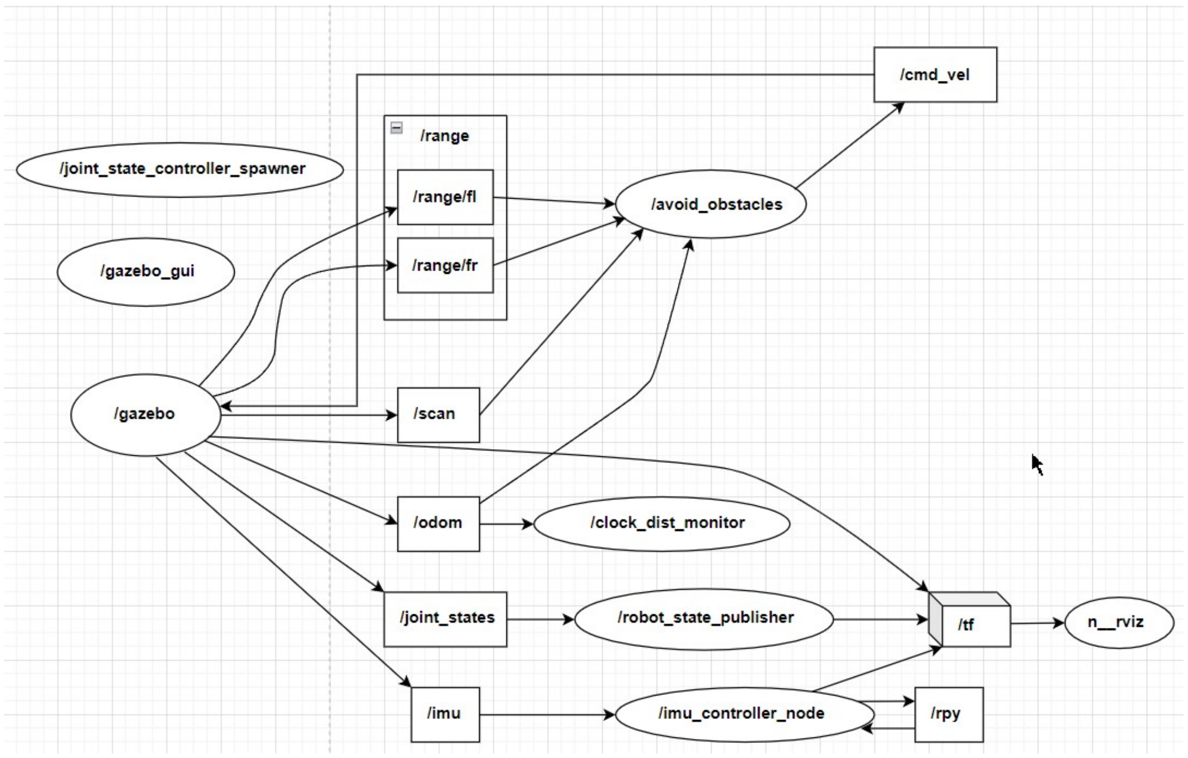


Figure 4: ROS graph

/avoid_obstacles is a node that takes into account of the */range*, */scan*, and */odom* nodes in order to decide what to do before publishing it to */cmd_vel*. */range* is the topic for the infrared distance of the front right(*/fr*) and front left(*/fl*) of the ROSbot. This topic can gives you the max. range, min. range, and the range from you to the object. This infrared sensors are good for scanning short obstacles, for example in our simulation are the coke cans and the bookshelves. If it scanned an obstacle infront of the ROSbot at certain range, */avoid_obstacles* will publish the */cmd_vel* topic to go left.

/scan itself is the A2 RPLIDAR, a 360° laser scanner for scanning the surroundings of the ROSbot. This sensors will be used to determine whether there is an obstacle in certain range, if it does, it will try to avoid it by going left. if not, it will keeps on going to the direction of the destination (which is the person).

/odom is a topic for estimating its position and orientation relative to a starting location given in terms of an x and y position and an orientation around the z axis. This topic is to help */avoid_obstacles* node to determine the direction of the ROSbot orientation, whether it is pointing to a correct direction or not. In this project, it will check if the ROSbot is pointing to the correct direction, if not, it will change the direction. By converting Quaternion into Euler and getting the theta(θ), whether the orientation is pointing to the correct direction could be determined. Moreover, This node is also used to set the Point() or goal of the destination of the ROSbot. So, by knowing the starting point, the current position and orientation, and the destination point of the ROSbot, it is able to navigate the ROSbot so that it can reach to the destination point that is set.

/odom topic is also helpful for getting information about the simulation time and distance travelled by the ROSbot. As mentioned before, */odom* topic is able to give the starting and current point of the ROSbot, and by knowing the starting point and current position it is in, the distance travelled could be determine by using this equation: $\sqrt{(X2-X1)^2+(Y2-Y1)^2}$.

From */avoid_obstacles* publishing to */cmd_vel* topic, for updating the ROSbot linear velocity of x and y, and the angular velocity of z, to the */gazebo* simulation.

4 Software Implementation

4.1 Packages needed

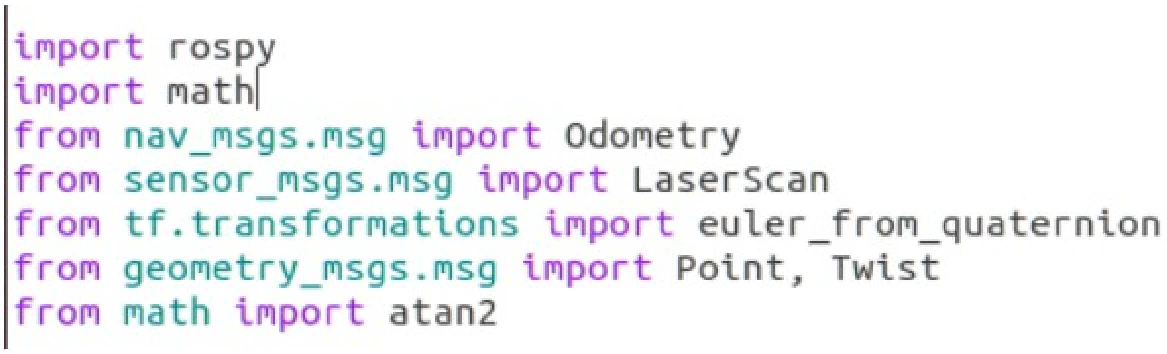

```
import rospy
import math
from nav_msgs.msg import Odometry
from sensor_msgs.msg import LaserScan
from tf.transformations import euler_from_quaternion
from geometry_msgs.msg import Point, Twist
from math import atan2
```

Figure 5: Imported Packages

Packages needed for this project are as follows:

- Odometry for getting information on estimating its position and orientation relative to a starting location given in terms of an x and y position and an orientation around the z axis, also for calculating distance travelled and getting the simulation time.
- LaserScan is for working with the Lidar.
- euler_from_quaternion is for getting the theta, to work with the directions of the ROSbot's orientation.
- Point is to set the destination for the ROSbot to reach.
- Twist is for updating the linear velocity of x and y, and angular velocity of z.

4.2 Navigate to The Destination

Initializing the variables for x, y, and theta. Position of x and y is basically the current positions of the ROSbot in the map. To get the theta, first need to get the orientation of the ROSbot. From the orientation of x, y, z, and w of the ROSbot, called Quaternion, could be transform into Euler (roll, pitch, and theta). Only theta is needed to be used for finding the correct directions and the orientation of the ROSbot.

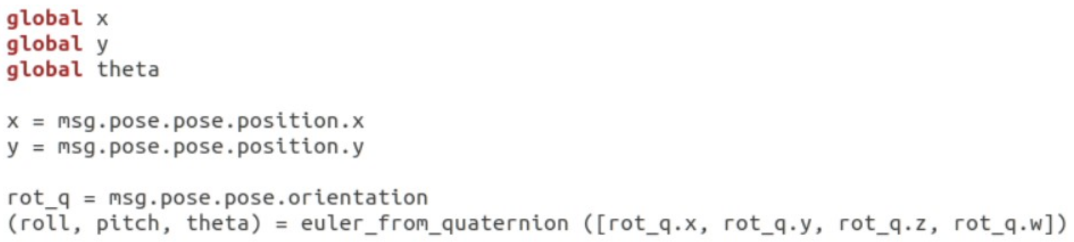
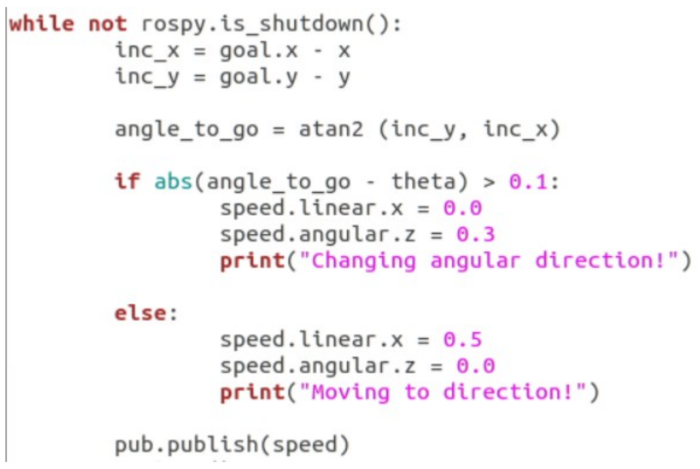

```
global x
global y
global theta

x = msg.pose.pose.position.x
y = msg.pose.pose.position.y

rot_q = msg.pose.pose.orientation
(roll, pitch, theta) = euler_from_quaternion ([rot_q.x, rot_q.y, rot_q.z, rot_q.w])
```

Figure 6: Initializing the variables.

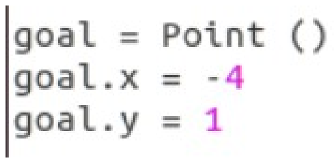
Checking if the ROSbot orientation is correct. First we need to find the arc tangent of increment y/increment x = angle_to_go. If the ROSbot orientation of angle_to_go - theta is more than 0.1, it needs to change the orientation of the ROSbot by angular of z for = 0.3. if it is in the correct direction, it will move with linear velocity of x = 0.5.



```
while not rospy.is_shutdown():
        inc_x = goal.x - x
        inc_y = goal.y - y

        angle_to_go = atan2 (inc_y, inc_x)

        if abs(angle_to_go - theta) > 0.1:
                speed.linear.x = 0.0
                speed.angular.z = 0.3
                print("Changing angular direction!")

        else:
                speed.linear.x = 0.5
                speed.angular.z = 0.0
                print("Moving to direction!")

        pub.publish(speed)
```

Figure 7: Navigate to the destination being set.

Setting the Point() of x = -4 and y = 1. This is the position of the person standing from the starting point of the ROSbot.



```
goal = Point ()
goal.x = -4
goal.y = 1
```

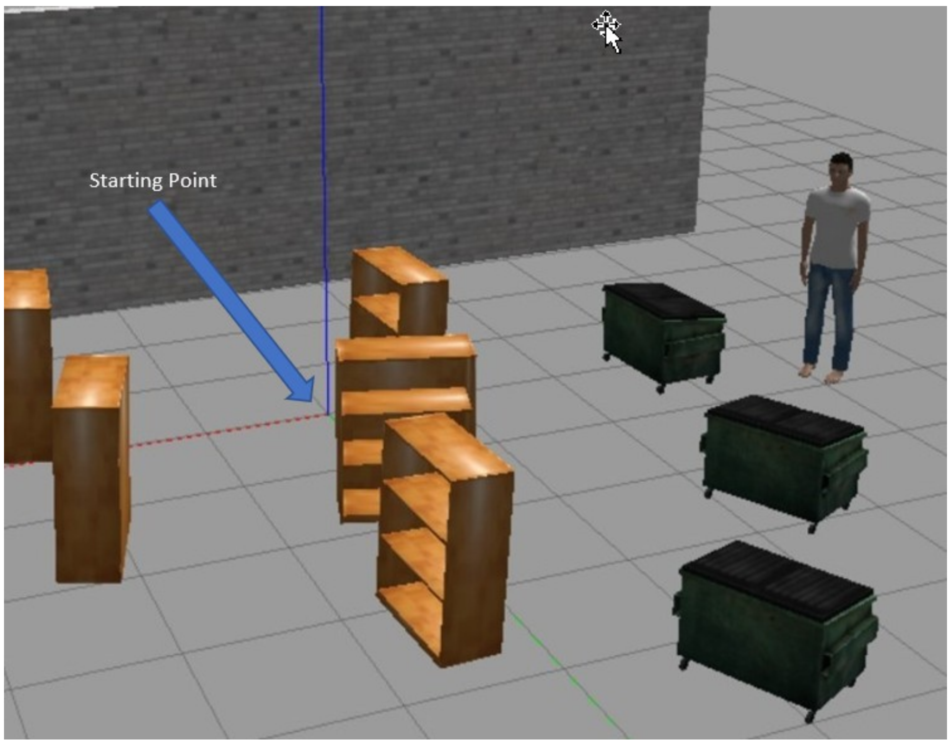
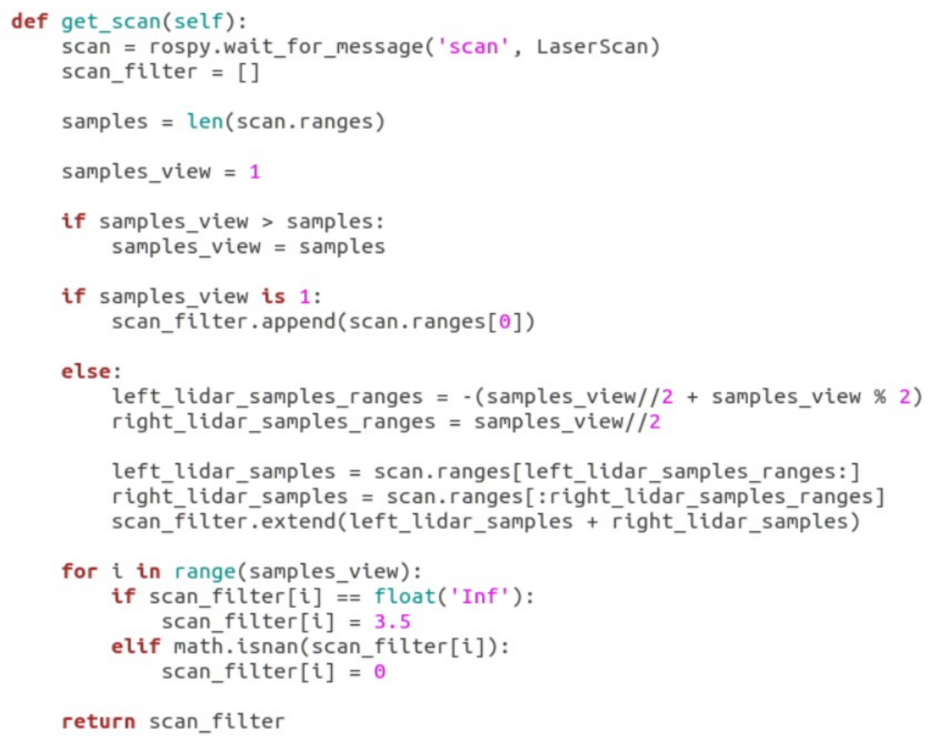


Figure 8: Setting the Point(), the destination or goal from the starting point.

4.3 Avoiding Obstacles

For avoiding obstacles, lidar sensor will be needed. However, dealing with the data that the lidar scanner returns is not easy. The lidar samples will be returning a lot of data samples. The samples are handled being dividing them into two regions, so array regions of the left of the lidar and the right of the lidar.

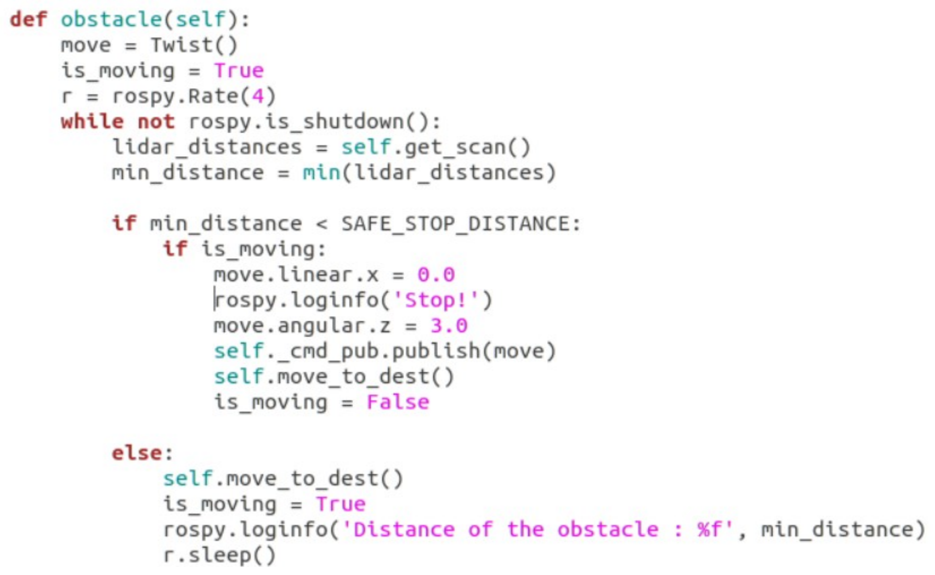


```
def get_scan(self):
    scan = rospy.wait_for_message('scan', LaserScan)
    scan_filter = []

    samples = len(scan.ranges)

    samples_view = 1

    if samples_view > samples:
        samples_view = samples

    if samples_view is 1:
        scan_filter.append(scan.ranges[0])

    else:
        left_lidar_samples_ranges = -(samples_view//2 + samples_view % 2)
        right_lidar_samples_ranges = samples_view//2

        left_lidar_samples = scan.ranges[left_lidar_samples_ranges:]
        right_lidar_samples = scan.ranges[:right_lidar_samples_ranges]
        scan_filter.extend(left_lidar_samples + right_lidar_samples)

    for i in range(samples_view):
        if scan_filter[i] == float('Inf'):
            scan_filter[i] = 3.5
        elif math.isnan(scan_filter[i]):
            scan_filter[i] = 0

    return scan_filter
```

Figure 9: Handling the lidar scanned samples

After handling the samples from the lidar, then we could decide when the ROSbot needs to avoid obstacles or keeps going forward. If the minimal distance from the ROSbot to the obstacle is smaller than the safe_stop_distance (meaning that it is close to the obstacle), it will update the cmd_vel to stop the ROSbot from going forward and change the angular of z for 3.0, which is going left. After avoiding the obstacle, then the move_to_dest() functions will be called, which is basically the navigation for reaching to the destinations wanted functions.

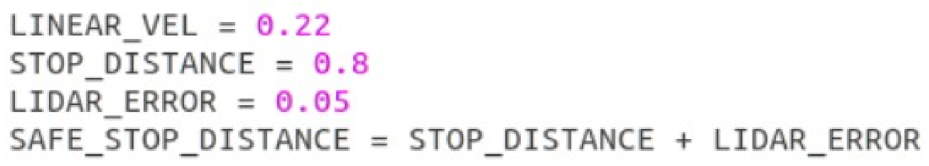
If the obstacle range is still larger than the safe_stop_distance (meaning that it is still far away from the obstacle), it will call move_to_dest() functions, which is the navigation to reach the destination.



```python
def obstacle(self):
    move = Twist()
    is_moving = True
    r = rospy.Rate(4)
    while not rospy.is_shutdown():
        lidar_distances = self.get_scan()
        min_distance = min(lidar_distances)

        if min_distance < SAFE_STOP_DISTANCE:
            if is_moving:
                move.linear.x = 0.0
                rospy.loginfo('Stop!')
                move.angular.z = 3.0
                self._cmd_pub.publish(move)
                self.move_to_dest()
                is_moving = False

        else:
            self.move_to_dest()
            is_moving = True
            rospy.loginfo('Distance of the obstacle : %f', min_distance)
            r.sleep()
```

Figure 10: Making decisions based on the lidar ranges

The settings for the safe_stop_distance in this simulation is stop_distance = 0.8 + the lidar_error = 0.05. the lidar_error is for tolerating the uncertainty of scanned data samples from the lidar sensor.



```python
LINEAR_VEL = 0.22
STOP_DISTANCE = 0.8
LIDAR_ERROR = 0.05
SAFE_STOP_DISTANCE = STOP_DISTANCE + LIDAR_ERROR
```

Figure 11: Settings for the safe_stop_distance.

5 Testing Approach

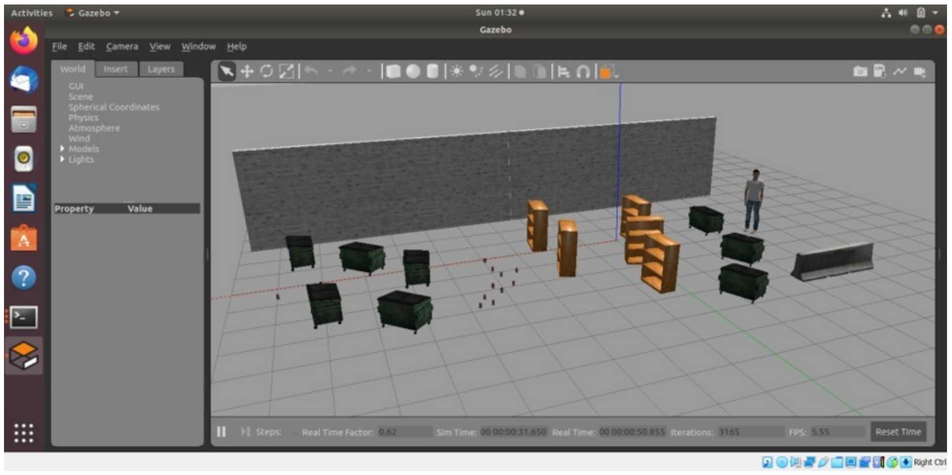


Figure 12: 2021_assessment.world

2021_assessment.world map is used for this testing on a Gazebo simulation. The distance travelled and the simulation time will be taken when trying the method mentioned above. The node should be able to make the ROSbot navigate from the coke can to the person, while avoiding obstacles by turning left. The method used in this testing assumes that the obstacles parameters are unknown and it should be able to be use on any map.

The ROSbot spawn is on the starting point position by default. However for this testing, the ROSbot spawn parameters are configured, so that the ROSbot will spawn right in front of the coke can (where it should start the navigation following the task given for this project).

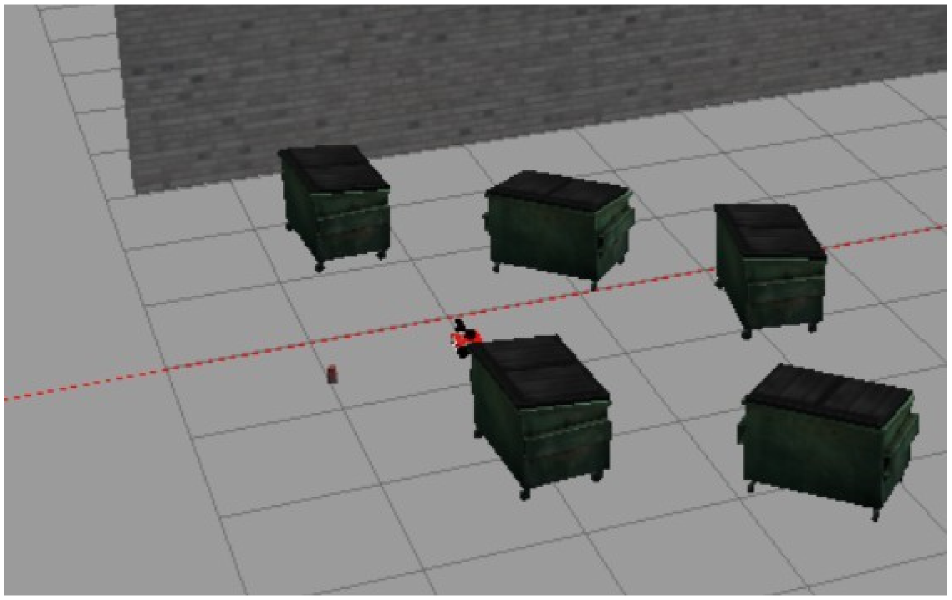


Figure 13: Spawning the ROSbot in front of the coke can.

The parameters could be configured in the ROSbot launch file. The parameters used in order for it to spawn in front of the coke can are as followed:

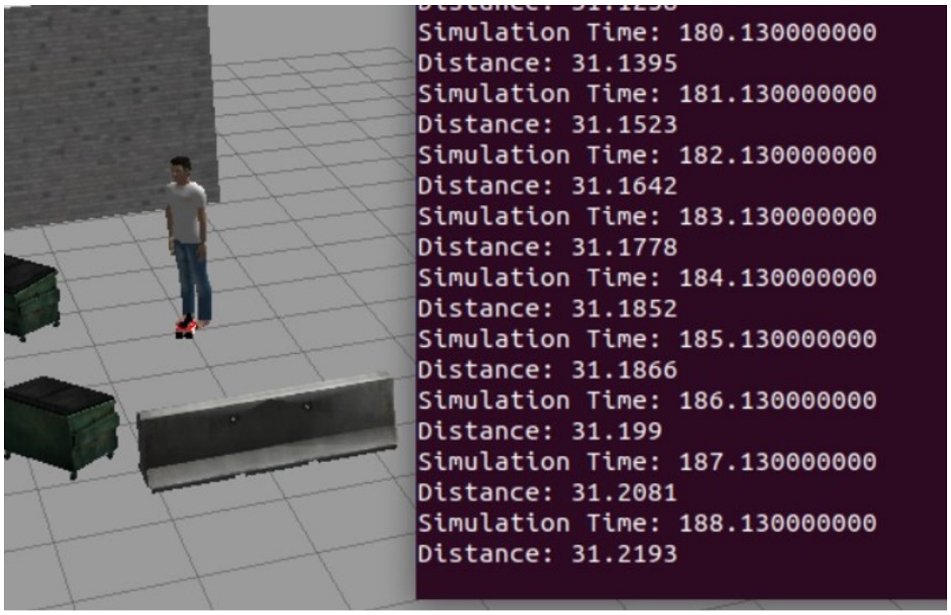
```
<include file="$(find rosbot_bath)/launch/spawn_robot.launch">
            <arg name="x" value="8.0"/>
     <arg name="y" value="0.5"/>
     <arg name="yaw" value="3.1416"/>
</include>
```

Settings the value of:

- x with the value of 8.0, spawning it 8 blocks to the left from the starting point of x.
- y with the value of 0.5, spawning it half-block below the starting point of y.
- yaw with the value of 3.1416, spawning it to face the correct direction.

6 Results and Discussions

The results considered for testing this approach are the distance travelled, the simulation time, and obstacles hit. The distance travelled and the simulation time results will be calculated by a node called */clock_dist_monitor*, while obstacles hit is counted manually by watching the simulation running. As could be seen from below figure, the simulation time takes 188.130 seconds, 31.2193 meters for the distance travelled, and as many as 1 obstacle collusion happens.



```
Simulation Time: 180.130000000
Distance: 31.1395
Simulation Time: 181.130000000
Distance: 31.1523
Simulation Time: 182.130000000
Distance: 31.1642
Simulation Time: 183.130000000
Distance: 31.1778
Simulation Time: 184.130000000
Distance: 31.1852
Simulation Time: 185.130000000
Distance: 31.1866
Simulation Time: 186.130000000
Distance: 31.199
Simulation Time: 187.130000000
Distance: 31.2081
Simulation Time: 188.130000000
Distance: 31.2193
```

Figure 14: The results of the simulation

The below figure demonstrates the path that the ROSbot has taken navigating itself from the coke can to the person.

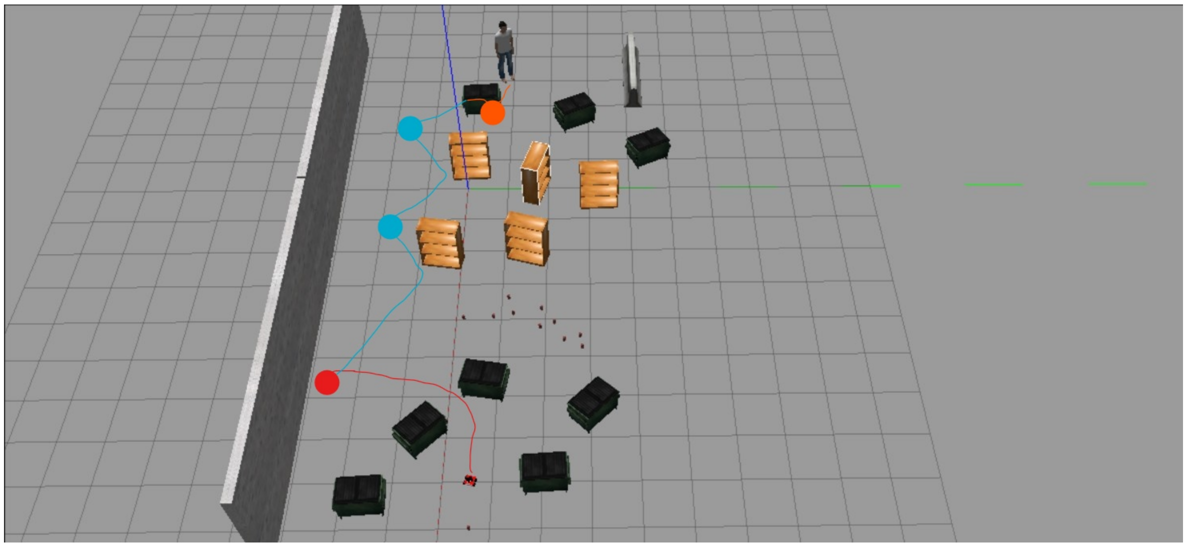


Figure 15: The path the ROSbot has taken (the big dot demonstrate the ROSbot changing the orientation for finding the correct direction).

From the above figure, it could be seen that when avoiding the first rubbish bin obstacle (shown by red line) it does not run the move_to_dest() function quite well because it run straight to the wall. Just before hitting the wall then it changes the direction slowly facing to the person. The big dot on the figure is to demonstrate the ROSbot changing it's orientation to face the correct direction. Avoiding both of the book shelves actually run pretty well (shown by the blue line), until it is trying to avoid the last rubbish bin obstacle that it somehow do not detect the obstacle and going forward colluding with one of the rubbish bin wheel (the collusion shown with orange line). After colluding with the rubbish bin wheel, it somehow going out of the course from reaching to the destination a bit before finally changing it's orientation to face towards the direction of the person and finally move forward reaching the destination that is set.

7 Conclusions

The methods of localization and moving towards the destination are used in this project. By using the odometry and the sensors, it is able to navigate and avoid obstacles quite well. Both the odometry and the sensors play the most important role for completing the task. Recognising the current position and orientation is vital in order to navigate in the correct direction. Moreover, handling the samples data given by the lidar sensors are really crucial on determining whether the ROSbot will be able to avoid the obstacles or not. Eventhough, there are some unexpected outcome from testing it in the simulation. It is still able to complete the task that is given.

8 Further Work

Modifying the node is recommended for further progression. The most recommended modification to be implemented is handling the lidar sensor samples data. Instead of just dividing them into two regions, it will be better to divide them into three regions (front, front right, front left), so that avoiding the obstacles is not by only turning left, but could be configured the best path whether to avoid it by going left or right depending on the position of the ROSbot facing the obstacles. Achieving this recommended method could result in shorter distance travelled.

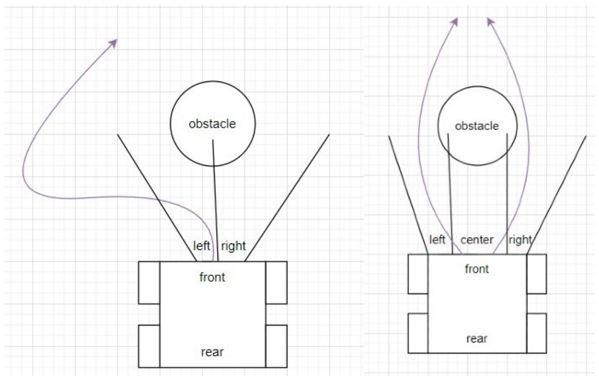


Figure 16: Current ROSbot avoiding obstacles (on the left side) and better ROSbot avoiding obstacles (on the right side).

Changing the orientation of the ROSbot direction on the current method is not efficient, the current method does not take into account where the ROSbot is facing, it will always rotate anti-clockwise until it is in the correct orientation. To make it more efficient, we can find whether rotating it clockwise or anti-clockwise is faster to meet the correct direction for the orientation of the ROSbot. This way, the simulation time can be significantly improve.

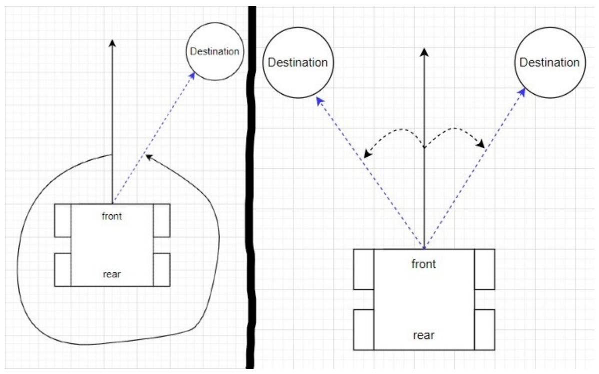


Figure 17: Current changing orientation method (on the left side) and better changing orientation method (on the right side).

References

[1] Shoudong Huang and Gamini Dissanayake. A critique of current developments in simultaneous localization and mapping. *International Journal of Advanced Robotic Systems*, 13(5):1729881416669482, 2016.

[2] Keith Kotay. Robo-rats. cs54 spring 2001. dartmouth college computer science department robot building course. https://groups.csail.mit.edu/drl/courses/cs54-2001s/odometry.html, 2001.

[3] Safdar Zaman, Wolfgang Slany, and Gerald Steinbauer. Ros-based mapping, localization and autonomous navigation using a pioneer 3-dx robot and their relevant issues. In *2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC)*, pages 1–5. IEEE, 2011.