Simulation of ROS bot to avoid obstacle

Gokulan Nithianandam

EE50237 Robotics Software

Department of Electrical and Electronics Engineering

Table of Contents

1-Introduction

1.1 Aim

The aim of the project is for the ROS bot to navigate to a destination while avoiding obstacles along the path.

1.2 Objective

The objective of the project is as follows

- To implement the obstacle avoidance algorithm in the ROS architecture.
- To use only the sensors that come with the ROS bot.
- Determine the total travel distance and time for the ROS bot to reach the destination.

2-Design approach

2.1 Action selection

As Bryson (2007) states the process by which an agent chooses what to do next at any instance is called Action selection. The agent for this project is the Ros bot. The representation of the action selection is shown in figure 1.

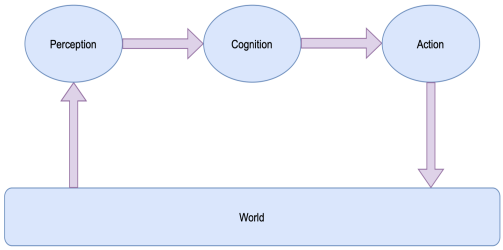

Figure-1 Action selection representation. Image adapts from Wortham (2021).

For the ROS bot to do the action selection, the Ros bot needs to sense the world using the sensor model and analyse the sensor data to get contextual information for the instance. Furthermore, employ the contextual information to determine the particular action at a particular moment through the reactive planner and perform an action. The representation of the ROS bot's action selection is shown in figure 2. The cognition state is called Reactive planner since the approach for the action selection is Dynamic/ Reactive planning. The reactive planning approach proves to be effective in a dynamic world and can deal with uncertainty. We use Hierarchical state machine design to implement reactive planning.

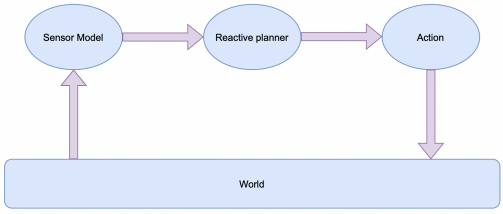

Figure-2 Representation of ROS bot action selection

2.2 Hierarchical State machine Design

The initial hierarchical state machine design has three states as shown in figure 3. The front, left, right region data comes from the Lidar sensor. The interpretation of the sensor data explains in section 2.4. The initial design is effective in avoiding obstacles through the Lidar data but ineffective in maintaining the initial angular position and it results in deviating from the destination point as shown in figure 4. The hierarchical state machine designs adapt from Wortham (2021).

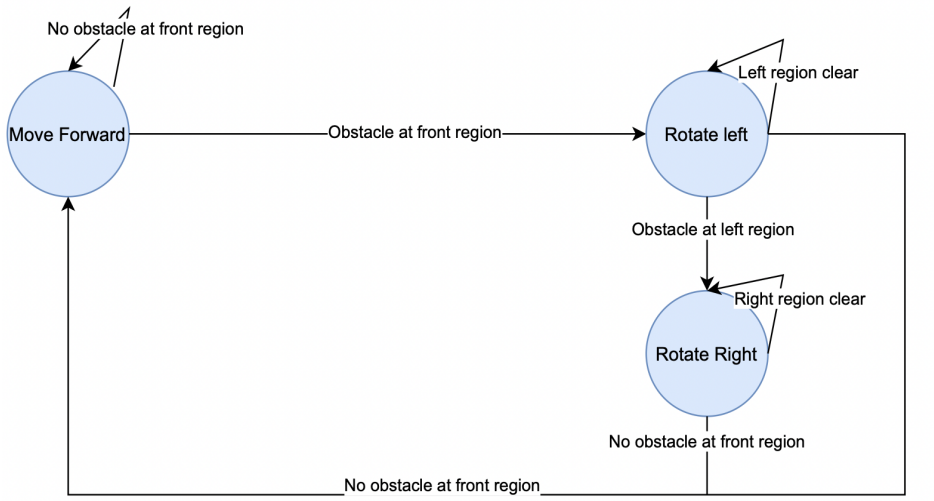

Figure 3- Hierarchical state machine design with three states

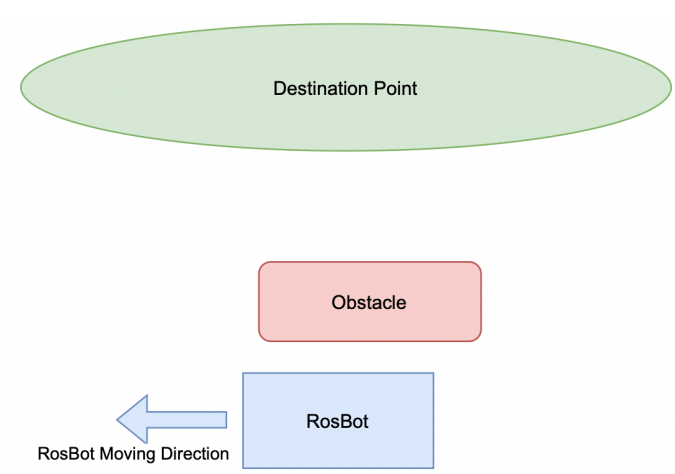

Figure 4- ROS bot moving direction after avoiding the obstacle ahead.

To correct the angular position of the ROS bot, we add another state as shown in figure 5. The new state helps to get the ROS bot orientation straight to reach the destination point as shown in figure 6. The ROS bot orientation data comes from the IMU unit and the interpretation of the IMU data is in section 2.4.

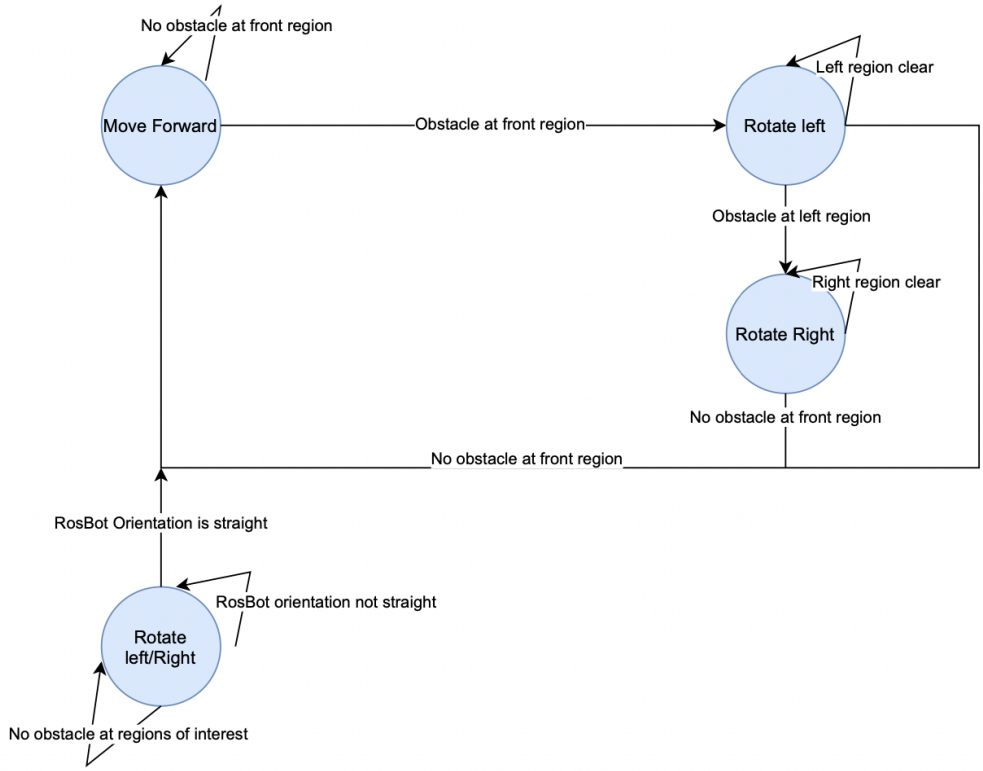

Figure 5- Hierarchical state machine design with four states

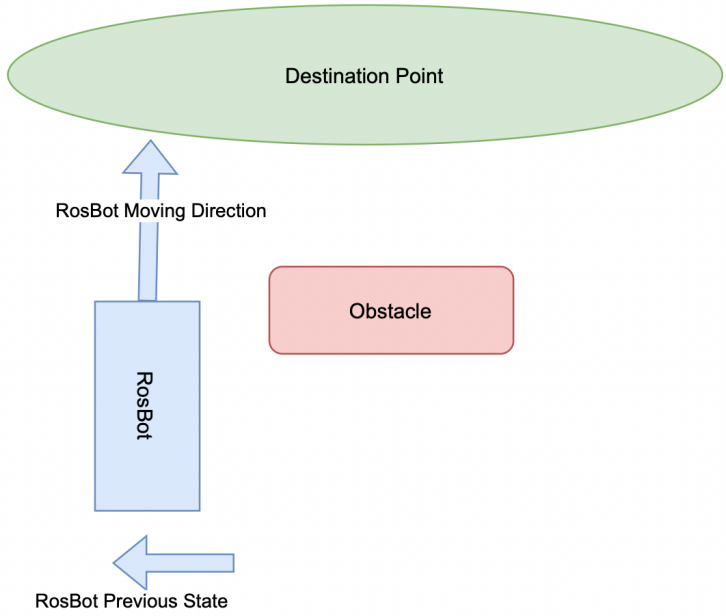

Figure 6- ROS bot in straight orientation after avoiding the obstacle in the previous state.

As shown in figure 8, all the region of interest has an obstacle and the ROS bot cannot move in any direction with the current hierarchical state machine design. For the above situation, the ROS bot employs a random state. The random state allows the ROS bot to move backwards for x amount of time. The Hierarchical stare machine design with the random state is shown in figure 7.

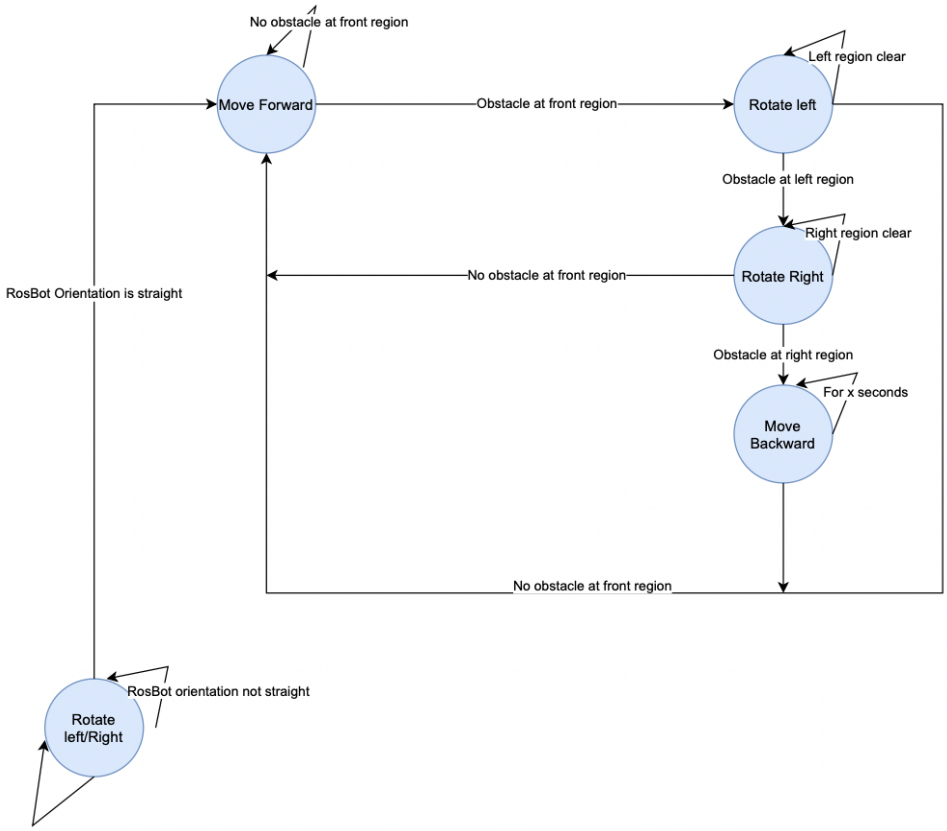

Figure 7- Hierarchical state machine design with five states

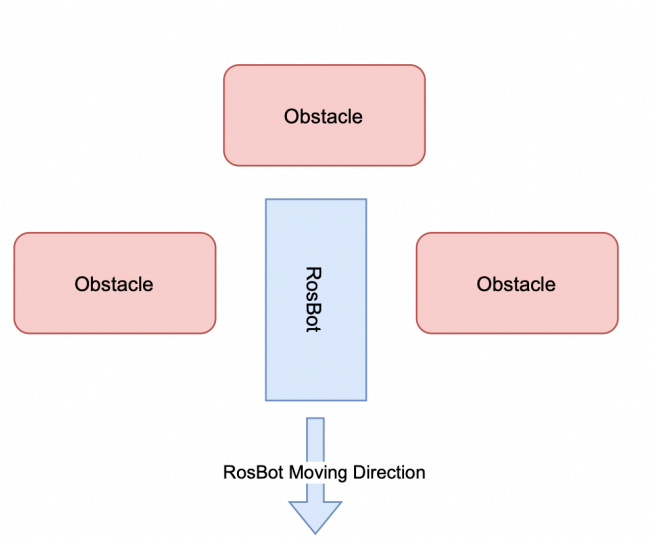

Figure 8- ROS bot between three obstacles.

2.3 Hierarchy of behaviours

The final design of the Hierarchical state machine has five states. The five states contribute to three behaviours such as Avoid obstacles, Orientation correction and Move forward. In figure 9, states highlight in red contributes to Avoid obstacles, green contributes to Orientation correction and blue contributes to Move forward behaviour. The behaviour has a hierarchy as shown in figure 10, to choose the right action selection in relation to the current sensory inputs. The lowest behaviour has the most priority and the highest behaviour has the least priority.

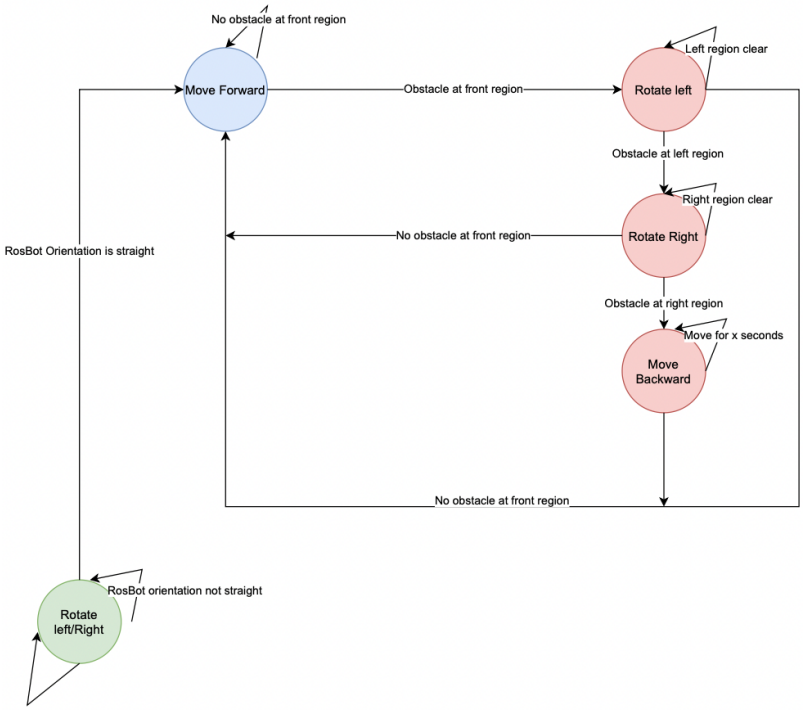

Figure 9- Hierarchical state machine design with five states representing behaviours

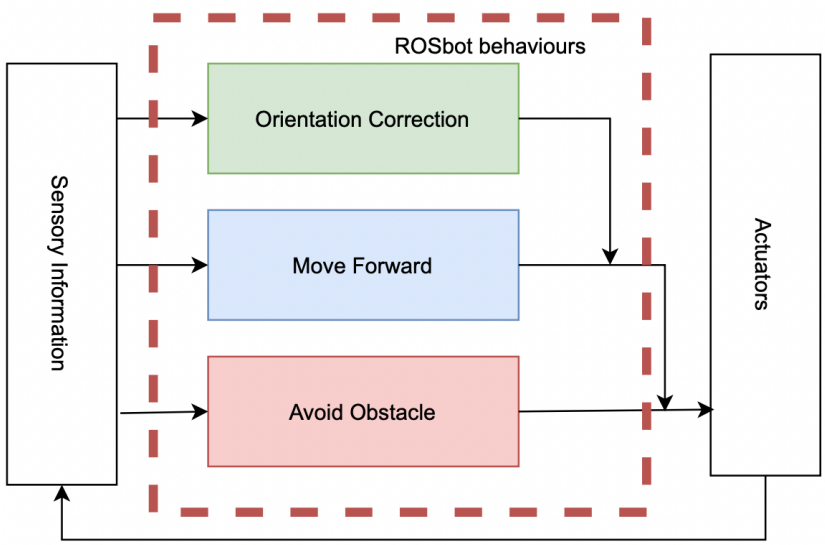

Figure 10- Representation of ROS bot behaviour hierarchy. Image adapts from Wortham (2021).

2.4 Sensor Selection

To detect obstacles, we use Lidar and IR sensors. The Lidar sensor act as a primary sensor for obstacle detection because of the measurement range advantage over the IR sensors. The range advantage allows the reactive planner to plan the ROS bot path ahead of the obstacle. The measurement range of lidar is 360 degrees. Furthermore, the lidar provides a measurement for every 0.5-degree interval. The maximum and minimum distance measurement is 0.2m and 12m respectively. For this project, we utilise the front region of the lidar in the form of an arc and the arc region is split into 5 regions as shown in figure 11. The interpretation of Lidar in regions allows a better understanding rather than interpreting the Lidar data in array indexes.

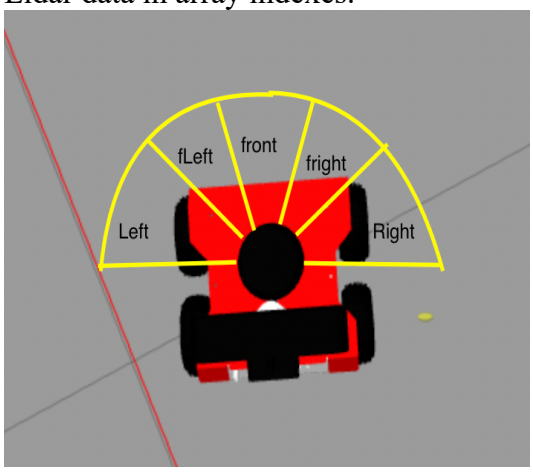

Figure 11- Representation of Lidar front region section.

The ROS bot has four IR sensors as shown in figure 12. For this project, we use two IR sensors at the front such as front left and front right since the ROS bot has the highest priority to move forward and backward motion is done on the basis of time. The front two IR sensors act as a redundant sensor because the IR sensors detect obstruction lower to the ground which the lidar sensor is unable to capture due to the lidar position as shown in figure 13. The maximum and minimum range of the IR sensor is 0.01m to 0.9m respectively.

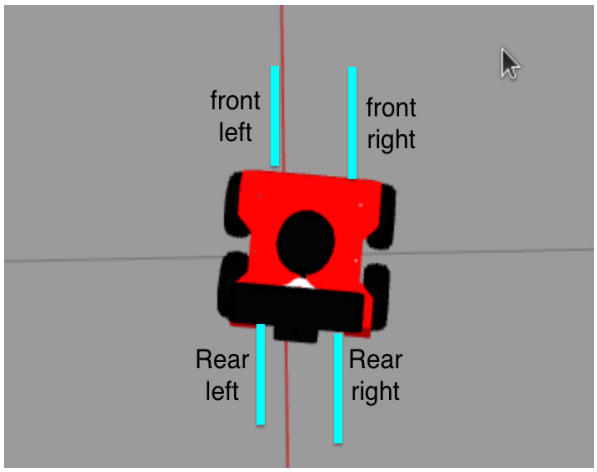

Figure 12- Representation of ROS bot IR sensors position.

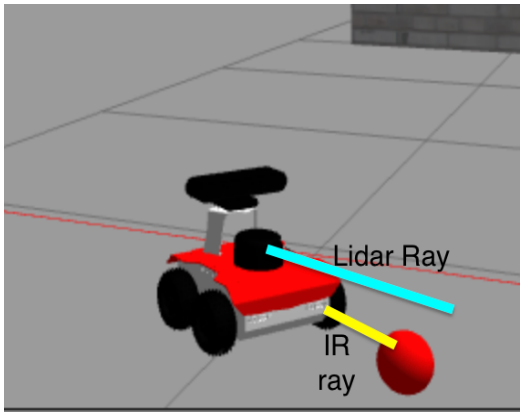

Figure 13- Representation of ROS bot IR ray detecting a ball while the lidar ray misses.

For the orientation correction behaviour, we use the IMU unit since it provides the angular position of the ROS bot. To calculate the distance travel, we employ the odometer sensor.

3-Software architecture

The software architecture of the ROS bot has three levels as shown in figure 14. The Task level outlines the plan to the Behaviour level and gets feedback on the plan progress from the Behaviour level. The Behaviour level sends commands to the Control level to exhibit behaviours such as Avoid obstacles, Move forward, Orientation correction and Log distance and Time. Furthermore, the Behaviour level requests contextual sensory information from the control level. The behaviour level has a new behaviour to log distance travel and time and it is not discussed in the hierarchical state machine design since it does not need a hierarchical state machine design due to its simplicity behaviour. The control level uses the ROS environment and two ROS nodes such as Obstacle_avoid and ROSbot_log. Obstacle_avoid node enables to implementation of the ROS bot behaviours such as Avoid obstacle, Move forward and Oientation correction. ROSbot_log node enables to implementation of the ROS bot behaviour such as Log distance travel and time.

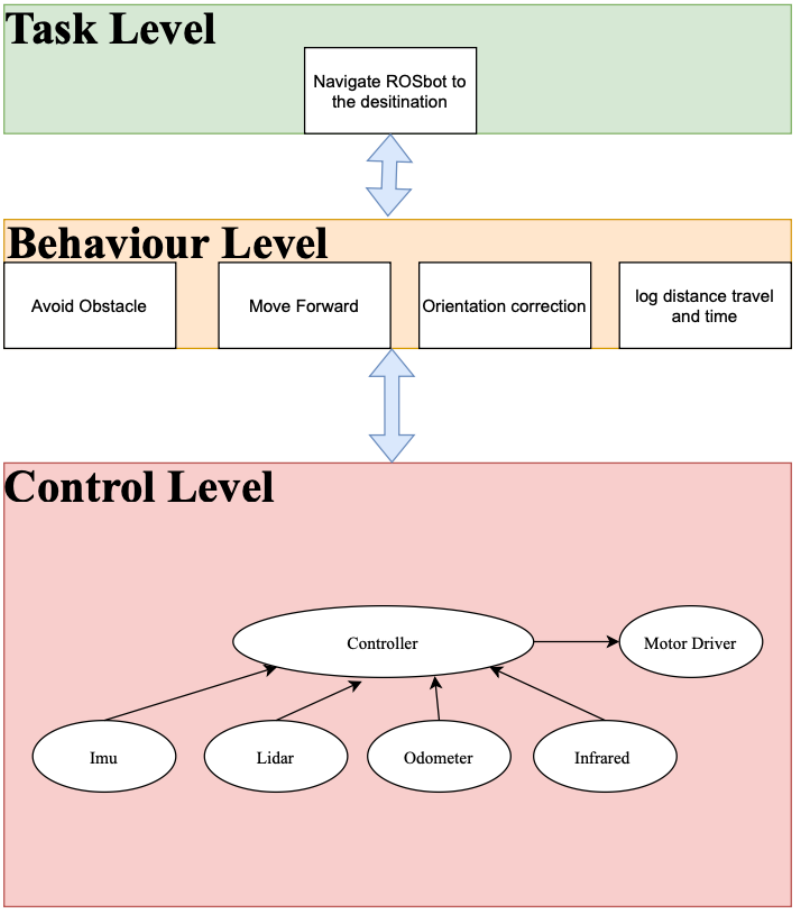

Figure 14- Representation of ROS bot software architecture. Image adapts from Meyer et al (2013).

3.1- Obstacle_avoid node

The software architecture for the ROS node Obstacle_avoid architecture is shown in figure 15, the node uses the python script “Obstacle_avoid.py”. The node subscribes to the topic Imu, scan, fl and fr to get the sensor feedback on ROS bot orientation, Lidar data, front left IR and front right IR. The node publishes to the topic cmd_vel to set the ROSbot speed command. The software architecture of obstacle avoid python script is shown in figure 16.

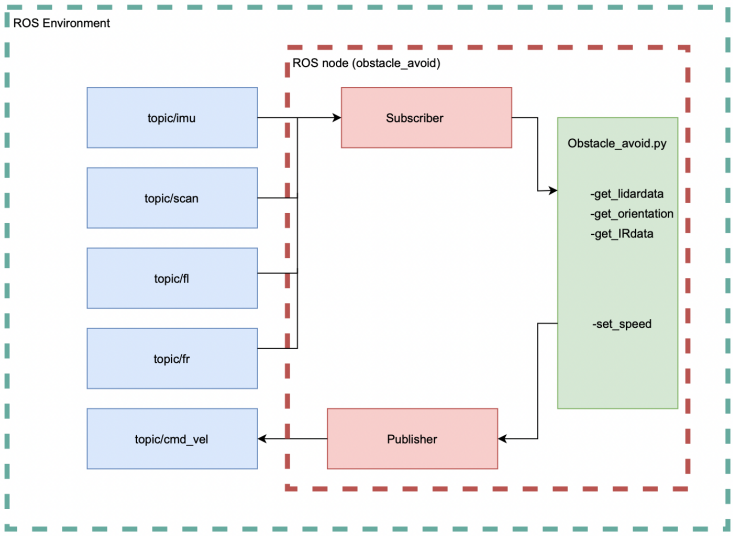

Figure 15- Software architecture of Obstacle_avoid node

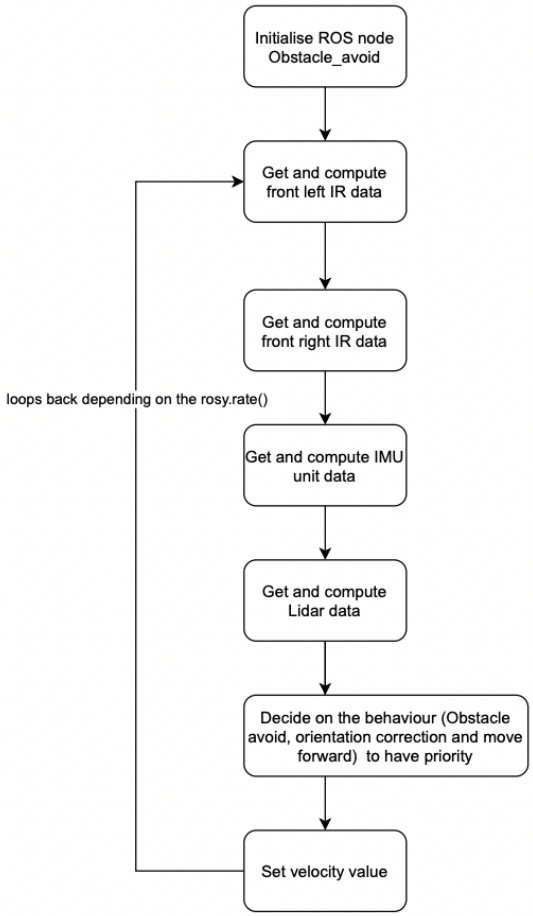

Figure 16- Software architecture of Obstacle_avoid python script

3.2 ROSbot_log node

The software architecture for the ROSbot_log node is shown in figure 17, the node uses the python script “ROSbot_log.py”. The node subscribes to the topic “odom” to get the odometer sensor data and logs the total travel distance of ROS bot. In addition, the node computes internally the total time to reach the destination. The software architecture of ROSbot _log python script is shown in figure 18.

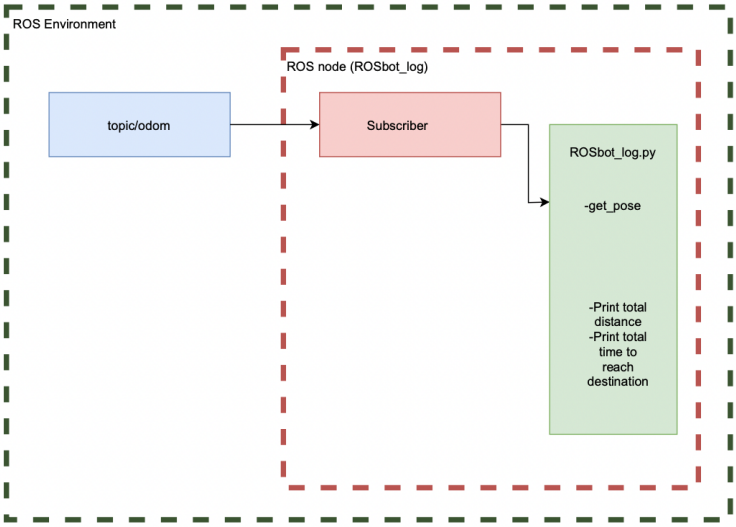

Figure 17- Software architecture of ROSbot_log node

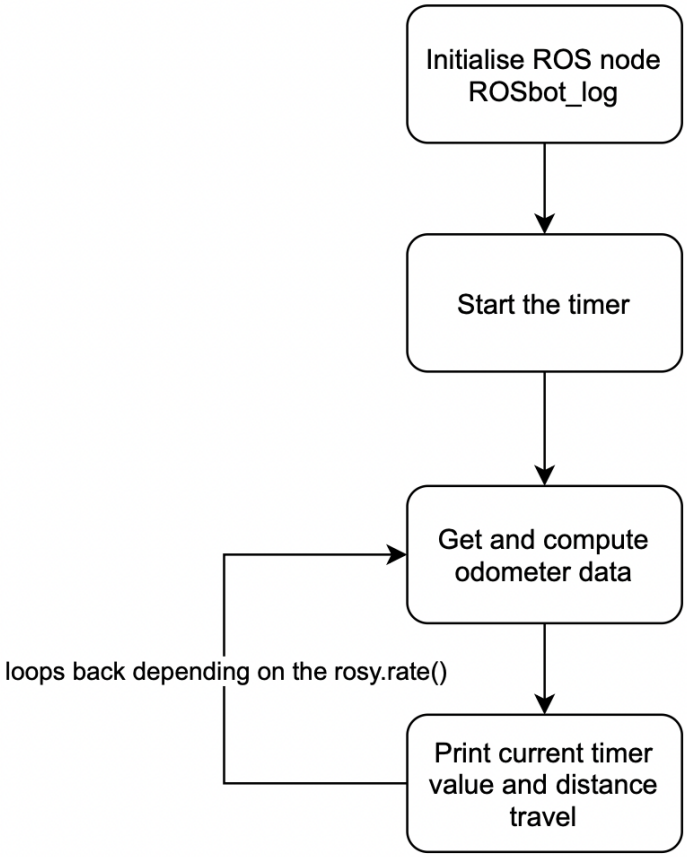

Figure 18- Software architecture of ROSbot_log python script

3.3 Rqt_Graph

The ROS computation graph for the task level “Navigate ROSbot to the destination” is shown in figure 19. The topics represent in a rectangle shape and nodes represent in the shape of the ellipse. The /obstacle_avoid and /rosbot_log nodes are user-defined whereas other nodes are system defined.

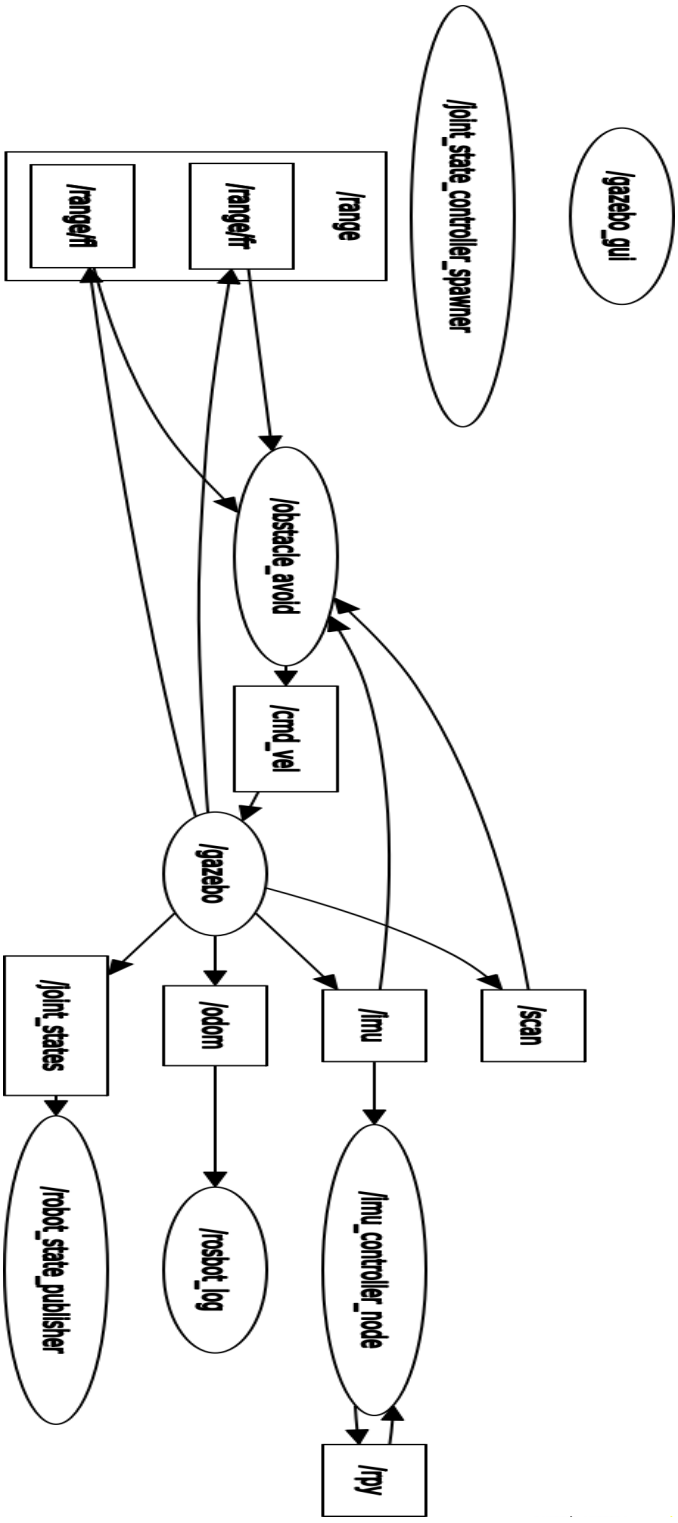

Figure 19- ROS computation graph for the ROS bot obstacle avoidance algorithm

4- Software implementation

4.1- Lidar data interpretation

The ROS node subscribes to the scan topic to aquire Lidar data. The code for the scan topic subscriber is "**rospy.Subscriber('scan',LaserScan,lidar_callback)**". Where the scan is the topic name, LaserScan is the message type and lidar_callback is the call back function to store and compute the Lidar data.

The lidar in the rosbot provides data in the form of arrays and the array size is 720. The array indexes respective to the regions of interest is shown in figure 20.

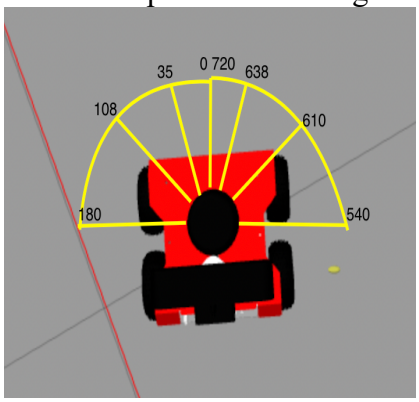

Figure 20- Representation of Lidar front region with the respective array index.

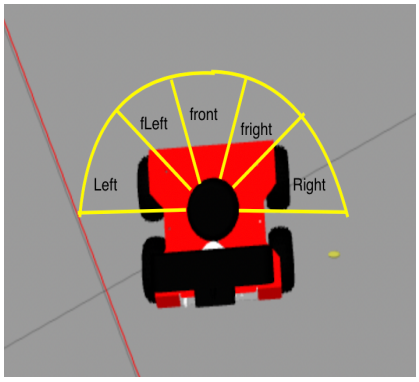

Figure 21- Representation of Lidar front region section.

The code for interpreting the Lidar data is shown in figure 22. The minimum function (min) gets the lowest value in the range of array indexes. The other minimum function returns the value 12 while the lidar data returns 'inf'. The inf value represents the lidar measuring outside the maximum distance range.

```
def lidar_callback(msg):
    global regions
    regions= {'right':min(min(msg.ranges[539:610]),12),       #right
    'fright':min(min(msg.ranges[611:684]),12),                #fright
    'front':min(min(msg.ranges[0:35]),12),                    #front1
    'front2':min(min(msg.ranges[683:719]),12),                #front2
    'fleft':min(min(msg.ranges[36:107]),12),                  #fleft
    'left':min(min(msg.ranges[108:179]),12)}                  #left
```

Figure 22- Computation of lidar data into regions. Code adapts from Arruda (2019).

4.2 IMU data interpretation

The ROS node subscribes to the IMU topic to acquire the orientation data as shown in figure 23. The code for the imu topic subscriber is "**rospy.Subscriber('/imu',Imu,IMU_callback)**". Where the /imu is the topic name, IMU is the message type and IMUcb is the callback function to store and compute the IMU data. As shown in figure 24, we extract the orientation data from the imu topic and convert the quaternion form of data to Euler form for better interpretation. Furthermore, Euler conversion provides roll, pitch and theta. The theta provides the angular position of the ROS bot. The theta information is in radians and the program converts to the degree for better understanding. The angular positions of the ROS bot are shown in figure 25.

```
orientation:
  x: -0.00033304610174338583
  y: 0.0009284930018677824
  z: 0.8601728249038378
  w: 0.5100017041912617
orientation_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
angular_velocity:
  x: 0.01778969412906469
  y: 0.003065723995960947
  z: -0.001914902555549605
angular_velocity_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
linear_acceleration:
  x: -0.025451588994566163
  y: -0.0007786357781467308
  z: 9.809851376201493
linear_acceleration_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Figure 23 – Representation of the IMU data. (Orientation data is highlighted in the blue box)

```
orientation_q=msg.orientation
orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
(roll, pitch, theta) = euler_from_quaternion (orientation_list)
theta_deg= math.degrees(theta)
```

Figure 24- Computation of imu data into ROS bot orientation. Code adapts from Rahman (2018).

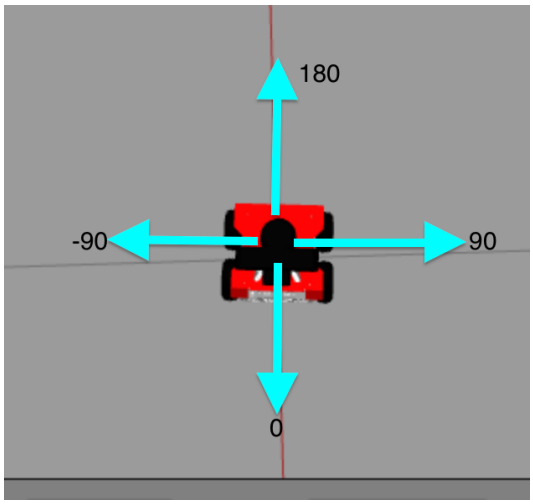

Figure 25 – Representation of the ROS bot angular position in the simulation environment.

4.3 IR sensor data interpretation

The ROS node subscribes to fl and fr topic to get the front left and front right IR sensor reading respectively. The code snippet for the fl and fr topic subscribers is shown in figure 26, where the fl and fr are the topics name, Range is the message type and fl_callback and fr_callback is the call back functions to store the IR sensor measurement data.

```
sub=rospy.Subscriber('range/fl',Range,f1_callback)
sub2=rospy.Subscriber('range/fr',Range,fr_callback)
```

Figure 26 – Code snipper of the IR sensors subscribers.

4.4 Odometer data interpretation

The ROS node subscribes to odom topic to acquire the ROS bot current position relative to the start position. The code snippet for the odom topic subscriber is "**rospy.Subscriber('/odom',Odometry,self.odometryCb)**", where the odom is the topic name, Odometry is the message type and self.odometryCb is the call back function to store and compute the Euclidean distance. Computation of the odometry data is shown in figure 27, where we extract the X and Y pose of the ROS bot and use the Pythagorean theorem to calculate the Euclidean distance as shown in figure 28. The sum of all Euclidean distance provides the total distance travel.

```
def odometryCb(self,msg):
    if self.firstrun:
        self.old_x=msg.pose.pose.position.x
        self.old_y=msg.pose.pose.position.y
    x= msg.pose.pose.position.x
    y= msg.pose.pose.position.y
    d_increment = math.sqrt(((x - self.old_x) * (x - self.old_x) )+
                ((y - self.old_y)*(y - self.old_y)))
    self.distance = self.distance + d_increment
    self.old_x=x
    self.old_y=y
```

Figure 27- Computation of Odometry data into Euclidean distance.

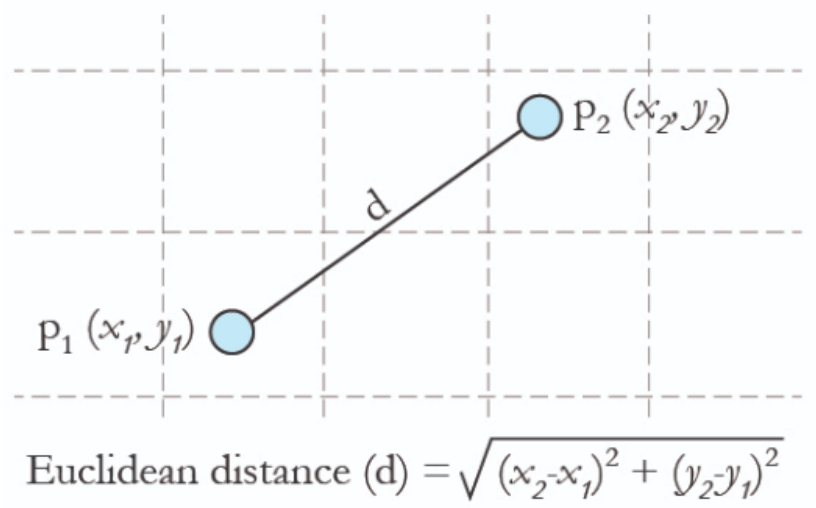

Figure 28- Representation of Euclidean distance calculation. (P1 and P2 are the first and second points, d is the Euclidean distance between the points). (Tutorial Example, 2020)

4.5 ROS bot velocity publisher

The ROS node publishes to the topic cmd_vel to communicate the value of angular velocity and linear velocity using the code "**rospy.Publisher('/cmd_vel',Twist, queue_size=1)**", where the cmd_vel is the topic name, Twist is the message type and queue_size is the message buffer size.

4.6 Orientation correction algorithm

ROS bot orientation correction algorithm is shown in figure 29. The ROS bot angular position logic has a tolerance of +-5 degrees, and the region of interest is shown in figure 28. At the angular position of 180 degrees, the ROS bot faces towards the destination point. We determine the value of X in section 6.2.

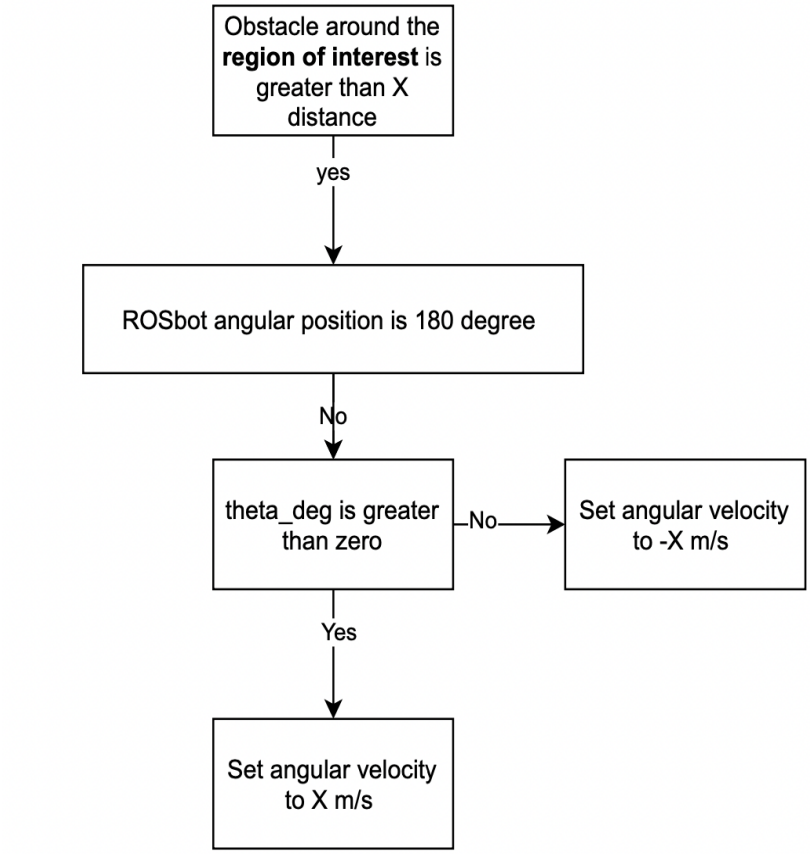

Figure 29- Flow chart representation of ROS bot orientation correction algorithm.

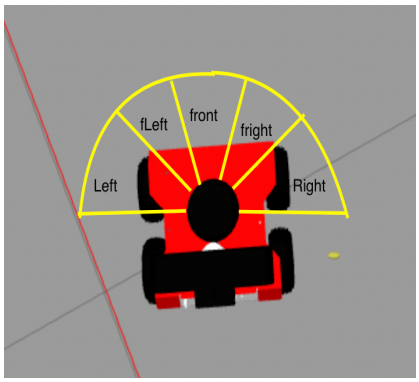

Figure 30- Representation of Lidar region of interest to detect obstacles.

4.7 Obstacle avoidance algorithm

ROS bot Obstacle avoidance algorithm is shown in figure 31. We determine the value of X in section 6.2.

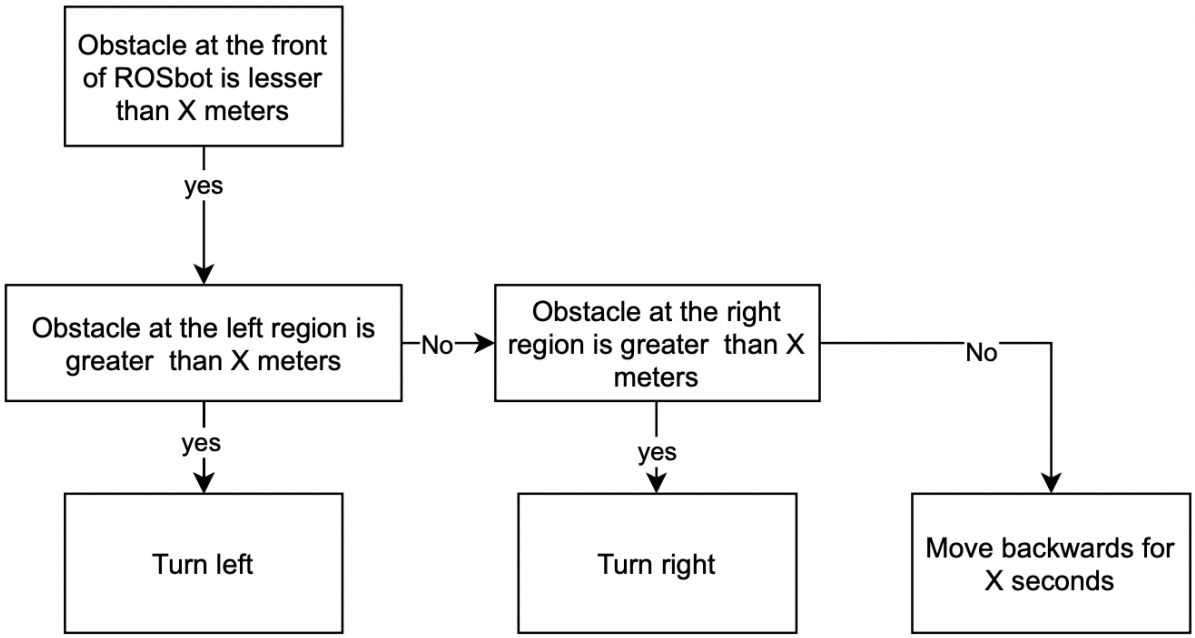

Figure 31- Flow chart representation of ROS bot obstacle avoidance algorithm.

4.8 Move forward algorithm

ROS bot move forward algorithm is shown in figure 32. The region of interest to move forward and determine the value of X in section 6.2.

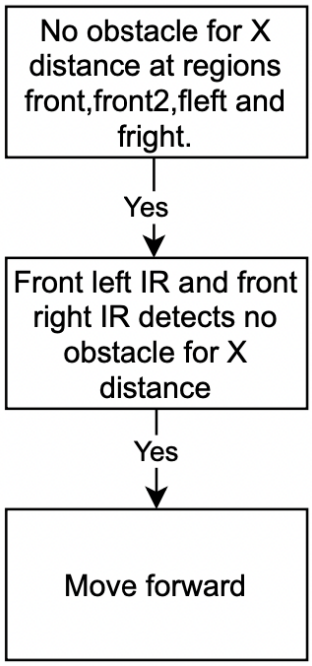

Figure 32- Flow chart representation of ROS bot Move forward algorithm.

5- Testing approach

5.1-Launch File

To simulate ROS bot in a 3D environment, we use the physics simulation software Gazebo. To initialise all the necessary parameters and files for the simulation, we employ a launch file. The world environment is added to the launch file using the code "**<arg name="world" default="$(find rosbot_bath)/worlds/2021_assessment.world"/>**".
The user defined ROS nodes such as Obstactle_avoid and ROSbot_log node is initialised in the launch file using the code ""**<node name="obstacle_avoid" pkg="rosbot_bath" type="obstacle_avoid.py" output="screen"></node>**
<node name="rosbot_log" pkg="rosbot_bath" type="rosbot_log.py" output="screen"></node>".
The complete launch file script is in the appendix section.

5.2-Robot start point

The ROS bot start point in the simulation world is changed for every run instance to avoid the ROS bot navigating to the destination under the same trajectory. Furthermore, the process helps to test the ROS bot behaviours under various trajectories and helps to optimise the obstacle avoidance algorithm.
The start point of the ROS bot is initialised in the launch file with the code
"<include file="$(find rosbot_bath)/launch/spawn_robot.launch">
<arg name="x" value="9.0"/>
<arg name="y" value="3.0"/>
<arg name="yaw" value="3.14"/>
</include>".
The code communicates the ROS bot initial world coordinate value to the launch file "Spawn_robot.launch".

5.3 Data acquisition

To optimise the total time and travel distance, we use the ROSbot_log node to publish the instantaneous distance travel and time in the terminal as shown in figure 33.

```
INFO] [1641469768.698026, 7.320000]: Distance= 1.398868648746578 meters
INFO] [1641469768.699650, 7.320000]: Time= 9.3851 seconds
INFO] [1641469768.799916, 7.410000]: Distance= 1.4188731289330763 meters
INFO] [1641469768.801018, 7.410000]: Time= 9.4870 seconds
INFO] [1641469768.900352, 7.510000]: Distance= 1.4389788158234098 meters
INFO] [1641469768.901834, 7.520000]: Time= 9.5875 seconds
INFO] [1641469769.007308, 7.620000]: Distance= 1.4589968781355407 meters
INFO] [1641469769.008765, 7.620000]: Time= 9.6944 seconds
INFO] [1641469769.106533, 7.720000]: Distance= 1.4790101271661606 meters
INFO] [1641469769.107527, 7.720000]: Time= 9.7936 seconds
INFO] [1641469769.205203, 7.820000]: Distance= 1.49901430145923 meters
INFO] [1641469769.207048, 7.820000]: Time= 9.8923 seconds
INFO] [1641469769.307655, 7.920000]: Distance= 1.5189715378269222 meters
INFO] [1641469769.308565, 7.920000]: Time= 9.9947 seconds
INFO] [1641469769.399498, 8.020000]: Distance= 1.538996217551203 meters
INFO] [1641469769.401042, 8.020000]: Time= 10.0866 seconds
INFO] [1641469769.501305, 8.120000]: Distance= 1.5590530306456023 meters
INFO] [1641469769.503331, 8.120000]: Time= 10.1884 seconds
INFO] [1641469769.602237, 8.220000]: Distance= 1.5790678731029781 meters
INFO] [1641469769.604334, 8.220000]: Time= 10.2893 seconds
INFO] [1641469769.704396, 8.320000]: Distance= 1.599081908143155 meters
INFO] [1641469769.705384, 8.320000]: Time= 10.3914 seconds
```

Figure 33- ROSbot_log node publishes the data of distance and time in the terminal.

5.4 ROS visualization

To visualize the sensor data, we launch the RVIZ software. The visualization of the sensor data in the 3D simulation world is shown in figure 34, the lidar sensor represents objects in the yellow colour while the grey cone represents the two front IR sensors. The 3D visualisation in the Gazebo software is shown in figure 35.

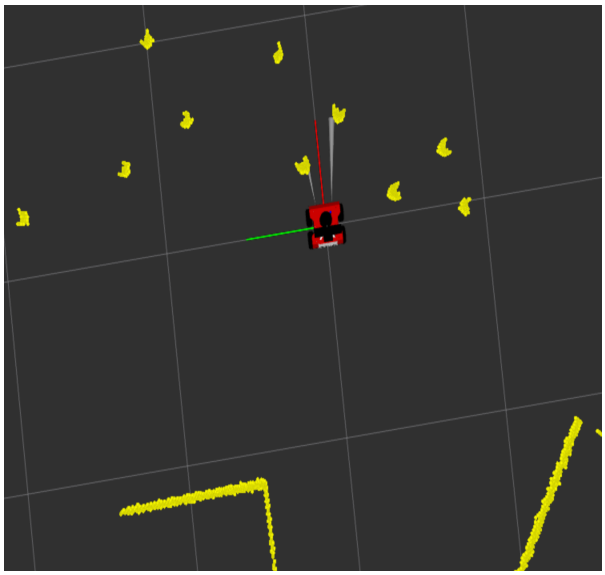

Figure 34- Visualization of ROS bot in the RVIZ software.

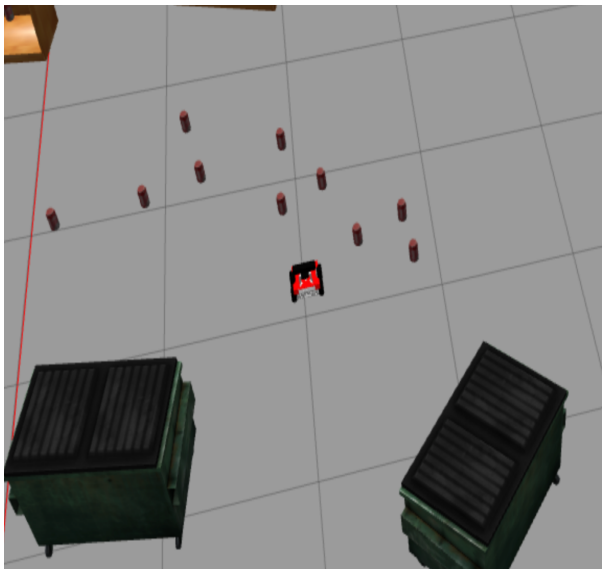

Figure 35- Visualization of ROS bot in the Gazebo simulation software.

6- Testing Result

6.1 Lidar regions of interest

For the initial test, the Lidar data is classified into 5 regions as shown in figure 36. The fleft and left regions are considered to detect obstacles at the left region. The fright and right regions are considered to detect obstacles at the right region. The front region and two IR are considered to detect obstacles at the front as shown in figure 37. The result is the ROS bot navigates around the obstacles successfully. However, occasionally clips the obstacle slightly at the side while moving forward because the front region is not wide enough as shown in figure 39.

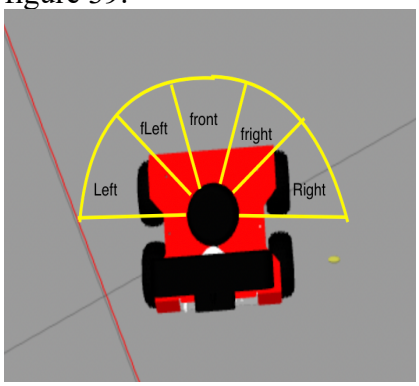

Figure 36- Representation of Lidar region of interest to detect obstacles.

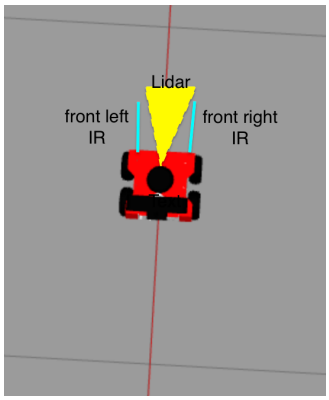

Figure 37- Representation of Lidar front region and rays of front IR sensors.

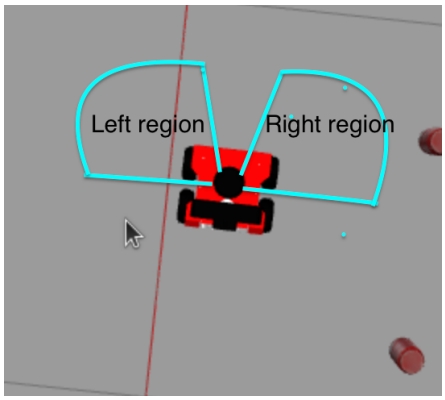

Figure 38- Representation of Lidar left and right region of interest.

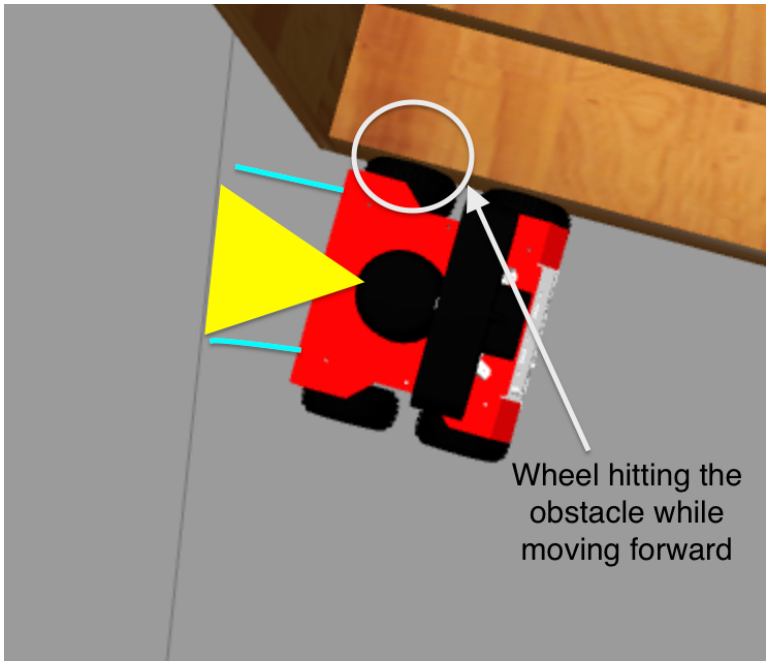

Figure 39- ROS bot hitting the obstacle at the side.

For the final test, the regions fleft, fright and front are considered to detect obstacles ahead prior to moving forward. As a result, the ROS bot does not have a blind spot while moving forward since the region of interest to detect obstacles ahead is wider than before as shown in figure 40.

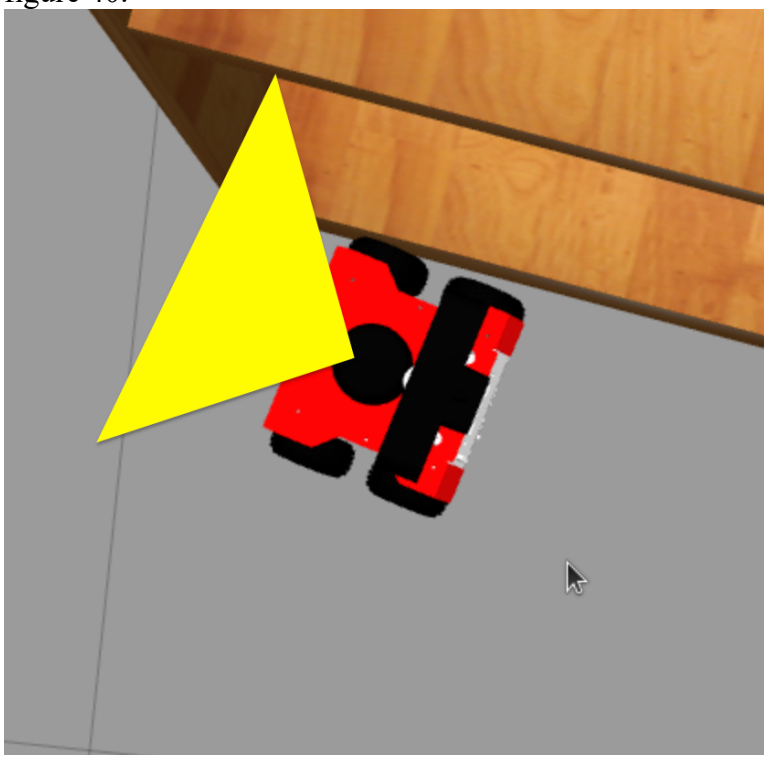

Figure 40- Representation of wider front region of interest to move forward.

6.2 Lidar range value

For the initial test, the obstacle detection range is set to one meter in the simulation environment. The outcome is the ROS bot navigates around the obstacle but takes a longer trajectory as shown in figure 41. The approach results in more time to reach the destination.

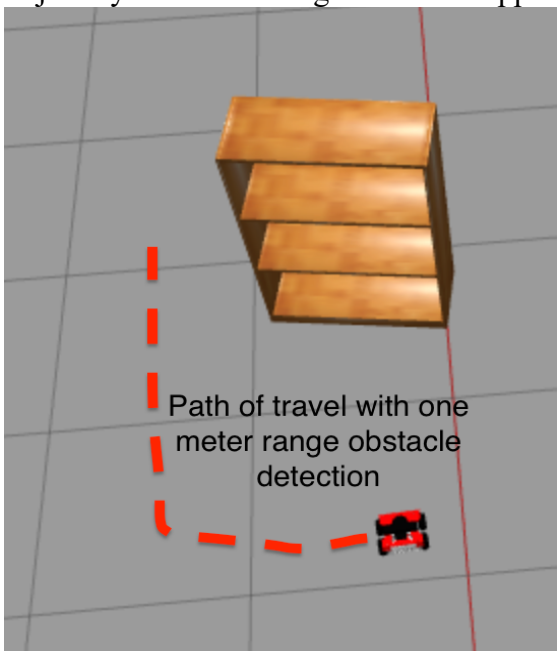

Figure 41- ROS bot navigating around the obstacle with a one-meter detection range.

The obstacle detection range is set to 0.1m. The outcome is the ROS bot navigate the obstacle at closer proximity as shown in figure 42. However, the ROS bot occasionally hits the obstacle because the action selection does not have enough time to process the Lidar data and react.

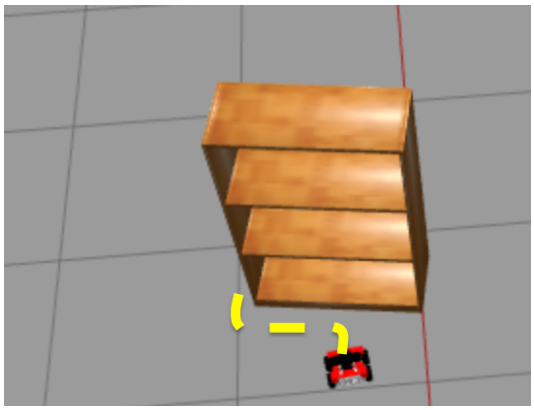

Figure 42- ROS bot navigating around the obstacle with a 0.1 meter detection range

After the reiterative approach, the obstacle detection range is set to 0.3 meters. The outcome is the ROS bot navigate around the obstacle and takes a shorter trajectory as shown in figure 43.

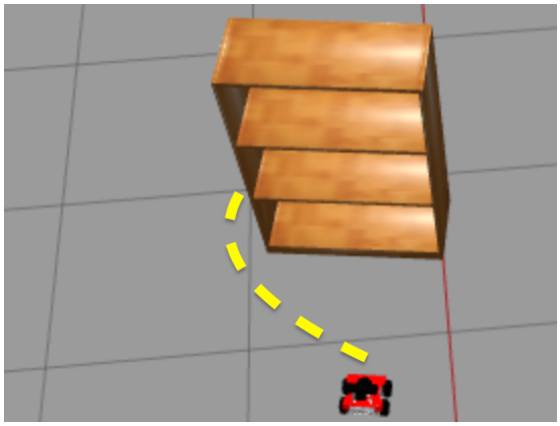

Figure 43- ROS bot navigating around the obstacle with a 0.3 meters detection range

6.3 Rospy loop rate

For the initial test, Rospy.rate() of the obstacle_avoid node is set to 1Hz which translates to the loop rate of 1 second (Frequency (Hz)=$\frac{1}{Time(S)}$). The ROS node publishes the new velocity command to the ROS bot and interprets sensor data with 1 second interval. Running the node at the rate of 1 second causes the ROS bot to dither while it steers around the obstacle as shown in figure 44, the ROS bot steers between left and right constantly.

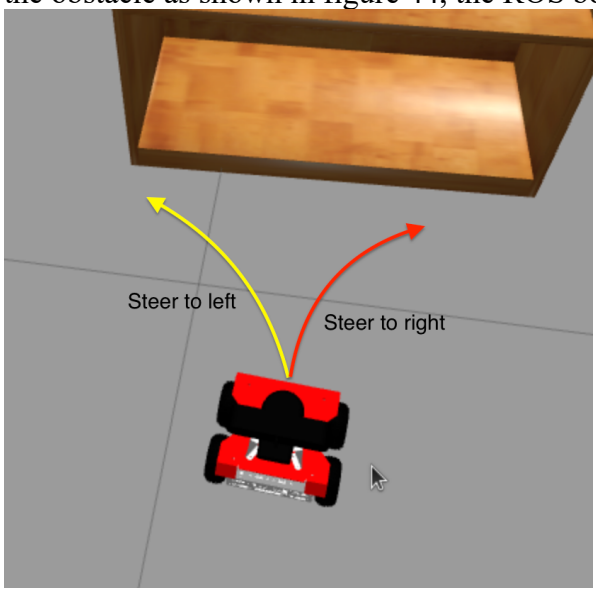

Figure 44- ROS bot in dithering state

To eliminate the dithering effect, the node loop rate is set to 2 seconds. The ROS bot starts to hit the obstacle ahead due to interpreting the sensor data at 2 second interval. To compensate for the effect, the range of detection cannot alter as per section 6.2 and the slower speed of the ROS bot results in a longer time to reach the destination. Through the reiterative test approach, the final ROS node rate is set to 1.4 seconds and the ROS bot steer around the obstacle without dithering.

To decrease the processor stress, the Rospy.rate() of ROSbot_log node is set to 0.2Hz which translates to the loop rate of 5 seconds. As a result, the distance travel and total time variable update with 5 seconds interval.

6.4 Angular and linear velocity

The initial value of the linear velocity is set to 0.5m/s. As a result, the ROS bot collides with the obstacle ahead due to less reaction time. The value of 0.1m/s results in a longer time. The value of 0.2m/s results in satisfactory total time for the ROS bot to reach the destination while avoiding the obstacle ahead as well.

The node loop rate at 1.4 seconds and the angular velocity of 0.2m/s results in a dithering effect as shown in figure 42. Therefore, a higher angular velocity value of 0.4m/s is set and ROS bot steers around the object without dithering.

The node loop rate at 1.4 seconds and the angular velocity of 0.4m/s for orientation correction result in the ROS bot spinning around because the angular velocity speed is higher than the IMU unit data interpretation rate. As a result, the ROS bot cannot stop at the specific angular position. Therefore, a lower angular velocity value of 0.2m/s is set only for the orientation logic correction and the ROS bot stops at the specific angular position with 5 degrees tolerance.

7- Conclusion

The ROS bot navigates to the destination while avoiding the obstacles along the path using only the sensors that come with the ROS bot and logs information such as total time and distance travel. The process of simulating the ROS bot helps to showcase the knowledge in ROS environment, interpretation of sensor data, Behaviour Oriented Design, Hierarchical state machine design and Linux operating system. In addition, optimising parameters such as ROS node loop rate, obstacle detection range, linear velocity and angular velocity to work in a harmony proves to be a tedious job. Even though the process is time-consuming, the process helps to understand the correlation between the parameters. Furthermore, obstacles of various morphology prove to be a challenge. However, with the reiterative testing, the obstacle avoidance algorithm is robust to exhibit dynamic adaptability behaviour.

8-Future work

Optimisation of various parameters to work in a harmony, we can automate the process by using a reinforcement learning algorithm. The approach helps to optimise the parameters in various world environments without the need for human interference.

9- References

Arruda, M., 2019. Exploring ROS using a 2 Wheeled Robot: Obstacle Avoidance | The Construct. [online] The Construct. Available at: https://www.theconstructsim.com/exploring-ros-2-wheeled-robot-part-5

Bryson, J. J. (2007) 'Mechanisms of Action Selection: Introduction to the Special Issue', Adaptive Behavior, 15(1), pp. 5–8. doi: 10.1177/1059712306076247.

Meyer, Benjamin & Ehlers, Kristian & Osterloh, Christoph & Maehle, Erik. (2013). Smart-E An Autonomous Omnidirectional Underwater Robot. Paladyn, Journal of Behavioral Robotics. 4. 10.2478/pjbr-2013-0015.

Rahman, A., 2018. How to Rotate a Robot to a Desired Heading? - The Construct. [online] The Construct. Available at: https://www.theconstructsim.com/ros-qa-135-how-to-rotate-a-robot-to-a-desired-heading-using-feedback-from-odometry

Tutorial Example (2020). Calculate Euclidean Distance in TensorFlow: A Step Guide - TensorFlow Tutorial. [online] Tutorial Example. Available at: https://www.tutorialexample.com/calculate-euclidean-distance-in-tensorflow-a-step-guide-tensorflow-tutorial

Wortham, R. (2021). 'Lecture 7 - Action Selection' [PowerPoint presentation]. EE50237: Robotics Software. Available at: https://moodle.bath.ac.uk/course/view.php?id=58288&section=12

10-Appendix

10.1- Obstacle node python script

```
#!/usr/bin/env python3
# BEGIN ALL

import math
import rospy
from geometry_msgs.msg import Twist,PoseStamped
from rospy.timer import sleep
from sensor_msgs.msg import LaserScan,Range,Imu
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion

def f1_callback(msg):
    global fl
    fl=msg.range

def fr_callback(msg):
    global fr
    fr=msg.range

def lidar_callback(msg):
    global regions
    regions= {'right':min(min(msg.ranges[539:610]),12),       #right
    'fright':min(min(msg.ranges[611:684]),12),                #fright
    'front':min(min(msg.ranges[0:35]),12),                    #front1
    'front2':min(min(msg.ranges[683:719]),12),                #front2
    'fleft':min(min(msg.ranges[36:107]),12),                  #fleft
    'left':min(min(msg.ranges[108:179]),12)}                  #left
    navigate()
    rotate()

def IMU_callback(msg):
    global theta
    global theta_deg
    global quadrant
    roll=0
    pitch=0
    orientation_q=msg.orientation
    orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z,
orientation_q.w]
    (roll, pitch, theta) = euler_from_quaternion (orientation_list)
    theta_deg= math.degrees(theta)

def navigate():
    if regions['front'] > 0.3 and regions['fright'] > 0.3 and regions['front2']>0.3
and regions['fleft'] > 0.3 and fl>0.3 and fr>0.3:
#front
        linear_x = 0.2
        angular_z = 0
    elif regions['left'] > 0.3 :         #left
```

```
        linear_x = 0
        angular_z = 0.4

    elif ( regions['right'] >0.3) :        #right
        linear_x = 0
        angular_z =-0.4

    else:
#back
        linear_x=-0.1
        angular_z=0

    move.linear.x=linear_x
    move.angular.z=angular_z

def rotate():
    if regions['front'] > 1 and regions['front2']>1 and regions['fleft'] > 0.3 and
regions['left']>0.3 and regions['fright'] > 0.3 and regions['right']>0.3 and
((theta_deg<175 and theta_deg>0)or (theta_deg>-175 and theta_deg<0)):
        if theta_deg>0:
                move.linear.x=0.0
                move.angular.z=0.2
        else:
                move.linear.x=0.0
                move.angular.z=-0.2

    pub.publish(move)

rospy.init_node('Obstacle_avoid')
sub=rospy.Subscriber('range/fl',Range,f1_callback)
sub2=rospy.Subscriber('range/fr',Range,fr_callback)
sub3=rospy.Subscriber('scan',LaserScan,lidar_callback)
sub4=rospy.Subscriber('/imu',Imu,IMU_callback)
pub= rospy.Publisher('/cmd_vel',Twist, queue_size=1)
move=Twist()
rospy.Rate(0.7)
rospy.spin()
```

10.2-ROSbot_log node python script

```
#!/usr/bin/env python3
# BEGIN ALL
import math
import time
import rospy
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion

class distancetravel:
    def __init__(self):
        self.firstrun=True
        self.distance=0
        self.old_x=0
        self.old_y=0
        self.startT=time.perf_counter()

    def listen(self):
        rospy.init_node('ROSbot_LogInfo')
        self.sub=rospy.Subscriber('/odom',Odometry,self.odometryCb)
        rospy.Rate(0.2)
        rospy.spin()

    def odometryCb(self,msg):
        if self.firstrun:
            self.old_x=msg.pose.pose.position.x
            self.old_y=msg.pose.pose.position.y
        x= msg.pose.pose.position.x
        y= msg.pose.pose.position.y
        d_increment = math.sqrt(((x - self.old_x) * (x - self.old_x) )+
                    ((y - self.old_y)*(y - self.old_y)))
        self.distance = self.distance + d_increment
        self.old_x=x
        self.old_y=y
        self.firstrun=False
        currentT=time.perf_counter()
        TotalT=(currentT-self.startT)
        rospy.loginfo(f'Distance= {self.distance} meters')
        rospy.loginfo(f'Time= {TotalT:0.4f} seconds')

if __name__ == '__main__':
    odom = distancetravel()
    odom.listen()
```

10.3 Launch file script

```
<?xml version="1.0"?>
<launch>

  <rosparam command="load" file="$(find
joint_state_controller)/joint_state_controller.yaml" />
  <node name="joint_state_controller_spawner" pkg="controller_manager"
type="spawner" output="screen" args="joint_state_controller" />

  <param name="robot_description" command="$(find xacro)/xacro '$(find
rosbot_bath)/urdf/rosbot.xacro'"/>

<include file="$(find rosbot_bath)/launch/spawn_robot.launch">

    <arg name="x" value="9.0"/>
    <arg name="y" value="3.0"/>
    <arg name="yaw" value="3.14"/>

</include>

  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher"/>
  <node name="imu_controller_node" pkg="rosbot_bath" type="imu_controller_node"/>

  <arg name="world" default="$(find rosbot_bath)/worlds/2021_assessment.world"/>
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(arg world)"/>
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="gui" value="$(arg gui)"/>
    <arg name="headless" value="$(arg headless)"/>
    <arg name="debug" value="$(arg debug)"/>
  </include>
  <node name="obstacle_avoid" pkg="rosbot_bath" type="obstacle_avoid.py"
output="screen"></node>
  <node name="rosbot_log" pkg="rosbot_bath" type="rosbot_log.py"
output="screen"></node>

</launch>
```