EG2310: Fundamentals of Systems Design

G2 Report

AY23/24 Semester 2

Group 11

Group members	Matriculation No.
Khoo Shi Xian	A0276011R
Kalyanasundaram Avaneesh	A0276914U
Gulati Shobhit	A0244507H
Srinivasan Udhaya	A0257466R
Goh Yee Ern	A0282618X

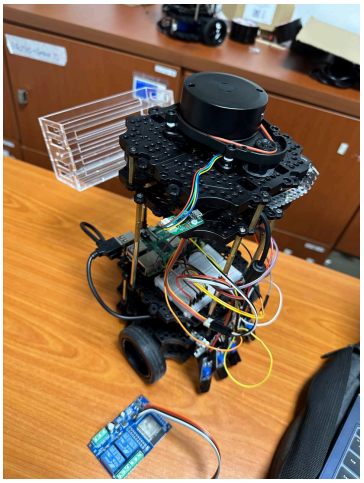


Table of Contents

1. Introduction

The document describes the system design process of our EG2310 robotic system. The system was designed to achieve the following purpose:

> *The objective is to design and implement a TurtleBot system that is able to autonomously navigate through a maze to the correct room and fire ping pong balls into the bucket in the room. The system is to also generate a map of the maze at the end of the mission time while avoiding collisions with any part of the maze.*

The system design process utilised follows the modified EG2310 V-Model as shown below:

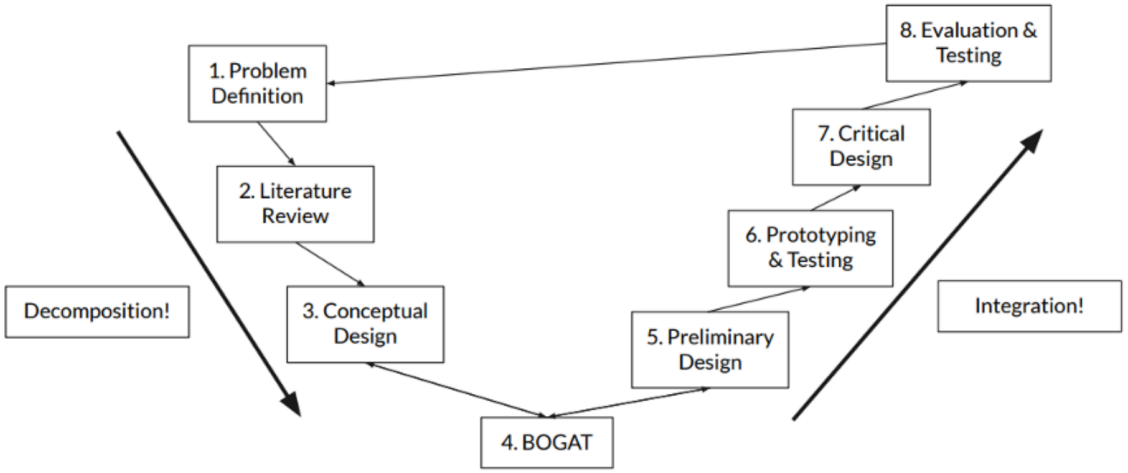


Fig 1.1 NUS EG2310 V-Model

2. Problem Definition

With this context set, the following table outlines the specific stakeholder requirements and project deliverables.

Stakeholder Requirements	Project Deliverables
The robot must navigate autonomously to the locked room.	The robot can start and enter the obstacle area on its own. The robot can also navigate through the maze while avoiding the obstacles.
The robot must develop a full simultaneous localization and mapping (SLAM) map of the area.	The robot can explore every part of the maze to map out all the maze elements and develop a full SLAM map.
The robot must communicate with a secured door to open the door.	The robot can connect to a server to unlock the door.
The robot must find a bucket in a locked room	The robot should be able to locate the randomly placed bucket within the elevator room and navigate to it, stopping a suitable distance for the next task

The robot must then fire up to 5 ping pong balls into a bucket	The robot can carry 5 ping pong balls safely to the room and fire the balls into the bucket.

3. Literature Review

3.1 Navigation

3.1.1 Pathfinding

This section explores different possible algorithms to find the shortest possible route to the bucket.

A* Search
A* search is a searching algorithm that plots a walkable path between multiple nodes, or points, on the graph[1]. It introduces a heuristic, such as a Manhattan distance, into a regular graph-searching algorithm, so a more optimal decision is made. This algorithm minimises the total cost between nodes, and in the right conditions will provide the best solution in optimal time.

Dijkstra's Algorithm
Dijkstra's algorithm discovers the shortest paths between nodes in a weighted graph by selecting a starting “source” node and determining the shortest path from it to all other nodes[2]. It utilises a criteria model to explore nearby unexplored nodes, maintaining a set of nodes with known shortest distances from the source and continues expanding this way until it reaches the destination or considers all reachable nodes.

Tape Following Method
The line/tape following method is a classic algorithm used in robotics for navigating predefined paths marked by lines or tapes on the ground[3]. It relies on sensors to detect the lines or tapes and employs a set of rules to determine the movement of the robot along the path. By adjusting the robot's direction based on the feedback from the sensors, it can maintain alignment with the path, effectively following it from start to end. It is easy to implement.

Decision & Rationale: We decided to use the tape following method due to its simplicity and ease of implementation in maze navigation. By relying on sensors to track predefined paths, it eliminates the need for complex path planning algorithms.

3.1.2 Mapping the maze

The turtlebot would have to travel to every single part of the maze, to be able to map it out entirely.

Depth First Search
Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures (in this case, the maze) [4]. The algorithm starts at the root node and explores as far as possible along each branch until it reaches a dead end before backtracking.

Breadth First Search
Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property[5]. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.

Frontier Algorithm

Frontier-based exploration algorithm is a method used for maze navigation that involves systematically expanding the frontier of explored areas. It starts with an initial node and continually expands the frontier by exploring adjacent cells, marking them as visited. This process continues until all reachable areas of the maze are explored, making it an effective approach for comprehensive maze traversal[6].

Decision & Rationale: Frontier exploration is dynamic and can work with any generated maze. DFS and BFS are relatively slower. Further, readily available frontier algorithms are also available which greatly reduces development time. Finally, frontier exploration guarantees complete maze exploration.

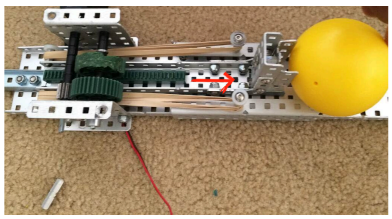
3.2 Server Response

HTTP is a set of protocols designed to enable communication between clients and servers. It works as a request-response protocol between a client and a server[7]. In our case, the Turtlebot serves as the client, and the server is responsible for giving the location of the opened door. We simply need to request a response from the server to determine which door is correct. For this purpose, we will be utilising the GET method to request data from the server. HTTP requests can be performed through different applications like Javascript, Angular, Python etc. However, for uniformity, we have chosen to implement Python scripts using the requests library[8]. As there is limited information available about the web server at the moment, our system is designed to make requests from the server's endpoint API.

3.3 Firing Mechanism

Shooter mechanism

This involves firing each ball as a projectile with some velocity straight into the bucket. This can be implemented with various types of shooters, such as a linear puncher, where a lever arm is used to “slap” the ball with a force to launch it or a flywheel powered shooter where the projectile comes in contact with two spinning wheels, and the friction between the the ball and the wheels shoot the ball[9].



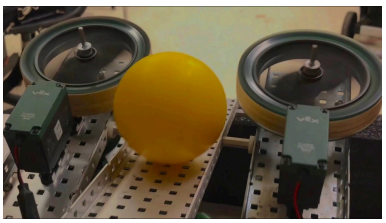


Fig 3.3.1 - Fig 3.3.: Examples of shooter

Catapult mechanism

This involves flinging all the balls together with a Mangonel catapult like mechanism. Two arms, connected to the payload (ping pong balls and mount) are rotated, causing the payload to gain momentum[10]. The arms are then abruptly stopped so that the object will continue its trajectory through the air. It requires the bot to come close to the bucket so that the balls, which are placed in the mount, can simply be thrown into the bucket.

Fig 3.3.3 - Fig 3.3.5: Examples of catapult

Turn and pour Mechanism

This involves the use of linear actuators and gates. The balls are stored inside a tube and are blocked by a gate. Upon reaching the bucket, the back of the tube is raised and the front is lowered. Then, the gate blocking the balls is released to allow the balls to roll and fall into the bucket under the influence of gravity.

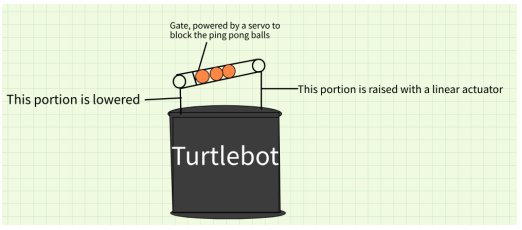


Fig 3.3.6 : Example of turn and pour

Decision & Rationale: We chose the catapult mechanism as it can deliver all five balls at once, optimising power. The simple turn-and-pour method was avoided due to potential Lidar blockage by raised linear actuators and the shooter mechanism might need a loading system to avoid varied launches and ensure that all balls reach the bucket.

4. Concept Design

4.1 Software System

Our plan is to deploy the tape following algorithm first to get to the bucket in the shortest amount of time. This can be implemented by having specific cases that the turtlebot will adhere to. The python code would direct the turtlebot's motions based on which line sensors are detecting black. Thus, the information published by the line sensors would be subscribed to by the turtlebot which would then accordingly trigger the response motion. Since the turtlebot needs to make a http call outside both rooms, a t-junction is made with the tape which would be the only unique case that would direct the turtlebot to make a http call. Then after firing the balls, the turtlebot would reverse back some distance and then the frontier based algorithm would be called upon to explore all the routes and develop the SLAM map. The frontier exploration would start from the room itself, after which the turtlebot would slowly explore the rest of the maze.

4.2 Electrical System

Upon entering the respective elevator room, we would be using 4 parallel line sensors which will be connected to 5V and four GPIO Pins on RPI for the navigation to the target bucket by employing a control algorithm. The line sensors consist of infrared detectors that can differentiate between light and dark surfaces. Since the flooring is dark, we would be using black tape as a guide for the robot to follow. The signals from the line sensor will be interpreted. If the robot drifts and the sensors detect the black tape, it would make steering changes accordingly to reach the bucket. The bot will follow to black tape to reach the bucket.

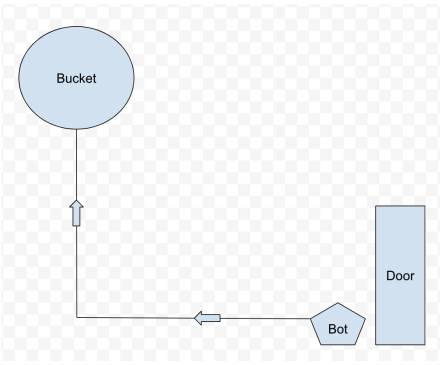


Fig 4.2.1: Alignment of tape in room

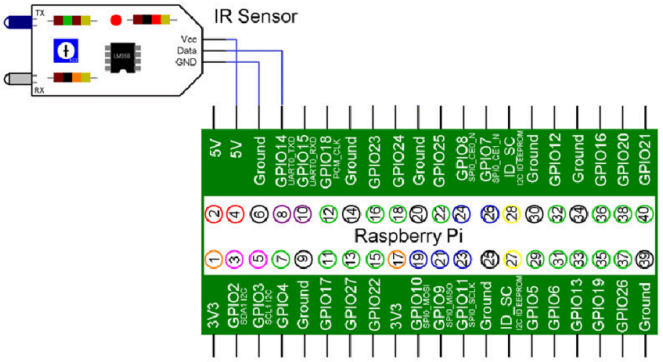


Fig 4.2.2 Circuit of IR Sensor

The servo motor input is connected to the GPIO Pin 12 on Raspberry Pi and the pin provides a PWM input signal to the servo motor. The servo motor will then rotate a certain angle when the duty cycle of the PWM input signal changes. The GND line is connected to GND on Raspberry Pi to act as ground.

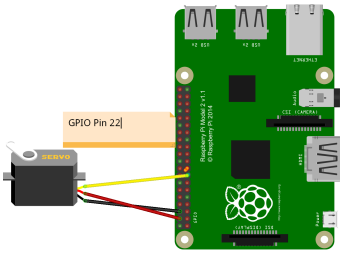


Fig 4.2.3: Servo motor wiring with RPI

4.3 Mechanical System

The chosen mechanism for the launching of the 5 ping pong balls is a double armed catapult. We have decided to adapt the catapult such that it would be releasing all 5 ping pong balls together in one go as it removes the need for a loading mechanism, reducing the complexity of the mount. The catapult mechanism can be easily controlled with servo motors and it also enables us to angle it below the lidar such that the mapping is not affected by the mount.

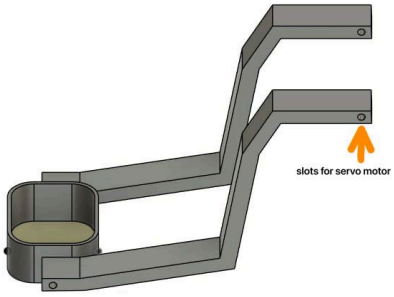


Fig 4.3.1 Catapult CAD

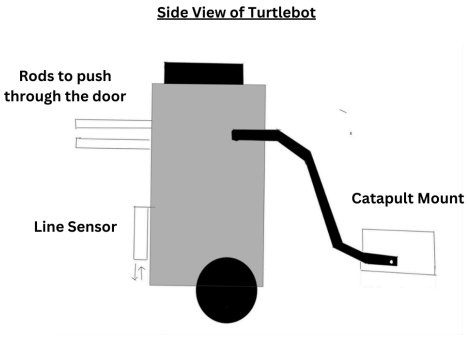


Fig 4.3.2 Side view of turtlebot

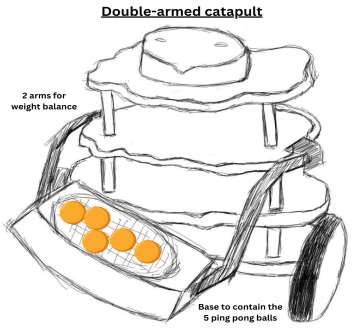


Fig 4.3.3 view of turtlebot

5. Preliminary Design

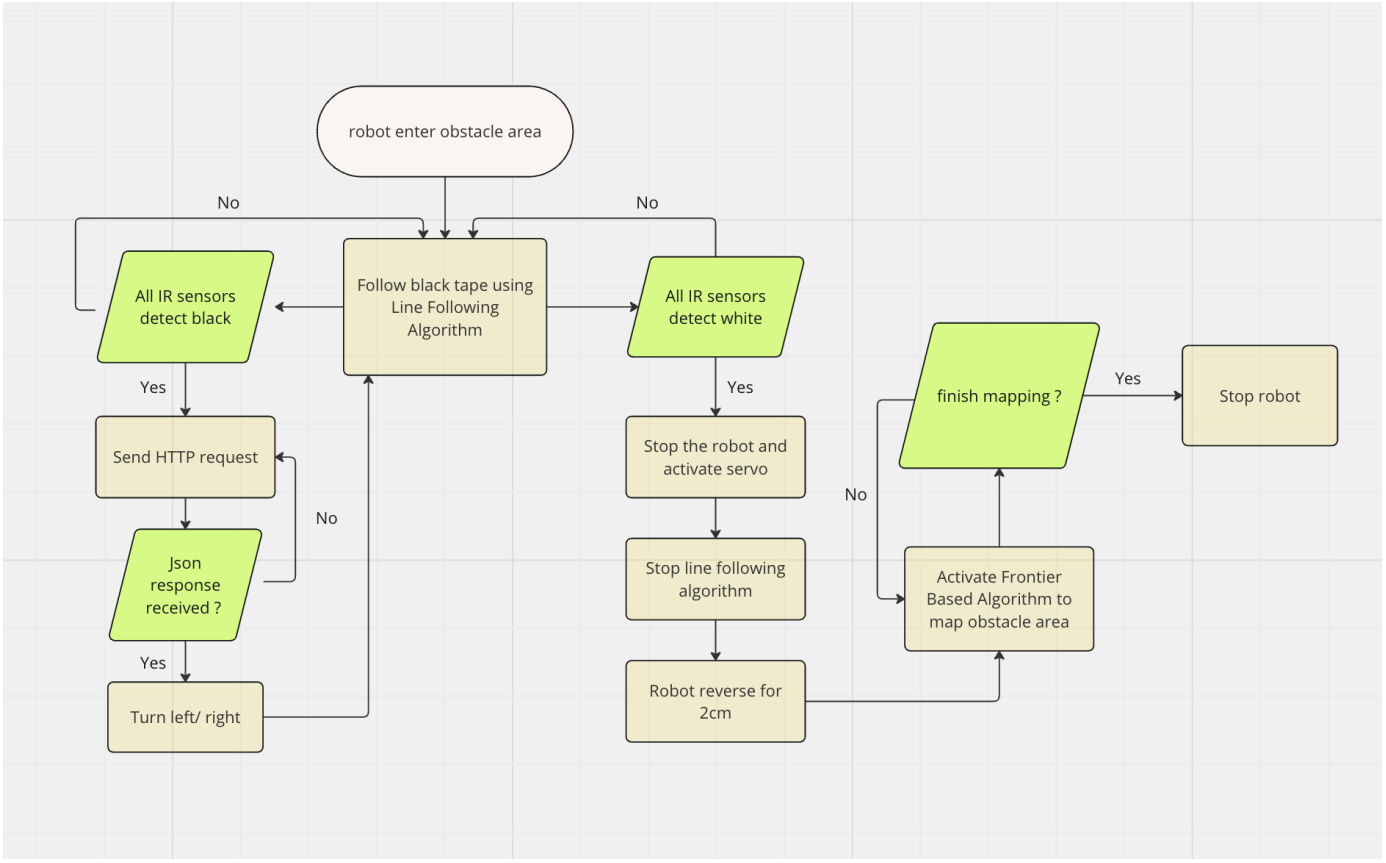


Fig 5 Overview of mission

6. Prototyping & Testing

6.1 Hardware and Electronics

1. Servo does not rotate as intended
    - After conducting trial and error adjustments, we modified the values within the time.sleep() and p.ChangeDutyCycle() functions to achieve the intended performance. Initially, the servo wasn't moving in alignment with the calculated angle derived from the duty cycle. This iterative approach allowed for fine-tuning and ensured the servo moved accurately according to the intended angle.
    - Ensured that the wires are fully inserted into the GPIO pin.
    - Ensured that the wires are connected to the correct GPIO pin (12).
    - Ensured that the battery was fully charged.
2. Turtlebot kept rebooting
    - Made sure that the metal part of the PCB was no longer in contact with the screw on the turtlebot.
    - Found out that the PCB was shorted and re-soldered a new one.
3. Line sensor caused turtlebot to keep rebooting
    - Placed tape on the aluminium sheet to prevent the metal part of the line sensor from coming into contact with the aluminium sheet, thus avoiding a short circuit.
4. Line sensor not detecting the tape correctly
    - Recalibrated the line sensor according to the tape using a screwdriver.

6.2 Fabrication of the payload

1. The initial solid walls of the design turned out to be too heavy for the motor to support.
    - Thus, we redesigned the box with slots to reduce the material and hence its weight while being functional
2. With a lower weight of payload, one servo was sufficient to support the weight and launch the balls. Moreover, there were some difficulties with synchronising both servos.
    - Thus, one servo was used instead of 2.

6.3 Frontier exploration

The frontier code was first tested in Gazebo and performed exceedingly well. It was tested in 2 custom maps and mapped 100% almost all the time. The mapping time was quite fast as well.

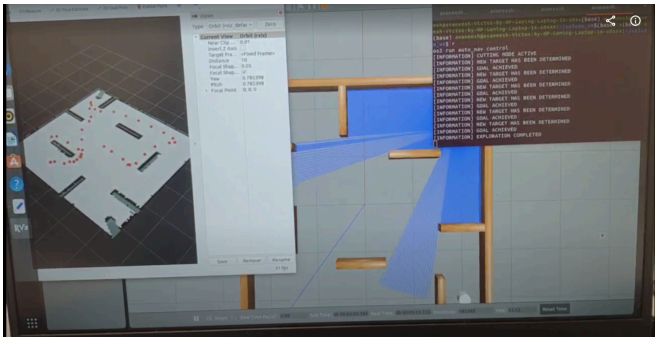


Fig. 6.3.1 Frontier in action on Gazebo.

However, it initially struggled a lot in real world mazes. A lot of parameter adjusting and configuration to our specific turtlebot, such as adjusting robot_r (robot radius around bot for obstacle detection) and lidar scan range (all of these are explained later) was done.

The exploration performed quite well with time in closed and rather open mazes.

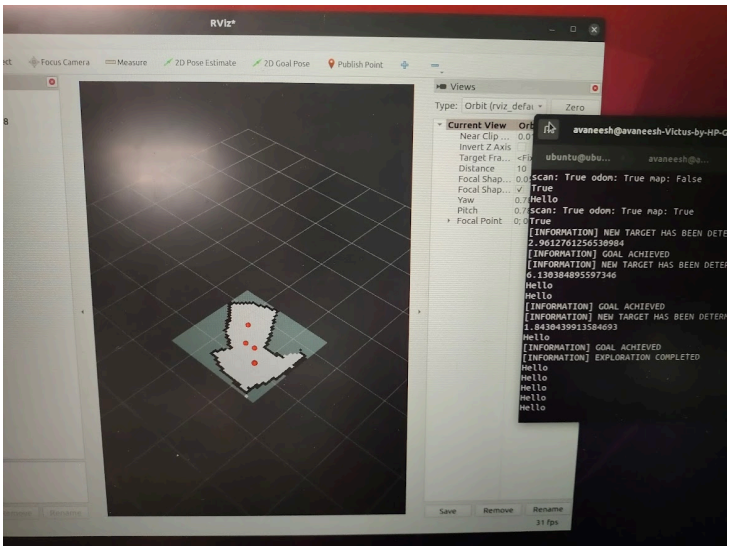


Fig. 6.3.2 Completed mapping #1 in real life

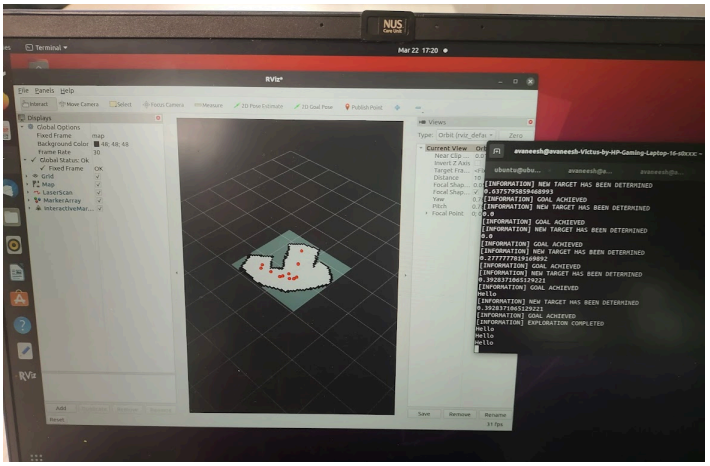


Fig. 6.3.3 Completed mapping #2 in real life

Fig. 6.3.4 Sample maze we constructed

However, it continued to struggle with complex and closed mazes which led to further parameter configuration before it was resolved.

Fig. 6.3.5 Initial struggles with more complicated mazes

7. Final design

7.1 Turtlebot specifications

List	**Specifications**	**Note**
Size (mm)	220 x 330 x 281	Full assembly (Turtlebot3 with the arm and box)
Weight	1.25 kg	
Drive Actuator	2 x L430-W250 Dynamixel	
Maximum Translational Speed	0.20 m/s	
Sensors on Board	LDS-01 Lidar Sensor	
Maximum Rotational Speed	2.84 rad/s	
Battery and its capacity	LiPo Battery 11.1V 1,800 mAh	
Expected Operation Time	2.5h	

7.2 Final CAD

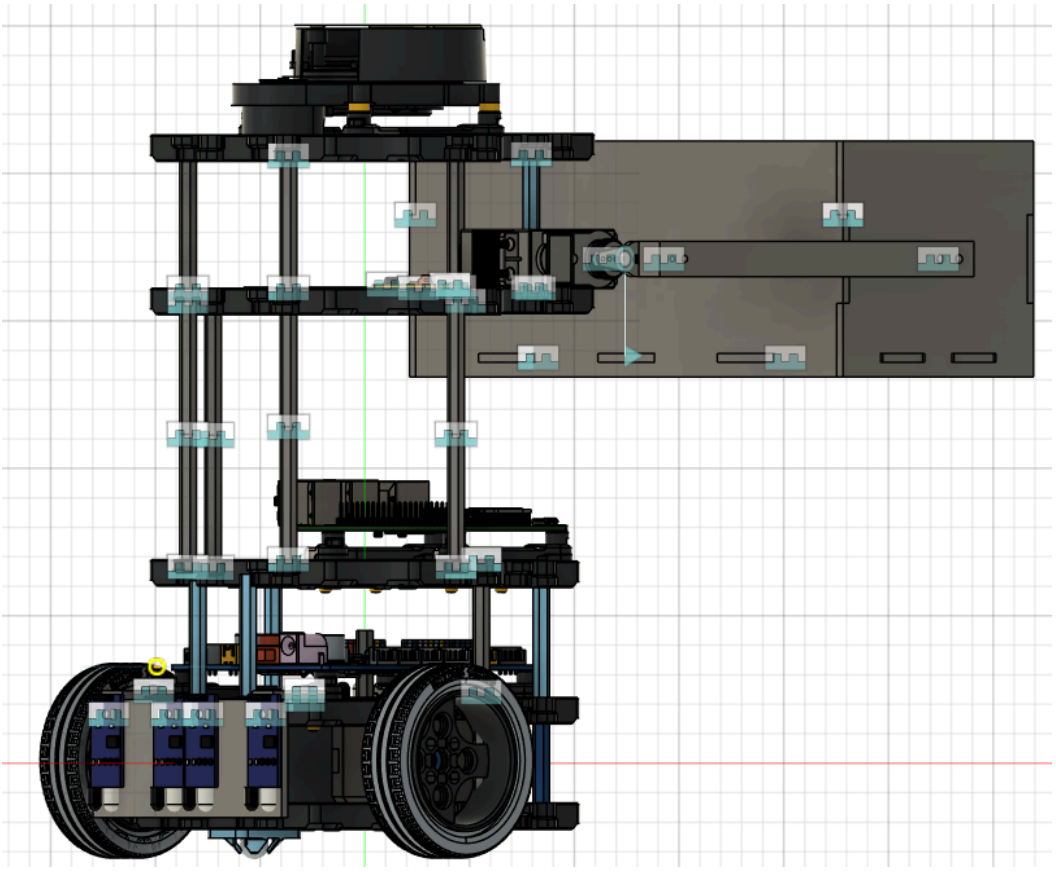


Fig 7.2.1 View of Turtlebot

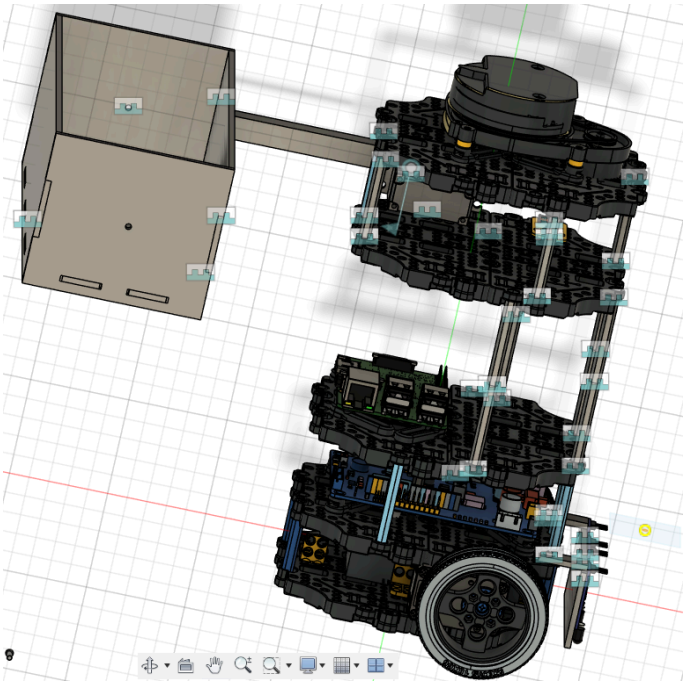


Fig 7.2.2 View of Turtlebot

7.3 Bill of materials (additional materials that was taken)

The bill of materials excludes everything that was handed to all the teams initially, such as the turtlebot, all its parts, toolbox, electronic components, picometer, ESP32, etc. Only materials that were additionally sourced are given here. Turtlebot parts can be derived from the manual as mentioned. All screws mentioned assume the same size nut is also used.

No.	Part	QTY	Unit Cost	Total Cost
1	Waffle Plate	2	Sourced from Lab	NA
2	Black tape	3 rolls		
3	IR sensors	5		
4	MG995 Servo motors (with horns)	2		
5	Acrylic arm	3		
6	Acrylic box	1		
7	Aluminium Sheet	1		
8	45mm waffle supports	4		
9	Saw (to cut aluminium sheet)	10		
10	M3x8 screws	23		
11	M2.5x12 screws	8		
12	M2x16 screws	2		

13	M2.5x16 screws	1		
14	M3x16 screws	2		
15	M3x10 screws	4		
16	M2.5x10 screws	2		
17	Cloth tape	1	$2.2	$2.2
18	Cloth Tape - 2” width	2	$3.90	$7.80
	TOTAL		**$10**	

8. Assembly Instructions

8.1 Mechanical Assembly

1. Assembly of up to the third layer of the turtlebot is to be done as per the instructions in the Turtlebot3 Burger Assembly Manual that can be found here.

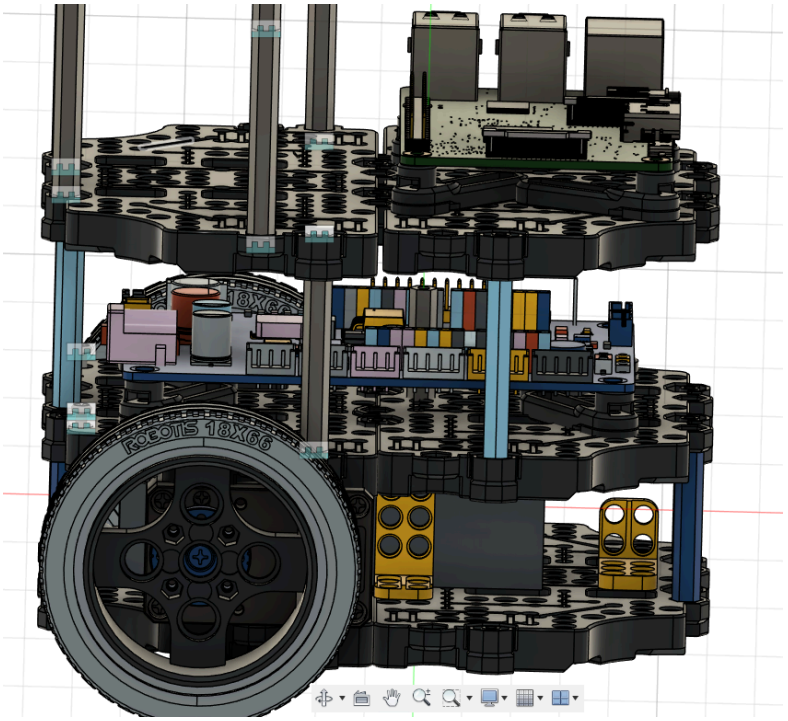


Fig 8.1.1 Zoom View of Turtlebot

2. Make 2 more (sets) of waffle plates as given in page 9 of the Manual

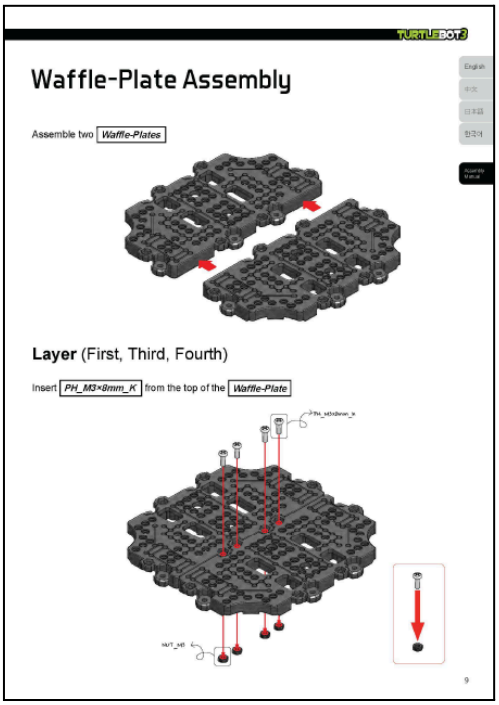


Fig 8.1.2 Assemble Manual

3. Next, make 5 sets of waffle support connections, by joining 4x45mm female to male waffle supports together, and finally 1x45mm male to male support at the end as shown. The height of the overall support (therefore the number of supports in each set) can be adjusted according to the bucket height.

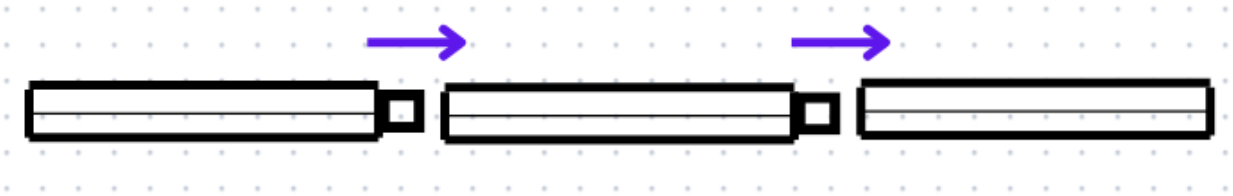


Fig 8.1.3 Axes Assemble

4. Next attach each of the 5 sets of waffle supports to the 3rd layer waffle plate in the shown holes. They are secured using M3X8 screws.

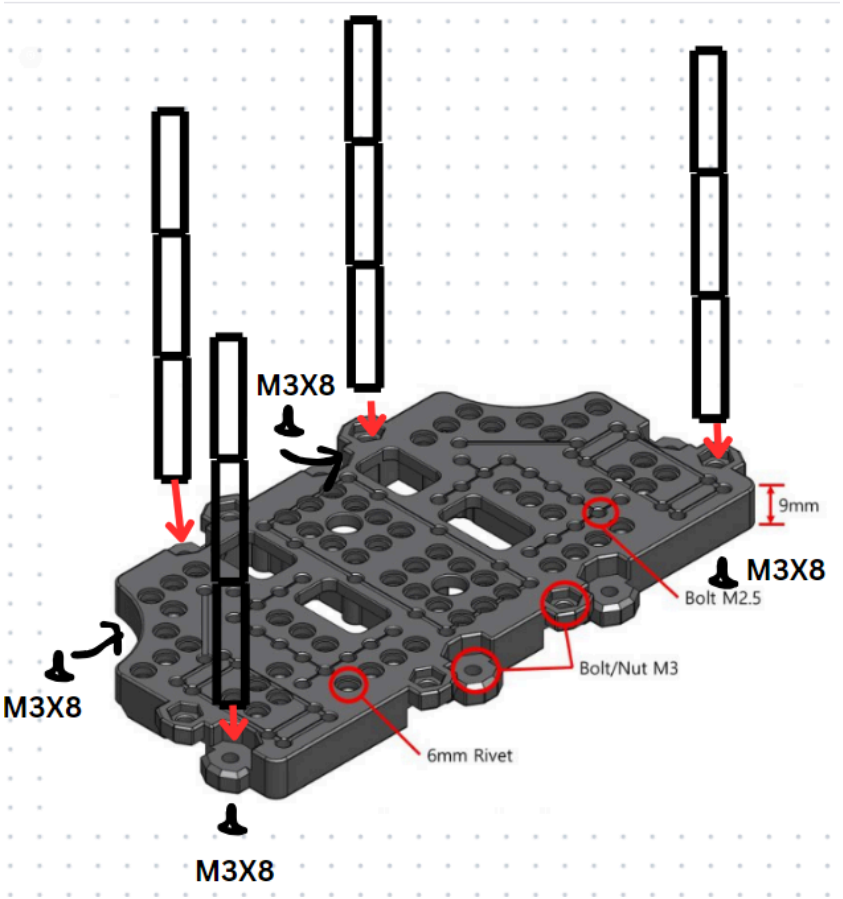


Fig 8.1.4 Front half of the waffle plate

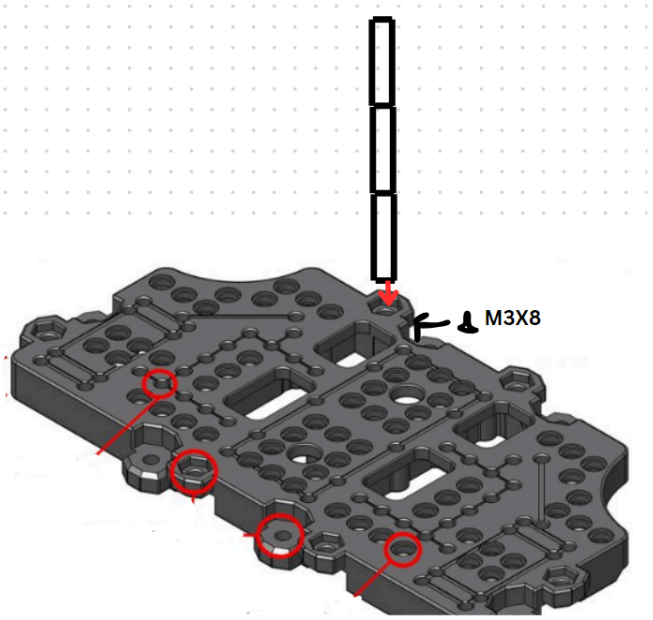


Fig 8.1.5 Back half of the waffle plate

5. Next, attach the 4th waffle plate on top of the waffle supports using M3x8 screws as well.

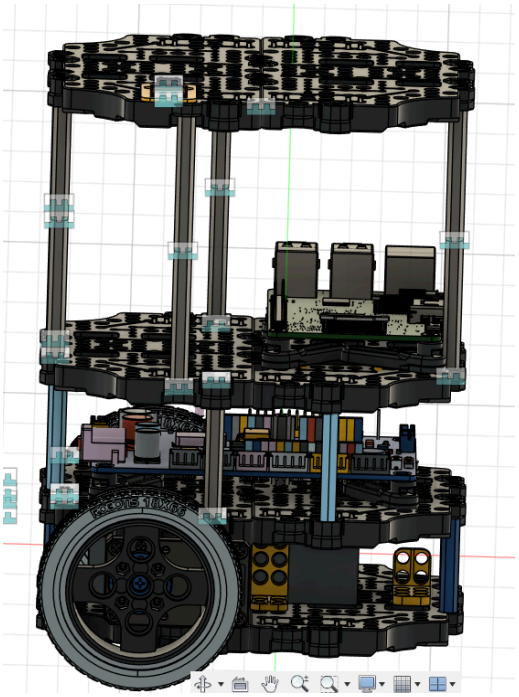


Fig 8.1.6 View of Bot

6. Next mount the servo motor onto the 4th waffle plate with the help of two custom brackets made with aluminium sheet. These brackets are to be made by bending an aluminium sheet with M2.5 holes. They are then screwed into the M2.5 holes on the waffle plate as well using M2.5x12 screws. (Since it is customised to the turtlebot and the servo motor used, it is flexible to screw in anywhere. Only concern that must be

addressed is that the arm, when attached to the servo motor, does not collide with the waffle plate).

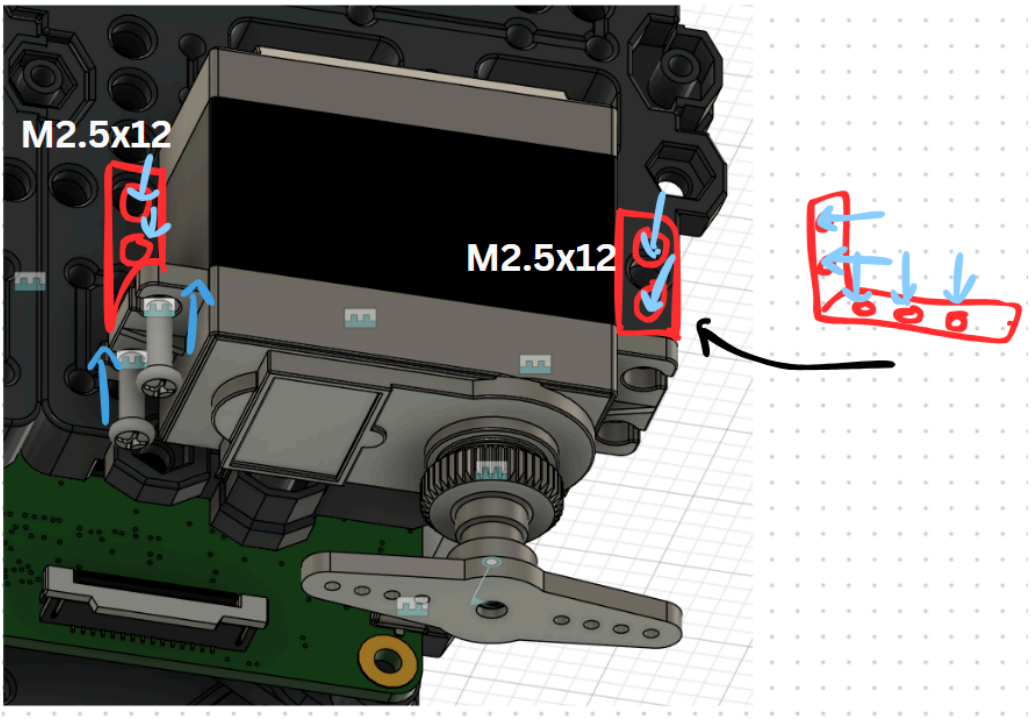


Fig 8.1.7 Assemble Servo Motor with Aluminium Sheet

7. The servo horn needs to be modified such that the hole sizes are 2.5mm, 2mm and 2mm as shown and that all three holes are 6mm apart from each other (if this is not possible, the holes in the arm have to be modified). The horn is screwed to the servo motor with an M3 screw.

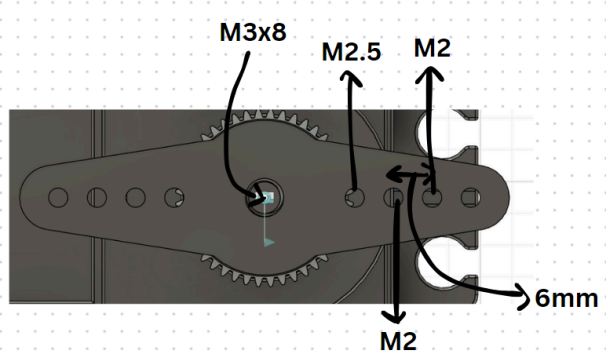


Fig 8.1.8 Screw Size on Servo Horn

8. Next, the acrylic arms are cut using the following dimensions. The arms are 10mm thick. All units are in mm.

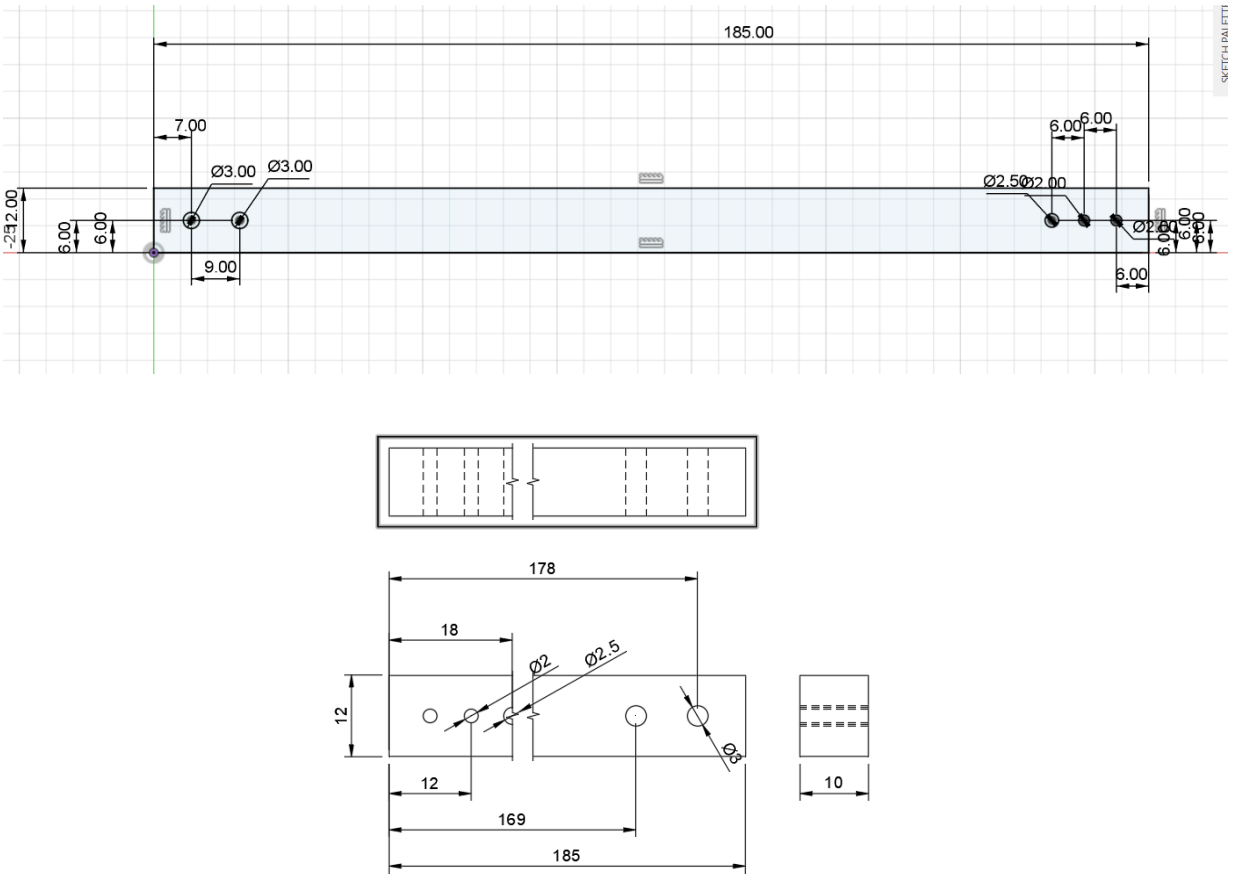


Fig 8.1.9 -8.1.10 Arm Dimension

9. Similarly, the open box is fabricated by laser cutting the 5 sides. The sides are then attached together using an acrylic bonding agent for bonding and curing.

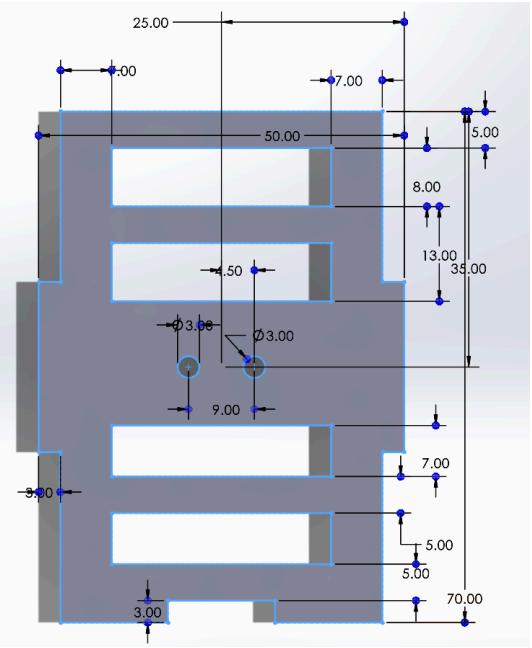


Fig 8.1.11 Side Plate of Box

2 holes of size m3 were cut in the side plates for attachment to the acrylic arm.

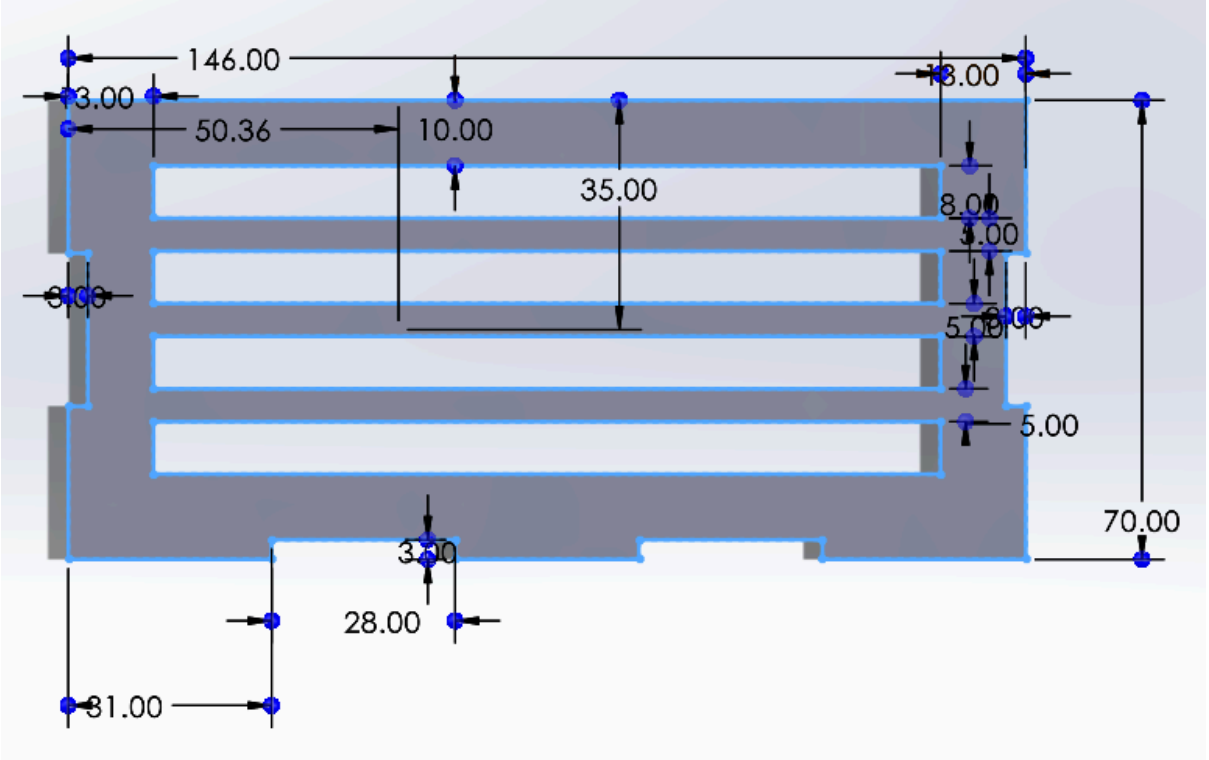


Fig 8.1.12 Frontal Plate of Box

2 copies of the above frontal and side plates were laser-cut.

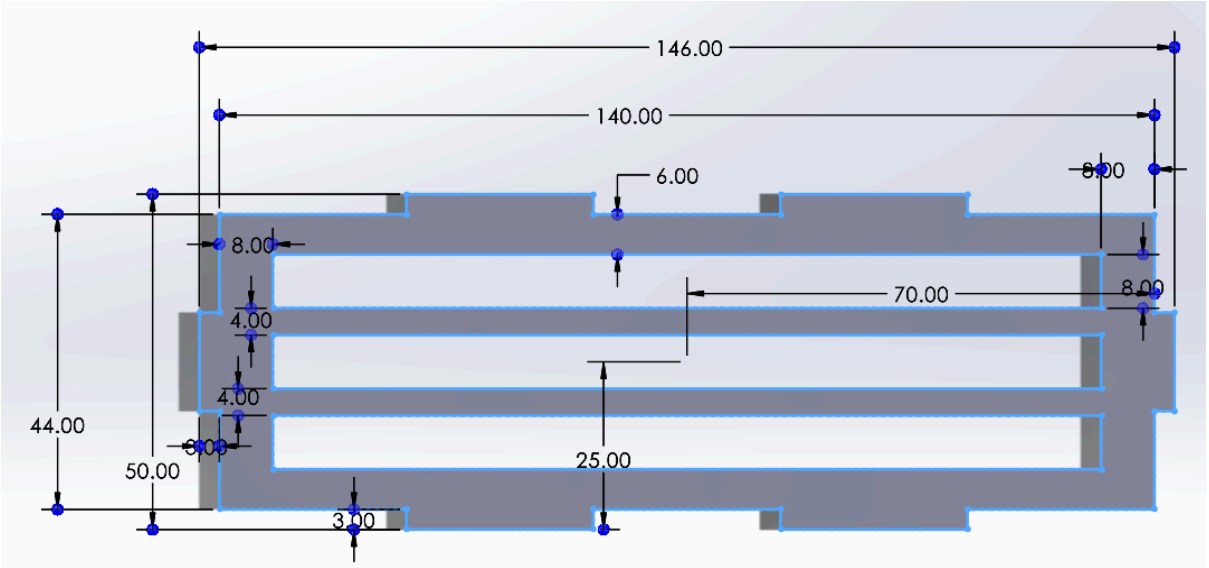


Fig 8.1.13 Base Plate of Box

One base plate with complementary ridges to the frontal and side plates were laser cut as well.

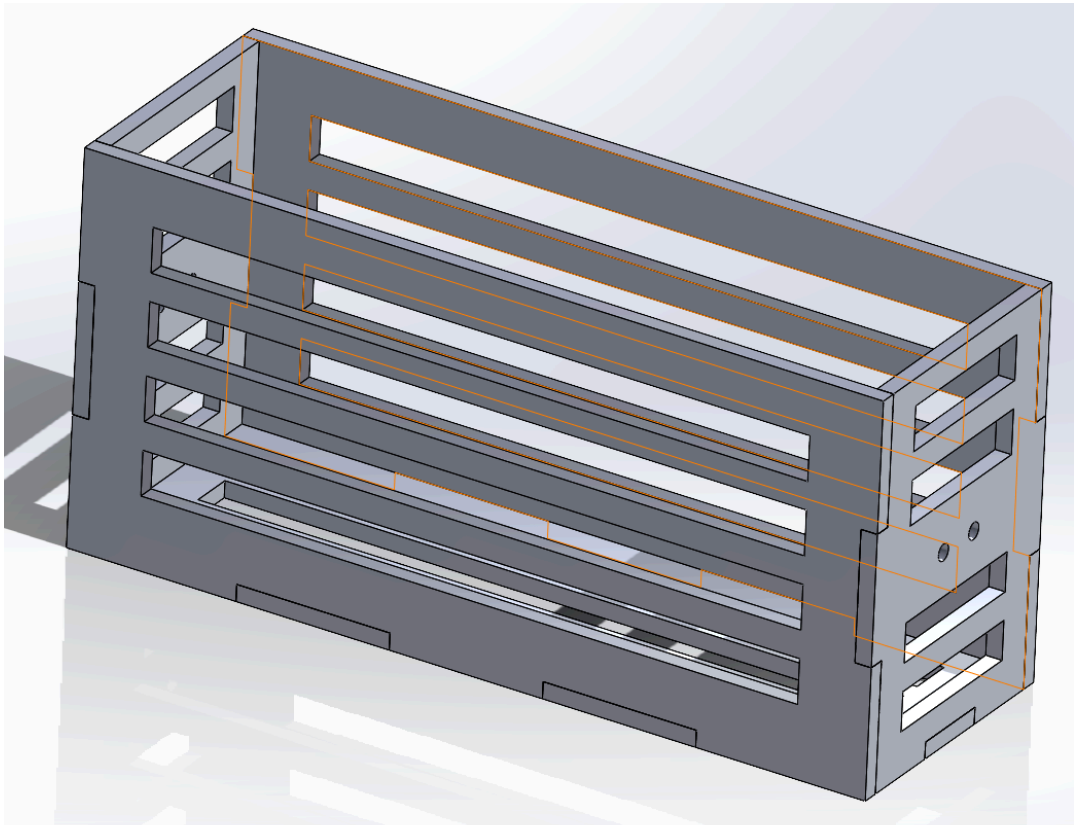


Fig 8.1.14 Final Assembly of Box

The final assembly of the box is as depicted in the figure above (Fig 8.1.14).

10. The arm is then screwed into the servo horn using 2 M2x16 screws and 1 M2.5x16 screw.

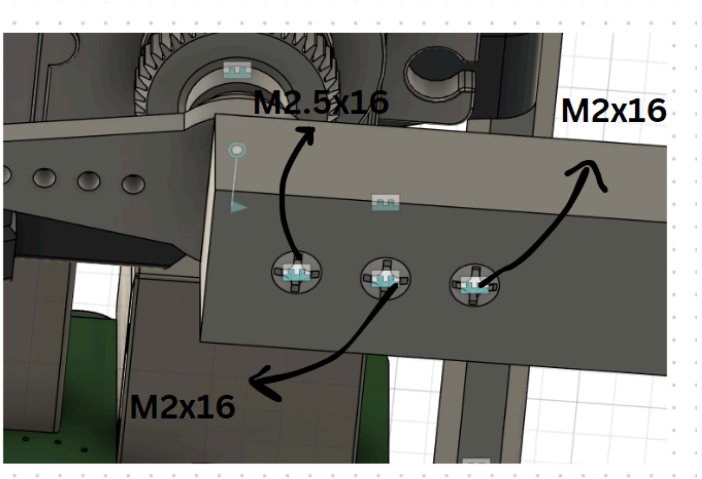


Fig 8.1.15 Screw size on Arm

11. Similarly, the box is screwed into the arm as well using 2 M3x16 screws.

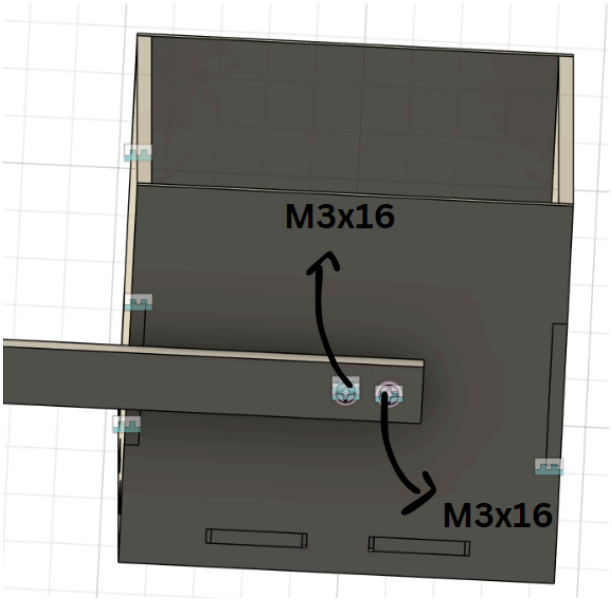


Fig 8.1.16 Screw Size on Box

12. Finally, four more 45mm waffle supports are attached to the places as shown in the figure. Then, a 5th waffle plate is mounted on top. The waffle supports are screwed with M3x8 screws on both ends.

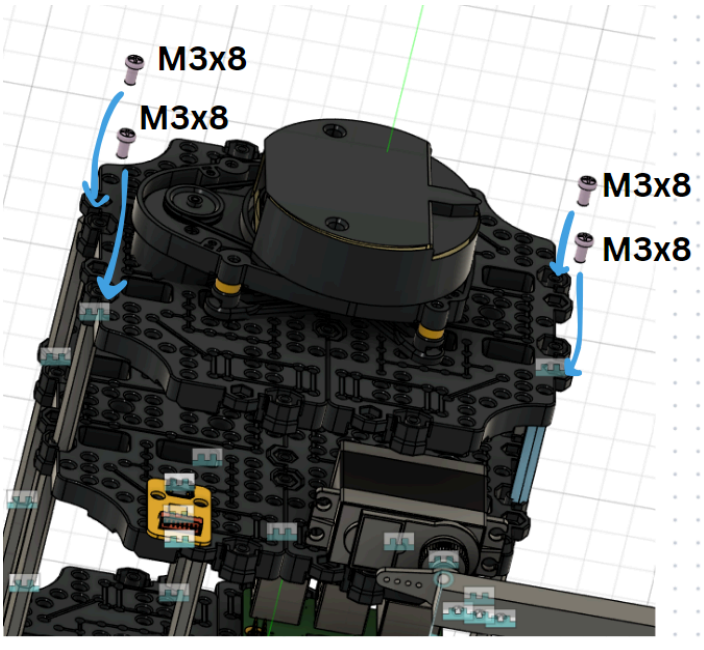


Fig 8.1.17 Zoom View of Bot

13. The LIDAR sensor is mounted onto the 5th layer as per the instructions given in the manual.

14. Next, a mount for the IR sensors is to be made using aluminium sheet and is screwed onto the 2nd layer using 2 M2.5x10 screws. The dimensions for the mount to be cut out are as follows.

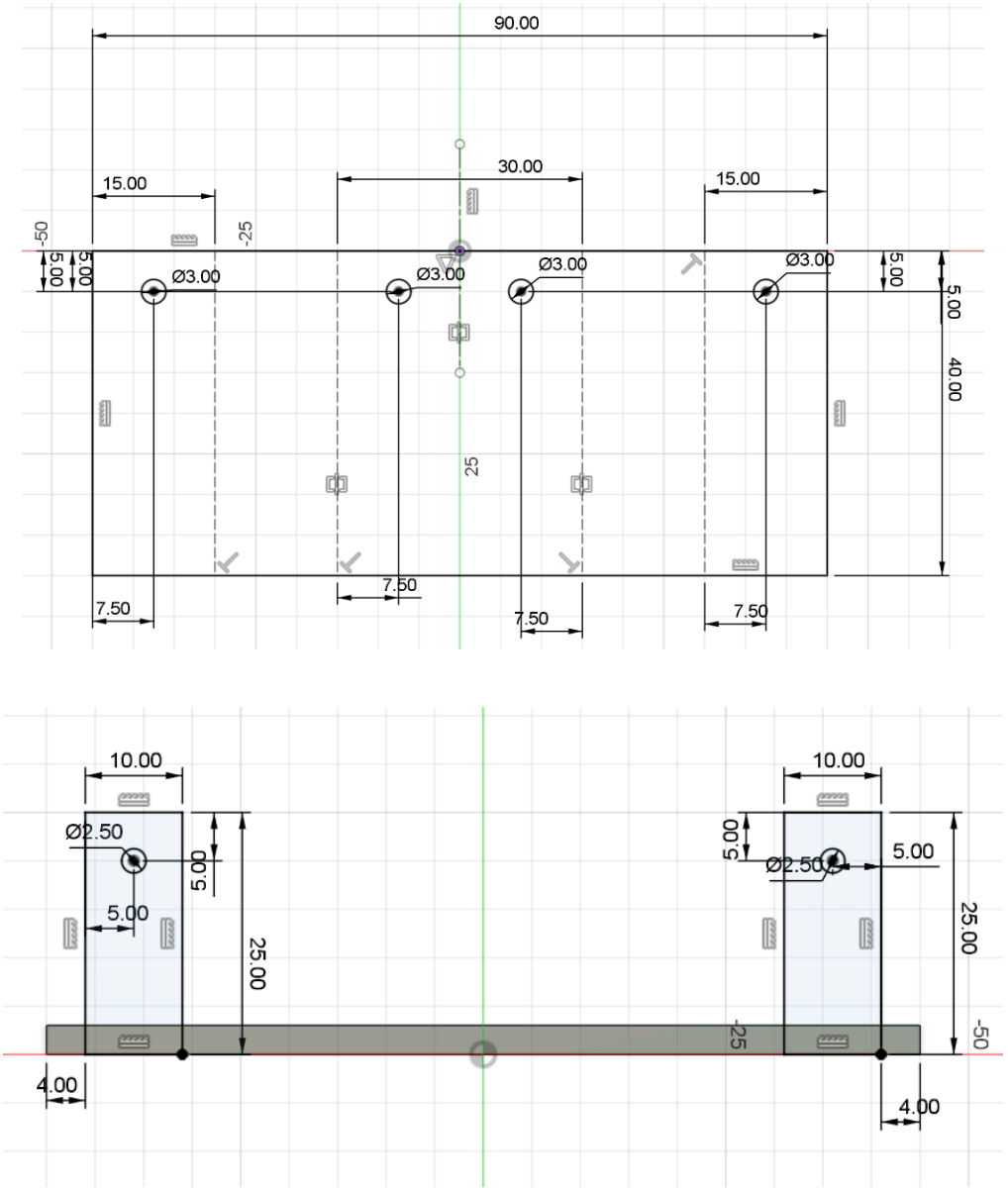


Fig 8.1.18 - 8.1.19 Dimension of Aluminium Sheet for IR Sensors

15. The IR sensors are then screwed into the sheet using M3x10 screws. The assembly is complete.

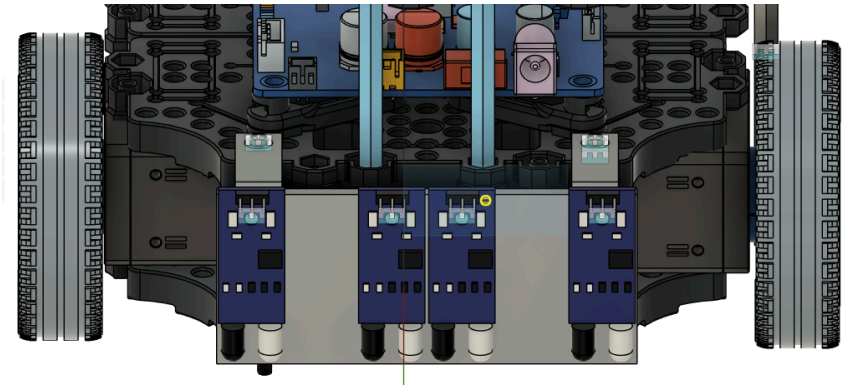


Fig 8.1.20 Zoom View of IR sensors on Bot

8.2 Electrical Diagram

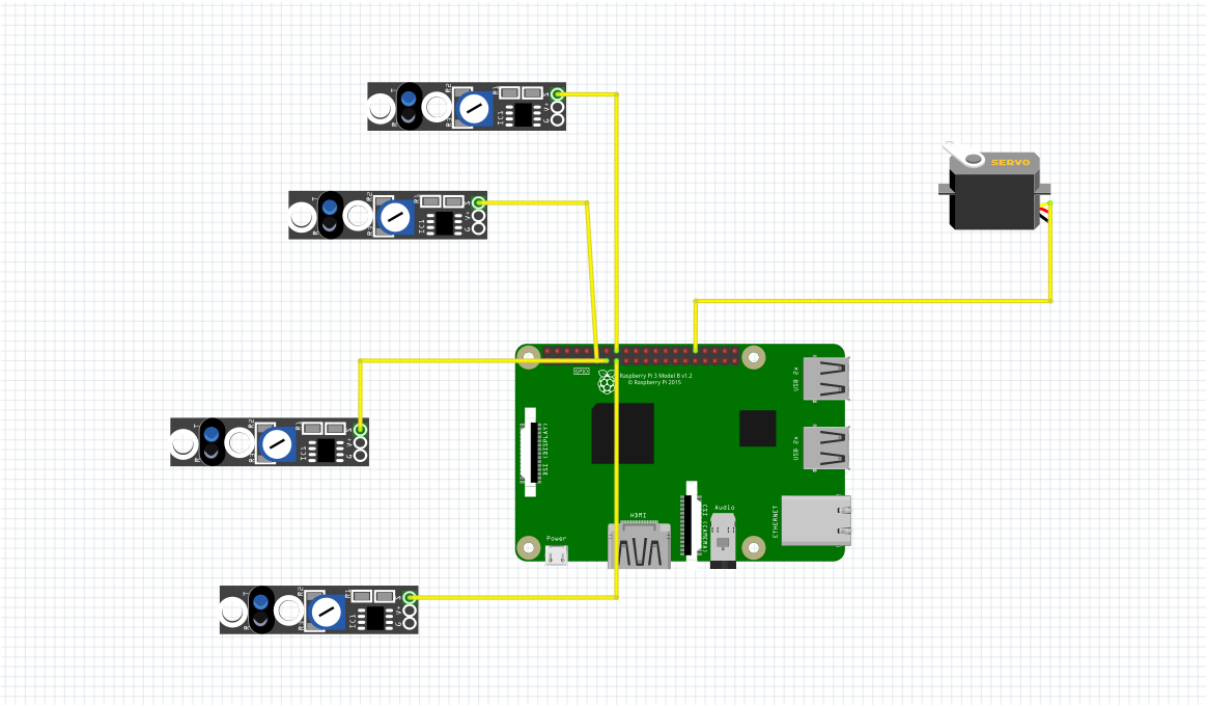


Fig 8.2.1 Electrical Diagram of Connecting Pi with IR sensors and Servo Motor

8.3 Software assembly

This section assumes that all setup until r2auto_nav.py and GPIO setup from the seminars have been set up in the Rpi.

1. Install slam toolbox via apt `apt install ros-eloquent-slam-toolbox`
2. Ensure that the `control.py` file, `servo.py` file, `reverse.py` `r2_mover3.py` file are all in the same directory as the `r2auto_nav.py` file.
3. Ensure that the config directory with the params.yaml file for the control.py file is also at the same directory as above.

4. In the `setup.py` file, ensure that the control.py file is added to the entry_points as shown below.

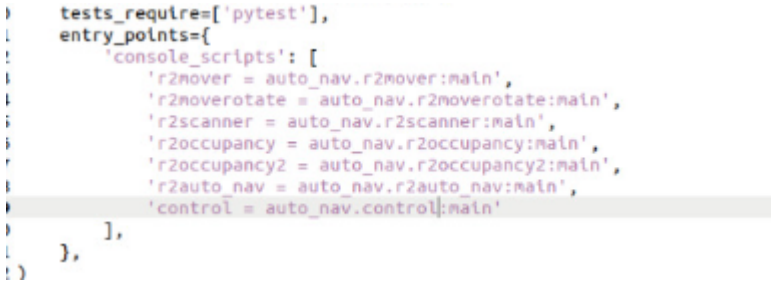


Fig 8.3.1 Code Editing in setup.py file

5. Add all the four python files to a shell file called `run_scripts`.sh

```
#!/bin/bash

python3 r2_mover3.py
python3 servo.py
python3 reverse.py
python3 control.py
```

6. Enter `source .bashrc`
7. To setup RVIZ open RVIZ and add the ‘map’ and ‘laserscan’ topics that we need manually
8. Top left of screen file -> save as -> name it and save it in the default directory. Close RVIZ
9. 4. Test:

```
cd ~/.rviz2
ls
```

You should see the file you saved in this folder

10. `rviz2 --display-config <filename>.rviz` (rviz should open with nodes already setup)
11. To automate, add to your .bashrc:

```
alias <shortcutname>='cd ~/.rviz2 && rviz2 --display-config
<filename>.rviz'
close .bashrc
source .bashrc
```

12. In the control.py file, ensure that the location address for the params.yaml file is updated correctly.

control.py params.yaml

```python
import rclpy
from rclpy.node import Node
from nav_msgs.msg import OccupancyGrid , Odometry
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan
import numpy as np
import heapq , math , random , yaml
import scipy.interpolate as si
import sys , threading , time
from rclpy.qos import qos_profile_sensor_data


with open("/home/avaneesh/colcon_ws/src/auto_nav/auto_nav/config/params.yaml", 'r') as file:
    params = yaml.load(file, Loader=yaml.FullLoader)

```

Fig 8.3.2 Code Editing for location address

8.4 Code Explanation

8.4.1 Line Follower:

This is the explanation for the line sensor script, used to traverse the maze.

1. In order to detect the tape on the map, 4 line sensors were used and arranged in a linear array format, according to their position with respect to the bot.

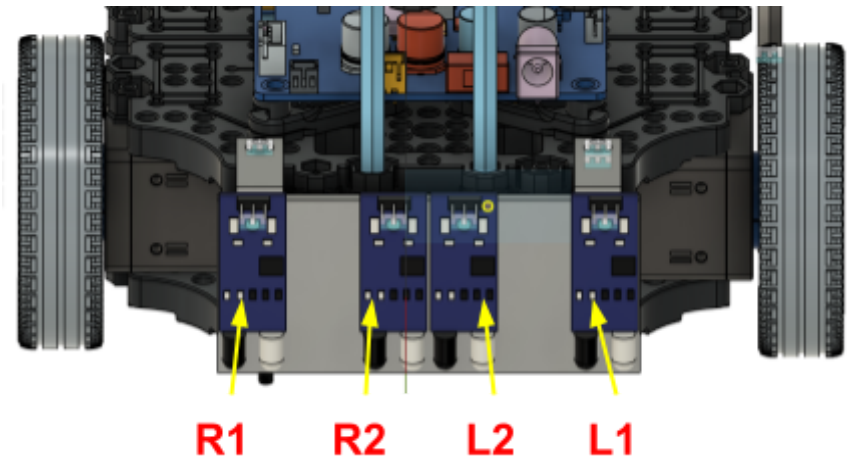


Fig 8.4.1 IR Sensor Numbering

2. Then according to the physical positioning on the bot they were declared in the script, with the same name. Other constants are also declared to tell how much the turtlebot must move in terms of linear and angular velocities.
   Here is the code snippet of how the constants are declared:

```
GPIO.setmode(GPIO.BCM)

PIN_L1 = 17  # most leftward pin
PIN_L2 = 24  # center leftward pin
PIN_R2 = 23  # center rightward pin
PIN_R1 = 27  # most rightward pin

GPIO.setup(PIN_L1, GPIO.IN)
GPIO.setup(PIN_L2, GPIO.IN)
GPIO.setup(PIN_R2, GPIO.IN)
GPIO.setup(PIN_R1, GPIO.IN)

# Constants
ROTATE_CHANGE = 0.4
NUDGE_CHANGE = 0.2
SPEED_CHANGE = 0.15
ROTATE_SPEED = 0.06
NUDGE_SPEED = 0.01
HTTP_CALL = True
```

Fig 8.4.2 Constant for Line Following Code

3. In order to move the bot on the tape, a publishes messages of type ‘geometry_msgs.msg.Twist’ to the topic 'cmd_vel' is created to tell the bot to move:

```
class Mover(Node):
    def __init__(self):
        super().__init__('mover')
        self.publisher_ = self.create_publisher(geometry_msgs.msg.Twist, 'cmd_vel', 10)

    def read_key(self):
        twist = geometry_msgs.msg.Twist()
```

Fig 8.4.3 Code Snippet for Line Following Code

4. In order to choose from one of the doors, a specific case is chosen, where all the line sensors detect the tape. In this case, the robot stops moving and initiates an **HTTP POST** request to an ESP32 server. It stores the JSON response and extracts the door information from it. Depending on the door indicated in the response, the robot adjusts its rotation accordingly. To prevent accidental duplicate requests, a flag is used to ensure the HTTP request is made only once.

```
  # detects black then print 1
  if L1 == 1 and L2 == 1 and R2 == 1 and R1 == 1:
      # Stop moving
      twist.linear.x = 0.0
      twist.angular.z = 0.0
      print("All black")
      time.sleep(0.05)
      self.publisher_.publish(twist)

      # HTTP request
      if HTTP_CALL:
          url = "http://192.168.60.159/openDoor"  # Change the IP address to the IP address of the server
          payload = {
              "action": "openDoor",
              "parameters": {
                  "robotId": "<TurtleBot3_ID>"
              }
          }
          headers = {
              "Content-Type": "application/json"
          }
          response = requests.post(url, json=payload, headers=headers)
          if response.status_code == 200:
              json_response = response.json()
              message = json_response["data"]["message"]
              print("Received message:", message)
              stored_message = message
              if stored_message == "door2":
                  twist.linear.x = ROTATE_SPEED
                  twist.angular.z -= ROTATE_CHANGE
                  print("turn right")
              else:
                  twist.linear.x = ROTATE_SPEED
                  twist.angular.z += 0.6
                  print("turn left")
          else:
              print("Error:", response.status_code)
          HTTP_CALL = False
      else:
          if stored_message == "door2":
              twist.linear.x = ROTATE_SPEED
              twist.angular.z -= ROTATE_CHANGE
              print("turn right")
          else:
              twist.linear.x = ROTATE_SPEED
              twist.angular.z += 0.6
              print("turn left")
```

Fig 8.4.4 Code for HTTP Call

NOTE: The IP - address of the ESP-32 is needed to do the HTTP Request, which is provided by the TA when everything is connected on the same local network.

5. There are other cases as well, which need to be considered, such as taking sharp 90 degree turns, as well as nudging on the tape. The bot stops at the final condition when there is no tape to detect, and exits from the loop.

```
elif L1 == 0 and L2 == 0 and R2 == 0 and R1 == 0:
    # Stop moving
    twist.linear.x = 0.0
    twist.angular.z = 0.0
    print("All white")
    break # Exits the loop

elif L2 == 1 and R2 == 1 and L1 == 1:
    # turn left
    twist.linear.x = ROTATE_SPEED
    twist.angular.z += ROTATE_CHANGE
    print("turn left")
elif R1 == 1 and L2 == 1 and R2 == 1:
    # turn right
    twist.linear.x = ROTATE_SPEED
    twist.angular.z -= ROTATE_CHANGE
    # self.publisher_.publish(twist)
    print("turn right")
elif L2 == 1 and R2 == 1:
    # Move forward
    twist.linear.x = SPEED_CHANGE
    twist.angular.z = 0.0
    # self.publisher_.publish(twist)
    print("move straight")
elif (L1 == 1 and L2 == 1) or L1 == 1 or (L2 == 1 and R2 == 0):
    # nudge left
    twist.linear.x = NUDGE_SPEED
    twist.angular.z += NUDGE_CHANGE
    print("nudge left")
elif (R1 == 1 and R2 == 1) or (L2 == 0 and R2 == 1) or R1 == 1 :
    # nudge right
    twist.linear.x = NUDGE_SPEED
    twist.angular.z -= NUDGE_CHANGE
    print("nudge right")
self.publisher_.publish(twist)
time.sleep(0.1)
```

Fig 8.4.5 Cases for Line Following

6. The main function is running in a try-finally block to avoid any crashes on Raspberry Pi. After the function stops running, the node is destroyed in the main.

```
        except KeyboardInterrupt:
            pass
            GPIO.cleanup()
        finally:
            # Stop moving before exiting
            twist.linear.x = 0.0
            twist.angular.z = 0.0
            self.publisher_.publish(twist)


def main(args=None):
    rclpy.init(args=args)

    mover = Mover()
    mover.read_key()

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    mover.destroy_node()

    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

Fig 8.4.6 Main Function for Line Following

8.4.2 Servo motor:

This is the explanation for the servo motor script, used to launch the ping pong balls into the bucket.

1. Firstly, GPIO pin 12 is used for the servo and the PWM with a frequency of 50Hz Is initialised on the pin. The numbering convention of the pin is set to BCM mode. p.start(2.5) allows the servo to start at its initial angle of zero degree.
   Below shows a code snippet of the setup:

```
import time
import RPi.GPIO as GPIO

# Set pin numbering convention
GPIO.setmode(GPIO.BCM)

# Choose PWM channels connected to the servos
servo_pin = 12

# Set the pin as outputs
GPIO.setup(servo_pin, GPIO.OUT)

# Initialize PWM on pin with 50 Hz frequency for servo control
p = GPIO.PWM(servo_pin, 50)

# Start PWM with a duty cycle that sets servos to initial position
p.start(2.5)
```

Fig 8.4.7 Initialising the Servo

2. p.ChangeDutyCycle(9.5) will initiate the servo to launch the ping pong ball. It will then return to the initial position through p.ChangeDutyCycle(2.5).
3. For the payload to stay perpendicular to the turtlebot to prevent the payload from hitting obstacles during Frontier Exploration, we implemented p.ChangeDutyCycle(6.9), followed by p.ChangeDutyCycle(6.5).
4. Time.sleep() is implemented to introduce delays in between servo movements, ensuring smooth transitions.
5. The loop exits after the payload stays perpendicular to the turtlebot using 'break' statement and the PWM operation will stop and the GPIO resources will be released, ensuring that the GPIO pin is properly reset.
6. 'except KeyboardInterrupt' catches a KeyboardInterrupt exception, which occurs when the user presses Ctrl+C to exit the program.
   Here is a code snippet for the function mentioned:

```
try:
    while True:
        p.ChangeDutyCycle(9.5)
        time.sleep(0.8)

        p.ChangeDutyCycle(2.5)
        time.sleep(1)

        p.ChangeDutyCycle(6.9)
        time.sleep(0.45)

        p.ChangeDutyCycle(6.5)
        time.sleep(0.1)

        break

except KeyboardInterrupt:
    # Cleanup on Ctrl+C exit
    pass
finally:
    # Stop PWM and clean up GPIO
    p.stop()
    GPIO.cleanup()
```

Fig 8.4.8 Main Function for Servo Motor

8.4.3 Frontier Exploration

1. The frontier exploration code is a slightly modified version of the code written by AbdulKadir Ture (link in references). The exploration works in the following way. It uses the occupancy grid map and determines all the cells that have a value of 0(unoccupied) with a neighbour of -1(occupied) as frontier cells. Then, all horizontally and vertically connected frontier cells are grouped into frontier groups. Next, it calculates the centroid for all the groups with at least 3 cells in them. Next, it sorts all the groups by increasing size and selects only the 5 biggest groups. And from its current position, it calculates the shortest centroid which is the next target the robot has to go to. This happens until there are no more groups that can be made, in which case the exploration is deemed completed.

```
def exploration(data,width,height,resolution,column,row,originX,originY):
        global pathGlobal #global variable
        data = costmap(data,width,height,resolution) #expand the barriers
        data[row][column] = 0 #Robot Current Location
        data[data > 5] = 1 # Those with a score of 0 are a go-to place, those with a score of 100 are a definite obstacle.
        data = frontierB(data) #Find breakpoints
        data,groups = assign_groups(data) #Group breakpoints
        groups = fGroups(groups) #Sort the groups from smallest to largest. Get the 5 biggest groups
        if len(groups) == 0: #If there is no group, the survey is completed
            path = -1
        else: #If there is a group, find the closest group
            data[data < 0] = 1 #-0.05 ones are unknown location. Mark it as ungoable. 0 = can go, 1 = can't go.
            path = findClosestGroup(data,groups,(row,column),resolution,originX,originY) #Find the closest group
            if path != None: #If there is a path, fix it with BSpline
                path = bspline_planning(path,len(path)*5)
            else:
                path = -1
        pathGlobal = path
```

Fig 8.4.9 Frontier exploration

2. The robot determines frontier cells and sets them to “2” in the frontierB function.

```
def frontierB(matrix):
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            if matrix[i][j] == 0.0:
                if i > 0 and matrix[i-1][j] < 0:
                    matrix[i][j] = 2
                elif i < len(matrix)-1 and matrix[i+1][j] < 0:
                    matrix[i][j] = 2
                elif j > 0 and matrix[i][j-1] < 0:
                    matrix[i][j] = 2
                elif j < len(matrix[i])-1 and matrix[i][j+1] < 0:
                    matrix[i][j] = 2
    return matrix
```

Fig 8.4.10 Assigning frontier cells

3. Next, the assign_groups function assigns all horizontally and vertically connected frontier cells using a Depth First Search algorithm.

```
def assign_groups(matrix):
    group = 1
    groups = {}
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] == 2:
                group = dfs(matrix, i, j, group, groups)
    return matrix, groups

def dfs(matrix, i, j, group, groups):
    if i < 0 or i >= len(matrix) or j < 0 or j >= len(matrix[0]):
        return group
    if matrix[i][j] != 2:
        return group
    if group in groups:
        groups[group].append((i, j))
    else:
        groups[group] = [(i, j)]
    matrix[i][j] = 0
    dfs(matrix, i + 1, j, group, groups)
    dfs(matrix, i - 1, j, group, groups)
    dfs(matrix, i, j + 1, group, groups)
    dfs(matrix, i, j - 1, group, groups)
    dfs(matrix, i + 1, j + 1, group, groups) # lower right diagonal
    dfs(matrix, i - 1, j - 1, group, groups) # upper left cross
    dfs(matrix, i - 1, j + 1, group, groups) # upper right cross
    dfs(matrix, i + 1, j - 1, group, groups) # lower left diagonal
    return group + 1
```

Fig 8.4.11 Assigning frontier groups

4. The function then filters out the five biggest groups and calculates centroid for each of them.

```
def fGroups(groups):
    sorted_groups = sorted(groups.items(), key=lambda x: len(x[1]), reverse=True)
    top_five_groups = [g for g in sorted_groups[:5] if len(g[1]) > 1]
    return top_five_groups

def calculate_centroid(x_coords, y_coords):
    n = len(x_coords)
    sum_x = sum(x_coords)
    sum_y = sum(y_coords)
    mean_x = sum_x / n
    mean_y = sum_y / n
    centroid = (int(mean_x), int(mean_y))
    return centroid
```

Fig 8.4.12 Fetching 5 biggest groups and calculate centroid

5. The closest centroid to the robot's current position is calculated and a path is planned using an A* algorithm.

```
def findClosestGroup(matrix,groups, current,resolution,originX,originY):
    targetP = None
    distances = []
    paths = []
    score = []
    max_score = -1 #max score index
    for i in range(len(groups)):
        middle = calculate_centroid([p[0] for p in groups[i][1]],[p[1] for p in groups[i][1]])
        path = astar(matrix, current, middle)
        path = [(p[1]*resolution+originX,p[0]*resolution+originY) for p in path]
        total_distance = pathLength(path)
        distances.append(total_distance)
        paths.append(path)
    for i in range(len(distances)):
        if distances[i] == 0:
            score.append(0)
        else:
            score.append(len(groups[i][1])/distances[i])
    for i in range(len(distances)):
        if distances[i] > target_error*3:
            if max_score == -1 or score[i] > score[max_score]:
                max_score = i
    if max_score != -1:
        targetP = paths[max_score]
    else: #If the groups are closer than target_error*2 distance, it selects a random point as the target. This allows the robot to escape some situations.
        index = random.randint(0,len(groups)-1)
        target = groups[index][1]
        target = target[random.randint(0,len(target)-1)]
        path = astar(matrix, current, target)
        targetP = [(p[1]*resolution+originX,p[0]*resolution+originY) for p in path]
    return targetP

def pathLength(path):
    for i in range(len(path)):
        path[i] = (path[i][0],path[i][1])
        points = np.array(path)
    differences = np.diff(points, axis=0)
    distances = np.hypot(differences[:,0], differences[:,1])
    total_distance = np.sum(distances)
    return total_distance
```

Fig 8.4.13 Calculation of closest centroid and path planning

6. The following is the implementation of the A* star algorithm.

```
def heuristic(a, b):
    return np.sqrt((b[0] - a[0]) ** 2 + (b[1] - a[1]) ** 2)

def astar(array, start, goal):
    neighbors = [(0,1),(0,-1),(1,0),(-1,0),(1,1),(1,-1),(-1,1),(-1,-1)]
    close_set = set()
    came_from = {}
    gscore = {start:0}
    fscore = {start:heuristic(start, goal)}
    oheap = []
    heapq.heappush(oheap, (fscore[start], start))
    while oheap:
        current = heapq.heappop(oheap)[1]
        if current == goal:
            data = []
            while current in came_from:
                data.append(current)
                current = came_from[current]
            data = data + [start]
            data = data[::-1]
            return data
        close_set.add(current)
        for i, j in neighbors:
            neighbor = current[0] + i, current[1] + j
            tentative_g_score = gscore[current] + heuristic(current, neighbor)
            if 0 <= neighbor[0] < array.shape[0]:
                if 0 <= neighbor[1] < array.shape[1]:
                    if array[neighbor[0]][neighbor[1]] == 1:
                        continue
                else:
                    # array bound y walls
                    continue
            else:
                # array bound x walls
                continue
            if neighbor in close_set and tentative_g_score >= gscore.get(neighbor, 0):
                continue
            if  tentative_g_score < gscore.get(neighbor, 0) or neighbor not in [i[1]for i in oheap]:
                came_from[neighbor] = current
                gscore[neighbor] = tentative_g_score
                fscore[neighbor] = tentative_g_score + heuristic(neighbor, goal)
                heapq.heappush(oheap, (fscore[neighbor], neighbor))
    # If no path to goal was found, return closest path to goal
    if goal not in came_from:
        closest_node = None
        closest_dist = float('inf')
        for node in close_set:
            dist = heuristic(node, goal)
            if dist < closest_dist:
                closest_node = node
                closest_dist = dist
        if closest_node is not None:
            data = []
            while closest_node in came_from:
                data.append(closest_node)
                closest_node = came_from[closest_node]
            data = data + [start]
            data = data[::-1]
            return data
```

Fig 8.4.14 Implementation of A algorithm*

7. The path is smoothed out using a bspline planning algorithm to improve navigation.

```
def bspline_planning(array, sn):
    try:
        array = np.array(array)
        x = array[:, 0]
        y = array[:, 1]
        N = 2
        t = range(len(x))
        x_tup = si.splrep(t, x, k=N)
        y_tup = si.splrep(t, y, k=N)

        x_list = list(x_tup)
        xl = x.tolist()
        x_list[1] = xl + [0.0, 0.0, 0.0, 0.0]

        y_list = list(y_tup)
        yl = y.tolist()
        y_list[1] = yl + [0.0, 0.0, 0.0, 0.0]

        ipl_t = np.linspace(0.0, len(x) - 1, sn)
        rx = si.splev(ipl_t, x_list)
        ry = si.splev(ipl_t, y_list)
        path = [(rx[i],ry[i]) for i in range(len(rx))]
    except:
        path = array
    return path
```

Fig 8.4.15 bspline planning algorithm

8. Using the path, the robot then navigates to the next target. It waits till it receives odom, map and scan data before calculating the path. If the path returns -1 (no path found), the exploration is deemed to have been completed.

```
class navigationControl(Node):
    def __init__(self):
        super().__init__('Exploration')
        self.subscription = self.create_subscription(OccupancyGrid,'map',self.map_callback,qos_profile_sensor_data)
        self.subscription = self.create_subscription(Odometry,'odom',self.odom_callback,10)
        self.subscription = self.create_subscription(LaserScan,'scan',self.scan_callback,qos_profile_sensor_data)
        self.subscription
        self.publisher = self.create_publisher(Twist, 'cmd_vel', 10)
        print("[INFORMATION] CUTTING MODE ACTIVE")
        self.kesif = True
        threading.Thread(target=self.exp).start() #Extract function as thread calistirir.

    def exp(self):
        twist = Twist()
        while True: #Wait until sensor data arrives.
            if not hasattr(self,'map_data') or not hasattr(self,'odom_data') or not hasattr(self,'scan_data'):
                time.sleep(0.5)
                print("scan:",hasattr(self,'scan_data'),"odom:",hasattr(self,'odom_data'),"map:",hasattr(self,'map_data'))
                print(self.kesif)
                continue
            if self.kesif == True:
                if isinstance(pathGlobal, int) and pathGlobal == 0:
                    column = int((self.x - self.originX)/self.resolution)
                    row = int((self.y- self.originY)/self.resolution)
                    exploration(self.data,self.width,self.height,self.resolution,column,row,self.originX,self.originY)
                    self.path = pathGlobal
                else:
                    self.path = pathGlobal
                if isinstance(self.path, int) and self.path == -1:
                    print("[INFORMATION] EXPLORATION COMPLETED")
                    sys.exit()
                self.c = int((self.path[-1][0] - self.originX)/self.resolution)
                self.r = int((self.path[-1][1] - self.originY)/self.resolution)
                self.kesif = False
                self.i = 0
                print("[INFORMATION] NEW TARGET HAS BEEN DETERMINED")
                t = pathLength(self.path)/speed
                print(t)
                t = t - 0.2 #0.2 seconds is subtracted from the calculated time according to the formula x = v * t. After t time, the discovery function is run.
                self.t = threading.Timer(t,self.target_callback) #Activates the reconnaissance function shortly before the target..
                self.t.start()

            #Route Tracking Block Start
            else:
                v , w = localControl(self.scan)
                if v == None:
                    v, w,self.i = pure_pursuit(self.x,self.y,self.yaw,self.path,self.i)
                if(abs(self.x - self.path[-1][0]) < target_error and abs(self.y - self.path[-1][1]) < target_error):
                    v = 0.0
                    w = 0.0
                    self.kesif = True
                    print("[INFORMATION] GOAL ACHIEVED")
                    self.t.join() #Wait until the thread finishes.
                twist.linear.x = v
                twist.angular.z = w
                self.publisher.publish(twist)
                time.sleep(0.1)
            #Route Tracking Block End
```

Fig 8.4.16 Code Snippet for Navigation

9. The obstacle avoidance logic works by checking if there are any obstacles in the 0 to ⅙ of lidar readings range. If detected, the bot swerves (right) the obstacles by setting a linear velocity of 0.15 and an angular velocity of $\frac{\pi}{2}$. Likewise, if an obstacle is detected in the ⅚ lidar readings to lidar readings range, the angular velocity is set to $-\frac{\pi}{2}$ instead so that the object swerves left. The scale was originally 0,60 and 300, 360 but since the lidar readings range is not consistently 360 and often varied from 270 to 290 in our case, the scan range was scaled.

```
def localControl(scan):
    scan_range=len(scan)
    v = None
    w = None
    #for i in range(int((1/6)*(scan_range)))
    for i in range(0,int((1/6)*(scan_range))):
    #range(60)
        if scan[i] < robot_r:
            v = 0.15
            w = -(math.pi/2) #pi/2
            break
    if v == None:
        for i in range(int((5/6)*(scan_range)),scan_range):
        #for i in range(int((5/6)*(scan_range)),scan_range)
        #(300,360)
            if scan[i] < robot_r:
                v = 0.15
                w = (math.pi)/2 #pi/2
                break
    return v,w
```

Fig 8.4.17 Obstacle avoidance

The reading range, linear and angular velocities can be adjusted according to the map conditions.

10. There are also various parameters found in the params.yaml file that can be adjusted.

```
lookahead_distance : 0.24 #forward looking distance
speed : 0.15 #maximum speed
expansion_size : 2 #wall expansion coefficient
target_error : 0.15 #margin of error to target
robot_r : 0.25 #robot distance for local security
```

Fig 8.4.18 Parameters in params.yml file

Lookahead distance specifies how dependent the robot will be on the path. Speed is the maximum speed of the robot. Expansion_size determines the factor by which obstacles would be expanded (we noticed that setting this to 3 prevented the bot from handling gaps that were exactly two times the width of the turtlebot, but

performed much better than 2 otherwise). Target_error is the margin of error in reaching the target. Robot_r is needed for the localControl function as mentioned above. It sets a radius around the robot which determines the closest distance that the algorithm considers when an obstacle is detected.

9. System Operation Manual

9.1 Software boot-up commands

1) Ensure laptop and RPi are connected to the same network.
2) In the first terminal,

```
sshrp #ssh into turtlebot
Rosbu #after successful ssh into RPi
```

3) In a new terminal,

```
ros2 launch slam_toolbox online_async_launch.py #run slam toolbox
```

4) In a new terminal,

```
rviz2 #to run rviz
```

5) In a new terminal,

```
sshrp #ssh into Rpi from another terminal
cd colcon_ws/src #go to the directory after ssh
./run_scripts.sh #run the bash file with all the scripts
```

10. Future scope of expansion

10.1 Mechanical

One of the key challenges with the mechanical design was the stability of the payload. Being connected to just one arm and servo motor. The payload was quite bouncy which could have affected the accuracy of the ball delivery. Furthermore, upon launching the balls, the payload did not always stand up straight and often fell back or fell forward. This meant that the frontier algorithm couldn't work perfectly since the payload was hitting walls and getting stuck in corners whenever the turtlebot worked. The frontier exploration worked much better when the payload was kept straight above the turtlebot. A decision for such a mechanical design was taken because it was believed that any payload design in front of the LIDAR may obstruct the LIDAR readings when mapping. However, towards the end of the design process, it was discovered that the payload mechanism, if too close to the LIDAR, wouldn't affect its readings. Hence, in the future, a more stable ball dropping mechanism can be potentially implemented which can be mounted on top of the LIDAR. The mechanism could store the balls during runs and have a gate which prevents the balls from falling out. The gate could be opened for the balls to drop out. This would also be much more stable, less bouncy and would not interfere with the frontier exploration as well.

10.2 Electrical

The electrical subsystem faced a significant challenge with unstable interconnections caused by wiring issues. For instance, there were instances of Raspberry Pi (SBC) rebooting due to insufficient power supply from the perfboard, hindering the computer's booting process. In future, implementing a robust electrical wiring system is crucial for the TurtleBot to overcome these issues and carry out its tasks effectively.

Furthermore, the line sensors encountered calibration issues, leading to inconsistencies among sensors. This inconsistency resulted in false junction detections or incorrect turns. Utilising a line sensor array would offer significant advantages in avoiding such calibration errors.

10.3 Software

For convenience, the scripts were divided based on their functions. In our case, we utilised a bash script to streamline automation. However, this approach encountered memory management issues and could overload the RPi's memory. Alternatively, going forward Python's inheritance functionality can be leveraged to enhance robustness and memory management.

10.3.1 Line Follower

The Line follower algorithm worked as intended but to make it more robust, a cache system can also be implemented in order to avoid accidental detections.

10.3.2 Servo Motor

The servo motor code successfully launches the ping pong balls into the bucket, but there are still inconsistencies in maintaining the payload at a perpendicular angle to the turtlebot. Therefore, more full trial runs of the maze should be conducted instead of just testing from the junction to the bucket. This will allow us to make necessary tweaks and ensure consistent performance.

10.3.3 Frontier Exploration

A key challenge with the frontier algorithm was the lack of control over the algorithm. Since it was a slightly modified version of someone else's code, it did not allow for too much modification or customisation. Hence, writing our own version of the frontier code could have been helpful in this case. Further, the frontier exploration often struggled to handle small gaps (twice the turtlebot's width exactly). Special conditions could be written to handle such specific cases. Finally, different parameters seemed to work in different maps. This leaves scope for the development of an algorithm that is dynamic enough to be able to handle different mazes using similar parameters.

11. References

[1] “A* search,” Brilliant Math & Science Wiki, https://brilliant.org/wiki/a-star-search/ (accessed Feb. 2, 2024).

[2]“(PDF) Analysis of algorithms for shortest path problem in parallel,” *ResearchGate*. https://www.researchgate.net/publication/304665934_Analysis_of_algorithms_for_shortest_path_problem_in_parallel

[3]“Line sensors and how to use them,” *FutureLearn*. https://www.futurelearn.com/info/courses/robotics-with-raspberry-pi/0/steps/75899

[4] Wikipedia Contributors, “Depth-first search,” *Wikipedia*, May 13, 2019. https://en.wikipedia.org/wiki/Depth-first_search

[5] Wikipedia Contributors, “Breadth-first search,” *Wikipedia*, May 28, 2019. https://en.wikipedia.org/wiki/Breadth-first_search

[6][A. TÜRE, “abdulkadrtr/ROS2-FrontierBaseExplorationForAutonomousRobot,” *GitHub*, Apr. 06, 2024. https://github.com/abdulkadrtr/ROS2-FrontierBaseExplorationForAutonomousRobot

[7]“GET and POST requests using Python,” *GeeksforGeeks*, Dec. 07, 2016. https://www.geeksforgeeks.org/get-post-requests-using-python/

[8]“Requests: HTTP for Humans™ — Requests 2.31.0 documentation,” *docs.python-requests.org*. http://docs.python-requests.org/en/master/ (accessed Feb. 02, 2024).

[9]“Robotics Ping Pong Ball Shooter Lab Report,” *Google Docs*. https://docs.google.com/document/d/150IR2ZaPNnfbJ02hdPq4gAjhxtInwHWYUCMfmHGjkgk/preview?hgd=1 (accessed Feb. 02, 2024).

[10]“Puncher Help,” *VEX Forum*, Sep. 03, 2018. https://www.vexforum.com/t/puncher-help/49129

12. Annex

12.1 Annex A - Power Budget Table

Elements	**Voltage (V)**	**Current (A)**	**Power (W)**
Turtlebot	11.1	Boot : 1.5	Boot : 12.77
		Operation : 1.13	Operation : 12.5
		Standby : 0.75	Standby : 8.33
4 * Line Sensor	3.3	0.02	0.066 * 4 = 0.264
Servo Motor	5	0.9	4.5

12.2 Annex B - Preliminary Monetary Budget

Materials	**Quantity**	**Cost (S$) (Links)**
Tape	2	2.1*2 (Link)
Line Sensor	4	1*4 (Link)
MG996R Servo motor	2	9 (Link)
M6x12 Bolts	10	1.3 * 5 (Link)
M6 Nuts	10	1.3 * 5 (Link)
M3 Bolts	4	1.58 *2 (Link)
M3 Bolts	4	1.58 * 2 (Link)
Acrylic sheet 10mm	1	$3 (Link)
Acrylic sheet 3mm	1	$5 (Link)
PLA	NA	$5/hr, 4hr-> $20
Steel Ring	1 piece	$15 (quote from CW)
Real Cost of Design		$88.5