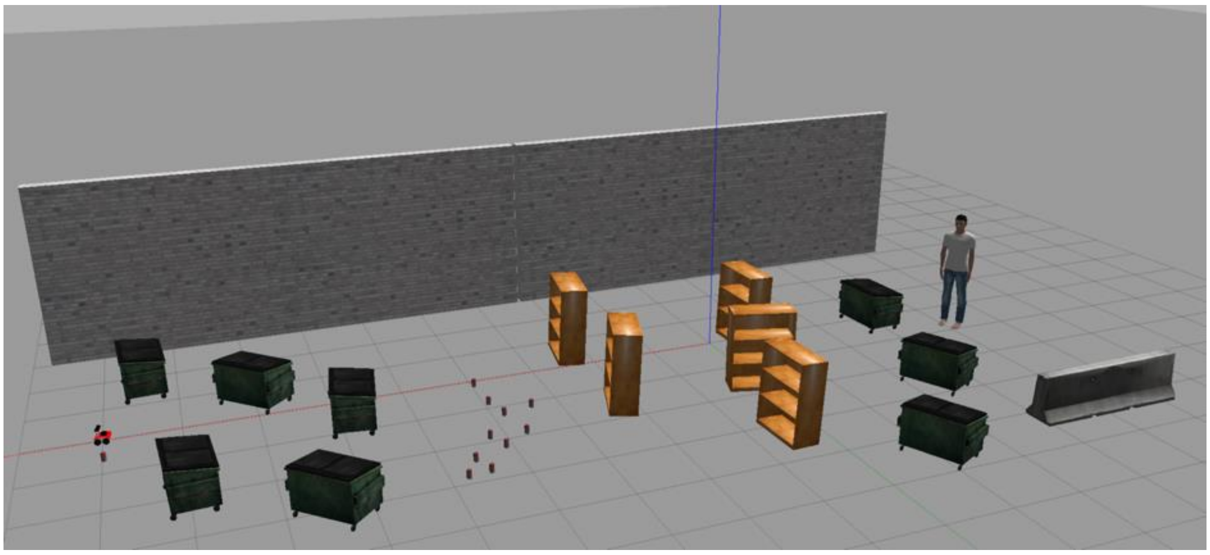
EE50237 Robotics Software
Assessed Project 2021 – The Robot Challenge
Project Report



Title: Reactive Motion Planning for Autonomous Obstacle Avoidance

Siu Kwan, YAU

Contents

List of Figures

List of Tables

1. Problem Analysis

The robot challenge is to build a robust autonomous robot navigation system that is capable of handling novel situation. The system should allow a Husarion ROSbot to travel across an obstacle course within the ROS Gazebo simulator.

The two major goals to be achieved in this project are:

- Obstacle course navigation [1.1]; and
- Distance travelled and time taken monitoring [1.2] for estimating the energy consumption and evaluating the performance of the navigation system.

1.1. Obstacle Course Navigation

1.1.1. Analysis of System Design Criteria

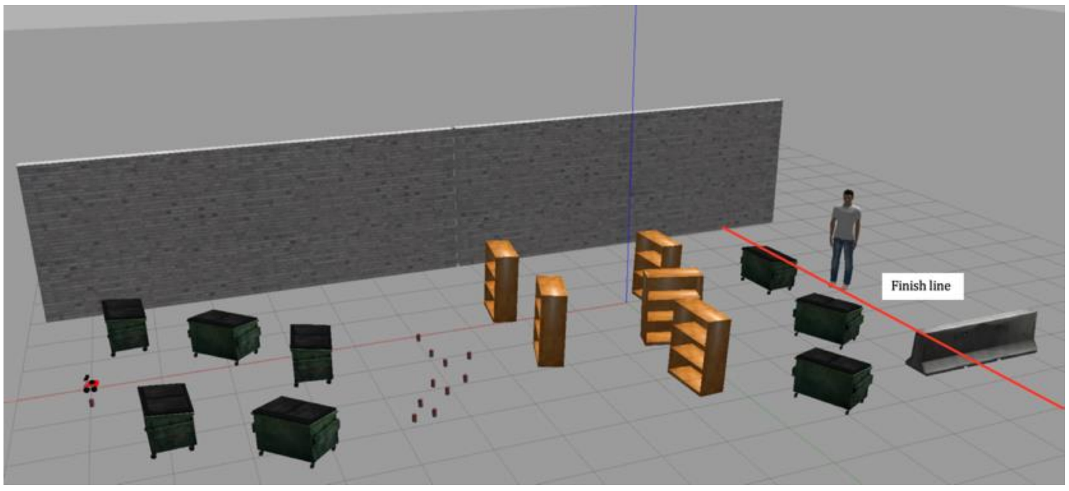
a. Specifications of the Obstacle Course

The following are the assumptions specified for the obstacle course:

- There is no direct line of sight path from starting point to destination, but a path is possible
- The obstacle course will remain static throughout the challenge
- No map or other parametric data about the environment of the course are provided in advance

The above assumptions imply that the ROSbot is placed in a dynamic and unpredictable environment, and must perform action selection in a timely manner based on information retrieved from the environment in order to achieve its goal, i.e. to reach the destination of the course. As clarified with teaching assistants in lab session, the destination of the course can be treated as a finish line at the position of the person on the obstacle course as indicated in *Figure 1.1-1*.

Figure 1.1-1: Destination of the obstacle course



b. Hardware Specifications of ROSbot

The Husarion ROSbot is a four-wheeled aluminum-framed mobile platform. The hardware components of interest for capturing information about obstacles on the course are the RGBD camera, the laser scanner, and the four infrared sensors shown in *Figure 1.1-2* below. The specifications of these components are listed in *Table 1.1-1*.

Figure 1.1-2: Obstacle-detecting sensors available on ROSbot (Husarion, n.d.)

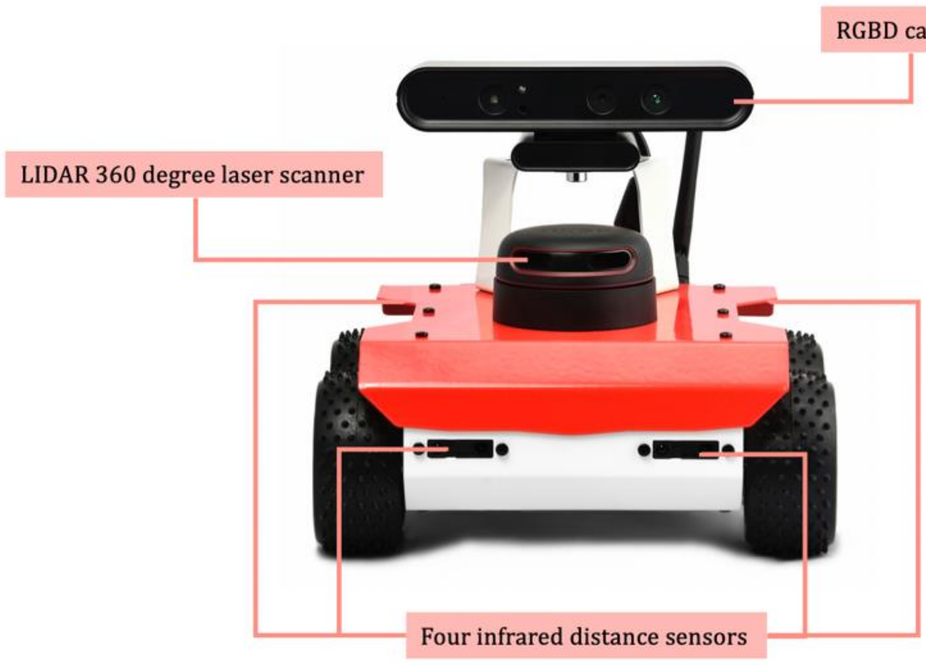

RGBD camera
LIDAR 360 degree laser scanner
Four infrared distance sensors

Table 1.1-1: Description of the obstacle detecting sensors on ROSbot (Husarion, n.d.; Mouser Electronics, 2018)

Component	Quantity	Description
RGBD camera	1	Model: Orbbec Astra RGB image size: 640 x 480 Depth image size: 640 x 480 Field of view: 60°H x 49.5°V x 73°D
LIDAR laser scanner	1	Model: RpLidar A2 Range: Up to 8 m Field of view: 360°
Infrared distance sensors	4	Model: VL53L0X Time-of-Flight distance sensors Range: Up to 200 cm Location: Front left, front right, rear left, rear right Field of view: Up to 27°

Apart from sensors that are helpful for detecting obstacles, the ROSbot is also equipped with an IMU sensor which provides data that helps interpreting the orientation, velocity and position of the ROSbot on the obstacle course.

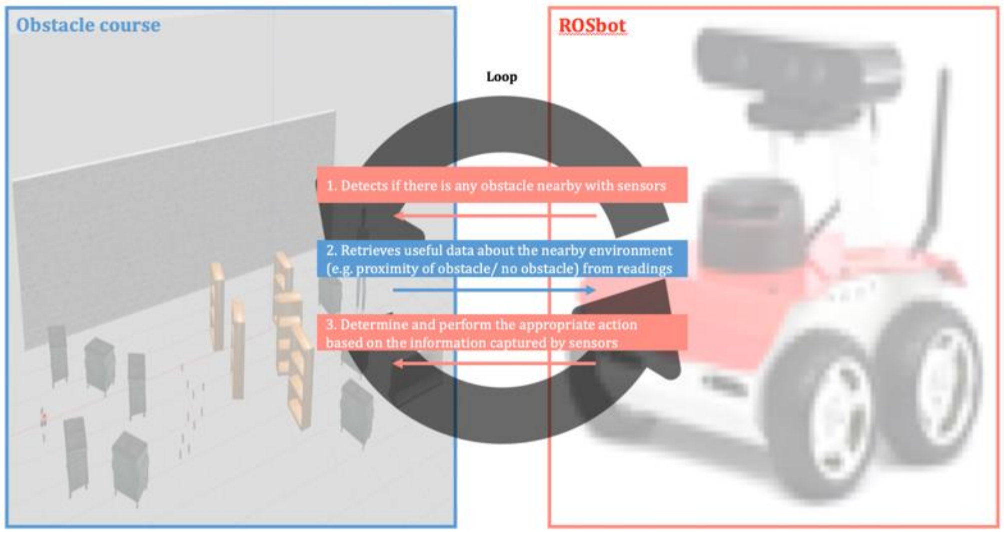
c. <u>Design Criteria</u>

Considering the specifications of the obstacle course and the hardware available on the ROSbot, the necessary features of the navigation system are identified as follows (also illustrated in *Figure 1.1-3*):

- **Retrieves data about the surrounding environment**: As mentioned in a, the ROSbot has no prior knowledge about the obstacle course initially. Therefore, the control system needs to make use of the sensors available on the ROSbot, as described in b, to obtain perceptions about the environment to help decide the next move.
- **Obtains and analyses the data received from sensors**: The navigation system should be able to retrieve useful information for the sensor readings including:
  - Existence of obstacles near the ROSbot;
  - Proximity of obstacles to the ROSbot; and
  - Heading direction of the ROSbot
- **Determine action to be taken based on data received from sensors**: The navigation system should make appropriate action(s) based on the information analysed from sensor readings, including:
  - If there is an obstacle in front of the ROSbot, steer to avoid the obstacle; and
  - If the ROSbot is not facing towards the destination, steer to face towards the destination.

The navigation system should perform the above processes repeatedly until the ROSbot reaches the destination.

Figure 1.1-3: Required features of the navigation system

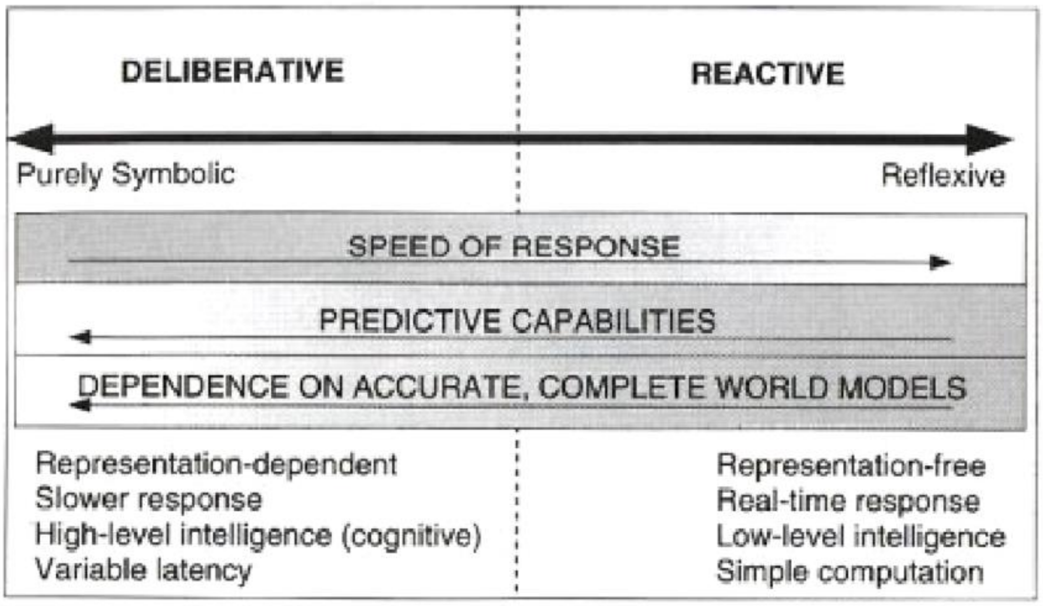

Obstacle course
ROSbot
Loop
1. Detects if there is any obstacle nearby with sensors
2. Retrieves useful data about the nearby environment (e.g. proximity of obstacle/ no obstacle) from readings
3. Determine and perform the appropriate action based on the information captured by sensors

1.1.2. Analysis of System Design Approach

a. <u>Choice of Navigation Approach</u>

Navigating a robot through an obstacle course is a problem of motion planning. Motion planning approaches can generally be classified into deliberative, reactive, or a hybrid of both (Arkin, 1998). *Figure 1.1-4* shows the characteristics of deliberative and reactive systems.

Figure 1.1-4: Characteristics of deliberative and reactive systems (Arkin, 1998)

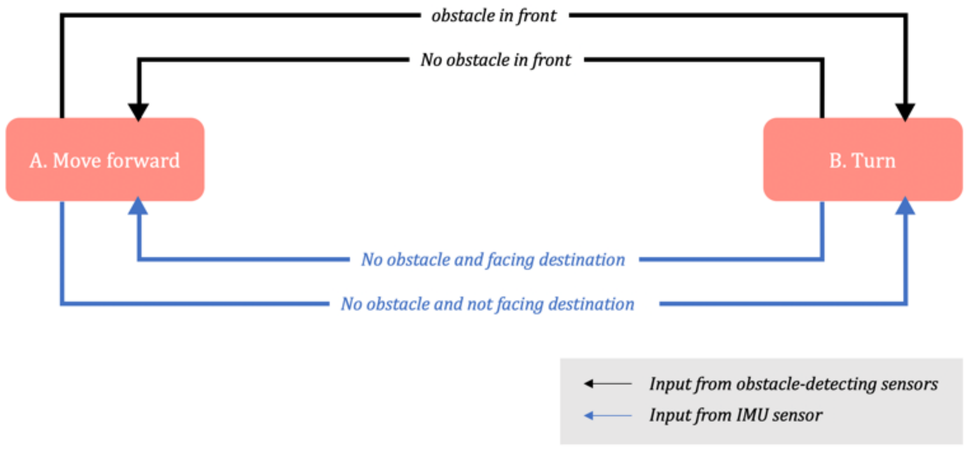


For this robot challenge, a reactive approach is deemed to be more appropriate than a deliberate approach because of the following reasons:

- **Reactive systems does not depend on a representation or map of the environment**: As no prior knowledge about the obstacle course is made available to the navigation system in advance, it is not possible to obtain sufficient information for constructing an accurate or meaning representation of the environment that allows deliberate planning.
- **Reactive systems are faster in response than deliberate systems**: As time taken for the ROSbot travel from starting point to destination is the performance evaluation criteria for this challenge, an approach which results in a faster speed of response is preferred.

In robotics, a reactive system can be expressed as a Finite State Machine (FSM). FSM presents a system as a sequence of state changes in response to some inputs (Wang and Tepfenhart, 2020). For this challenge, the inputs are readings from the ROSbot's sensors, and the high-level states are defined as "move forward" and "turn". *Figure 1.1-5* shows a high-level diagram illustrating transition of states in response to the information sensed by the ROSbot's sensors. The navigation system constantly senses the surrounding of the ROSbot's current location, then react to sensors' input in a timely manner, which mimics the way how biological organisms interacts with the world (Wortham, Gaudl and Bryson, 2019).

Figure 1.1-5: High-level State Machine Diagram of the navigation system



obstacle in front
No obstacle in front
A. Move forward
B. Turn
No obstacle and facing destination
No obstacle and not facing destination
Input from obstacle-detecting sensors
Input from IMU sensor

b. <u>Choice of Obstacle-detecting Sensors</u>

To reduce the computational complexity of the navigation system and allow the ROSbot to react in a timelier manner to the sensor input, an incremental approach for introducing sensors to the navigation system design is adopted. *Table 1.1-2* compares the characteristics of the three types of obstacle-detecting sensors available on the Husarion ROSbot. By comparing these characteristics and the hardware specifications listed in 1.1.1-b, LIDAR seems to be a more all-rounded sensor with the longest range, widest field of view (FoV), and unaffected by light variations. Hence, it was the first obstacle-detecting sensor being considered for the navigation system design. However, upon testing, it was found that the LIDAR scanner is unable to detect obstacles that are too short as it is installed on top of the ROSbot's aluminum frame (as shown in *Figure 1.1-6*). An additional sensor is therefore necessary.

Comparing the remaining two sensors available for obstacle detecting, although the RGBD camera has a longer range and wider FoV than the infrared distance sensor, it is positioned even higher than the LIDAR scanner (as shown in *Figure 1.1-2*). Hence, it might not be the optimal choice for complementing LIDAR in sensing short obstacle. The two front infrared distance sensors are therefore being incorporated in the system instead. During testing, the infrared sensors successfully detected the short coke can obstacles on the course.

With both the LIDAR and the two front infrared sensors incorporated in the navigation system, all obstacles on the course (i.e. the green bins, coke cans, and book shelves) can be detected. Hence, they become they final choice of obstacle-detecting sensors in the design. The scan range of the sensors are illustrated in *Figure 1.1-7* and *Figure 1.1-8* respectively. The FoV of the two front infrared distance sensors are obtained from the "/range/fl" and "/range/fr" ROS topics shown in *Figure 1.1-9*.

Table 1.1-2: Comparison on the three types of obstacle-detecting sensors available on ROSbot (El-Sheimy and Li, 2021; Lohar et al., 2021)

Characteristics	**Sensors**		
	RGBD camera	LIDAR laser scanner	Infrared distance sensors
Sensitive to illumination	Yes	No	No
Range	Medium	Long	Short
Camouflage detection	No	Yes	Yes
Resolution	Dense	Sparse	Sparse
Depth sensation	Yes	Yes	Yes
Cost	Low	High	Low

Figure 1.1-6: Obstacles on the course undetectable by LIDAR laser scanner

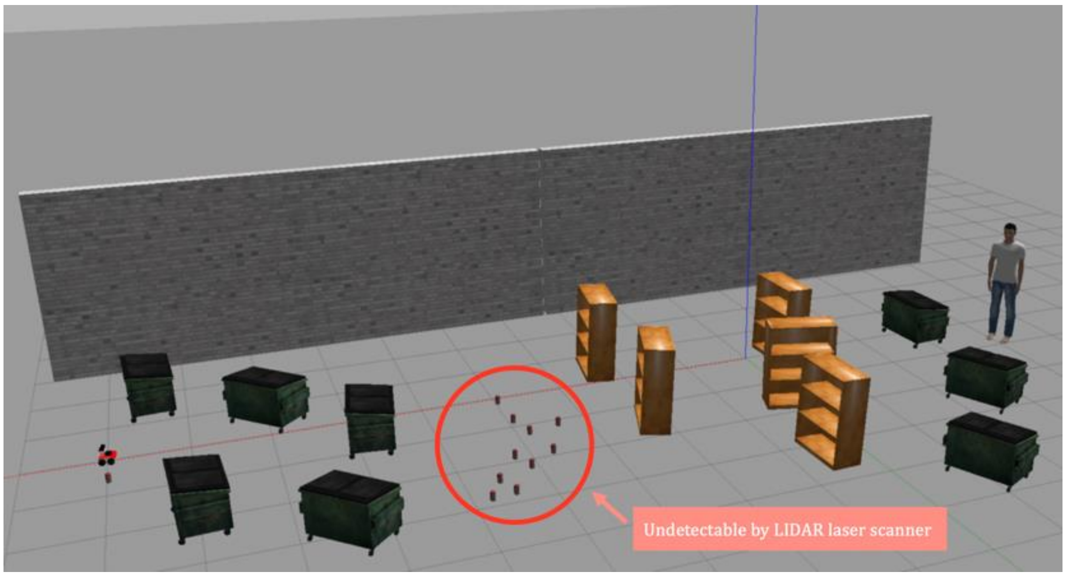


Figure 1.1-7: Scan range of the LIDAR laser scanner on ROSbot

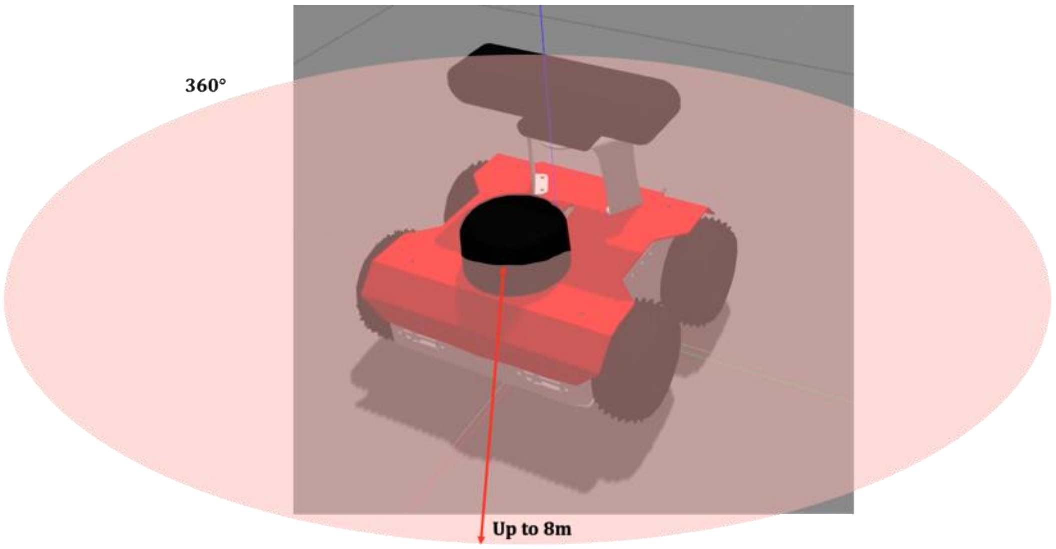


Figure 1.1-8: Scan range of the infrared distance sensors on ROSbot

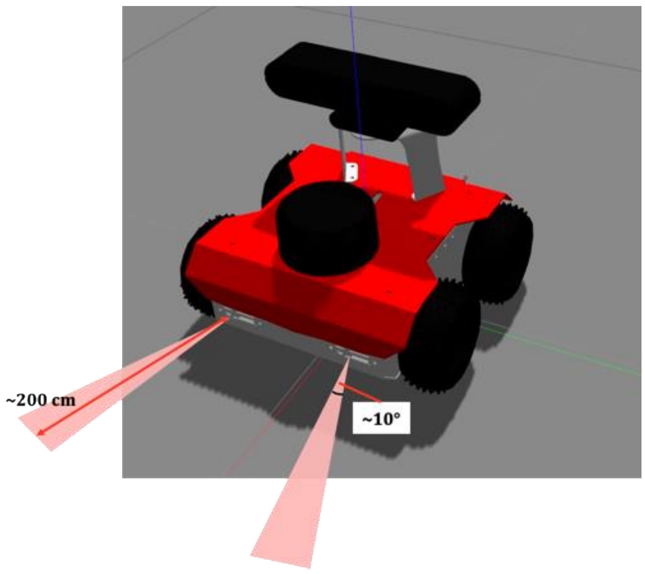
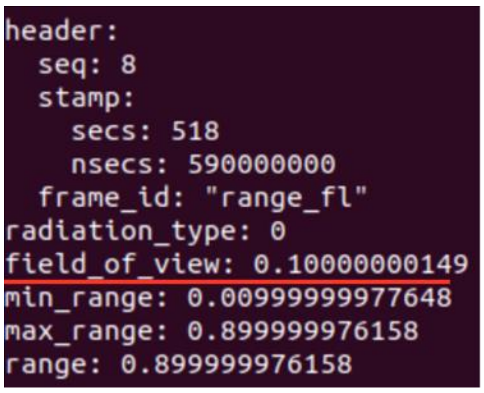
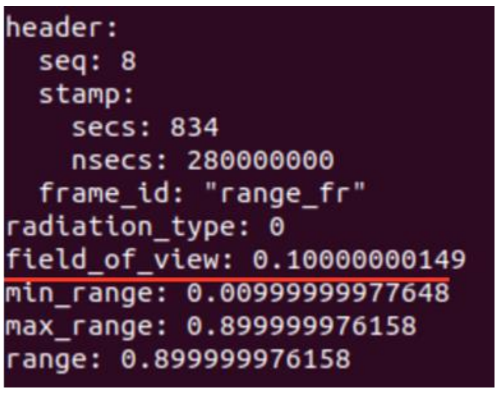


Figure 1.1-9: Field of view values of infrared distance sensors obtained from ROS topics



```
header:
  seq: 8
  stamp:
    secs: 518
    nsecs: 590000000
  frame_id: "range_fl"
radiation_type: 0
field_of_view: 0.10000000149
min_range: 0.00999999977648
max_range: 0.899999976158
range: 0.899999976158
```



```
header:
  seq: 8
  stamp:
    secs: 834
    nsecs: 280000000
  frame_id: "range_fr"
radiation_type: 0
field_of_view: 0.10000000149
min_range: 0.00999999977648
max_range: 0.899999976158
range: 0.899999976158
```

1.2. Distance Travelled and Time Taken Monitoring

Distance travelled and simulated time are specified to be the criteria for measuring the performance of the navigation system. The following describes the design for capturing these information:

1.2.1 Distance Travelled

With the x and y positions of the ROSbot derived from the IMU sensor readings, the distance travelled by ROSbot from its old position to its new position can be estimated with the distance formula below:

$$Movement\ Distance\ = \sqrt{(new\ x - old\ x)^2 + (new\ y - old\ y)^2}$$

The cumulative distance travelled by the ROSbot is then estimated by summing all the movement distances.

1.2.2 Total Time Taken

The navigation system calculates the total time from the simulated time data from the *"/clock"* topic published by Gazebo. The time taken is displayed in seconds to enhance human readability.

2. Software Architecture

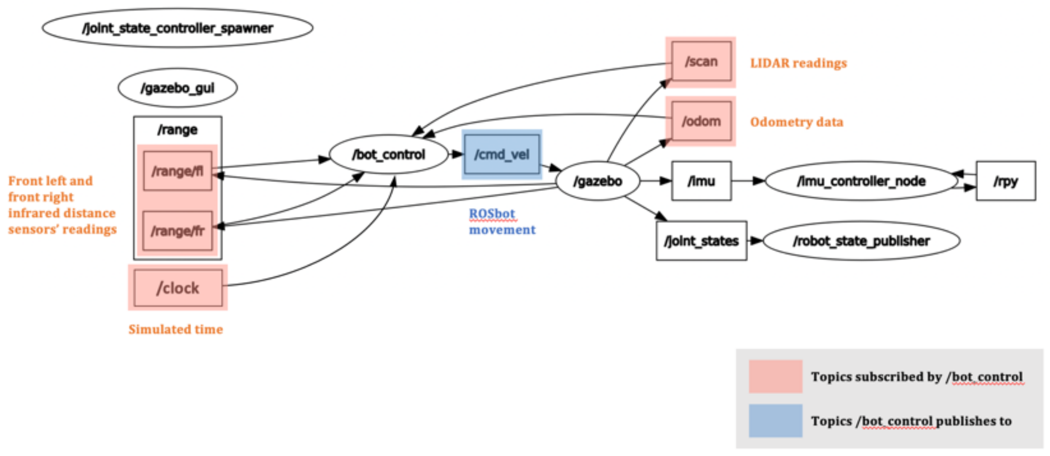
2.1 ROS Architecture

The navigation system is designed as a single ROS node *"/bot_control"*. The node subscribes to five ROS topics:

- ***"/scan"*** topic which publishes the LIDAR readings as an array of size 720. The value of reading ranges from 0.2 to infinity, infinity represents that no obstacle is detected within the scanner's range.
- ***"/range/fl"*** and ***"range/fr"*** which publish the front left and front right infrared distance sensor readings respectively. The value of reading ranges from 0.009 to 0.899, with 0.899 representing no obstacle detected within the sensor's range.
- ***"/odom"*** topic which publishes the odometry data for estimating the cumulative distance travelled.
- ***"/clock"*** topic which publishes the simulated time for displaying the total time taken.

It publishes to only one ROS topic which is ***"/cmd_vel"*** for sending velocity commands that controls the movement of the ROSbot.

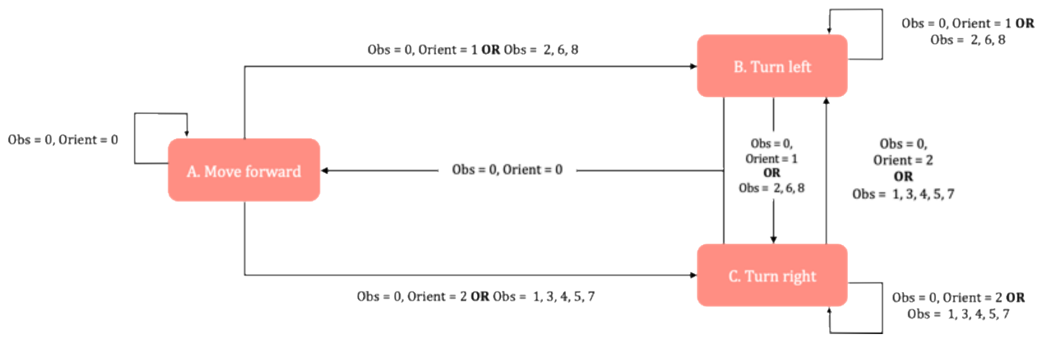
Figure 2.1-1: ROS architecture of the designed navigation system



2.2 Logical Architecture

Figure 2.2-1 is an elaborated version of the State Machine Diagram presented in *Figure 1.1-5*. It shows a more detailed logical architecture of state transition mechanism of the navigation system, i.e. the *"/bot_control"* ROS node. The two major components in the diagram are states and input. The values of input in the diagram are results from the logic flows illustrated in *Figure 2.2-2* and *Figure 2.2-6*.

Figure 2.2-1: Elaborated State Machine Diagram of the navigation system


Obs = 0, Orient = 1 **OR** Obs = 2, 6, 8
Obs = 0, Orient = 1 **OR** Obs = 2, 6, 8
B. Turn left
Obs = 0, Orient = 0
A. Move forward
Obs = 0, Orient = 0
Obs = 0, Orient = 1 **OR** Obs = 2, 6, 8
Obs = 0, Orient = 2 **OR** Obs = 1, 3, 4, 5, 7
C. Turn right
Obs = 0, Orient = 2 **OR** Obs = 1, 3, 4, 5, 7
Obs = 0, Orient = 2 **OR** Obs = 1, 3, 4, 5, 7

2.2.1 States

The states in the *Figure 2.2-1* represent the movement commands to be sent to the ROSbot by the navigation system. There are three states which are **"A. Move forward"**, **"B1. Turn left"** and **"B2. Turn right"**. The states "B1. Turn left" and "B2. Turn right" are elaborated from the "B. Turn" in *Figure 1.1-5*.

2.2.2 Input

The input in the diagram represents the information available by analysing readings from the LIDAR, the two front infrared distance sensor and the IMU sensor. The information that can be obtained through these sensor readings are the existence and proximity of obstacle, as well as the orientation of the ROSbot relative to the starting position.

a. Existence and Proximity of Obstacle

As mentioned in 1.1.2-b, the LIDAR laser scanner is the primary sensor for obstacle detection, while the two front infrared sensors complements the LIDAR scanner in detecting short obstacles that are undetectable by LIDAR. The overall logic and process flow for obstacle detection is illustrated in *Figure 2.2-2*. The system determines whether a right or left turn is needed by analysing the region in front of the ROSbot. Based on reading from the LIDAR scanner, the front region is segmented into three regions, namely "**front**", "**front left**" and "**front right**" (as illustrated in *Figure 2.2-3*). The minimum LIDAR reading from each region, which is the closest distance from an obstacle detected in that region, is used for the obstacle detection flow illustrated in *Figure 2.2-2*. For infrared sensors readings, the front region is segmented into "**front left**" and "**front right**" which are scanned by the front left and front right sensors respectively (as illustrated in *Figure 2.2-4*).

Figure 2.2-2: Flow chart of the obstacle-detection process in the designed navigation system

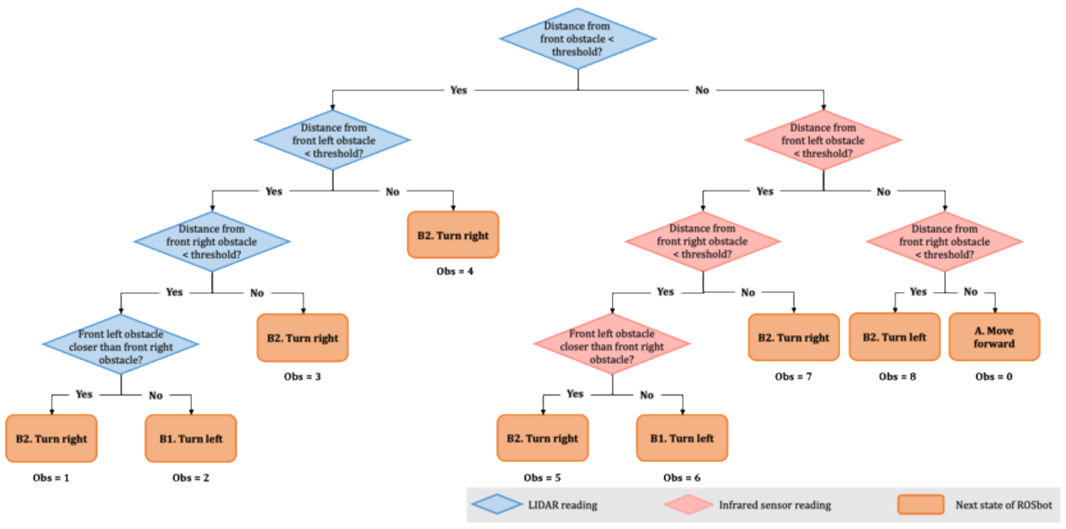

Distance from front obstacle < threshold?
Yes
No
Distance from front left obstacle < threshold?
Distance from front left obstacle < threshold?
Yes
No
Yes
No
Distance from front right obstacle < threshold?
B2. Turn right
Obs = 4
Distance from front right obstacle < threshold?
Distance from front right obstacle < threshold?
Yes
No
Yes
No
Yes
No
Front left obstacle closer than front right obstacle?
B2. Turn right
Obs = 3
Front left obstacle closer than front right obstacle?
B2. Turn right
Obs = 7
B2. Turn left
Obs = 8
A. Move forward
Obs = 0
Yes
No
Yes
No
B2. Turn right
Obs = 1
B1. Turn left
Obs = 2
B2. Turn right
Obs = 5
B1. Turn left
Obs = 6
LIDAR reading
Infrared sensor reading
Next state of ROSbot

Figure 2.2-3: Front region segmentation from LIDAR reading

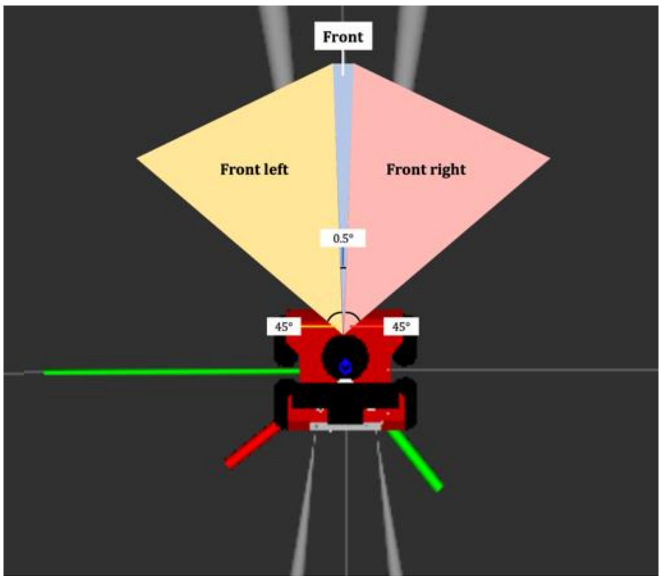
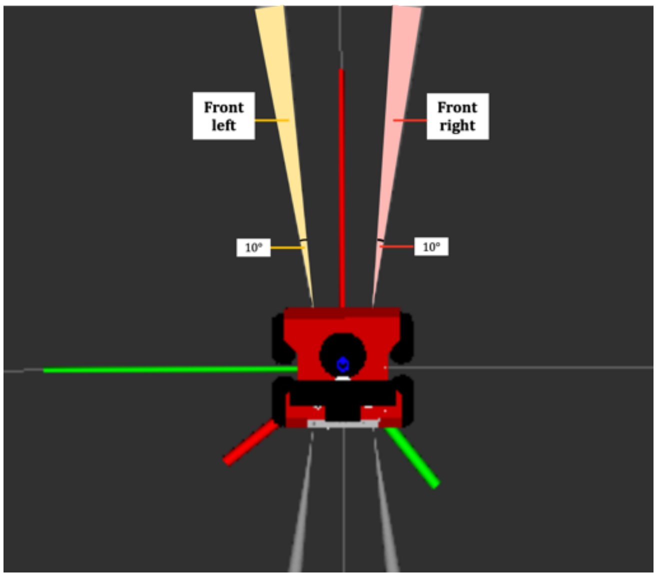


Figure 2.2-4: Front region segmentation from infrared sensor readings



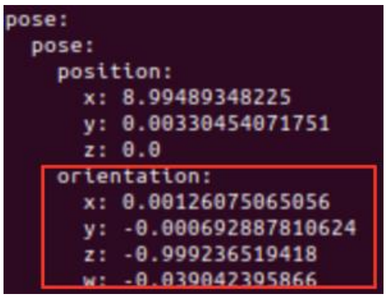
b. <u>Orientation of the ROSbot Relative to the Starting Orientation</u>

The ROSbot always face towards the finish line at the beginning of a simulation, hence, the starting position is used as the baseline for deriving the orientation of the ROSbot. The orientation of the ROSbot is determined from the IMU sensor readings. The orientation provides information about the current heading direction of the ROSbot and allow the system to determine whether steering angle adjustment is needed to drive the ROSbot towards the destination.

The IMU sensor provides data about the quaternion of the ROSbot. The quaternion of the ROSbot is published at the *"/odom"* ROS topic under orientation as shown in *Figure 2.2-5*. As the orientation of interest is the rotation along the yaw axis, only the values z and w are used in determining orientation adjustment needed.

Table 2.2-1 lists the z and w quaternion values of ROSbot when different orientation adjustments are needed to make the ROSbot face towards the destination finish line, i.e. same orientation as the starting orientation. From the values listed, it is found that when the absolute value of z is close to 1, no adjustment is needed. As the absolute value of z decreases, left or right adjustments need to be made. It is also noticed that when an orientation adjustment is needed, the sign of z*w determines whether left or right adjustment is needed. When z*w is a positive number, left adjustment is needed, vice versa. To avoid dithering, a threshold is set to allow for a certain extent of deviation of absolute z value from 1. *Figure 2.2-6* summaries the abovementioned logic.

Figure 2.2-5: Sample quaternion reading from "/odom" ROS topic

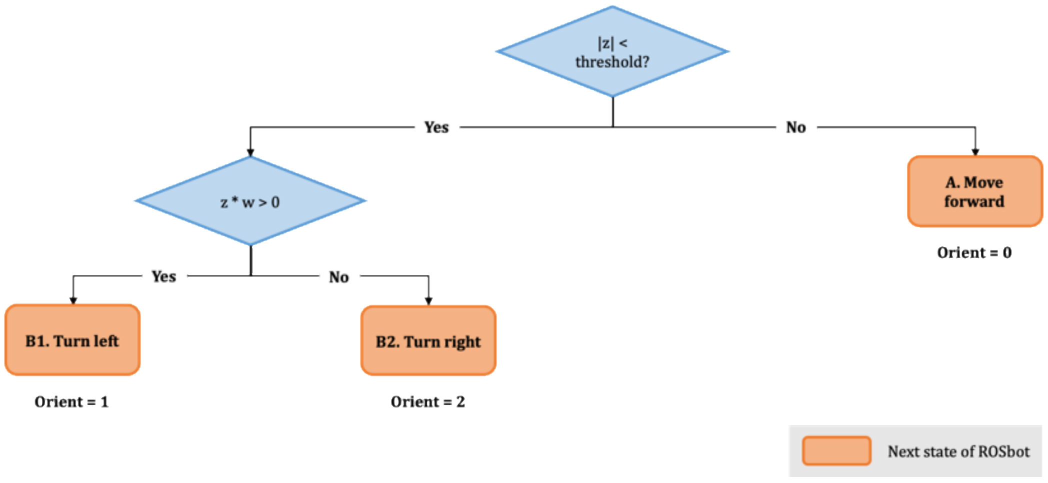


```
pose:
  pose:
    position:
      x: 8.99489348225
      y: 0.00330454071751
      z: 0.0
    orientation:
      x: 0.00126075065056
      y: -0.000692887810624
      z: -0.999236519418
      w: -0.039042395866
```

Table 2.2-1: Quaternion values when different orientation adjustments are needed

Adjustment needed to face towards the finish line	Quaternion values	
	z	w
No	-0.99924	-0.03904
Turn right	-0.95484	0.297134
Turn right	-0.72677	0.68688
Turn left	-0.95888	-0.28381
Turn left	-0.69768	-0.7164

Figure 2.2-6: Logic flow for determining orientation adjustment

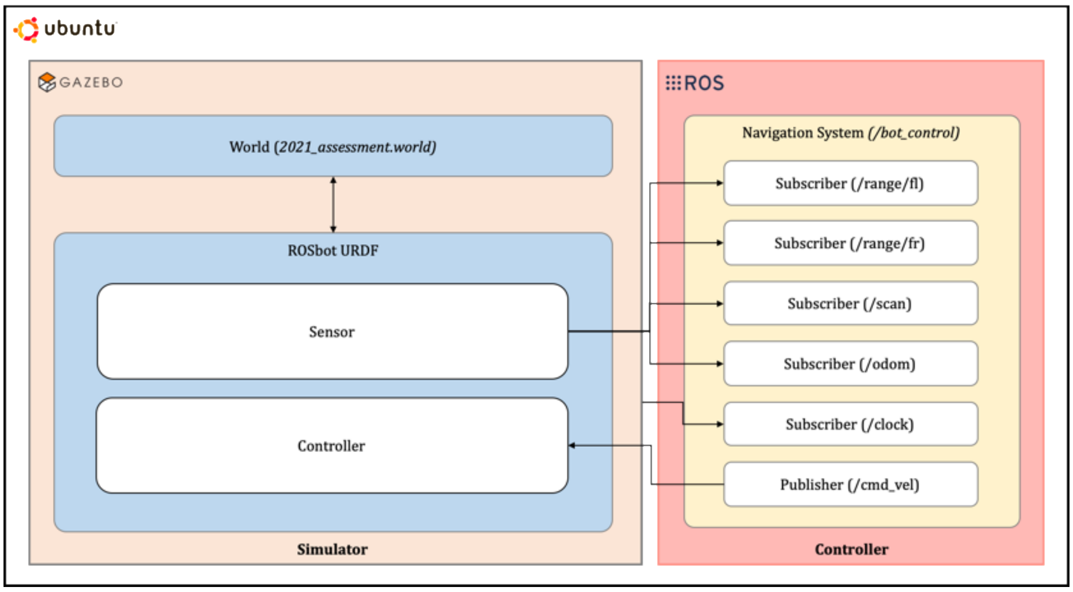


2.3 Structural Architecture

2.3.1 Interaction Structure

Figure 2.3-1 illustrates the overall interaction structure between the navigation system designed and the Gazebo simulator. The ROSbot URDF model is put into the simulation world. The navigation system subscribes to topics published by the sensor plugins to obtain sensor data, and subscribes to the *"/clock"* topic published by Gazebo. Based on the data collected by sensor reading subscribers, the navigation system publishes velocity commands to the controller plugin to move the ROSbot model in the simulation world.

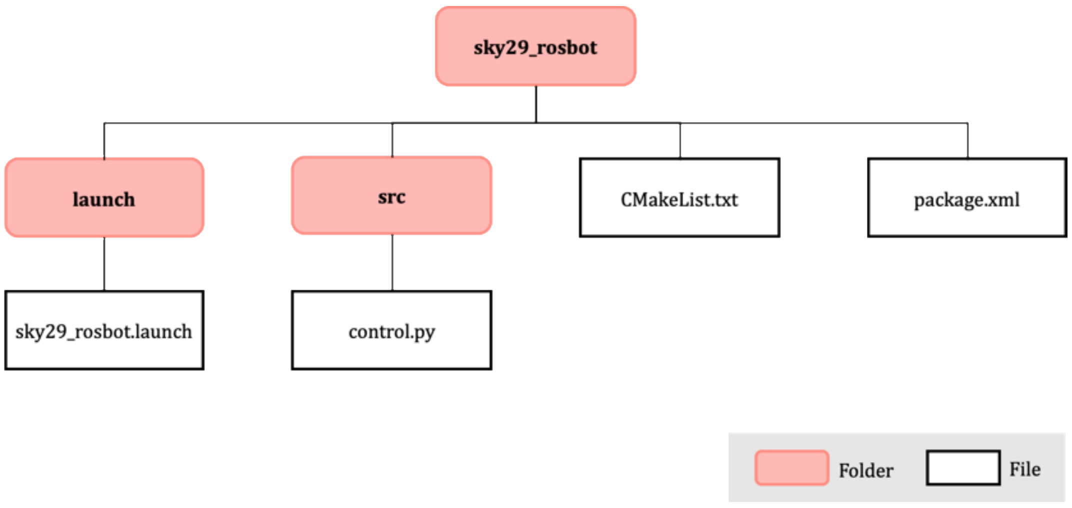
Figure 2.3-1: Structure of interactions between the designed navigation system and the Gazebo simulator



2.3.2 File Structure

The navigation system is built with the catkin build system as a single ROS package named "**sky29_rosbot**". The package is placed in the source space of the catkin workspace "**~ros_workspace/src**". The file structure of the package is illustrated in *Figure 2.3-2*. The script "**control.py**" contains all the logic of the system. The system can be launched with the launch file "**sky29_rosbot.launch**".

Figure 2.3-2: File structure of the navigation system



3. Software Implementation

3.1 Dependencies

3.1.1 ROS Package Dependencies of the System

Figure 3.1-1 illustrates the ROS package dependencies of the navigation system ROS package "**sky29_rosbot**". *Table 3.1-1* lists some brief descriptions of the use of these packages.

Figure 3.1-1: Dependency graph of the navigation system package

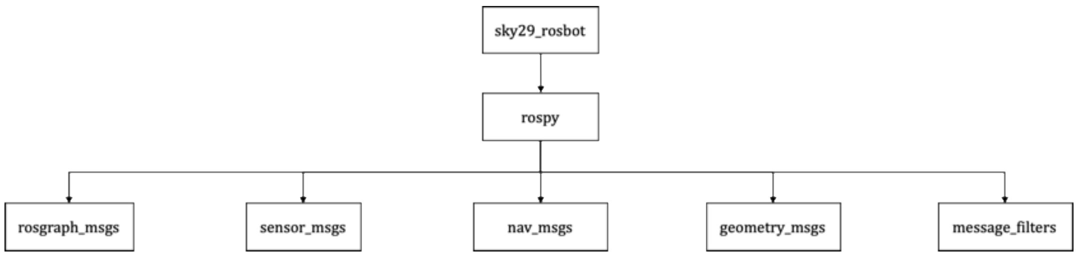


Table 3.1-1: Descriptions of the ROS packages used in the navigation system

ROS Package	Description
"rospy"	The pacakage provides a Python client library for interacting with ROS in Python.
"rosgraph_msgs"	The package contains the "**Clock**" message type which the simulated time is published in.
"sensor_msgs"	The package contains the "Range" and "LaserScan" message types which the LIDAR and infrared sensor readings are published in.
"nav_msgs"	The package contains the "Odometry" message type which is used by the "/odom" topic in publishing data such as position and orientation of the ROSbot.
"geometry_msgs"	The package contains the "Twist" message type which the "/cmd_vel" topics take for controlling the movement of the ROSbot.
"message_filters"	The package contains the "TimeSynchronizer" filter for synchronising messages received different ROS topic subscribers by their timestamps. This allows handling of multiple messages received from different ROS topics with a single callback function.

3.1.2 Python Module Dependencies of the System

Apart from the ROS packages listed in 3.1.1, the system also uses the Python **math** module for the calculation of total distance travelled.

3.2 Implementation of the Navigation Controlling Logic

All the logics of the controller described in 2.2 are implemented in the Python Script "**control.py**". The script can be divided into five components. The following sections discuss the logic implemented in each of the component in detail.

Refer to Appendix I: Source Code for the complete source code the Python controller script and the launch file for launching the script.

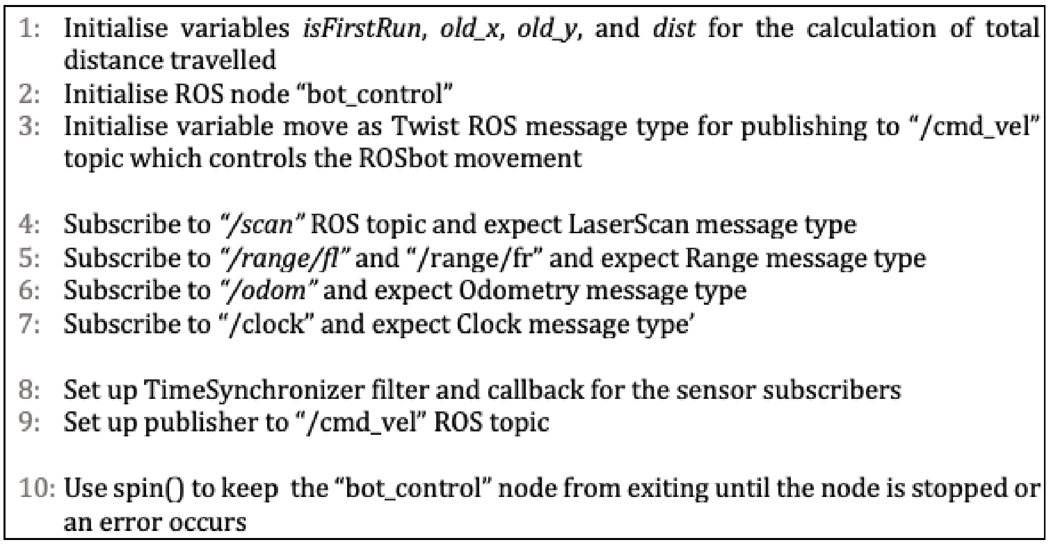
3.2.1 The Main Function

This is the starting point of the Python script. It initialises the following:

- the ROS node *"bot_control"*
- the variables to be used in other functions
- the ROS topic subscribers for receiving messages to be used in other functions
- the ROS topic publisher for sending velocity command to ROSbot

Figure 3.2-1 shows the pseudocode of the function.

Figure 3.2-1: Pseudocode of the main function



```
1:  Initialise variables isFirstRun, old_x, old_y, and dist for the calculation of total
    distance travelled
2:  Initialise ROS node "bot_control"
3:  Initialise variable move as Twist ROS message type for publishing to "/cmd_vel"
    topic which controls the ROSbot movement

4:  Subscribe to "/scan" ROS topic and expect LaserScan message type
5:  Subscribe to "/range/fl" and "/range/fr" and expect Range message type
6:  Subscribe to "/odom" and expect Odometry message type
7:  Subscribe to "/clock" and expect Clock message type'

8:  Set up TimeSynchronizer filter and callback for the sensor subscribers
9:  Set up publisher to "/cmd_vel" ROS topic

10: Use spin() to keep  the "bot_control" node from exiting until the node is stopped or
    an error occurs
```

3.2.2 The Simulation Time Display Function

The function is the callback function of the *"/clock"* topic subscriber. It calculates the difference between the current simulated time published by the topic publisher and the simulated time when the node was first activated, then prints the result on screen. *Figure 3.2-2* shows the pseudocode of the function.

Figure 3.2-2: Pseudocode of the simulation time display function

```
INPUT: message received from "/clock" topic in "Clock" type
1:  Initialise the variables start_time and time
2:  If the function is being called the first time during the simulation:
3:      start_time = current simulated time
4:      total_time = current simulated time

5:  total_time = current simulated time – start_time
6:  Print total_time on screen
```

3.2.3 Sensor Reading Consolidation Function

This function is the callback function of the TimeSynchronizer filter of the sensor topic subscribers. It retrieves the useful data from the messages received by the subscribers, processes them and pass them to other functions in the script for calculating cumulative distance travelled and determines the ROSbot's next state. *Figure 3.2-3* shows the pseudocode of the function.

Figure 3.2-3: Pseudocode of the sensor reading consolidation function

```
INPUT:
1) message received from "/scan" topic in "LaserScan" type;
2) message received from "/range/fl" topic in "Range" type;
3) message received from "/range/fr" topic in "Range" type; and
4) message received from "/odom" topic in "Odometry" type

1:  Calls the cumulative distance travelled calculation function [3.2.4]
2:  Retrieve front left, front and front right LIDAR readings from the "/scan" message
3:  Retrieve the z and w quaternion values from the "/odom" message
4:  Calls the movement determination function [3.2.5]
```

3.2.4 Cumulative Distance Calculation Function

This function estimates the total distance travelled by the ROSbot with the distance formula mentioned in 1.2.1 and displays the result on screen. The x and y values used in the calculation is rounded to two decimal places to eliminate sensor noises. *Figure 3.2-4* shows the pseudocode of the function.

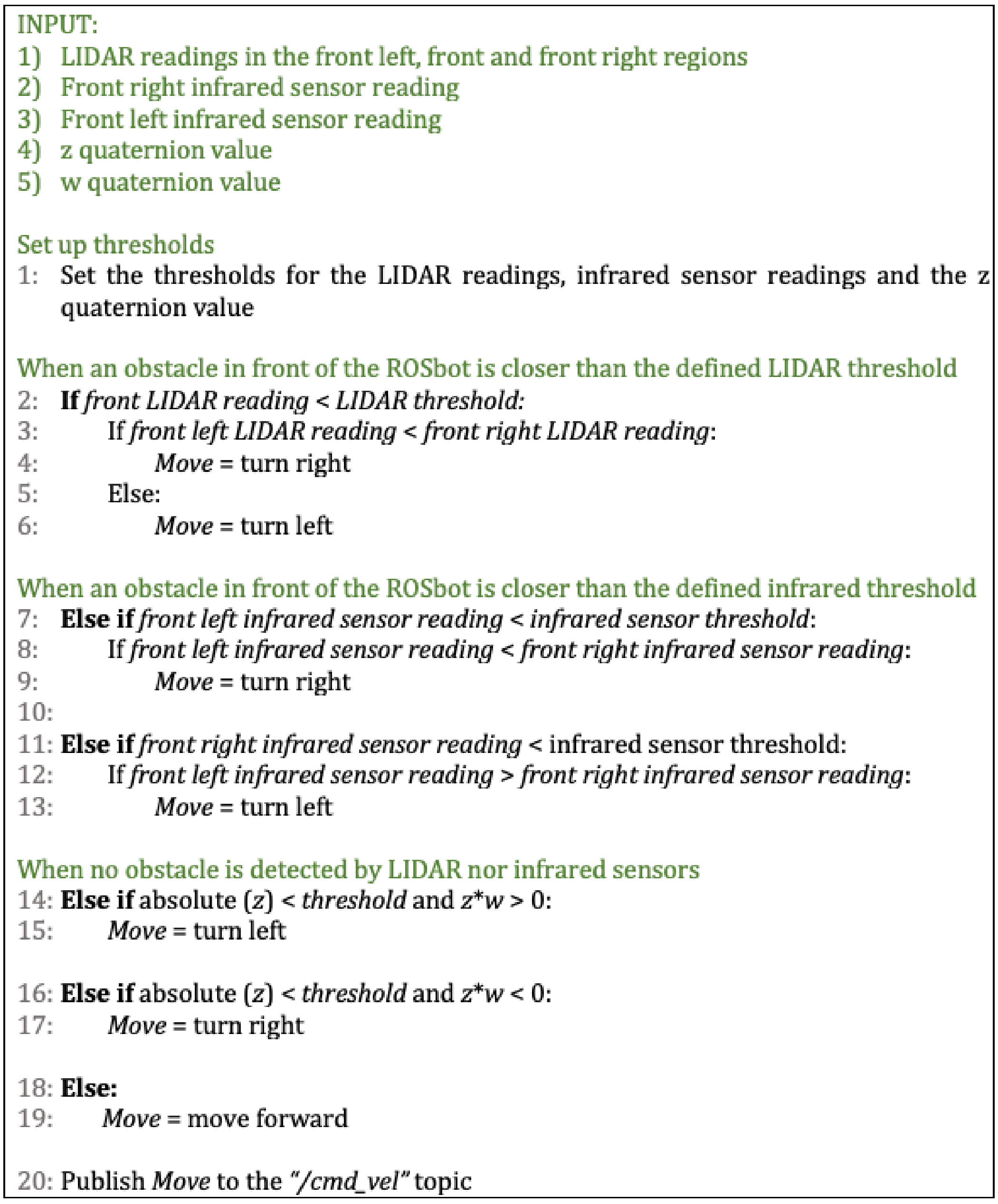
Figure 3.2-4: Pseudocode of the cumulative distance calculation function

```
INPUT: message received from "/odom" topic in "Odometry" type

1:  If the function is being called the first time during the simulation:
2:      old_x = current x position of ROSbot retrieved from "/odom" topic
3:      old_y = current y position of ROSbot retrieved from "/odom" topic

4:  new_x = current x position of ROSbot retrieved from "/odom" topic
5:  new_y = current y position of ROSbot retrieved from "/odom" topic

6:  distance = distance between old and new position using the distance formula
7:  Cumulative distance travelled = Cumulative distance travelled + distance

8:  Print Cumulative distance travelled on screen
```

3.2.5 Movement Determination Function

This function implements the logic described in section 2.2. The thresholds used in the function are selected by the method of trial-and-error based on the testing result in section 5. *Figure 3.2-5* shows the pseudocode of the function.

Figure 3.2-5: Pseudocode of the movement determination function



```
INPUT:
1) LIDAR readings in the front left, front and front right regions
2) Front right infrared sensor reading
3) Front left infrared sensor reading
4) z quaternion value
5) w quaternion value

Set up thresholds
1:  Set the thresholds for the LIDAR readings, infrared sensor readings and the z
    quaternion value

When an obstacle in front of the ROSbot is closer than the defined LIDAR threshold
2:  If front LIDAR reading < LIDAR threshold:
3:      If front left LIDAR reading < front right LIDAR reading:
4:          Move = turn right
5:      Else:
6:          Move = turn left

When an obstacle in front of the ROSbot is closer than the defined infrared threshold
7:  Else if front left infrared sensor reading < infrared sensor threshold:
8:      If front left infrared sensor reading < front right infrared sensor reading:
9:          Move = turn right
10:
11: Else if front right infrared sensor reading < infrared sensor threshold:
12:     If front left infrared sensor reading > front right infrared sensor reading:
13:         Move = turn left

When no obstacle is detected by LIDAR nor infrared sensors
14: Else if absolute (z) < threshold and z*w > 0:
15:     Move = turn left

16: Else if absolute (z) < threshold and z*w < 0:
17:     Move = turn right

18: Else:
19:     Move = move forward

20: Publish Move to the "/cmd_vel" topic
```

3.3 Launch Process

To simulate the ROSbot navigation, both the Gazebo simulation environment and the navigation system need to be launched. The launch process of the two components are described in sections below.

3.3.1 Launch Gazebo Simulation

The “**rosbot_bath**” package contains all the components required to launch the Gazebo simulation. The package is stored under the directory **“~/ros_workspace/src**”. The process to launch the simulation is as follows:

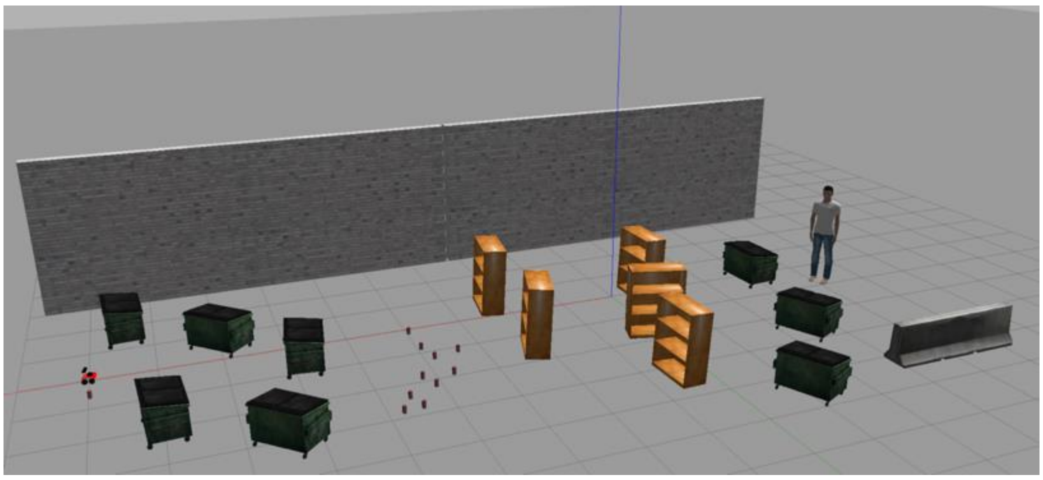
i. Open a new terminal window and source the catkin workspace
source ~/ros_workspace/devel/setup.sh

ii. Move to the “**rosbot_bath**” directory
cd ~/ros_workspace/src/rosbot_bath

iii. Launch the simulation with the shell script “run_rosbot.sh”
./run_rosbot.sh

iv. The Gazebo simulator should be launched. The obstacle course together with the ROSbot located next to a coke can should be displayed (as shown in *Figure 3.3-1*).

Figure 3.3-1: The obstacle course and initial position of ROSbot after launching the Gazebo simulation



3.3.2 Launch Navigation Controller

The “**sky29_rosbot**” package contains all components of the navigation controller and is also placed under the directory **“~/ros_workspace/src**”. With the simulation running, the controller can be launched with the following steps:

i. Open a new terminal window and source the catkin workspace
source ~/ros_workspace/devel/setup.sh

ii. Launch the navigation controller with the launch file “**sky29_rosbot.launch**”
roslaunch sky29_rosbot sky29_rosbot.launch

4. Testing Approach

4.1 Testing Objective

The objective of the testing performed is to find the optimal values for the following parameters which are used for deciding the next state of the ROSbot:

- **Orientation threshold**: The threshold for deviation allowed from forward movement identified by IMU sensor before a steering adjustment is made according to the logic illustrated in 2.2.2-b.
- **LIDAR threshold**: The threshold for the closest distance allowed between an obstacle and the ROSbot identified by LIDAR sensor before a steering adjustment is made according to the logic illustrated in 2.2.2-a.
- **IR threshold**: The threshold for the closest distance allowed distance between an obstacle and the ROSbot identified by infrared sensors before a steering adjustment is made according to the logic illustrated in 2.2.2-a.

4.2 Testing Environment

The testing is performed within the Gazebo simulator. The obstacle course used for testing is defined in the Gazebo world file "*2021_assessment.world*" within the "*rosbot_bath*" ROS package.

4.3 Performance Evaluation Criteria

The performance of the navigation system is evaluated with the following criteria:

- Total distance travelled by ROSbot to reach the destination (α_1)
- Total time taken to reach the destination (α_2)
- Number of obstacles hit (α_3)
- Occurrence of manual intervention, for instance, when the ROSbot is stuck or tipped over (α_4)

A performance score is computed with the multipliers listed in *Table 4.3-1*, which is similar to the score sheet used in the lab competition held. The lower the score, the better the system performs. The performance score is calculated according to the formula below:

$$score = \alpha_1 \times \beta_1 + \alpha_2 \times \beta_2 + \alpha_3 \times \beta_3 + \alpha_4 \times \beta_4$$

Table 4.3-1: Multipliers of evaluation critera for performance score calculation

Evaluation Criteria	**Multiplier ID**	**Multiplier Value**
Total distance travelled	β_1	3
Total time taken	β_2	1
Number of obstacles hit	β_3	20
Occurrence of manual intervention	β_4	25

4.4 Testing Tools

Bash scripts are used to automate the testing process. The scripts automatically substitute the values of parameters to be tested in “control.py” and output the total distance travelled and time taken as text files. The number of obstacles hit and occurrence of manual intervention are measured manually.

5. Test Results

5.1 Test Cases

A total of two trials were performed in the testing. In each of the trial, different values of orientation threshold, LIDAR threshold and IR threshold are used. The values tested in the second trial are chosen based on the findings in the first trial. The values tested in each trial are listed in *Table 5.1-1*.

Table 5.1-1: Values of parameter used in testing

	Trial	
Parameter	**1**	**2**
Orientation threshold	0.85, 0.90, 0.95	0.85, 0.95
LIDAR threshold	1.00, 1.50, 2.00	1.00, 1.50
IR threshold	0.55, 0.65, 0.75	0.75, 0.55

5.2 Test Results

5.2.1 Results of the First Trial

a. Findings

A total of 27 runs were performed with three different values for each of the parameter tested. The navigation system was able to drive the ROSbot to arrive at the finish line in all the 27 runs, which confirms the effectiveness of the navigation logic described in 2.2. The results of this trial are listed in *Table 8.2-1*.

By analysing the results of the runs, the following were observed:

- **A higher value of orientation threshold results in better performance** (*Figure 5.2-1*): According to the Pythagoras' Theorem, the shortest trajectory for the ROSbot to reach the finish line would be a trajectory that is perpendicular to the finish line, as illustrated in *Figure 5.2-4*. However, a lower value of orientation threshold leads to a lower tendency for the system to return the heading direction of ROSbot to a position that is more perpendicular to the finish line, which results in a longer path. Based on this observation, the orientation threshold at 0.97 was tested for a few runs, yet the ROSbot failed to complete the course either due to dithering or tipping over after hitting obstacle. This might be a result of obstacle avoiding adjustments being cancelled out by the orientation adjustments before an obstacle has been avoided successfully. It is therefore concluded that 0.95 is the maximum and optimal orientation threshold for the system.

- **Minimal difference in performance caused by the value of LIDAR threshold** (*Figure 5.2-2*): The overall performance of the system does not change significantly with the three values being tested. The value 1.50 results in a slightly better performance than the other two values.

- **A lower value of IR threshold results in better performance** (*Figure 5.2-3*): With a high value of IR threshold, the system starts avoiding the coke cans when the ROSbot is still very far from them and resulting in a longer distance travelled and time taken. This shows that with a high value of IR threshold, the system becomes too sensitive to short objects and might overlook possible shorter trajectories. Based on this finding, the value 0.50 was tested for a few runs, however, the system become more likely to hit the coke cans or dither as it only reacts to the coke cans until the ROSbot is at a position very close to them (as shown *Figure 5.2-5*). It is therefore decided that 0.55 is the minimum and optimal threshold for the system.

Figure 5.2-1: Effect of Orientation Threshold on System Performance

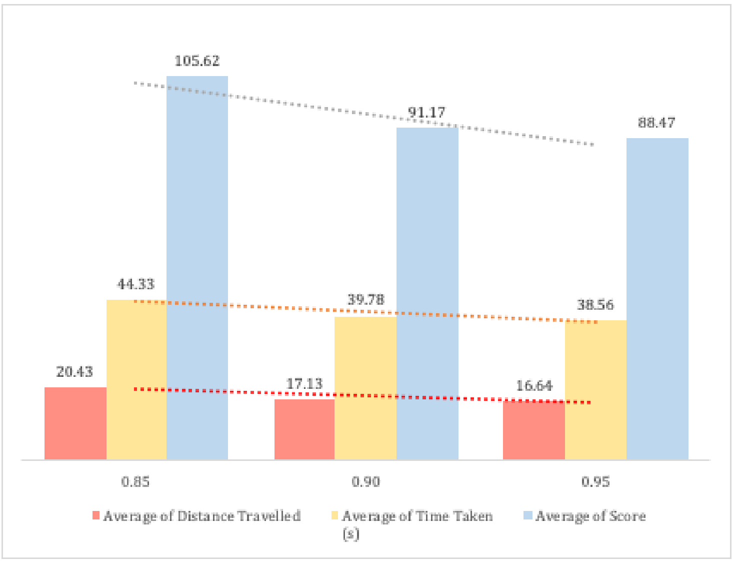


Figure 5.2-2: Effect of LIDAR Threshold on System Performance

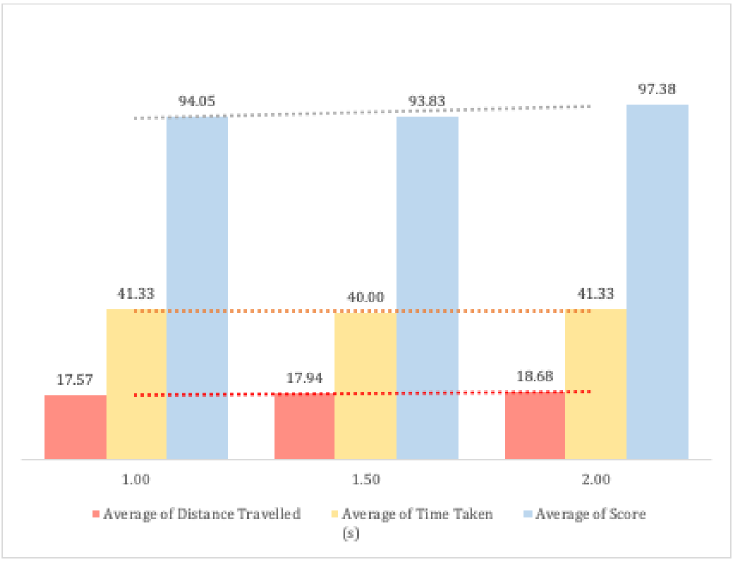


Figure 5.2-3: Effect of IR Threshold on System Performance

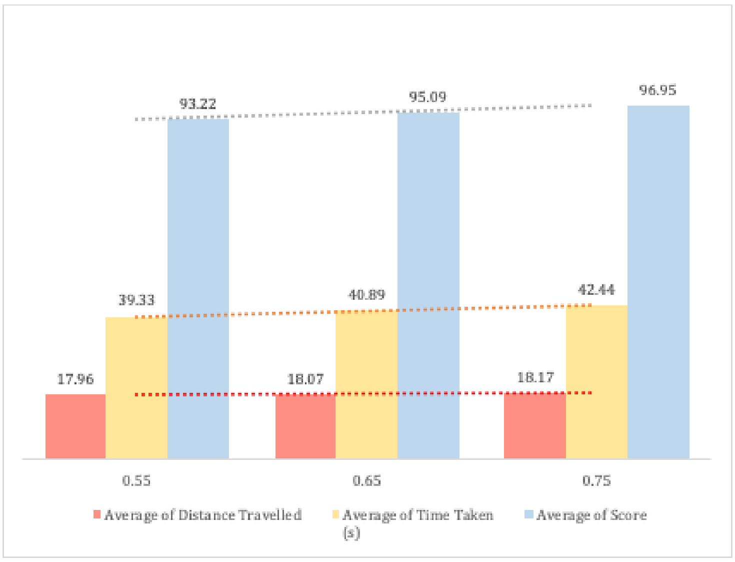


Figure 5.2-4: Optimal trajectory and trajectory chosen with low orientation threshold

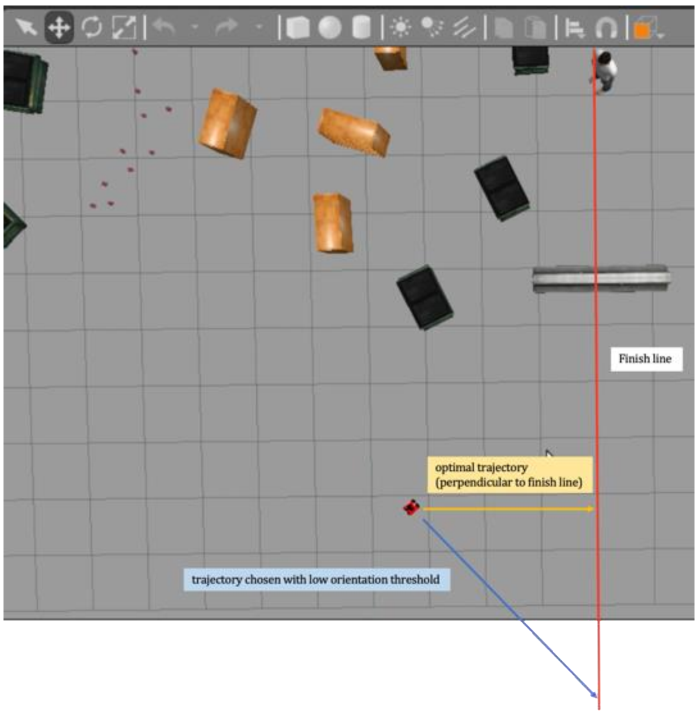
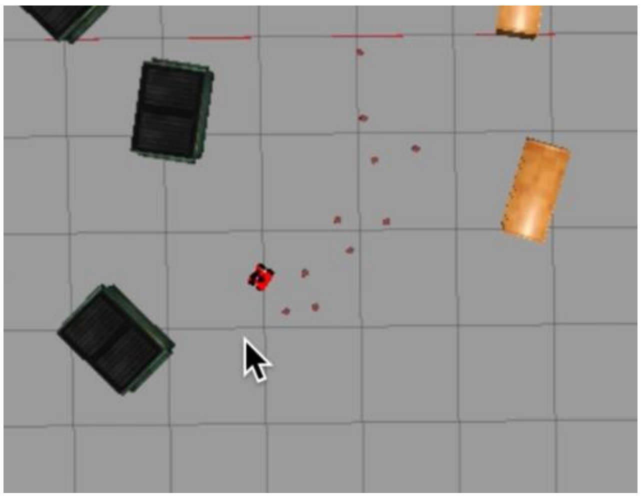

Finish line
optimal trajectory (perpendicular to finish line)
trajectory chosen with low orientation threshold

Figure 5.2-5: Reaction location of system to short obstacles with IR threshold = 0.50



b. Parameter Values to be Tested in the Second Trial

Among the 27 runs performed in this trial, run 3 with orientation threshold 0.85, LIDAR threshold 1.00 and IR threshold 0.75 results in the best performance. However, these values do not agree with the observations made in a. Hence, one more trial is performed to compare the performances of the system with parameter values the same as the best run in the first trial, and with parameter values determined according to findings in a.

5.2.2 Results of the Second Trial

a. <u>Findings</u>

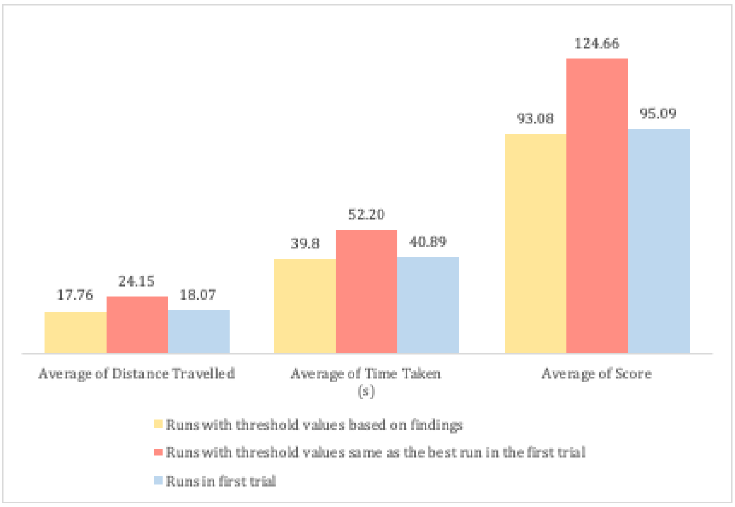
The following two sets of parameter values are tested in this trial:

- Orientation threshold: 0.95; LIDAR threshold: 1.50; IR threshold: 0.55
- Orientation threshold: 0.85; LIDAR threshold: 1.00; IR threshold: 0.75

For each set of parameter values, five runs were performed. The results of the ten runs are listed in *Table 8.2-2*.

Upon testing, it is found that the average performance of the first set of values (i.e. parameter values chosen according to findings) is significantly better than that of the second set of values (i.e. parameter values of the best run in the first trial). Although the second set of values results in a run with the best performance among the 10 runs performed, it also results in the worst run. The first set of values is therefore more preferable in order to achieve a more stable performance. The first set of parameter values also has a higher average performance score than that of all the runs performed in first trial.

Figure 5.2-6: System performances in the second trial of testing



b. <u>Final parameter values chosen for the system</u>

Based on the testing result, the orientation threshold 0.95, LIDAR threshold 1.50 and IR threshold 0.55 are chosen as the parameter values for the system.

6. Conclusion and Further Work

6.1 Conclusion

To conclude, a biologically-inspired reactive autonomous navigation system is designed for this robot challenge. The system has three states and the transition between states are determined by the obstacle avoidance and orientation mechanisms of the system. The threshold values used for these mechanisms are tested and determined using a trial-and-error approach over a total of 37 runs.

6.2 Further Work

6.2.1 Testing on Physical Robot

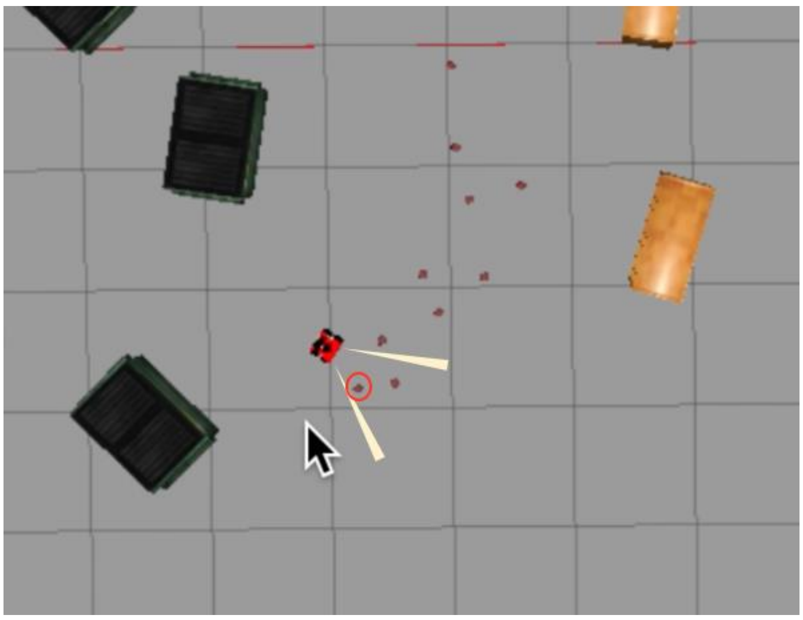
Unfortunately, the navigation system was not tested on the physical ROSbot due to insufficient time for altering the code to adapt to the physical ROSbot environment. It is noticed that the ROS topics published in the Gazebo environment and the physical ROSbot environment do not fully align. Hence, alterations such as re-mapping of ROS topics, developing a new system for total distance travelled and simulation time measurements are required in order to deploy the code onto the physical ROSbot.

6.2.2 Use of RGBD Camera

At the design phase, it was decided not to deploy the RGBD camera as the LIDAR and infrared sensors have the technical ability to detect all obstacles on the course. However, during testing, it is found that due to the limited FoV of the infrared sensors, the system fails to search for a trajectory when the IR threshold set to a low value or when short obstacles are located very close to the ROSbot and are not in the front infrared sensors' FoV. In one of the runs performed for testing the IR threshold value of 0.50, when the ROSbot was passing through the coke can mine on the obstacle course, a low value of IR threshold caused the system to react to the coke cans only when it reaches at the position indicated in *Figure 6.2-1*. As the infrared sensor fails to detect any obstacle at this position, the navigation believes the forward path is clear and sends command to the ROSbot to move forward, which caused the ROSbot to hit the coke can and tip over. Thus, it might be helpful to incorporate readings from the RGBD camera which has a wider FoV into the obstacle avoidance logic to have a more thorough analysis of the environment.

Apart from the ability to detect objects, the RGBD camera is also capable of object classification, which cannot be performed with LIDAR or infrared sensors. However, as object classification significantly increases the complexity of the navigation system, it could affect the react time of the system to an obstacle (Bolanos et al., 2006). Further experiment is required to evaluate whether the decline in performance is justifiable by the value of information gain.

Figure 6.2-1: Location of ROSbot when infrared sensor fails to detect obstacle



6.2.3 Dynamic Speed and Steering Angle Adjustments

Due to time constraint, the speed of movement and the steering angle adjustment for the turning states are held constant. In terms of movement speed, as total time taken is one of the performance evaluation criteria, it would be sensible to speed up the ROSbot when the likelihood of collision is minimal, for instance, when a direct line of sight path between the ROSbot and the finish line is observed. In terms of steering angle adjustment, a static angle adjustment is likely to result in over or under adjustment. There have been studies showing the achievement of near-optimal navigation path with the use of artificial neural network and fuzzy logic representation for steering control during obstacle avoidance (Gharajeh and Jond, 2022; Pandey, 2016; Marichal et al., 2001). Further effort can be spent on exploring the implementation of these techniques in the system.

7. Bibliography

Arkin, R.C. (1998). *Behavior-based robotics*. Cambridge, Mass.: MIT Press.

Bolanos, J.M., Meléndez, W.M., Fermín, L., Cappelletto, J., Fernández-López, G. and Grieco, J.C. (2006). Object Recognition for Obstacle Avoidance in Mobile Robots. *Artificial Intelligence and Soft Computing – ICAISC 2006*, pp.722–730.

El-Sheimy, N. and Li, Y. (2021). Indoor navigation: state of the art and future trends. *Satellite Navigation*, 2(1).

Gharajeh, M.S. and Jond, H.B. (2022). An intelligent approach for autonomous mobile robots path planning based on adaptive neuro-fuzzy inference system. *Ain Shams Engineering Journal*, 13(1), p.101491.

Husarion (n.d.). *ROSbot 2.0 & ROSbot 2.0 PRO*. [Online] Available at: https://husarion.com/manuals/rosbot/ [Accessed 27 Dec. 2021].

Lohar, S., Zhu, L., Young, S., Graf, P. and Blanton, M. (2021). Sensing Technology Survey for Obstacle Detection in Vegetation. *Future Transportation*, 1(3), pp.672–685.

Marichal, G.N., Acosta, L., Moreno, L., Méndez, J.A., Rodrigo, J.J. and Sigut, M. (2001). Obstacle avoidance for a mobile robot: A neuro-fuzzy approach. *Fuzzy Sets and Systems*, 124(2), pp.171–179.

Mouser Electronics (2018). *STMicroelectronics VL53L1X Time-of-Flight Proximity Sensor*. [online] Mouser Electronics. Available at: https://www.mouser.co.uk/new/stmicroelectronics/stm-tof-proximity-sensor/ [Accessed 29 Dec. 2021].

Pandey, A. (2016). Cascade Neuro-Fuzzy Architecture Based Mobile- Robot Navigation and Obstacle Avoidance in Static and Dynamic Environments. *International Journal of Advanced Robotics and Automation*, 1(3), pp.1–9.

Wang, J. and Tepfenhart, W.M. (2020). *Formal methods in computer science*. Boca Raton London New York Crc Press, Taylor & Francis Group.

Wortham, R.H., Gaudl, S.E. and Bryson, J.J. (2019). Instinct: A biologically inspired reactive planner for intelligent embedded systems. *Cognitive Systems Research*, 57, pp.207–215.

8. Appendix

8.1 Appendix I: Source Code

8.1.1 Launch File “sky29_rosbot.launch”

```
<launch>

<node name = "bot_control" pkg = "sky29_rosbot" type = "control.py" output =
"screen" />

</launch>
```

8.1.2 Python Controller Script “control.py”

```
#! /usr/bin/env python

import rospy
from sensor_msgs.msg import Range
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from rosgraph_msgs.msg import Clock
import math
import message_filters
from sensor_msgs.msg import LaserScan

def sensor_callback(lidar_msg, odom_msg, fl_msg, fr_msg):

  x = odom_msg.pose.pose.position.x

  if x > -4.178339:
    calc_dist_travelled(odom_msg)

    regions = {
      'a': lidar_msg.ranges[0],
      'b': min(lidar_msg.ranges[89:179]),
      'd': min(lidar_msg.ranges[539:629])
    }

    y_pos = odom_msg.pose.pose.position.y
    z = odom_msg.pose.pose.orientation.z
    w = odom_msg.pose.pose.orientation.w

    movement_determination(regions, z, w, fl_msg.range, fr_msg.range)

  else:
    global dist
    global time
    move.linear.x = 0
    move.angular.z = 0
    pub.publish(move)
```

```
        print("reached destination dist: " + str(dist) + " time:" +str(time))
        rospy.signal_shutdown("reached destination dist: " + str(dist) + "time:" +str(time))

def movement_determination (regions, z, w, fl, fr):

    orient_thres = 0.95
    lidar_thres = 1.5
    ir_thres = 0.55

    # if there is obtacle in front and detectable by laser scanner
    if regions['a'] < lidar_thres:
        move.linear.x = 0
        move.angular.z = 0
        pub.publish(move)

        #if left obtacle closer than right obstacle, turn right
        if regions['b'] < regions['d']:
            print("left obstacle is closer than right obstacle, turn right")
            move.linear.x = 0
            move.angular.z = -1

        #if right obstacle closer than or equal to right obstacle, turn right
        else:
            print("right obstacle closer than/equal to left obstacle, turn left")
            move.linear.x = 0
            move.angular.z = 1

    # if there is obstacle in front detected by infrared sensor
    elif fl < ir_thres:
        if fl < fr:
            print("front left obstacle detected by infrared, turn right")
            move.linear.x = 0
            move.angular.z = -1

    elif fr < ir_thres:
        if fr < fl:
            print("front right obstacle detected by infrared, turn left")
            move.linear.x = 0
            move.angular.z = 1

    else:
        if abs(z) < orient_thres and z*w > 0 and regions['b'] > lidar_thres:
            print("turning left to face towards goal")
            move.linear.x = 0.0
            move.angular.z = 1

        elif abs(z) > orient_thres and z*w < 0 and regions['d'] > lidar_thres:
            print("turning right to face towards goal")
            move.linear.x = 0.0
            move.angular.z = -1

        else:
            print("no obstacle, move forward")
            move.linear.x = 0.5
        move.angular.z = 0

    pub.publish(move)


def calc_dist_travelled(msg):
    global isFirstRun
    global dist
    global old_x
    global old_y

    if isFirstRun ==  True:
        old_x = round(msg.pose.pose.position.x*100.0)/100.0
        old_y = round(msg.pose.pose.position.y*100.0)/100.0

    new_x = round(msg.pose.pose.position.x*100.0)/100.0
    new_y = round(msg.pose.pose.position.y*100.0)/100.0

    dist = dist + math.sqrt((new_x - old_x)**2 + (new_y - old_y)**2)
    old_x = new_x
    old_y = new_y
    isFirstRun = False

    print("Distance travelled: " + str(dist))

def clock_callback(msg):
    global start_time
    global time
    global isFirstRun

    if isFirstRun == True :
        start_time = msg.clock.secs
        time = msg.clock.secs

    if time != msg.clock.secs:
        time = msg.clock.secs - start_time

        print("Simulation Time: " + str(time) + " secs")


if __name__ == '__main__':

    global isFirstRun
    isFirstRun = True
    global atDest
    atDest = False
    global old_x
    old_x = 0.0
    global old_y
    old_y = 0.0
    global dist
    dist = 0.0

    rospy.init_node('bot_control')
    move = Twist()

    sub_fl = message_filters.Subscriber('/range/fl', Range)
    sub_fr = message_filters.Subscriber('/range/fr', Range)
    sub_odom = message_filters.Subscriber('/odom',  Odometry)
    sub_clock = rospy.Subscriber('/clock', Clock, clock_callback)

    laser_sub = message_filters.Subscriber('/scan', LaserScan)
    ts = message_filters.ApproximateTimeSynchronizer([laser_sub, sub_odom, sub_fl,
sub_fr], 1,1)
    ts.registerCallback(sensor_callback)

    pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)

    rospy.spin()
```

8.2 Appendix II: Testing Results

Table 8.2-1: Testing results of the first trial

Run	Orientation Threshold	Lidar Threshold	Infrared Threshold	Time Taken (s)	Obstacles Hit	Distance Travelled	Manual Interventions	Score
1	0.85	1.00	0.55	47	0	19.48	0	105.45
2	0.85	1.00	0.65	45	0	23.26	0	114.79
3	0.85	1.00	0.75	33	0	14.22	0	75.66
4	0.85	1.50	0.55	43	0	20.56	0	104.68
5	0.85	1.50	0.65	40	0	17.64	0	92.93
6	0.85	1.50	0.75	43	0	20.08	0	103.25
7	0.85	2.00	0.55	42	0	19.78	0	101.33
8	0.85	2.00	0.65	53	0	24.34	0	126.02
9	0.85	2.00	0.75	53	0	24.48	0	126.45
10	0.90	1.00	0.55	36	0	18.07	0	90.22
11	0.90	1.00	0.65	36	0	16.02	0	84.07
12	0.90	1.00	0.75	46	0	13.66	0	86.98
13	0.90	1.50	0.55	40	0	17.30	0	91.89
14	0.90	1.50	0.65	35	0	15.61	0	81.82
15	0.90	1.50	0.75	46	0	21.42	0	110.26
16	0.90	2.00	0.55	36	0	16.25	0	84.76
17	0.90	2.00	0.65	40	0	16.87	0	90.62
18	0.90	2.00	0.75	43	0	18.98	0	99.95
19	0.95	1.00	0.55	42	0	18.71	0	98.13
20	0.95	1.00	0.65	43	0	16.75	0	93.26
21	0.95	1.00	0.75	44	0	17.97	0	97.92
22	0.95	1.50	0.55	35	0	16.26	0	83.77
23	0.95	1.50	0.65	40	0	16.23	0	88.68
24	0.95	1.50	0.75	38	0	16.40	0	87.19
25	0.95	2.00	0.55	33	0	15.25	0	78.76
26	0.95	2.00	0.65	36	0	15.87	0	83.62
27	0.95	2.00	0.75	36	0	16.31	0	84.92
			Average:	40.89	0.00	18.07	0.00	95.09
🟧	: Run with the best score							

Table 8.2-2: Testing results of the second trial

Run	Orientation Threshold	Lidar Threshold	Infrared Threshold	Time Taken (s)	Obstacles Hit	Distance Travelled	Manual Interventions	Score
1	0.85	1.00	0.75	41	0	19.35	0	99.04
2	0.85	1.00	0.75	35	0	15.09	0	80.28
3	0.85	1.00	0.75	37	0	17.28	0	88.83
4	0.85	1.00	0.75	44	0	21.11	0	107.32
5	0.85	1.00	0.75	104	0	47.95	0	247.85
			Average:	52.2	0.00	24.15	0.00	124.66
6	0.95	1.50	0.55	39	0	16.13	0	87.39
7	0.95	1.50	0.55	47	0	22.41	0	114.24
8	0.95	1.50	0.55	35	0	15.10	0	80.30
9	0.95	1.50	0.55	36	0	16.07	0	84.21
10	0.95	1.50	0.55	42	0	19.08	0	99.25
			Average:	39.80	0.00	17.76	0.00	93.08
🟧	: Run with the best score							