NUS

National University
of Singapore

EG2310 Fundamentals of Systems Design G2 Report

“OpenBurger”

| Group 4 |

Soh Sze Juin	A0276220M
Teh Wei Sheng	A0276148W
Tong Jing Yen	A0288445N
Chang Shu Kai	A0288452U
Ling Zi Yan	A0273061M

Table of Contents

Chapter 1 | Introduction

1.1 Problem Definition

The central objective of this project is to design and construct a robotic system capable of successfully navigating, communicating with secured doors (referred to as the "Elevator"), and executing a ping pong ball delivery mechanism. The mission, denoted as **the "Elevator" Problem**, is divided into three distinct components: *autonomous navigation*, *communication with secured doors*, and the *implementation of a ping pong ball delivery mechanism*. The map comprises three key sections: the start zone, random maze element zones, and the locked room featuring electronic secured doors, also known as the delivery zone. The mission initiates from the start zone, prompting the robot to autonomously traverse through the random maze elements, subsequently locate the locked room with secured doors, gain access through the secured door, identify the bucket within the room, and ultimately execute the delivery of five ping pong balls into the designated bucket, thereby concluding the mission. Additionally, bonuses are given for the robot's capability to autonomously generate a complete SLAM map of the maze during the mission.

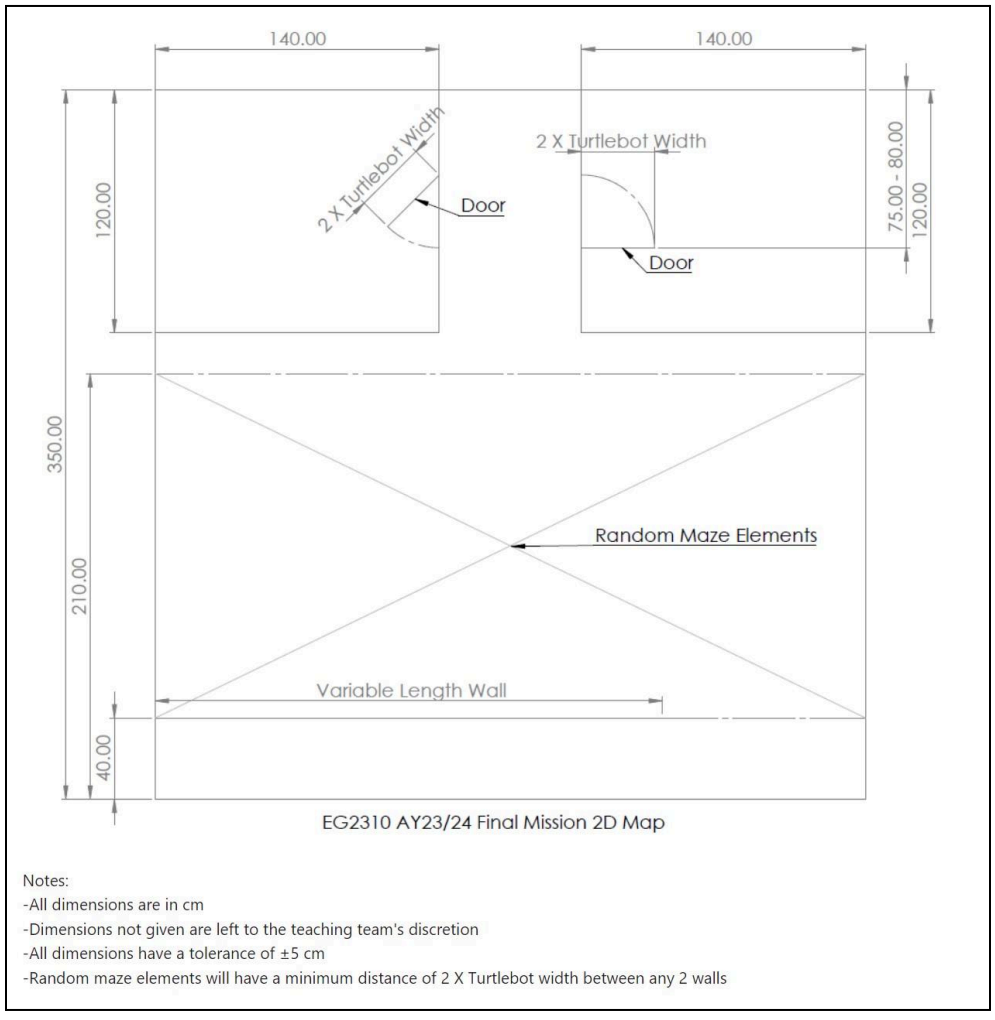


Figure 1.1: Mission 2D Map

1.2 System Requirements Analysis

1.2.1 Project Deliverables

As per the stipulated mission requirements provided by stakeholders, the ensuing table presents an overview of the anticipated project deliverables.

Stakeholder's Requirements	Project Deliverables
Traverse the map autonomously from the start zone	System shall implement mapping techniques to create a real-time map of the maze environment
Navigate through the random maze element zones autonomously	System shall develop the capability for obstacle detection and avoidance during navigation
Recognise different sections of the map	System shall incorporate a system to identify different zones within the map
Recognise the "Elevator" doors	System shall develop the capability to recognise the "Elevator" doors
Communicate with the "Elevator" doors	System shall design a mechanism for the robot to communicate with the "Elevator" door
Navigate through the "Elevator" doors	System shall develop the capability for the robot to enter the correct "Elevator" door
Locate the bucket in the locked room	System shall develop capability to accurately locate the bucket
Deliver ping pong balls into the bucket	System shall implement a ping pong ball delivery mechanism capable of delivering up to 5 ping pong balls into the bucket.

Table 1.2.1: Project Deliverables

1.2.2 Functional Requirements

Stakeholder's Requirements	Functional Requirements
Traverse the map autonomously from the start zone	System shall implement Simultaneous Localization and Mapping (SLAM) techniques to create a real-time map of the maze environment. System shall continuously update and refine the map during exploration.
Navigate through the random maze element zones autonomously	System shall utilise sensors and developed map to promptly detect obstacles and navigate around them to ensure smooth traversal.
Recognise different sections of the map	System shall implement the use of temporary markers to accurately recognize and differentiate between the start zone, random maze element zone, and delivery zone.
Recognise the "Elevator" doors	System shall implement the use of temporary markers to designate points of interest, such as "Elevator" doors.
Communicate with the "Elevator" doors	System shall develop the capability to securely communicate with the server by sending HTTP calls to request door unlocking based on predefined protocols.
Navigate through the "Elevator" doors	System shall implement reliable communication protocols for the robot to enter the unlocked "Elevator" door after sending an HTTP call.
Locate the bucket in the locked room	System shall implement and employ sensor data and marker recognition algorithms for precise identification of the bucket within the delivery zone.
Deliver ping pong balls into the bucket	System shall ensure precise positioning and release of the 5 ping pong balls for successful delivery.

Table 1.2.2: Functional Requirements

1.2.3 Non-functional Requirements

Stakeholder's Requirements	Non-functional Requirements
Performance	System shall accomplish the entire mission, including the setup of temporary markers, generation of SLAM map, and removal of markers, within a time limit of 20 minutes.
Reliability	System shall operate reliably under various environmental conditions, including different maze configurations and lighting conditions. System shall be resilient to sensor noise and external disturbances.
Safety	System shall prioritise safety during operation to prevent collisions with obstacles and map elements.
Scalability	System shall be designed with scalability in mind to accommodate future enhancements or modifications. System shall be capable of adapting to changes in maze layouts or task requirements.
Usability	System shall have a user-friendly interface for easy setup, operation, and monitoring by human operators.
Accuracy	System shall achieve high accuracy in map generation, obstacle detection, and task execution. System shall minimise errors in localization and navigation to ensure precise performance.
Maintainability	System shall be designed for ease of maintenance and repair. System shall allow for quick diagnosis and replacement of any faulty components.
Verification	System shall record the process of generating SLAM map for verification purposes
Power Efficiency	System shall optimise power usage to prolong battery life by employing energy-efficient algorithms and hardware components.

Table 1.2.3: Non-functional Requirements

1.2.4 Constraints

Constraint	Description
Physical Size and Weight	The robot must adhere to size and weight limitations to ensure compatibility with the maze environment and manoeuvrability through narrow passages and the ‘Elevator’ door.
Power Supply Limitations	The robot must operate within constraints imposed by the power source, such as the battery capacity and voltage requirements.
Sensor and Actuator Limitations	The selection and integration of sensors and actuators are constrained by factors such as cost, availability, and compatibility with the robot operating platform - ROS2.
Processing power and Memory	The onboard processing capabilities and memory resources of the robot system component (Raspberry Pi) are limited, affecting the complexity and efficiency of algorithms and computations.
Communication Range and Bandwidth	The robot system's communication capabilities are limited by the range and bandwidth of wireless communication protocols, such as Wi-Fi or Bluetooth.
Environmental	Environmental factors such as variations in lighting conditions, surface textures, and ambient noise levels may affect sensor performance, navigation accuracy, and overall system reliability.
Cost	The development and deployment of the robot system are constrained by a budget limitation of $100. Thus, cost-effective solutions and prioritisation of resources are essential to stay within budget constraints while meeting project objectives.

Table 1.2.4: Constraints

1.3 Safety Precautions

Adherence to the following safety precautions is imperative when handling the robot system. Read all safety measures before operating the system.

1.3.1 Handling

Handle the system with the utmost care and attention to detail. Avoid subjecting it to harsh conditions such as dropping, burning, puncturing, crushing, or exposure to water, as such actions can compromise its integrity and functionality. Any observed damage should prompt immediate disconnection from power sources for inspection and repair.

When working with joints or connections within the system, exercise caution to prevent injury from sharp edges. To mitigate the risk of cuts or abrasions, ensure that all the joints are folded or trimmed neatly to eliminate any protruding or jagged edges. Additionally, wear appropriate protective gloves to shield hands from potential sharp edges while handling the robot.

1.3.2 Static Electricity Management

In environments susceptible to static electricity, take proactive measures to mitigate risks. Grounding oneself before handling the system is essential to dissipate any accumulated static charge and safeguard sensitive electronic components from damage.

1.3.3 Battery Charging Practices

Adhere strictly to the prescribed battery charging procedures to safeguard against potential hazards. Use only the designated charger provided with the system, as employing incompatible chargers or prolonged charging periods may compromise battery performance and pose risks of fire or explosion.

Chapter 2 | System Overview

2.1 Description of the Robot System

Our robot, also known as the **OpenBurger**, is a platform robot built upon the ROS2 framework and utilising the TurtleBot chassis. It serves a specific mission of launching a payload of 5 ping pong balls into a designated bucket while autonomously navigating through maze-like environments. Derived from the TurtleBot3 Burger, the OpenBurger has been modified to incorporate a payload system and a line-following system employing OpenCV.

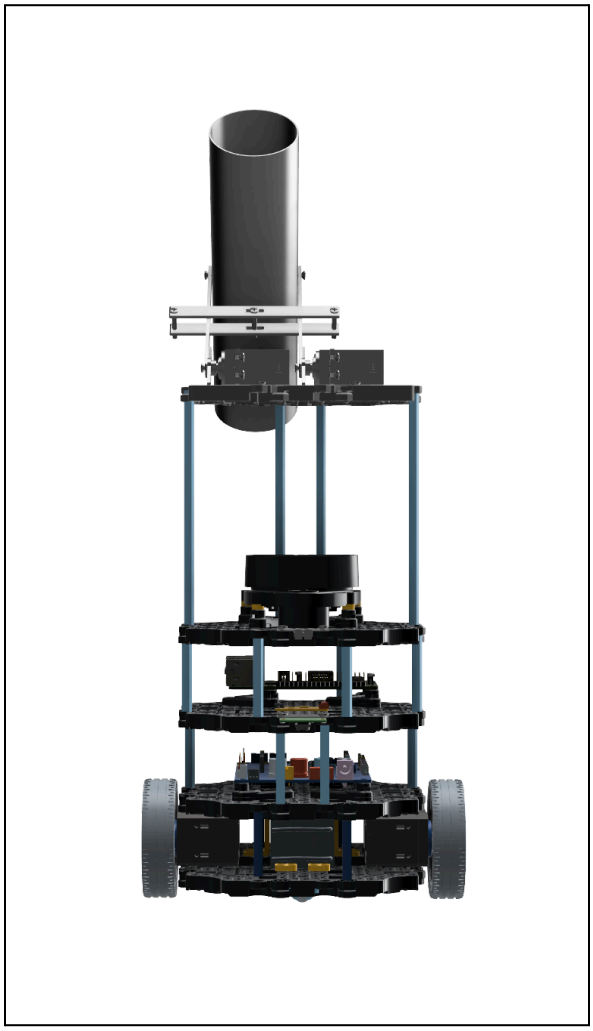


Figure 2.1a: Front view of the OpenBurger

Figure 2.1b: Side view of the OpenBurger

The OpenBurger consists of four distinct subsystems: the **TurtleBot3 system**, the **line follower system**, the **mechanical arm system**, and the **ping pong ball holder system**. These subsystems are depicted and labelled below for clarity:

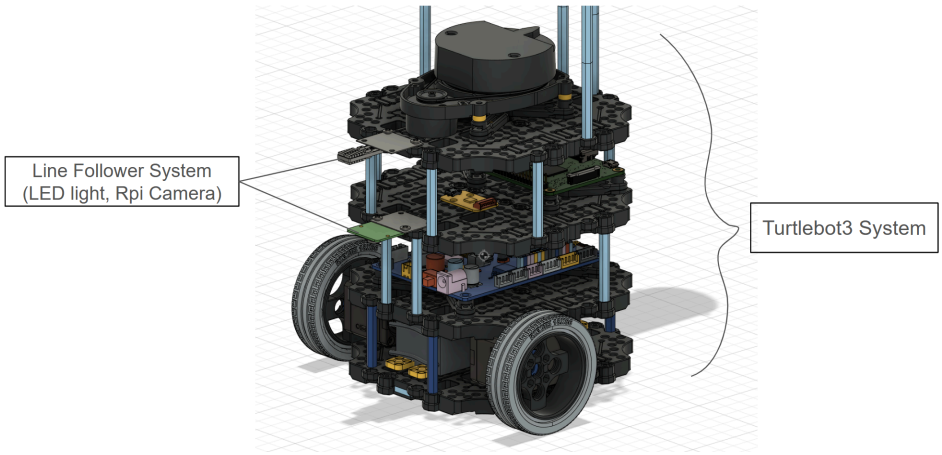


Figure 2.1c: Turtlebot3 system & line follower system

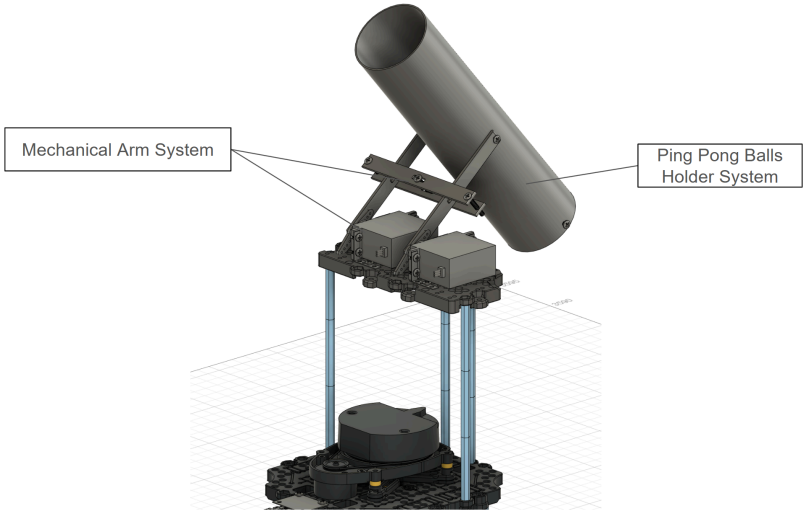


Figure 2.1d: Mechanical subsystems

The **TurtleBot3 system** is assembled following the official guidelines outlined in the ROBOTIS e-manual. According to the ROBOTIS website, TurtleBot3 is a compact, cost-effective, programmable, ROS-based mobile robot designed for applications in education, research, hobbies, and product prototyping. Key features of TurtleBot3 include SLAM, Navigation, and Manipulation technologies, rendering it well-suited for tasks in home service robotics. With its capability to execute SLAM (Simultaneous Localization and Mapping) algorithms for map construction, the TurtleBot is aligned with the objectives of our mission.

The primary purpose of the **line follower system** on our robot is for autonomous navigation. We implemented a coloured line following strategy using a Raspberry Pi camera and the OpenCV library. Our system consists of laying tape of multiple colours, each with a distinct action or acting as a guide for the robot to follow.

Instead of relying on line sensors, our line follower system relies on camera vision, using the OpenCV software stack with input from Raspberry Pi Camera 2. The image feed is passed through two main algorithms, colour detection and contour detection.

The colour detection algorithm determines the action to be carried out by identifying the colour of the tape segment (or segments) in front of the OpenBurger. Hence, the algorithm runs multiple times per frame, once per colour. Here is the list of recognized colours and their actions:

Colour of tape	Action to take
Blue	Continue moving by following the tape (from the maze to one of the buckets)
Yellow	This tape occurs twice. For each occurrence, its function differs: 1. Send HTTP request and lower the mechanical arm 2. Raise the mechanical arm and drop the ping-pong balls into the bucket
Red	Continue moving by following the tape (one of the paths to bucket)
Purple	Same action as blue (not used for final run)
Orange	Same action as yellow (not used for final run)

Table 2.1: Colour of tape and Actions

The contour detection algorithm is used to drive the OpenBurger based on the route charted out by the blue or red tape. The two main parameters of the drive system obtained by the algorithm are the angle of the tape and its placement to the left or right of the robot. These outputs are passed to a drive function with calibrated values to determine the angular velocity and linear velocity of OpenBurger.

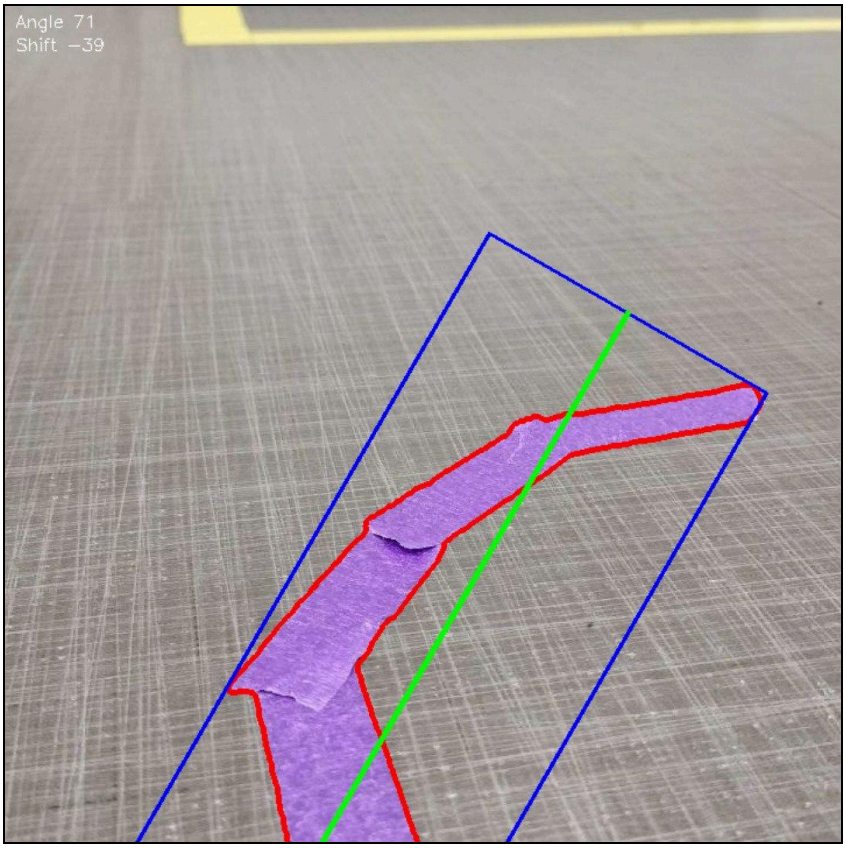


Figure 2.1e: Contour detection

Initially, OpenBurger looks for the blue coloured tape and follows it to navigate the maze. The yellow coloured tape signifies the end of the maze and the start of the elevator zone. At the elevator lobby zone, the path marked by the blue tape splits into paths of blue and red to the two elevators and buckets. The yellow colour will alert OpenBurger to send a HTTP request to the ESP32, and receive a response packet containing the door ID. The OpenBurger will pick the path according to the packet sent by the ESP32.

Continuing from the path, the OpenBurger will pass through the elevator and encounter another segment of yellow tape in front of the bucket. The OpenBurger will stop and drop the ping-pong balls into the bucket, and return to the starting zone of the elevator, where it will switch to autonomous maze exploration upon detecting the yellow marker.

The **mechanical arm system** of the OpenBurger features a strong mechanical structure specifically crafted to support the **ping pong ball holder system** securely. We modified the original Turtlebot3 system by adding an additional waffle plate atop the LIDAR layer, elevating its height through the use of supports. Two MG995 servo motors are fixed onto this waffle plate, and the 3D printed arms are attached onto the servo motors. These servo motors facilitate movements, and the mechanical arms allow the system to accommodate a PVC pipe capable of holding up to 5 ping pong balls. The design incorporates 1 degree of freedom, enabling

adjustments both at the servo motors joints and at the joint connecting the mechanical arm to the PVC pipe. This configuration empowers our payload system to finely tune the launching angle in accordance with the height of the target bucket.

The OpenBurger is also capable of sending **Hypertext Transfer protocol (HTTP) requests** to the server. The only requirement is that the Raspberry Pi on the OpenBurger is connected under the same local network with the server, it will act as a client and send a request to the server. The request will then be transferred to the ESP32 server through the local network in the format of bits. Once the ESP32 receives the request, it will send its response which will also be sent to the OpenBurger Raspberry Pi through the local network in the format of bits. Finally, the OpenBurger will then navigate to the opened door based on the response obtained.

2.2 System Technical Specifications

2.2.1 Turtlebot Specifications

Items	Specifications
Maximum translational velocity	0.22 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)
Maximum payload	15kg
Size (L x W x H)	138mm x 178mm x 192mm
Weight	1kg
Threshold of climbing	10 mm or lower
Expected operating time	2h 30m
Expected charging time	2h 30m
SBC (Single Board Computers)	Raspberry Pi
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Actuator	XL430-W250
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01
Camera	Raspberry Pi Camera Module 2
IMU	Gyroscope 3 Axis Accelerometer 3 Axis
Power connectors	3.3V / 800mA 5V / 4A 12V / 1A
Expansion pins	GPIO 18 pins Arduino 32 pin
Peripheral	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5 pin OLLO x4
DYNAMIXEL ports	RS485 x 3, TTL x 3
Audio	Several programmable beep sequences
Programmable LEDs	User LED x 4
Status LEDs	Board status LED x 1

	Arduino LED x 1 Power LED x 1
Buttons and Switches	Push buttons x 2, Reset button x 1, Dip switch x 2
Battery	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C
PC connection	USB
Firmware upgrade	via USB / via JTAG
Power adapter (SMPS)	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A

Table 2.2.1: Turtlebot Specifications

2.2.2 Ping Pong Ball Launcher Specifications

Items	Specifications
Overall	
Size (L x W x H)	220mm x 110mm x 237mm
Weight	245g
MG995 Servo Motor	
Torque	208 oz-in at 6V, 180 oz-in at 4.8V
Speed	0.13 sec/60° at 6V, 0.17 sec/60° at 4.8V
Weight	55 g
Size (L x W x H)	40.7mm x 19.7mm x 42.9mm
Operating Voltage	4.8V - 7.2V
Mechanical arms (rectangular shaped)	
Size (L x W x H)	98mm x 10mm x 2mm
Weight	15g
PVC pipe	
Length	226mm
Internal diameter	50mm

External diameter	52mm
Weight	75g

Table 2.2.2: Ping Pong Ball Launcher Specifications

2.2.3 Line Follower System Specifications

Items	Specifications
Raspberry Pi Camera Module 2	
Size (L x W x H)	25mm x 23mm x 9mm
Weight	3g
Features	8 megapixel native resolution sensor-capable of 3280 x 2464 pixel static images
	Supports 1080p30, 720p60 and 640x480p90 video
	Connects to the Raspberry Pi board via a short ribbon cable
	Supported in the latest version of Raspberry Pi OS,
LED light	
Colour	Cool white
Luminous intensity	150-200 mcd
Forward Current	30mA
Reverse Voltage	5V

Table 2.2.3: Line Follower System Specifications

2.3 Design Subsystems

2.3.1 Mechanical Subsystem

For the ping pong ball delivery mechanism, we implemented a straightforward yet consistent ball dropper mechanism. It operates by rotating the mechanical arm precisely by 90 degrees. This rotation is achieved by adjusting the PWM signal value of the servo motors. The image below illustrates the concept of our ping pong ball delivery system, showing the process from the HTTP request area to the ping pong ball delivery area.

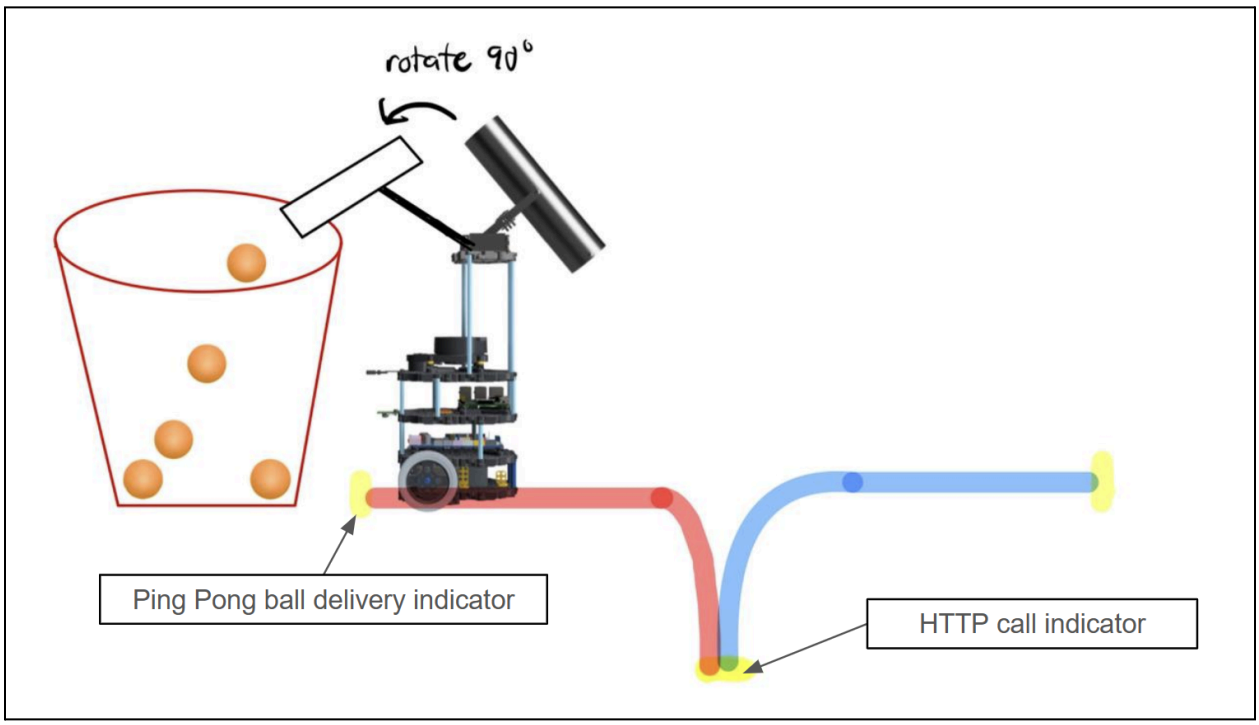


Figure 2.3.1a: Concept of ball dropper mechanism

The yellow coloured tape placed in front of the bucket serves as a marker for initiating the ping pong ball dropping mechanism, which entails steadily rotating the servo motors by 90 degrees. We have meticulously calibrated the servo motor to ensure precise delivery of all 5 ping pong balls into the bucket, without the PVC pipe making contact with the bucket, thus avoiding penalties during our mission.

We have integrated an additional feature into the ping pong ball delivery system: the ability to lower the mechanical arm to pass through doors without contacting the ceiling. This action occurs when the robot reaches the HTTP call marker, simultaneously triggering both the sending of the HTTP request and the lowering of the mechanical arm. The image below illustrates the system in the 'lowered arm' state, accompanied by an image showing the robot successfully passing through the doorway.

Figure 2.3.1b: 'lowered arm' state of robot

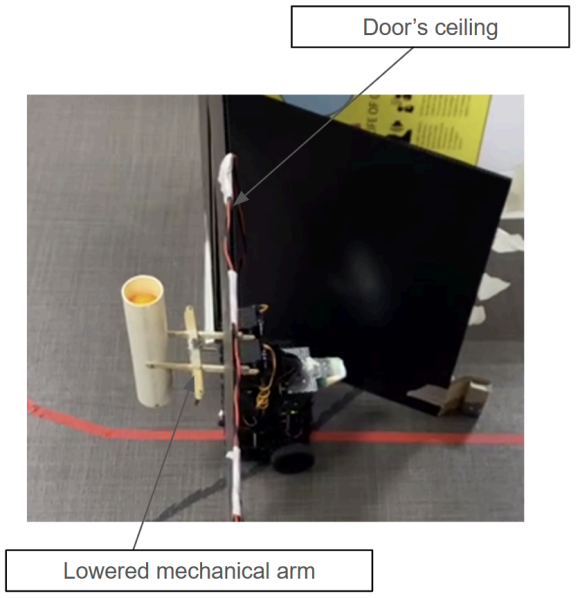


Figure 2.3.1c: Robot passing through doorway

We have calculated the overall centre of gravity for the OpenBurger, excluding screws, nuts, and parts with negligible mass. The centre of gravity for the OpenBurger is located at coordinates (x = 0, y = 0.17, z = 12.52) cm with reference to the labelled axes and a reference point as shown in figure 2.3.1d. Based on this centre of gravity analysis, we are confident in the stability of our robot during mission execution. The proximity of the x and y coordinates to the centre, combined with the z axis not being half of the OpenBurger's height, ensures balanced weight distribution.

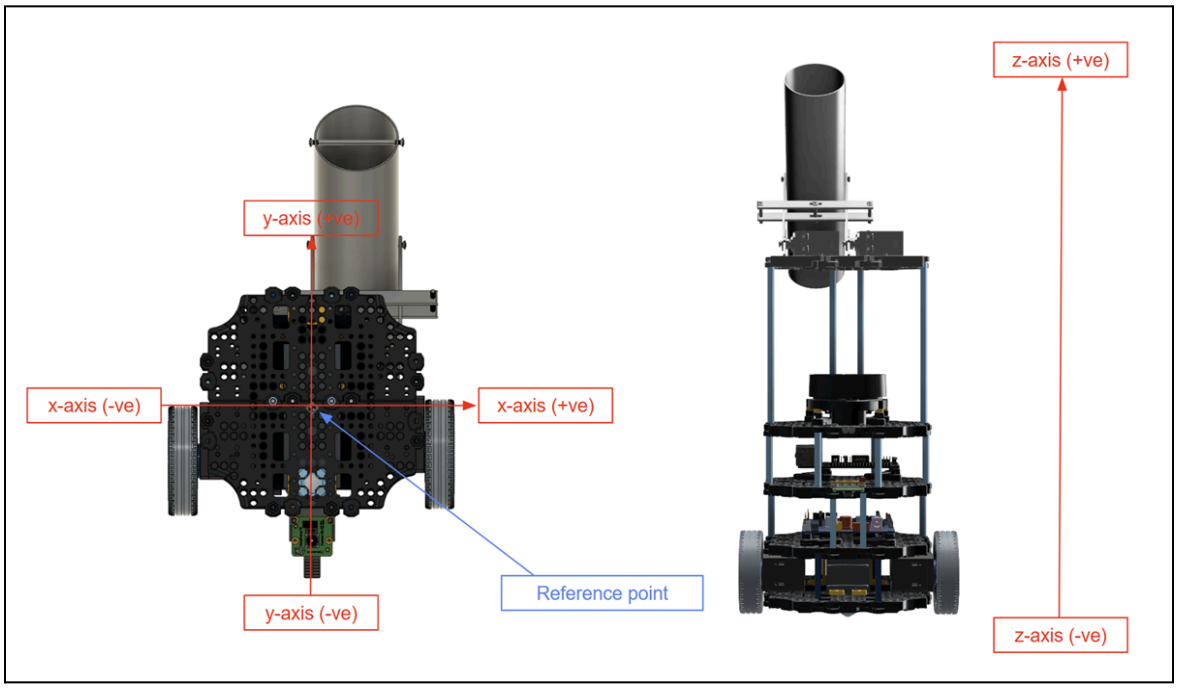


Figure 2.3.1d: Reference point and label of axes for calculation of centre of gravity

Below is the detailed breakdown of the centre of gravity calculation for the OpenBurger, excluding screws, nuts, and components with negligible mass:

Part	Weight(g)	X-position(cm	Weight x X-position	Y-position(cm)	Weight x Y-position	Z-position(cm)	Weight x Z-position
Turtlebot3 Components:							
Dynamixel motor (Right)	57	5	285	-2	-114	2.5	142.5
Dynamixel motor (Left)	56	-5	-280	-2	-112	2.5	140
RPI 4	45	0	0	3	135	11	495
OpenCR1.0	62	0	0	0	0	6	372
Lidar	129	0	0	0	0	17.5	2257.5
LED light + veroboard	50	0	0	-7	-350	11.8	590
Rpi Camera Module 2	3	0	0	-7	-21	6	18
Battery	139	0	0	0	0	2.5	347.5
PCB support x 4 (2nd layer)	9.5	0	0	0	0	5	47.5
PCB support x 4 (3rd layer)	9.5	0	0	3	28.5	10	95
PCB support x 4 (4th layer)	9.5	0	0	0	0	15	142.5
Wheel + tyre (Right)	42	8	336	-3.5	-147	2	84
Wheel + tyre (Left)	42	-8	-336	-3.5	-147	2	84
Waffles plates (1st layer)	35.2	0	0	0	0	0.4	14.08
Waffles plates (2nd layer)	35.2	0	0	0	0	4	140.8
Waffles plates (3rd layer)	35.2	0	0	0	0	8.9	313.28
Waffles plates (4th layer)	35.2	0	0	0	0	13.7	482.24
Support M3x45 (1st to 2nd layer)	6.6	-2.5	-16.5	-6.5	-42.9	2	13.2
Support M3x45 (1st to 2nd layer)	6.6	2.5	16.5	-6.5	-42.9	2	13.2
Support M3x45 (1st to 2nd layer)	6.6	-2.5	-16.5	6.5	42.9	2	13.2
Support M3x45 (1st to 2nd layer)	6.6	2.5	16.5	6.5	42.9	2	13.2
Support M3x45 (2nd to 3rd layer)	6.6	-6.4	-42.24	2.4	15.84	6.9	45.54
Support M3x45 (2nd to 3rd layer)	6.6	6.4	42.24	2.4	15.84	6.9	45.54
Support M3x45 (2nd to 3rd layer)	6.6	-1.25	-8.25	-6.5	-42.9	6.9	45.54
Support M3x45 (2nd to 3rd layer)	6.6	1.25	8.25	-6.5	-42.9	6.9	45.54
Support M3x45 (3rd to 4th layer)	6.6	-2.5	-16.5	-6.5	-42.9	11.8	77.88
Support M3x45 (3rd to 4th layer)	6.6	2.5	16.5	-6.5	-42.9	11.8	77.88
Support M3x45 (3rd to 4th layer)	6.6	-6.5	-42.9	-2.5	-16.5	11.8	77.88
Support M3x45 (3rd to 4th layer)	6.6	6.5	42.9	-2.5	-16.5	11.8	77.88
Support M3x45 (3rd to 4th layer)	6.6	-2.5	-16.5	6.5	42.9	11.8	77.88
Support M3x45 (3rd to 4th layer)	6.6	2.5	16.5	6.5	42.9	11.8	77.88
Bracket 1	3.75	0.9	3.375	-5.5	-20.625	1.4	5.25
Bracket 2	3.75	-0.9	-3.375	-5.5	-20.625	1.4	5.25
Bracket 3	3.75	2.5	9.375	0.7	2.625	1.6	6
Bracket 4	3.75	-2.5	-9.375	0.7	2.625	1.6	6
Support M3x45 (4th to 5th layer)	6.6	-6.4	-42.24	1.2	7.92	23	151.8
Support M3x45 (4th to 5th layer)	6.6	6.4	42.24	1.2	7.92	23	151.8
Support M3x45 (4th to 5th layer)	6.6	-1.25	-8.25	6.5	42.9	23	151.8
Support M3x45 (4th to 5th layer)	6.6	1.25	8.25	6.5	42.9	23	151.8
Waffles plates (5th layer)	17.6	0	0	3.25	57.2	26.8	471.68
Payload:							
MG995 Servo 1	55	-2.2	-121	3.1	170.5	27.2	1496
MG995 Servo 2	55	2.2	121	3.1	170.5	27.2	1496
Mechanical Arm Structure	60	2.3	138	3.5	210	30	1800
Ping Pong ball holder (PVC pipe)	75	2.3	172.5	4.5	337.5	34	2550
Total	1190.7		315.5		196.72		14911.52
Centre of gravity			0		0.17		12.52

Table 2.3.1: Breakdown of calculation for centre of gravity

2.3.2 Electrical Subsystem

The electrical subsystem outlines the crucial hardware components utilised by the OpenBurger system, including a Functional Block Diagram, a Schematics Diagram, and Power Calculation. Accurate power calculation is crucial for managing the energy requirements of various components to prevent overheating and ensure component longevity. A steady power source is key to efficient energy use, which in turn reduces the frequency of battery replacements or recharging.

2.3.2.1 OpenBurger

For the OpenBurger, the following table shows the hardware components required with their specific uses.

Type of Component	Function
Microcontroller (Raspberry Pi 4 Model B)	Main brain for program flow of these component
	Main power for the Raspberry Pi Camera, 2x MG995 servo motor and a LED
Raspberry Pi Camera Module 2	Position as such it always faced the ground perpendicularly
	Live camera feed to the microcontroller to follow the line
MG995 Servo Motor	2 servo motors that act as a robotic arm that drop the ball into the bucket
LED Ultra Bright 5mm White	The LED provide the camera with sufficient light to obtain the data regarding the colour and contour of the tape

Table 2.3.2.1: Function of hardware components

Figure 2.3.2.1a displays the functional block diagram of the OpenBurger system. This diagram visually demonstrates the structure and interactions within the subsystem by segmenting it into discrete blocks, with each block representing a key function or component. The diagram indicates that all additional components are powered by the Raspberry Pi microcontroller, which, in turn, is driven by a Li-Po Battery. It also shows how energy flows from the Li-Po Battery to each component and how input data is transmitted throughout the system.

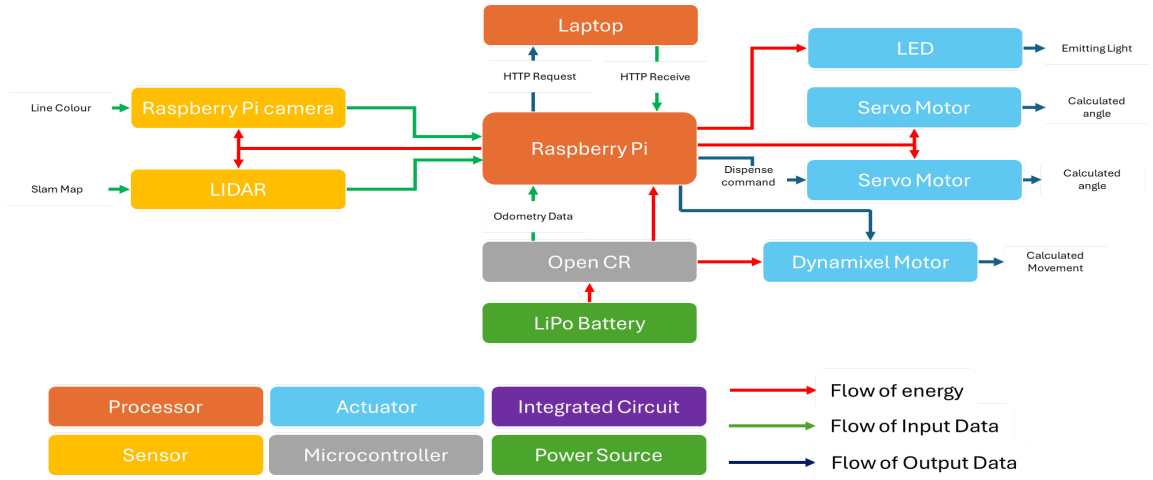


Figure 2.3.2.1a: functional block diagram of OpenBurger system

Figure 2.3.2.1b provides a schematic diagram of the OpenBurger, illustrating how the TurtleBot is connected to additional components used during the mission. This diagram emphasises the connections of each pin within the system, showcasing how the various components interface with one another.

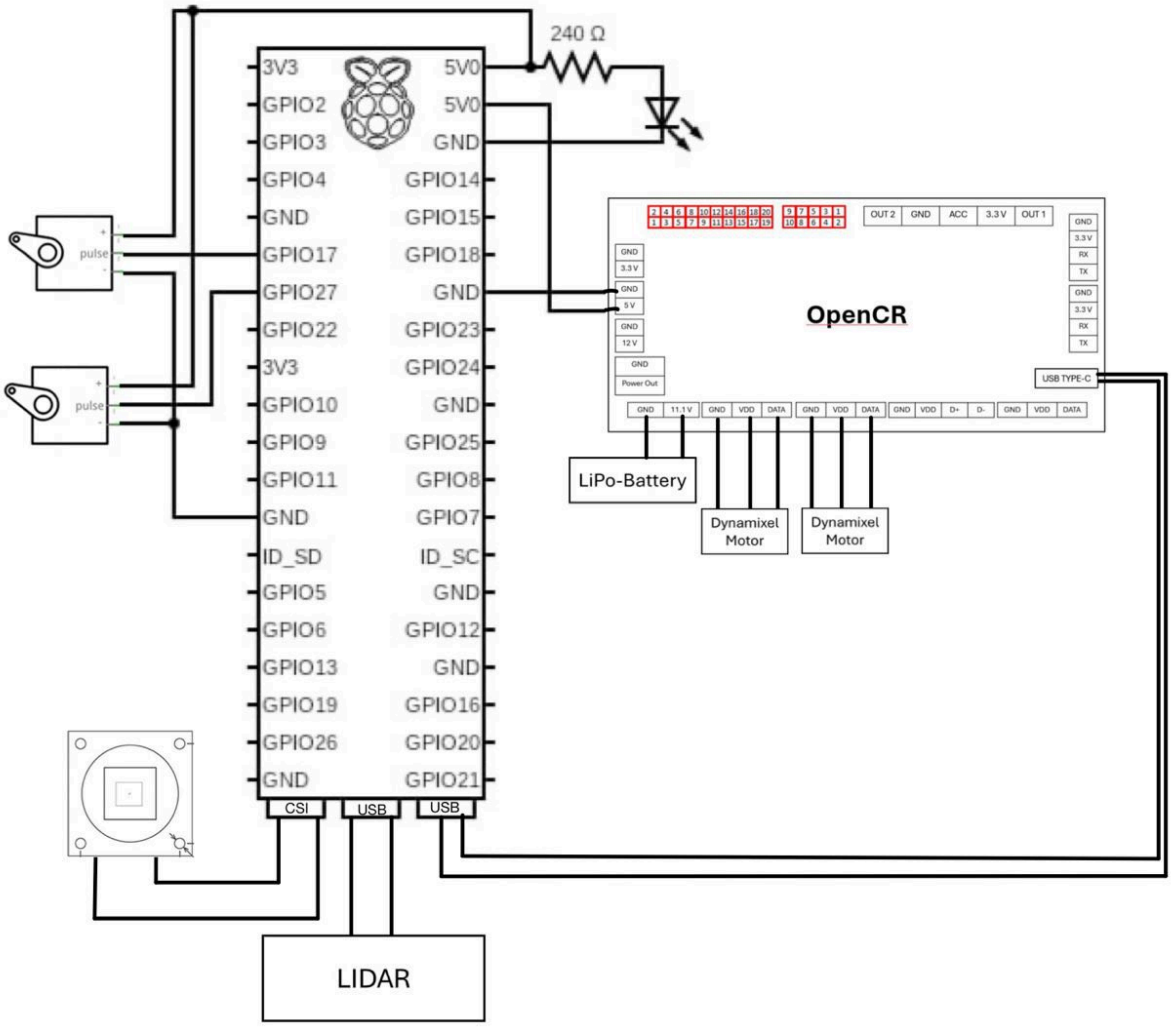


Figure 2.3.2.1b: Schematic diagram of OpenBurger system

In the schematic, MG 995 servo motors are wired to GPIO pins 17 and 27, and they also use the 5V power pins on the Raspberry Pi. LEDs are also connected to the Raspberry Pi's 5V pin, with a 240-ohm resistor in place to ensure that the current remains within the safe range of 10 to 30 mA, bringing it down to about 21 mA. The Raspberry Pi camera is linked to the Camera Serial Interface (CSI) port. The LIDAR sensor and OpenCR board connect to the Raspberry Pi through the USB ports. The Raspberry Pi gets its power from its 5V pin, which is supplied by a connection to the 5V socket on the OpenCR board.

The figure 2.3.2.1c present the power budget table that outlines the power requirement and distribution of the hardware component. The OpenBurger is powered by a LIPO 11.1V 1,800mAh LB-012 Battery. It is responsible for powering up the R-Pi and OpenCR in the TurtleBot, which then power up the other required components for the robot itself.

Component	Quantity	Voltage (V)	Current (A)	Power (W)	Runtime (s)	Energy Required (Wh)
MG 995 Servo motor (Before and after shooting)	2	5.0	0.333	3.330	1200	1.110
MG 995 Servo motor (During shooting)	2	5.0	0.651	6.510	60	0.109
TurtleBot Burger (Boot up)	1	11.1	0.784	8.702	180	0.435
TurtleBot Burger (Stand By)	1	11.1	0.590	6.549	300	0.546
TurtleBot Burger (During Operation)	1	11.1	0.848	9.413	1200	3.138
Raspberry Pi Camera	1	2.8	0.038	0.106	1200	0.035
LED Ultra Bright 5mm White	1	5.0	0.021	0.105	1200	0.035
Total	9			34.716		5.407

Figure 2.3.2.1c: Power budget table

The following are then the calculations performed to assess the ability of the battery's capacity at full charge to power the TurtleBot system throughout the entire mission.

$$Total\ Power\ Required \times Operation\ Time = Total\ Energy\ Consumed$$

$Energy\ Required$: $1.110Wh + 0.109Wh + 0.435Wh + 0.546Wh + 3.138Wh + 0.035Wh + 0.035Wh = 5.407Wh$

$Energy\ Provided\ by\ Battery$: $11.1V \times 1800mAh = 19980mWh = 19.98Wh$

$Considering\ Safety\ Factor\ of\ 60\%$: $5.77Wh \times 1.6 = 8.65Wh < 19.98Wh$

Even after accounting for a safety factor and assuming the turtlebot operates throughout the 20 minute mission , the energy demand of the TurtleBot during the mission is found to be lower than the battery capacity, which suggests that the battery chosen is adequate to support the TurtleBot's operation.

2.3.2.2 ESP32

For the ESP32, the following table shows the hardware components required with their specific uses.

Component	Function
ESP32	Main brain to communicate with OpenBurger
	Receive HTTP request from OpenBurger
	Open the door of the room and inform the OpenBurger which room is opened
USB TTL Adapter	To flash the ESP32 with the code from our Laptop

Table 2.3.2.2: Function of hardware components (2)

Figure 2.3.2.2a presents the functional block diagram of the ESP32 system. This visual depiction outlines the layout and interactions within the subsystem, with each block symbolising a core function or component. The diagram reveals that all supplementary components derive power from the ESP32 microcontroller, which is itself supplied by a Power Source. It further illustrates how power is distributed from the Power Source to each component, and how input data is transmitted across the system.

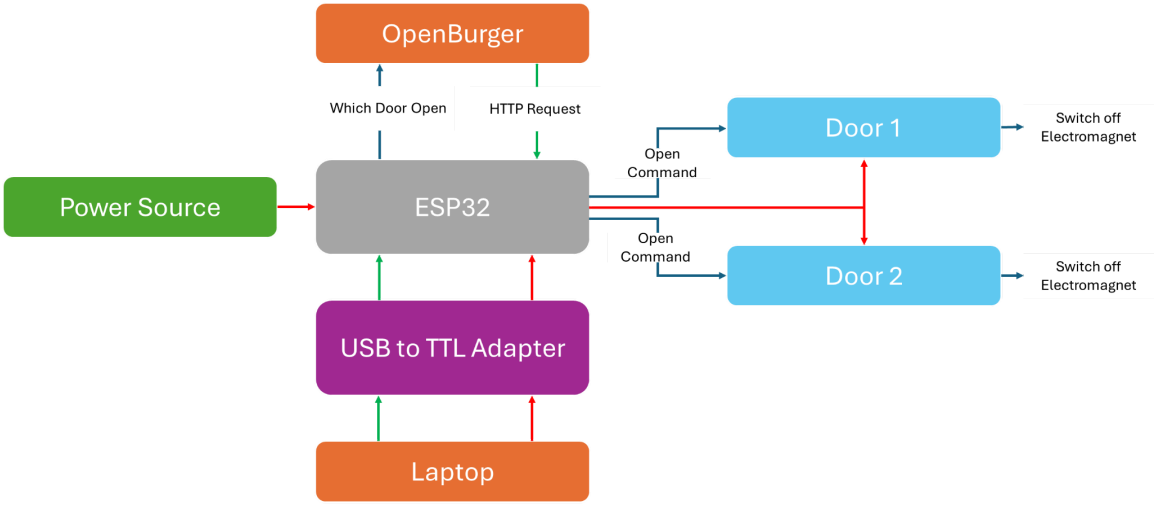


Figure 2.3.2.2a: Functional block diagram of ESP32 system

Figure 2.3.2.2b presents the actual diagram of how the ESP32 was connected to the USB to TTL adapter. This diagram emphasises the connections of each pin within the system, showcasing how the various components interface with one another.

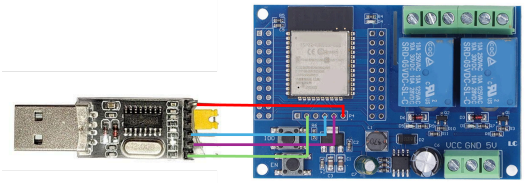


Figure 2.3.2.2b: Actual diagram of ESP32 system

2.3.3 Software Subsystem

We employed computer vision techniques, specifically leveraging the OpenCV library, to process video input from a webcam for coloured line-following navigation and colour detection tasks. Here's a concise explanation of the software we used to complete the mission.

Step 1: Colour Detection

As we are using multiple coloured tape, there is a need for the robot to differentiate between the colours used. This process is critical for the robot to accurately identify and follow the designated coloured paths. The colour detection logic is encapsulated within the `handle_Frame(inputImage)` function. The image below shows a before and after applying the mask.

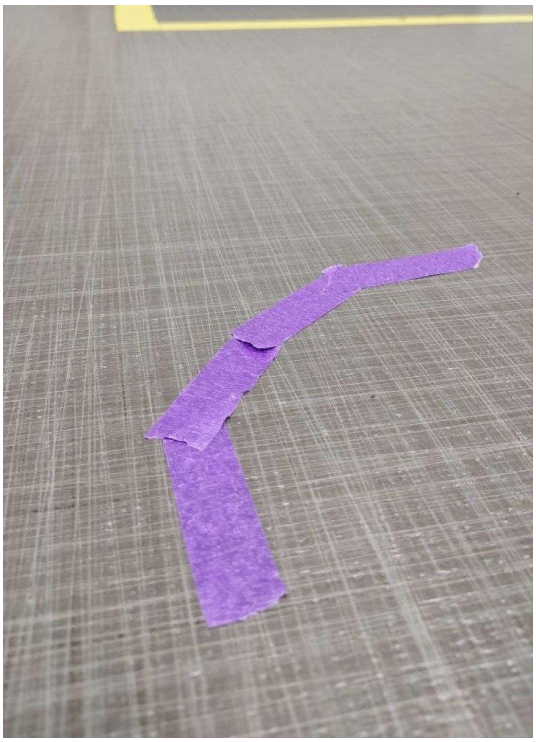


Figure 2.3.3a: Colour detection (before)

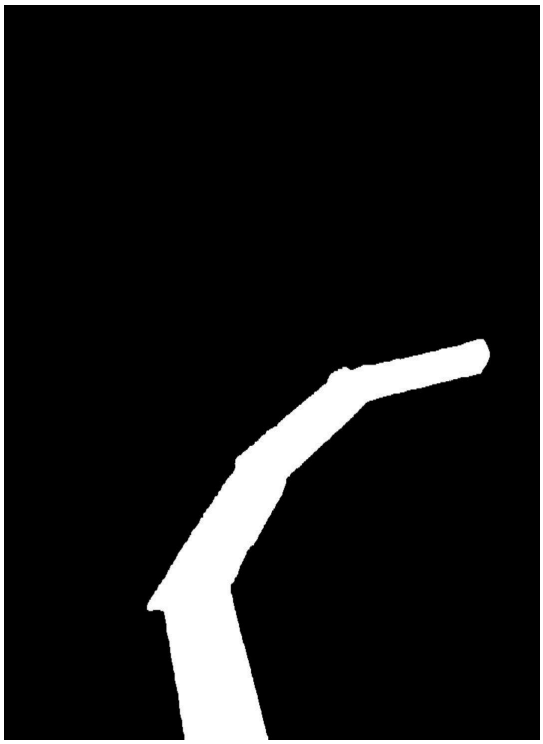


Figure 2.3.3b: Colour detection (after)

Initially, we convert the input image from the RGB colour space to the HSV (Hue, Saturation, Value) colour space. This conversion is strategic as HSV offers superior performance over RGB for colour differentiation tasks. In HSV, the Hue component captures the colour information, while the Saturation and Value components control the intensity and brightness, respectively.

Following the conversion to HSV, we create a mask to filter the input image based on predefined HSV values corresponding to the desired colour. The mask selectively retains pixels in the image that fall within the specified HSV colour range. This ensures that only elements of the desired colour are retained in the processed image, effectively 'hiding' items that do not match the desired colour criteria.

Additionally, we crop the input image to optimise the field of view. This cropping action serves to obscure the body of the robot from view, preventing the robot from looking too far ahead, and focusing the analysis on the immediate surroundings.

```
top_left = (0, int(height*0))  # Top-left corner
bottom_right = (width, int(height*0.85))  # Bottom-right corner

# Define the region to keep visible (inverse of hiding).
cv2.rectangle(mask, top_left, bottom_right, (255, 255, 255), -1)
croppedImage = cv2.bitwise_and(inputImage, mask)

# Convert to HSV colour space
blurredImage = cv2.GaussianBlur(croppedImage, (21, 21), 0.5)
hsvFrame = cv2.cvtColor(blurredImage, cv2.COLOR_BGR2HSV)

# Apply yellow colour mask for detecting yellow special marker
yellow_mask = cv2.inRange(hsvFrame,
np.array(color_dict_HSV['yellow'][1]),np.array(color_dict_HSV['yellow'][0])
)
yellow_mask = cv2.dilate(orange_mask, kernel)
```

Relevant code from handleFrame(input_image)

Step 2: Contour detection

To detect the lines, we've employed the findContours() function from OpenCV. This function not only allows us to draw on the original photo but also provides a range of contour values for analysis.

However, the contour values returned include not just the lines we're interested in, but also the contours of small objects in the background. To discern and extract the primary lines intended for tracking, we rely on the find_main_contour(image) function. This function iterates through all detected contours and selects the largest one as our main contour, which represents the primary line for tracking.

Subsequently, the identified main contour is visualised through the drawing of a prominent red line overlay, providing a clear representation of the line we intend to track. This overlay is shown during our debugging sessions using the laptop webcam.

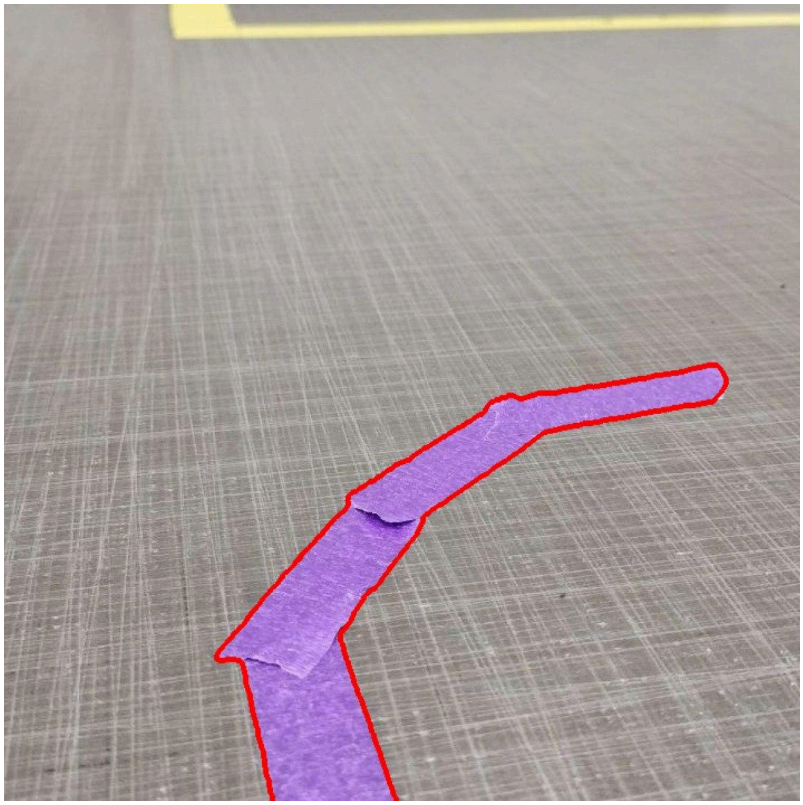


Figure 2.3.3c: After applying contour detection

```
def find_main_countour(image):

    cnts,_ = cv2.findContours(image,cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
    C = cnts
    #take the biggest contour with the biggest area
    if cnts is not None and len(cnts) > 0:
        C = max(cnts, key = cv2.contourArea)

    if len(C)<=0:
        return None, None
    #Return coordinate top left, top right, bottom right, bottom left
    if len(C)>0:
        rect = cv2.minAreaRect(C)
        box = cv2.boxPoints(rect)
        box = np.intp(box)
        box = geom.order_box(box)
        return C, box
```

The find_main_countour(image) function

Step 3: Rectangle and line fitting to main contour

With the main contour identified, our next objective is to determine the direction of the line for navigation. However, contours can exhibit varying shapes and orientations, making it challenging to accurately ascertain the overall direction.

To address this challenge, we employed a strategy leveraging on the information provided by the detected main contour. Initially, we encapsulate the main contour within a bounding box using the cv2.minAreaRect() function. This function returns the vertices of the minimum-area rectangle, which represents the bounding box around the contour. These vertices are then converted to integer values using np.intp.

Subsequently, the function geom.order_box() organises the vertices of the bounding box in a consistent order. Once the vertices are ordered, the calc_box_vector(box) function is invoked to compute the centre points (p1 and p2) of the longer sides of the quadrilateral represented by the bounding box. These are used to compute the green line which runs through the middle of the rectangle.

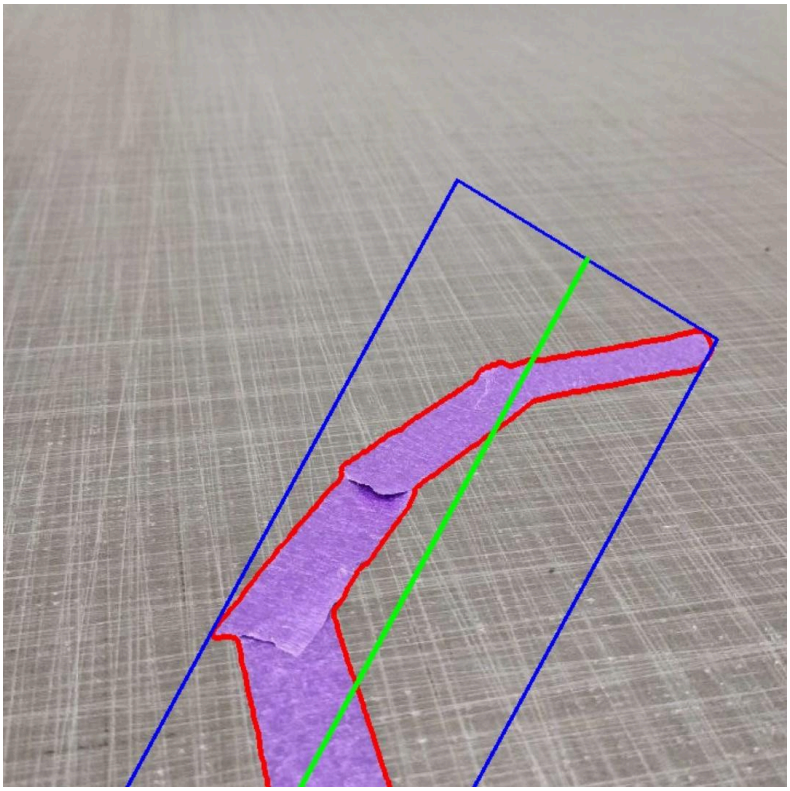


Figure 2.3.3d: After drawing blue box and green line

Step 4: Extracting the parameters of the characteristic rectangle and line
After getting our green line, we need to tell our robot mathematically how far the line is, and at what angle it is pointing at. To achieve this, two key values, "angle" and "shift," are computed to mathematically characterise the line's orientation and position relative to the robot's perspective.

Parameter	Description	Range	Interpretation
Angle	Represents the orientation of the line	0 - 180 degrees	0 to 90 degrees: Line points rightward 90 to 180 degrees: Line points leftward
Shift	Indicates the lateral displacement of the line from the image frame centre	-100 to 100	Zero shift: Line middle point aligns with the centre of the image frame. Negative shift: Line deviates to the right. Positive shift: Line deviates to the left.

Table 2.3.3: Parameters

The "angle" parameter is derived using the get_vert_angle() function within geom_util. However, it takes the vertical axis as 90 degrees and the maximum range of angle is only from 0 to 180 degrees. This configuration implies that if the line points towards the right side, the angle ranges from 0 to 90 degrees, whereas if it points leftward, the angle spans from 90 to 180 degrees.

Shift quantifies the lateral displacement of the middle point of the green line relative to the centre of the image frame. The middle point of the photo is used as the origin whereby it has a shift value of 0. If the middle point of the green line is located to the right of the photo, it will have a shift value of 0 to -100. If it is to the left of the photo, it will have a shift value of 0 to 100. Essentially, the value of shift represents a percentage width of the photo, whereas the absolute shift value of 100 represents a 50% width of the entire image frame.

To enhance consistency and facilitate comparative analysis, the ranges of both parameters are normalised to the range -100 to 100. This normalisation process ensures that both angle and shift are uniformly scaled for calculations in Step 5, where the direction of movement is determined based on these normalised parameters.

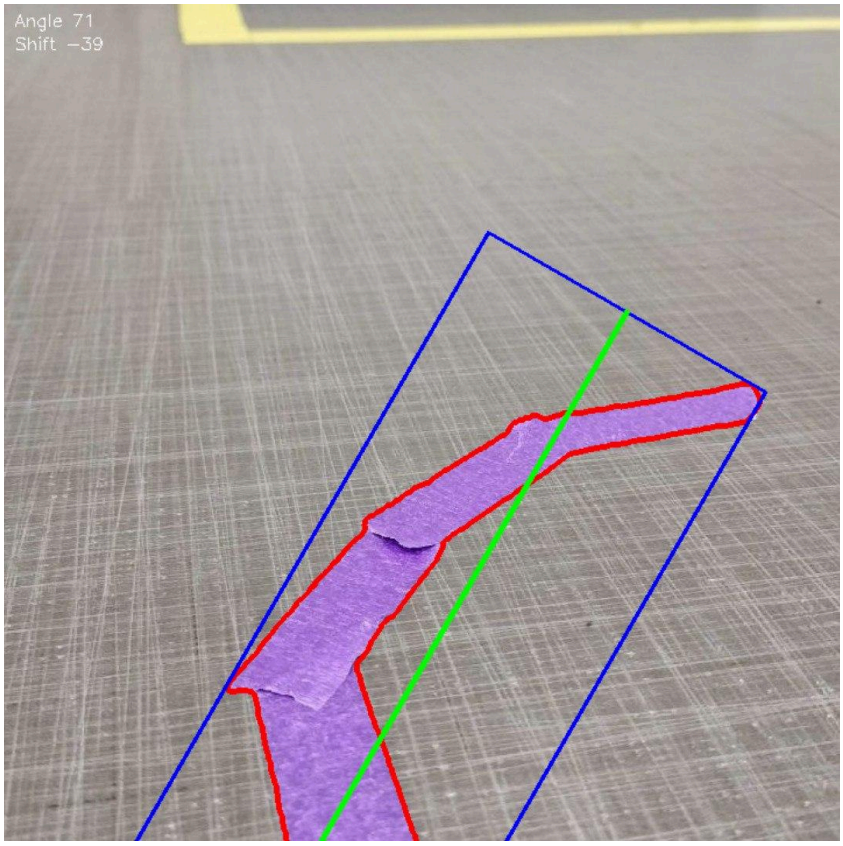


Figure 2.3.3e: Values of shift and angle seen on top left of photo

Step 5: Determining direction of movement

The direction of movement is determined based on the derived angle and shift parameters. While both parameters contribute to the calculation of driving angle, the angle holds a greater significance compared to shift. This prioritisation ensures that the robot aligns itself with the direction of the path (as indicated by the green line), even if it results in slight deviation from the centre of the line.

```
def get_turn(angle_state, shift_state):
    overall_state = angle_state + shift_state*0.33
    turn_dir = np.sign(overall_state) # -1 for left, 1 for right
    turn_val = abs(overall_state/100.0*turn_step)
    if abs(angle_state)>20: # turn more in sharp turns
        turn_val*=1.5
    elif abs(angle_state)<10: # turn less on straights
        turn_val*=0.6
    if turn_val>2.5: # Prevent angular velocity from exceeding limit
        turn_val=2.5
```

```
    return turn_dir, turn_val
```

The get_turn function code

Figure 2.3.3f: An illustration of output mask generated in the hypothetical scenario

To illustrate this decision-making process, consider a scenario where the shift of OpenBurger is highly negative (indicating the line is to the far left), while the angle is less than 90 degrees, suggesting a rightward path.

The shift parameter suggests the robot to move to the left to centre itself on the line, while the angle parameter suggests the robot to turn right as that is the direction of the path ahead. By taking both angle and shift to be of equal priority, the robot will cancel both signals out and move forward, or even move left due to the extreme shift value, eventually losing sight of the line and becoming lost.

Hence, we give precedence to the angle parameter, as it emphasises following the direction of the path ahead. After extensive experimentation and calibration, the ratio of angle to shift influencing the driving angle is set to 3:1. This calibration ensures a balanced approach to navigation, emphasising directional alignment while accounting for lateral displacement.

Another optimization we made was to conditional modify the magnitude of the driving direction based on the angle parameter. When the angle is small, the magnitude of turn is reduced to prevent overcorrection and keep the robot driving in a straight line. When the angle is large, the magnitude of turn is increased to maintain path adherence and prevent understeering.

Similarly, the linear velocity of the robot was maximised on straight segments, and reduced in sharp turns, reducing the navigation time while ensuring the reliability of line detection.

```
twist = geometry_msgs.msg.Twist()
if abs(turn_val)<0.15:
        twist.linear.x = 0.15
    elif abs(turn_val)<0.25:
        twist.linear.x = 0.06
    else:
        twist.linear.x = 0.04
```

Relevant code to conditionally modify the linear velocity

Step 6: Handling special markers.
To coordinate crucial actions such as launching HTTP requests and activating our launching mechanism, we integrated special yellow tape markers into our system. These markers serve as signals to prompt specific actions from the robot.

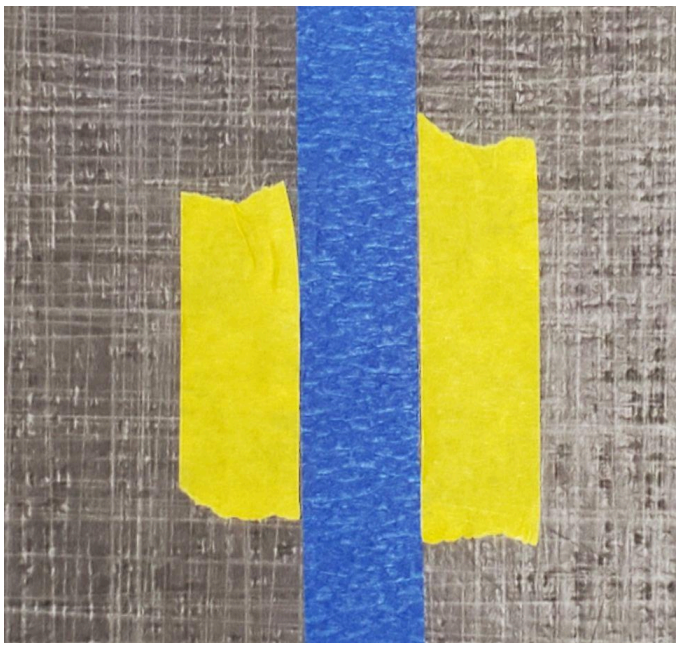


Figure 2.3.3g: The yellow markers positioned at the sides of the navigation line

When the presence of yellow tape is initially detected using the colour detection mechanism described in Step 1, the system triggers an HTTP request to open the door associated with the detected marker. Upon receiving a response confirming the execution of the request, a boolean variable httpSent is set to "true".

Subsequently, whenever the robot encounters yellow tape in future frames, it checks the status of the httpSent flag. If httpSent is true, indicating that the HTTP request has already been sent previously, the system activates the launching mechanism, stopping the OpenBurger and raising the mechanical arm.

Regardless of the current line colour being followed, our algorithm continuously scans each frame for the presence of yellow markers. This ensures that the system remains responsive to these special markers at all times, effectively acting as an interrupt mechanism. Upon detecting yellow tape, the robot temporarily suspends locomotion and prioritises the execution of the specified special action.

Step 7: Servo code

To manage servo control in our project, we opted for the pigpio library over the default RPI.GPIO library. This choice was made primarily to address performance issues observed with the servo when operating alongside ROS2, where the default library exhibited jittery behaviour.

The pigpio library offers a superior solution by leveraging hardware PWM timers, as opposed to software PWM timers utilised by the RPI.GPIO library. This hardware-based approach ensures more stable servo arm movements, especially in scenarios where the Raspberry Pi experiences high CPU loads. Unlike software PWM, hardware PWM operates independently of CPU activity, minimising the risk of delays and enhancing overall servo performance.

The initial pulse width of the servo was 1700, changing to 2100 when approaching the elevator and 950 when depositing the balls. To maintain smooth servo operation and slow down its movement such that the torque produced does not destabilise the OpenBurger, the pulse width is incrementally modified by 1 unit:

```
current = initial
increment = 1
angle_input = end

for i in range(int(abs(initial - angle_input))):
    if initial>angle_input:
        current -=  increment
    else:
        current += increment
    p1.set_servo_pulsewidth(servo_pin_1,current)
    p2.set_servo_pulsewidth(servo_pin_2,current)
    time.sleep(0.001)
```

Relevant code to control the servo motors through pigpio

The code sudo pigpiod initiates the pigpio library as a daemon process on the Raspberry Pi. This daemon process provides server services, enabling communication between a client application and the Raspberry Pi's GPIO pins. The communication occurs over local pipes or sockets.

By subscribing to this service, our system gains the capability to precisely control the frequency and PWM (Pulse Width Modulation) signal sent from the Raspberry Pi to a servo motor. This control allows us to manipulate various parameters such as the turning angle, speed, and overall motion of the servo motor.

Although the code snippet is not present in r2mover.py, it is executed during the boot-up process of the Raspberry Pi through a boot-up script. This ensures that the pigpio daemon is automatically launched and ready for use whenever OpenBurger is started up.

Step 8: HTTP request code

Our system utilises Python's requests library to facilitate communication between the Raspberry Pi, acting as a HTTP client, and the ESP32 server, functioning as a HTTP server controlling the elevator doors. The process involves sending POST requests from the Pi to the ESP32 server, located at http://<ESP_IP_ADDRESS>/openDoor. This request aims to open the elevator door and acquire the corresponding door ID. The JSON payload of the POST request is shown below:

```
{
  "action": "openDoor",
  "parameters": {
    "robotId": "TurtleBot3_ID"
  }
}
```

Upon receiving this request, the ESP32 server, configured to listen on port 80, verifies the integrity of the request payload. Subsequently, it employs a random number generator to select one of the two elevator doors, unlocking it via the appropriate relay mechanism. Following this action, the ESP32 sends the 200 response code back to the Pi, indicating the success status along with the ID of the opened door:

```
{
  "status": "success",
  "data": {
    "message": "door1" // OR "door2" randomly
  }
}
```

The handleRequest function within the ESP32 server operates as a blocking function, taking approximately 60 seconds to execute before locking the previously unlocked door. Within this time interval, the ESP32 does not respond to any new HTTP requests.

Hence, to ensure the robustness of the communication process, the Pi implements a recursive approach within the HTTP_requests function. In case of a failed request, this function automatically resends the POST request until a successful response is received. This iterative behaviour ensures that OpenBurger accurately navigates to the designated elevator and deposits the balls into the correct bucket. In the simplified version of the code below, the recursive code is highlighted in yellow.

```
def HTTP_requests():
    if response.status_code == 200:
        if response_data["status"] == "success":
            # handle door ID data
        else:
            message = HTTP_requests()
    else:
        message = HTTP_requests()
    return message
```

The HTTP_requests() function

Step 9: Navigating to the lift room

After obtaining the string which contains the opened door, the OpenBurger will lower down the servo arm and navigate to the opened door. The movement of the servo arm is to prevent it from hitting the upper part of the door while passing through:

```
            throw_my_big_balls(1700,2100)
            print('True, start HTTP')
            twist.linear.x = 0.0
            twist.angular.z = 0.0
            myMover.publisher_.publish(twist)
            time.sleep(0.3)
            Door = HTTP_requests()
            if Door == 'door1' or Door == 'door2':
                print(Door)
                httpSent = True
                color_change_counter = 0
                if Door == 'door1':
```

```
                color_num = 2
            else:
                color_num = 0
        break
```

Relevant code for sending HTTP request after the yellow marker

Upon encountering the yellow marker in front of the bucket, OpenBurger will stop moving and raise its servo arm to drop the ping pong balls into the bucket:

```
        print('shooting now')
        twist.linear.x=0.0
        twist.angular.z=0.0
        myMover.publisher_.publish(twist)
        throw_my_big_balls(2100,950) # Drop ping pong balls
        time.sleep(3)
        throw_my_big_balls(950,1250) # Move the servo back
```

Relevant code for shooting PingPong Balls

Step 10: Autonomous Navigation triggering

After the ball dropping process, the robot will then backtrack and rotate 180° in order to navigate back to the maze area. The yellow marker mentioned above will act as the marker to trigger the autonomous navigating function of the robot. For the code below, the yellow highlighted part will in-charge of the backtracking, and the blue highlighted part will in-charge of the rotating

```
twist.linear.x = -0.1
myMover.publisher_.publish(twist)
time.sleep(1)
twist.linear.x=0.0
twist.angular.z=0.5
myMover.publisher_.publish(twist)
time.sleep(7.5)
backtrack = True
color_change_counter = 0
break
```

Relevant code to control backtrack and rotate movement

The auto navigating function is then triggered after detecting the yellow marker, the code is shown here:

```
if backtrack and color_change_counter > 100 :
      twist.linear.x=0.0
      twist.angular.z=0.0
      myMover.publisher_.publish(twist)
      main1() —> The name of auto navigating function
```

Main part to trigger the autonomous navigation function

Step 11: Data Obtaining and Interpretation

In the auto navigating function, the main logic is to detect the longest distance in the 90° and navigate toward that direction. Thus to obtain the information of the lidar and the slammed RViz map, we will subscribe to three topics under ROS2 which are:

1. Scan topic which return the 360° distance information which obtained via the lidar

```
      self.scan_subscription = self.create_subscription(
                  LaserScan,
                  'scan',
                  self.scan_callback,
                  qos_profile_sensor_data)
```

Relevant code to subscribe on the scan topic

2. Occupancy topic which return the data of the real time slammed occupancy grid map

```
      self.occ_subscription = self.create_subscription(
                  OccupancyGrid,
                  'map',
                  self.occ_callback,
                  qos_profile_sensor_data)
```

Relevant code to subscribe on the Map topic

3. Odometry topic which return the current position of the OpenBurger

```
self.odom_subscription = self.create_subscription(
            Odometry,
            'odom',
            self.odom_callback,
            10)
```

Relevant code to subscribe on the Odometry topic

Then we will proceed to interpreting the data:

1. **Odometry:**

Then we will convert the quaternion data obtained via odometry topic into euler angles (roll, pitch, yaw) and save it into three variables which are roll, pitch and yaw respectively.

- roll is rotation around x in radians (counterclockwise)
- pitch is rotation around y in radians (counterclockwise)
- yaw is rotation around z in radians (counterclockwise)

```
def euler_from_quaternion(x, y, z, w):
    t0 = +2.0 * (w * x + y * z)
    t1 = +1.0 - 2.0 * (x * x + y * y)
    roll_x = math.atan2(t0, t1)
    t2 = +2.0 * (w * y - z * x)
    t2 = +1.0 if t2 > +1.0 else t2
    t2 = -1.0 if t2 < -1.0 else t2
    pitch_y = math.asin(t2)
    t3 = +2.0 * (w * z + x * y)
    t4 = +1.0 - 2.0 * (y * y + z * z)
    yaw_z = math.atan2(t3, t4)
    return roll_x, pitch_y, yaw_z # in radians
```

```
def odom_callback(self, msg):
       # self.get_logger().info('In odom_callback')
        orientation_quat =  msg.pose.pose.orientation
        self.roll, self.pitch, self.yaw =
euler_from_quaternion(orientation_quat.x, orientation_quat.y,
orientation_quat.z, orientation_quat.w)
```

Relevant code to interpret the odometry data

2. **Scan:**

The scan data from lidar will be saved into a file called lidar.txt and also a numpy array called laser_range. Then, the data with no value, which is shown to be NaN will be replaced using 0.

```
def scan_callback(self, msg):
        # self.get_logger().info('In scan_callback')
        # create numpy array
        self.laser_range = np.array(msg.ranges)
        # print to file
        np.savetxt(scanfile, self.laser_range)
        # replace 0's with nan
        self.laser_range[self.laser_range==0] = np.nan
```

Relevant code to interpret the lidar data

3. **Occupancy Grid Map:**

The occupancy grid map data obtained from the cartographer app will then be evaluated here, where we save it into a numpy array and plot the histogram of data. The data we obtained is the occupied point, non-occupied point and unknown point, which are interpreted as 1-100, 0 and -1 in the cartographer app. We added a counter to tackle the OpenBurger trap in a close loop, so when the number of occupied points is not changing for a certain period, the OpenBurger will detect the largest distance over its 360° to map out of that area and look for unknown areas.

```
def occ_callback(self, msg):
       # self.get_logger().info('In occ_callback')
```

```
            # create numpy array
            msgdata = np.array(msg.data)
            self.get_logger().info('map'%(msgdata))
            # compute histogram to identify percent of bins with -1
            self.occ_counts = np.histogram(msgdata,occ_bins)
            if self.occ_counts[0][0]<self.last_occ+200 and
    self.occ_counts[0][0]>self.last_occ-200:
                self.occ_counter +=1
            else:
                self.last_occ = self.occ_counts[0][0]
                self.occ_counter = 0
            # calculate total number of bins
            total_bins = msg.info.width * msg.info.height
            # log the info
            self.get_logger().info('Unmapped: %i Unoccupied: %i Occupied:
    %i Total: %i' % (self.occ_counts[0][0], self.occ_counts[0][1],
    self.occ_counts[0][2], total_bins))

            # make msgdata go from 0 instead of -1, reshape into 2D
            oc2 = msgdata + 1
            # reshape to 2D array using column order
            self.occdata =
    np.uint8(oc2.reshape(msg.info.height,msg.info.width,order='F'))
            self.occdata =
    np.uint8(oc2.reshape(msg.info.height,msg.info.width))
            # print to file
            np.savetxt(mapfile, self.occdata)
```

Relevant code to interpret the Occupancy grid map data

Step 12: Robot movement for autonomous navigation

Main code flow for navigating:

First, we will pick a desired angle for the turtlebot to turn using the pick_direction() function. In the function, we will check through the 90° in front of the OpenBurger in order to pick the longest distance to go.

```
    def pick_direction(self):
            global full_range_counter
            # self.get_logger().info('In pick_direction')
            if self.laser_range.size != 0:
                if full_range_counter<5:
                     # print(self.laser_range)
                     # use nanargmax as there are nan's in laser_range
    added to replace 0's
                     lr2i =
    np.nanargmax(self.laser_range[possible_angles])
                     self.get_logger().info('Picked direction: %d %f m'
    % (lr2i, self.laser_range[lr2i]))
                     full_range_counter +=1
                # elif self.occ_counter >=15:
                 #     lr2i =
    np.nanargmax(self.laser_range[small_angles])
                  #    self.get_logger().info('Picked direction: %d %f m'
    % (lr2i, self.laser_range[lr2i]))

                else:
                     lr2i = np.nanargmax(self.laser_range)
                     self.get_logger().info('Picked direction: %d %f m'
    % (lr2i, self.laser_range[lr2i]))
            else:
                lr2i = 0
                self.get_logger().info('No data!')
            # rotate to that direction
            self.rotatebot(float(lr2i))
            # start moving
            self.get_logger().info('Start moving')
            twist = Twist()
```

```
            twist.linear.x = speedchange1
            twist.angular.z = 0.0
            # not sure if this is really necessary, but things seem to
      work more
            # reliably with this
           # time.sleep(1)
            self.publisher_.publish(twist)
```

Relevant code for the OpenBurger to pick direction

A stopbot function is also used to stop the bot. We set the linear and angular speed to 0 and publish it to the OpenBurger for it to stop.

```
      def stopbot(self):
            self.get_logger().info('In stopbot')
            # publish to cmd_vel to move TurtleBot
            twist = Twist()
            twist.linear.x = 0.0
            twist.angular.z = 0.0
            # time.sleep(1)
            self.publisher_.publish(twist)
```

Relevant code to stop the robot

Using these two functions, we will then set the main movement of the OpenBurger. Via interpreting the yaw data in the imaginary number format, we would be able to avoid the problems when the value of angle goes from 360° to 0°. In the main code, while the OpenBurger turns, it will keep checking its odometry data until reaching the desired angle. After that, the OpenBurger will stop and go straight until reaching the next blocked area

```
      def rotatebot(self, rot_angle):
            # self.get_logger().info('In rotatebot')
            # create Twist object
            twist = Twist()

            # get current yaw angle
```

```
            current_yaw = self.yaw
            # log the info
            self.get_logger().info('Current: %f' %
    math.degrees(current_yaw))
            # we are going to use complex numbers to avoid problems when
    the angles go from
            # 360 to 0, or from -180 to 180
            c_yaw = complex(math.cos(current_yaw),math.sin(current_yaw))
            # calculate desired yaw
            target_yaw = current_yaw + math.radians(rot_angle)
            # convert to complex notation
            c_target_yaw =
    complex(math.cos(target_yaw),math.sin(target_yaw))
            self.get_logger().info('Desired: %f' %
    math.degrees(cmath.phase(c_target_yaw)))
            # divide the two complex numbers to get the change in
    direction
            c_change = c_target_yaw / c_yaw
            # get the sign of the imaginary component to figure out
    which way we have to turn
            c_change_dir = np.sign(c_change.imag)
            # set linear speed to zero so the TurtleBot rotates on the
    spot
            twist.linear.x = 0.0
            # set the direction to rotate
           # if random.randint(3,9)%2==0:
            if True:
                 twist.angular.z = c_change_dir * rotatechange1
            else:
                 twist.angular.z = -c_change_dir * rotatechange1
            # start rotation
            self.publisher_.publish(twist)
```

```
        # we will use the c_dir_diff variable to see if we can stop
rotating
        c_dir_diff = c_change_dir
        # self.get_logger().info('c_change_dir: %f c_dir_diff: %f' %
(c_change_dir, c_dir_diff))
        # if the rotation direction was 1.0, then we will want to
stop when the c_dir_diff
        # becomes -1.0, and vice versa
        while(c_change_dir * c_dir_diff > 0):
            # allow the callback functions to run
            rclpy.spin_once(self)
            current_yaw = self.yaw
            # convert the current yaw to complex form
            c_yaw =
complex(math.cos(current_yaw),math.sin(current_yaw))
            # self.get_logger().info('Current Yaw: %f' %
math.degrees(current_yaw))
            # get difference in angle between current and target
            c_change = c_target_yaw / c_yaw
            # get the sign to see if we can stop
            c_dir_diff = np.sign(c_change.imag)
            # self.get_logger().info('c_change_dir: %f c_dir_diff:
%f' % (c_change_dir, c_dir_diff))

        self.get_logger().info('End Yaw: %f' %
math.degrees(current_yaw))
        # set the rotation speed to 0
        twist.angular.z = 0.0
        # stop the rotation
        self.publisher_.publish(twist)
        time.sleep(1)
```

Relevant code to integrate the pick_direction() and stop() functions

Final main code

We will then integrate all the functions into a final main code, where the OpenBurger will start to go straight, until it reaches a place where the 30° angle area in front of it contains a distance lower than 0.36. Then, it will run the rotate() function in order to find the longest distance within the 90° area in front of it. After backtracking for 0.2 seconds, it will then turn to the desired angle and go straight again until the 30° in front detected a distance lower than 0.36 again.

```
def mover(self):
        try:
            # initialize variable to write elapsed time to file
            # contourCheck = 1

            # find direction with the largest distance from the Lidar,
            # rotate to that direction, and start moving
            time.sleep(5)
            self.pick_direction()

            while rclpy.ok():
                if self.laser_range.size != 0:
                    # check distances in front of TurtleBot and find values
less
                    # than stop_distance
                    lri =
(self.laser_range[front_angles]<float(stop_distance)).nonzero()
                    # self.get_logger().info('Distances: %s' % str(lri))

                    # if the list is not empty
                    if(len(lri[0])>0):
                        # stop moving
                        self.stopbot()
                        twist = Twist()
                        twist.linear.x = -speedchange1
                        twist.angular.z = 0.0
        # not sure if this is really necessary, but things seem to work more
        # reliably with this
       # time.sleep(1)
```

```
                        self.publisher_.publish(twist)
                        time.sleep(0.2)
                        self.stopbot()

                        # find direction with the largest distance from the
Lidar
                        # rotate to that direction
                        # start moving
                        self.pick_direction()

                # allow the callback functions to run
                rclpy.spin_once(self)

        except Exception as e:
            print(e)

        # Ctrl-c detected
        finally:
            # stop moving
            self.stopbot()


def main1(args=None):
   # rclpy.init(args=args)

    auto_nav = AutoNav()
    auto_nav.mover()

    # create matplotlib figure
    # plt.ion()
    # plt.show()

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

Relevant code which integrated all functions and determine the main flow

Additional: Mapping Software

Cartographer is a robust system designed to provide real-time Simultaneous Localization and Mapping (SLAM) capabilities in both 2D and 3D environments, across a variety of platforms and sensor configurations. SLAM allows OpenBurger to build a map of its surroundings while simultaneously determining its own position. We have opted to utilise Cartographer as our primary mapping software to generate maps for our robotic system.

To optimise system performance and resource utilisation, Cartographer is executed on a laptop rather than on the Raspberry Pi. This decision allows us to offload computational tasks from the Pi, thereby reducing CPU load and enabling the Pi to focus its limited resources on navigation-related tasks.

Visualising the SLAM mapping process in real-time is facilitated through RViz, a powerful 3D visualisation software commonly used in ROS (Robot Operating System) environments. In RViz, we can observe the progress of map generation as Cartographer processes sensor data and constructs the environment map.

In addition to the map itself, we augment the visualisation in RViz with pose information, providing a visual representation of the robot's current location and orientation within the environment. This pose information is invaluable for developing autonomous exploration algorithms and ensuring accurate navigation of the robot.

By leveraging Cartographer with RViz visualisation, we gain valuable insights into the mapping process and the robot's spatial awareness, empowering us to refine and optimise our robotic system for various applications.

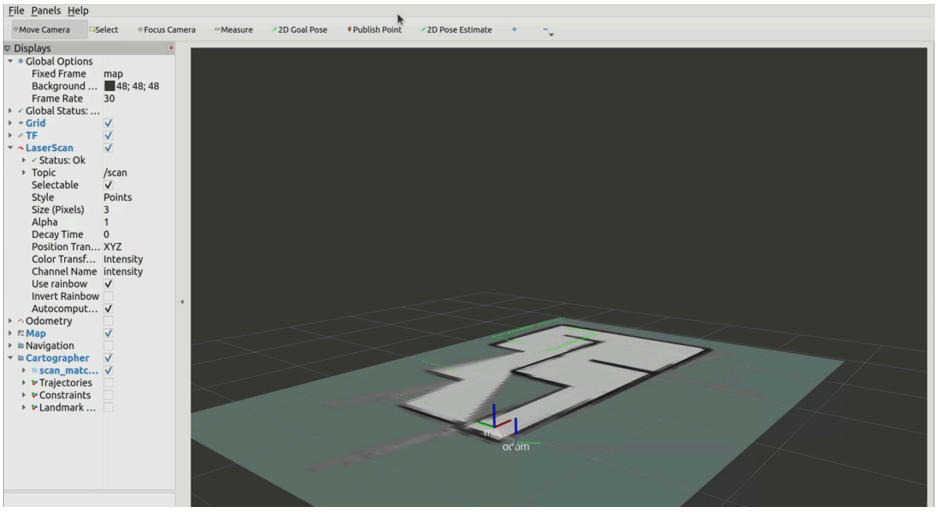


Figure 2.3.3h: A screenshot of RViz during the final run

Chapter 3 | System Fabrication Procedures

This section serves as a guide and offers a detailed explanation of the fabrication process for the OpenBurger from scratch. The fabrication procedures are divided into three distinct subsections, covering the mechanical, electrical, and software aspects of the system.

3.1 Mechanical Fabrication

Step 1: Building the original Turtlebot3 Burger

To initiate the project, we constructed the original Turtlebot3 Burger model by following the instructions outlined in the ROBOTIS Turtlebot3 Burger manual and utilising all the materials provided with the robot kit. The detailed assembly manual of the Turtlebot3 Burger can be found on the official website of ROBOTIS.

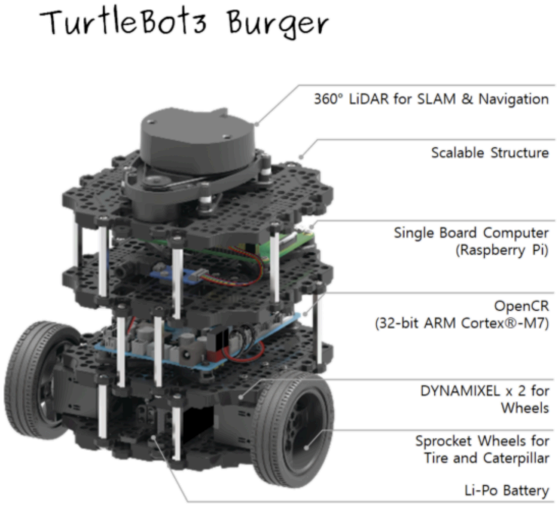


Figure 3.1a: Labelled Turtlebot3 Burger

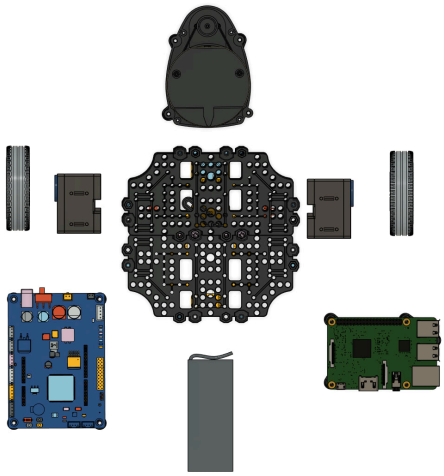


Figure 3.1b: Breakdown of Turtlebot3 Burger components

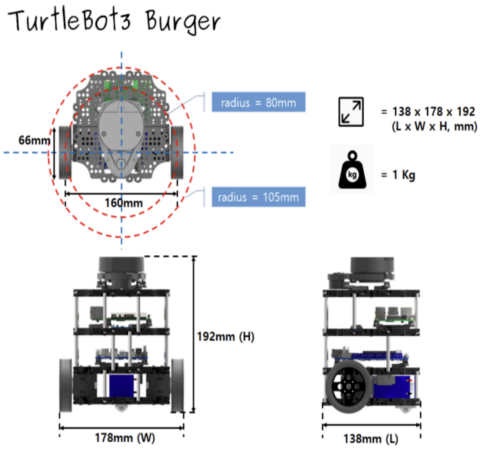


Figure 3.1c: Original Turtlebot3 Burger dimensions

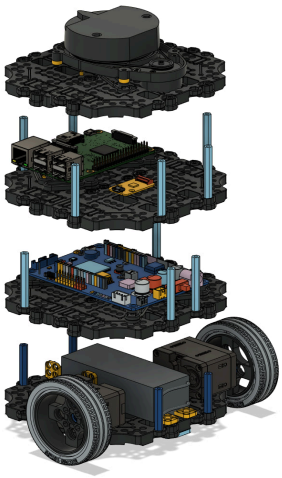


Figure 3.1d: 4 layers of Turtlebot3 Burger

Step 2: Adding an additional waffle plate layer above the Turtlebot3 Burger

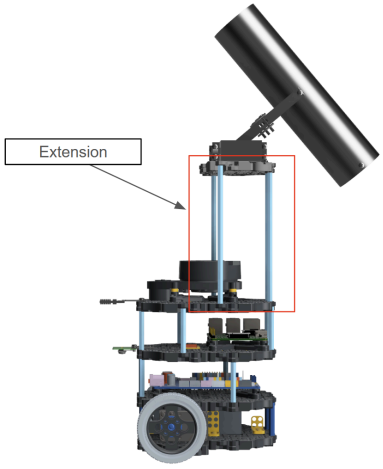


Figure 3.1e: Extension layer

Now that we have assembled the original Turtlebot3 Burger model, we will extend the height of the robot to accommodate the attachment of our payload system. This extension ensures that the robot has sufficient height to effectively deliver the ping pong balls into the bucket.

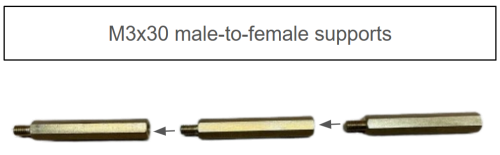


Figure 3.1f: Extension supports

To construct the extension, we will need 12 M3x30 male-to-female supports, as depicted in figure 3.1f. Combine three of these supports to form a set, resulting in a total of four sets of supports.

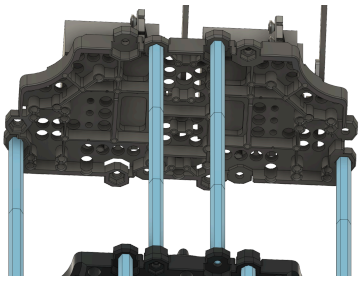


Figure 3.1g: Supports position

Next, affix the four sets of supports onto the uppermost layer waffle plate, arranging them as illustrated in figure 3.1g. Utilise M3 screws and nuts to secure the joints.

Step 3: Assembling the payload system

Figure 3.1h: Overall payload system assembly

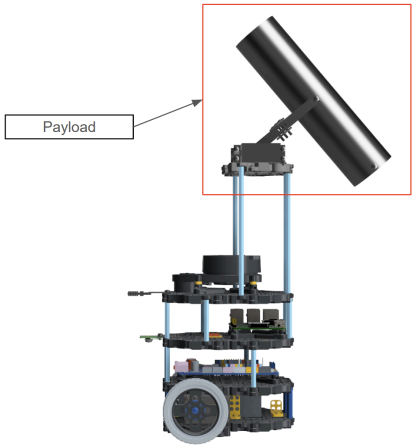
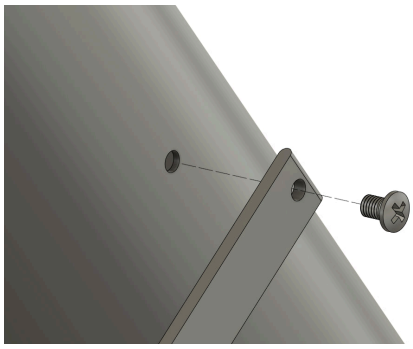
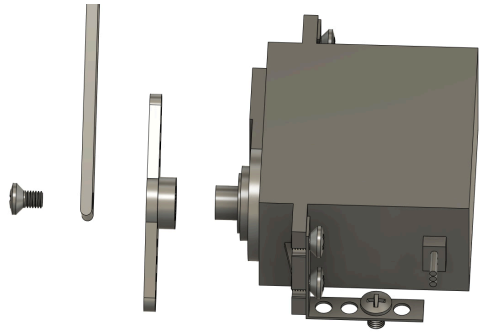
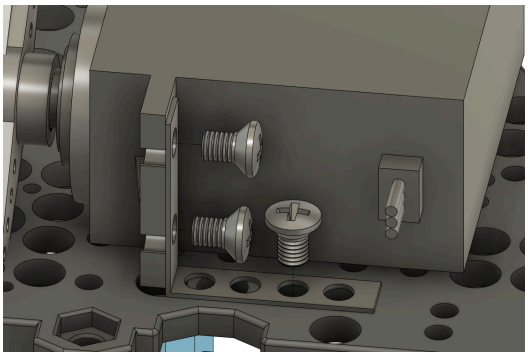


Figure 3.1i: Payload system

We commenced the customization of the Turtlebot3 Burger by constructing our intended payload system, enabling the OpenBurger to deliver 5 ping pong balls into a designated bucket. Figure 3.1h illustrates the CAD drawing of the payload system assembly. The essential components required for fabrication include two MG995 servo motors, four 3D printed mechanical arms, and a single 50 mm diameter PVC pipe.

Figure 3.1j: Sealing PVC pipe

Drill two 3mm holes at the end of the PVC pipe and secure the M3 x 50 support using screws to seal the end of the pipe, preventing the ping pong balls from falling out.

 ***Figure 3.1k: Mechanical Arm***	3D print four mechanical arms measuring 98mm x 10mm x 2mm, each featuring M3 holes at both ends. Assemble them together using M3 screws and nuts, following the design as shown in figure 3.1h.
 ***Figure 3.1l: Connecting the mechanical arm to the PVC pipe***	Fasten the 3D printed mechanical arms to the central portion of the PVC pipe using M3 screws and nuts. Optionally, employ additional fasteners to ensure a more secure connection at the joint.
 ***Figure 3.1m: Servo motors joint***	Assemble the servo motor, servo horn, and mechanical arm as depicted in figure 3.1m, securing the joint with M3 screws.
 ***Figure 3.1n: Fixing servo motors on waffle plate***	Finally, attach both MG995 servo motors onto the uppermost layer waffle plate, aligning them as indicated in figure 3.1i. Secure the joints between the servo motor and the waffle plate using L-brackets made from 3mm round hole aluminium plate, as illustrated in figure 3.1n.

Step 4: Attaching the Raspberry Pi Camera and LED light

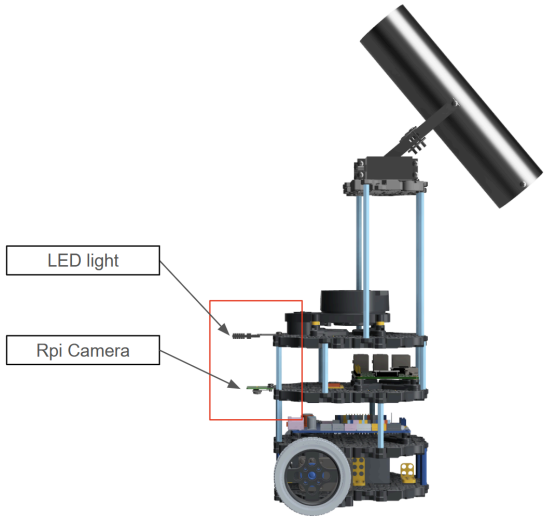


Figure 3.1o: Rpi camera and LED light

The final step involves mounting the Raspberry Pi camera onto the 3rd layer and the LED light onto the 4th layer of the OpenBurger. These additions equip the robot with the capability to utilise OpenCV for line following navigation.

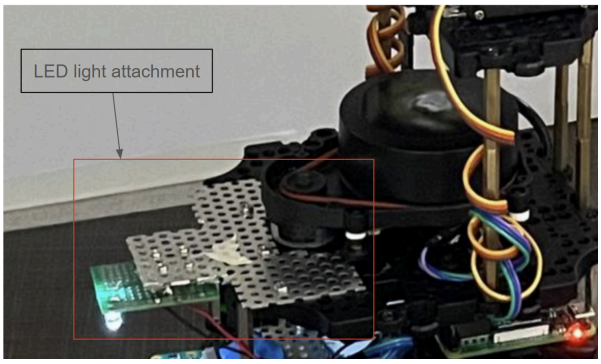


Figure 3.1p: LED light attachment

Cut a piece of 3mm round hole aluminium plate to match the shape depicted in figure 3.1p. Utilise M3 screws and nuts to fasten it securely onto the 4th waffle plate layer. Mount the veroboard and LED circuits onto the aluminium plate. (refer to section 3.2.1 for LED circuit assembly)

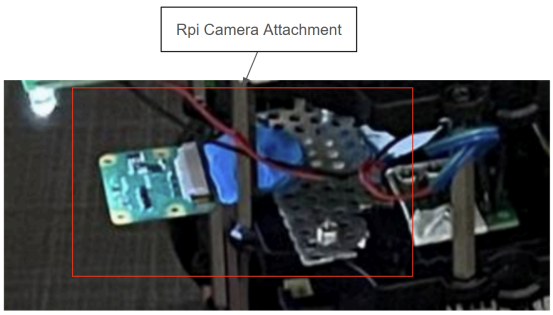


Figure 3.1q: Rpi camera attachment

Repeat the same process for mounting the Raspberry Pi camera onto the 3rd waffle plate layer. Handle it with care to avoid damaging the delicate Raspberry Pi camera belt.

3.2 Electrical Assembly

3.2.1 OpenBurger

Component required:

1. TurtleBot
2. Raspberry Pi Camera (Rpi Camera)
3. MG995 Servo Motor x2 (Servo Motor)
4. 5mm Ultra Bright White LED (LED)
5. Vero board
6. Single Core wire
7. Breadboard

Step 1: testing on breadboard:

1. Set up the system on a breadboard for testing of functionality of each output components as shown in diagram below:

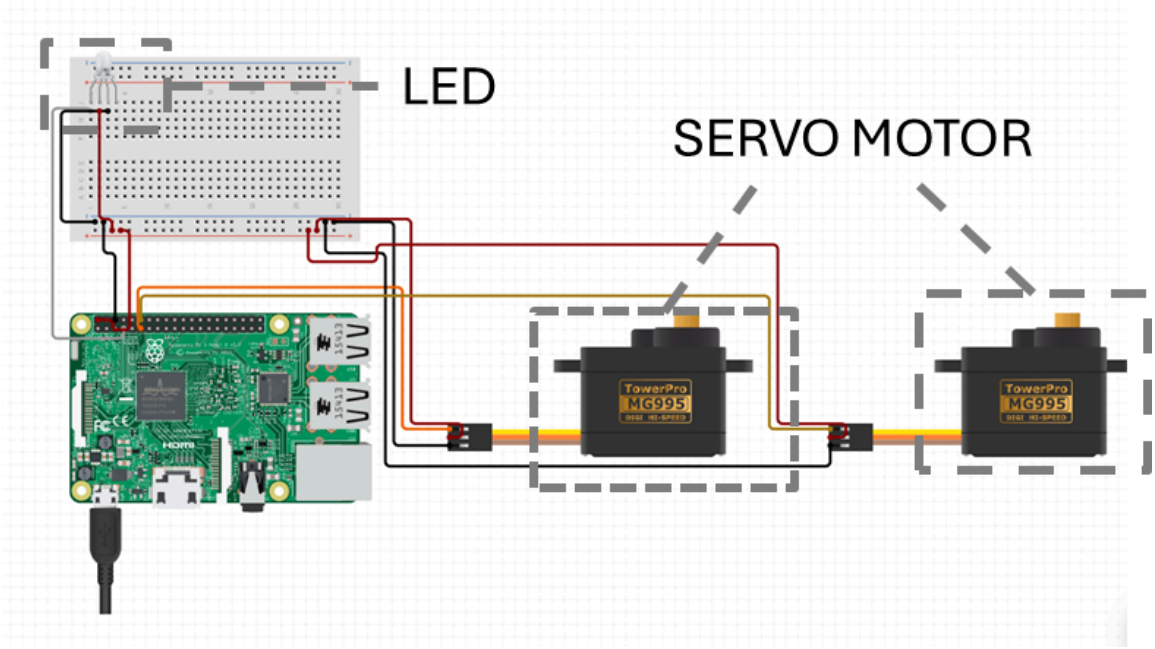


Figure 3.2.1a: Connections of components for testing

Figure 3.2.1a shows how each component is connected to the correct pin for testing of functionality. The input pins of both Servo Motors are connected to the GPIO Pin 17 and 27.

2. Set up the system on a breadboard for testing of functionality of Rpi Camera as shown in diagram below:

Figure 3.2.1b: Rpi camera CSI cable connection

Figure 3.2.1b show how the Rpi camera CSI cable are connected to the Raspberry Pi

Step 2: Solder the required component onto vero board:

1. Solder the common ground and 5V pin of each Servo motors with single core wire onto a vero board
2. Solder the LED and its resistor onto another vero board to attach it on the bracket of the OpenBurger.
3. Solder the 5V wire and GND pin wire of the LED vero board in step 2 onto the vero board that connects the common ground and 5V pin in parallel in step 1.

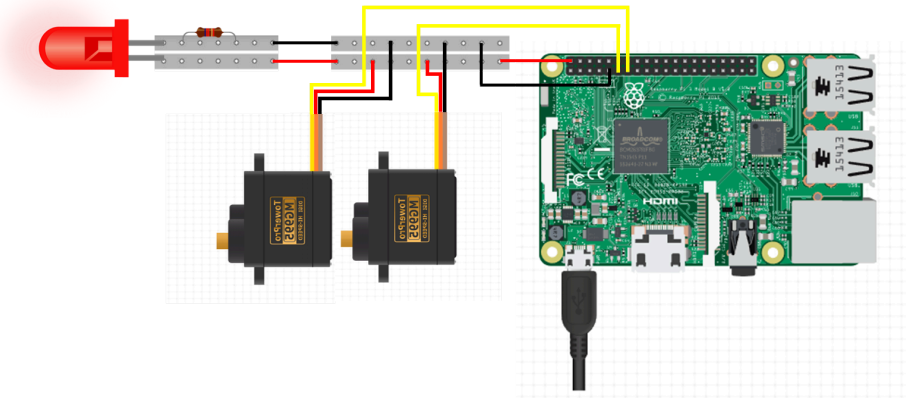


Figure 3.2.1c: Soldering of components on veroboard

Figure 3.2.1c shows how the components are soldered onto the vero board and connected to the Raspberry Pi. The resistor is added to limit the current flow to the LED to prevent the LED from overheating and burning out. The input pins of both Servo motors are connected to the GPIO Pin 17 and 27.

Step 3: Final Testing:

1. After the components are attached to the Turtlebot, the functionality of each component is tested again to ensure that it works well.

3.2.2 ESP32

Component Required:

1. ESP32
2. USB to TTL Adapter (adapter)
3. Female to female wire
4. Pin Header

Step 1: Solder the pin header onto the ESP32:

1. The ESP32 comes without pin headers to connect with the adapter.
2. The pin from GND to 5V including the RX and TX pin are required to solder with pin header to connect with the adapter by using female to female wire.

Figure 3.2.1d shows the pin that are required to be soldered by the pin headers:

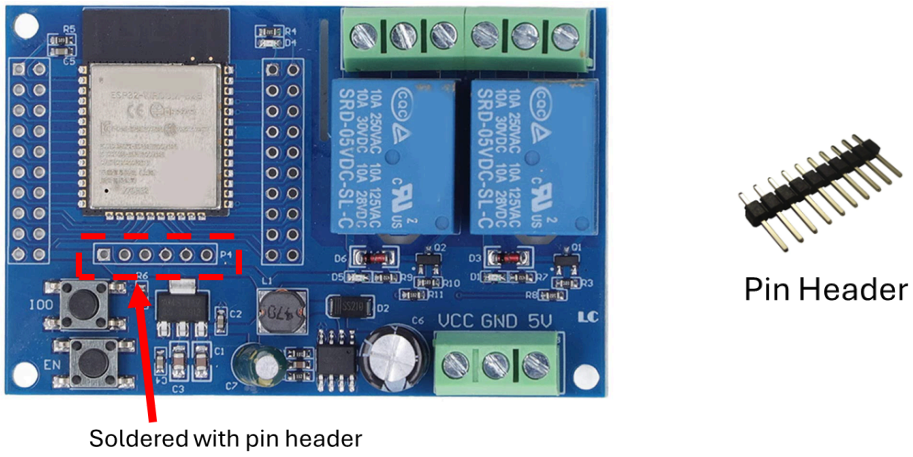


Figure 3.2.1d: Soldering of pins

Step 2: Connect the ESP32 to the adapter

1. The 5V and GND of the ESP are connected to the adapter
2. The TX of the ESP32 is connected to the RX of the adapter
3. The RX of the ESP32 is connected to the TX of the adapter

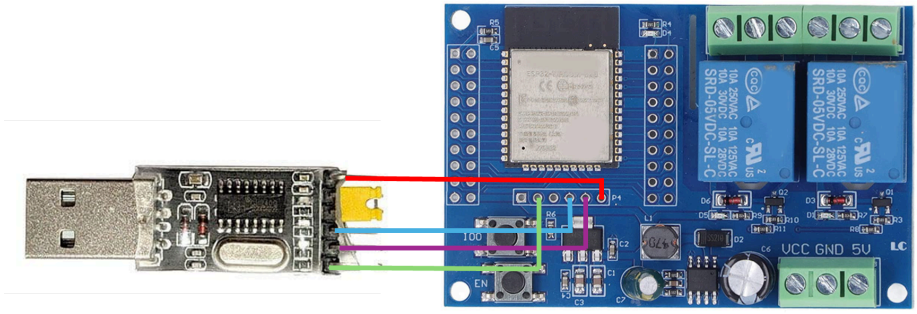


Figure 3.2.1e: Connection of ESP32 to the adapter

Step 3: Setting up the ESP32

1. Follow the github link to set up the ESP32 (https://github.com/edic-nus/EG2310/blob/main/src/main.ino)

3.3 Software Setup

Installation Requirements:

- Ubuntu 20.04
- ROS2 Foxy
- Python 3.8
- Robotis Co. TurtleBot3 Burger with Raspberry Pi 4 Model B Single Board Computer

Note: OpenBurger has only been tested and verified on this specific configuration. Modifications may be required for different operating systems, software, or hardware.

3.3.1 Setup Instructions

1. Installation:
    - Ensure that your system meets the installation requirements mentioned above.
    - Follow the instructions under the "Foxy" tab in the ROS2 documentation here to set it up on the Remote PC.
2. Testing ROS2 Environment:
    - Test the ROS2 environment on the PC by creating a publisher and subscriber as instructed here.
3. Setting up the TurtleBot3:
    - Write the ROS 2 Foxy Image onto the SD card inserted into the Raspberry Pi of the TurtleBot3. Follow the "Quick Start Guide" to set up a fully operational TurtleBot3.
    - Install OpenCR firmware to the connected OpenCR robot controller as instructed here.
4. Configuration for PC:
    - Add the following lines to the .bashrc file in the root directory of the Remote PC. These lines define the TurtleBot model and register aliases that simplify the operation of teleoperation and SLAM nodes.

      ```
      export TURTLEBOT3_MODEL=burger
      alias rteleop='ros2 run turtlebot3_teleop teleop_keyboard'
      alias rslam='ros2 launch turtlebot3_cartographer
      cartographer.launch.py’
      ```

5. Configuration for Raspberry Pi:
    - Add the following lines to the .bashrc file in the root directory of the Raspberry Pi, to define the TurtleBot model and register the TurtleBot bringup launch script:

```
export TURTLEBOT3_MODEL=burger
alias rosbu='ros2 launch turtlebot3_bringup robot.launch.py'
```

6. ROS 2 Package Creation:
    - Create a ROS 2 package on the remote laptop:

```
cd ~/colcon_ws/src
ros2 pkg create --build-type ament_python auto_nav
cd auto_nav/auto_nav
mv init.py ..
```

7. Clone GitHub Repository:
    - Clone the GitHub repository to the desired directory:

```
cd desired/directory/path
git clone https://github.com/LingZiyann/r2auto_nav
```

8. Install Required Libraries:
    - Install OpenCV and NumPy libraries:

```
pip install opencv-python numpy
```

9. Build the Package:

```
cd ~/colcon_ws
colcon build
. install/setup.bash
```

3.3.2 Operation Parameters

After careful calibration with OpenBurger, we obtained a set of constants that worked best to complete the mission. The OpenBurger was stable during line following, the autonomous exploration was able to work (but not in the final run) and the servo motor held the mechanical arm in the correct position. Hence, we recommend using these constants and modifying them with caution.

```
# Line Following, Color, and Speed Constants
rotatechange = 1.5 # Rotation speed for line following
speedchange = 0.2 # Linear speed for line following
scanfile = 'lidar.txt' # File to store LiDAR data
mapfile = 'map.txt' # File to store map data
turn_angle = 3 # Angle threshold for turning
shift_max = 10 # Horizontal shift threshold for turning
turn_step = 3.0 # Duration of turning
colour_area = 10 # Minimum area threshold for colour detection

# Autonomous Exploration Constants
occ_bins = [-1, 0, 100, 101] # Bins for occupancy grid histogram
stop_distance = 0.36 # Distance to stop from obstacles
front_angle = 15 # Angle range for front obstacle detection
front_angles = range(-front_angle, front_angle+1, 1) # Range of front
angles for obstacle detection
possible_angles = range(-115, 115, 1) # Range of possible angles for
direction selection
small_angles = range(-119, 119, 1) # Smaller range of angles for
direction selection

# Servo Motor Constants
servo_pin_1 = 17 # GPIO pin for servo motor 1
servo_pin_2 = 27 # GPIO pin for servo motor 2
freq = 50 # Frequency for servo motor control
initial = 1700 # Initial servo position
end = (950, 2100) # End servo positions for shooting balls
```

Chapter 4 | Operation Plan

4.1 Mission Flow Plan

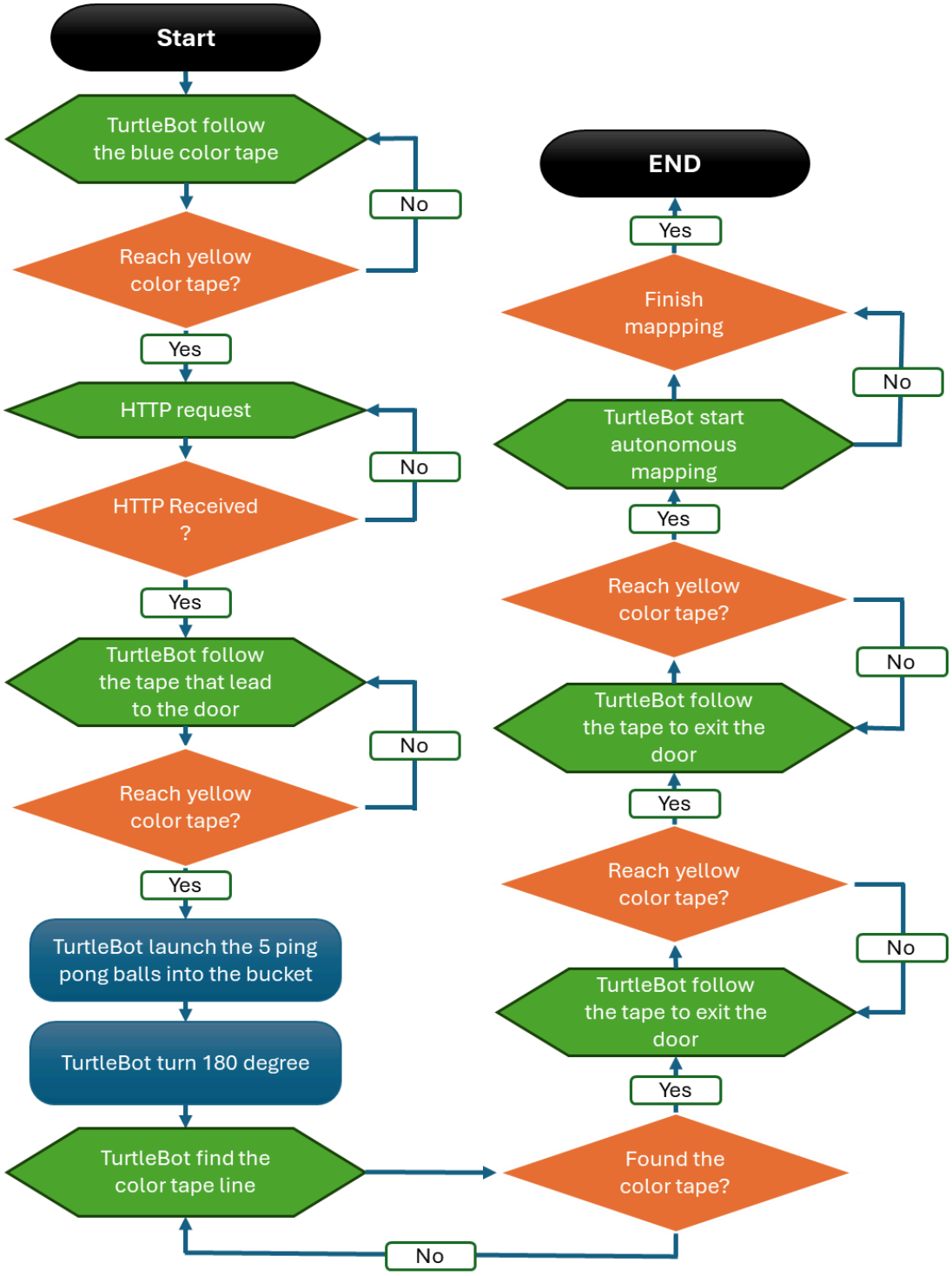


Figure 4.1: Mission flow plan

4.1.1 Planned Timing

Time (minutes)	Planned activity
0 - 5	Preparatory work: - Explain the user manual - Flash the ESP32 and update OpenBurger with its IP address - Stick the tape on the floor for line following
5 - 9	Start the first run attempt, using a slower speed on the robot to ensure the mission success of depositing the ball successfully.
9 - 14	If the mission has failed on the first attempt, prepare for a second attempt. If the mission was successful, let our robot continue running to try and map out the entire maze.
14 - 18	If the second attempt in depositing the ball fails, surrender and start the mission from the front of the door instead. If the mission and mapping are successful and time allows, try a second run with a faster-calibrated speed on the robot to achieve better timing.
After 18	Remove all tapes on the floor and prepare to leave.

Table 4.1.1: Planned timings

4.1.2 Distribution of tasks during the run

Team Member	Task
Teh Wei Sheng	Explain the user manual to the judging panel
Tong Jing Yen	- Flash the ESP32 with Wi-Fi credentials and update OpenBurger with its IP address - Setup the scripts for OpenBurger operation
Chang Shu Kai	Lay out tape from initial OpenBurger position to first half of maze
Soh Sze Juin	Lay out tape for the second half of maze
Ling Zi Yan	Lay out tape for the elevator junction until the location of both buckets

Table 4.1.2: Distribution of tasks

The preparatory tasks are delegated such that they are done in parallel, allowing for the run to be started as soon as the explanation of the user manual is complete. In addition, all members of the team will be responsible for removing the tape and OpenBurger after the run.

4.2 Pre-Ops Check

OpenBurger needs to pass the pre-operations check, before the run is allowed to proceed. A variant of the table below is included in the user manual to systematically verify that each component is working as intended.

Component	Intended condition	Pre-operations Check	Intended Observation
OpenCR	Able to be powered by the LiPo Battery	Connect the LiPo battery to the OpenCR	Green LED lights up and boot-up tune is played
Raspberry Pi	Able to turn on when connected to OpenCR	Connect the Pi to the OpenCR	Red light turns on while green light flashes
	Connect to network and be accessible from remote laptop	Run ‘sshrp2’ from a laptop connected to the same Wi-Fi network	Terminal hostname change to “ubuntu” when “sshrp2” is run on laptop
Lidar	Able to spin and collect data	Run ‘rosbu’ on the Pi, and ‘rslam’ on the laptop	Mapping with SLAM works and it is displayed in RViz
MG995 servo motor	Daemon subscription established and the servo is able to rotate the mechanical arm for ball dropping	Run ‘servo.py’ testing script on the Pi	Servo motor rotates and all ping-pong balls drop out of the PVC pipe
RPi Camera 2	Detected by the Pi, and be able to capture video and do line following properly	Run ‘ctry.py’ testing script on the Pi	The script runs without errors and outputs an image file ‘ctry.jpg’
Wheels	Able to rotate clockwise and anticlockwise freely	Run ‘rosbu’ on the Pi, and ‘rteleop’ on the laptop	Motors moves in the right direction and can be controlled properly
Veroboard	Able to attach the wires firmly under all circumstances	Examine the wire connections and soldering	Wires are not detached and soldering is clean
Ball caster	Able to roll in all directions freely	Spin the ball continuously	It is not stuck and able to move around in all directions smoothly
LED light	Able to illuminate the ground for the line detection with Camera under shadowed areas	-	White colour light is emitted
Structural stability	Structural platforms and components installed correctly	Shake OpenBurger	All components are mounted securely and no loose parts are heard
	All joints are secured	Examine the screws and nuts	All screws and nuts are installed and tightened

Table 4.2: Pre-Ops Check

4.3 Setting up the Code

A total of 3 terminals are required to operate OpenBurger, consisting of 2 terminals remotely connected to the Raspberry Pi and 1 terminal on the Laptop. Ensure that the OpenBurger, ESP32 and Laptop are connected to the same Wi-Fi connection. It is necessary to flash the ESP32 with the Wi-Fi SSID and password for this purpose.

For some commands i.e. connecting to the Raspberry Pi through SSH and initiating SLAM, we utilise aliases defined in our .bashrc file, to quicken our operation and minimise typing errors.

On the laptop:

```
alias sshrp2='ssh ubuntu@`ssh aws cat rpi2.txt`'
alias rslam='ros2 launch turtlebot3_cartographer
cartographer.launch.py'
```

On the Raspberry Pi:

```
alias rosbu='ros2 launch turtlebot3_bringup robot.launch.py'
```

In Terminal 1: Bring up TurtleBot3

```
sshrp2
rosbu
```

The code will attempt to bring up the various components of OpenBurger e.g. the LIDAR sensor and the Dynamixel motors, by executing the robot.launch.py file from the turtlebot3_bringup package. The terminal will notify us of the overall status, and we can move on when it is ready.

In Terminal 2: Initiate SLAM on laptop, and run RViz visualisation

```
rslam
```

The SLAM (Simultaneous Localization and Mapping) algorithm is executed by calling cartographer.launch.py launch file from the turtlebot3_cartographer package. RViz is launched to display the SLAM map generated on-the-fly. An initial incomplete map will be displayed.

Launch screen recording to document the SLAM map generation by pressing Ctrl+Alt+Shift+R on the laptop.

In Terminal 3: Configure IP address of ESP32, and run navigation code

```
sshrp2
cd ~/colcon_ws
vim src/r2auto_nav/r2auto_nav/r2mover.py
colcon build --packages-select r2auto_nav
source install/local_setup.bash
ros2 run r2auto_nav r2mover
```

Update the r2mover.py code in the r2auto_nav ROS2 package to include the IP address of the ESP32. Then, rebuild the package and run the package using ROS2. The terminal will start displaying debug data, and OpenBurger will begin to navigate.

Chapter 5 | Bill of Materials

S/N	Item	Quantity	Price
1	3D printing parts	4	$20
2	Mini Brush 6V DC Motor 140 (Not used in final run)	1	$9.36
3	AS568 O Ring Seal Gaskets Part 2, Nitrile NBR Rubber, 029 x 10 pieces (Not used in final run)	1	$12.80
4	PVC Pipe White Color Aquarium Hard Pipe Drinking Water Outer Diameter 50mm	1	$12.68
5	6 pcs Coloured Masking Tape	1	$16.99
6	RPI Camera	2	Borrowed from lab
7	MG 995 Servo Motor	2	Borrowed from lab
8	Waffle-Plate	2	Borrowed from lab
9	5mm Ultra Bright LED white	2	Borrowed from lab

Table 5: Bill of Materials

Chapter 6 | Initial Conceptual Design

This section presents detailed information and comparisons of our initial conceptual designs for the OpenBurger system. **The "final decision" mentioned here does not necessarily reflect the design ultimately adopted.** Instead, it represents the decision made during week 7 of the semester for the product design review. Certain design concepts were adjusted or discarded after testing and validation, which will be further discussed in Chapter 7.

6.1 Autonomous Exploration through Maze

6.1.1 Mapping techniques

One of the critical aspects of navigating through the maze is generating an accurate map of the environment. The following table compares two popular mapping techniques, GMapping and Cartographer SLAM, along with their respective pros and cons:

	Pros	Cons
Cartographer SLAM	-Able to create 2D and 3D maps for complex environments -Integrates real-time loop closure for more accurate mapping -Performs well in dynamic environments	-High computational demand in terms of processing power -Harder to use -Requires more memory and storage
GMapping	-Easier to use -Effective in indoors and small environments -Less computationally heavy	-Not effective in large and dynamic environments -Not as accurate as Cartographer

Table 6.1.1: Mapping Techniques

Cartographer SLAM (Simultaneous Localization and Mapping) is chosen as it is able to generate a more accurate and precise map for navigation, which is the most essential part of the mission.

While GMapping is easier to use and less computationally demanding, it may not be effective in the potentially large and random maze environment. Its mapping accuracy is also lower compared to Cartographer SLAM, which could lead to navigation errors or missed paths.

Although Cartographer SLAM has higher computational requirements and may be more challenging to set up and use initially, the requirements are met by offloading the processing load to the remote PC, which is much more capable than the Raspberry Pi.

6.1.2 Pathing Algorithm

Our plan for autonomous exploration and navigation of the maze was to execute a pathing algorithm using SLAM map data generated on-the-fly. This way, the OpenBurger can map and navigate out of the maze of unknown layout by itself to the elevator. The following table compares the possible pathing algorithms to be used, along with their respective pros and cons:

	Pros	Cons
Dijkstra's Algorithm	-Easier to implement compared to A* algorithm -Able to help find the shortest path to destination	-Slower than A* algorithm
A* Algorithm	-Higher levels of accuracy due to using heuristics -If used correctly, able to find the path to destination the fastest compared to the other 2 algorithm	-Harder to implement -Requires good heuristic
Breadth-First Search (BFS)	-Very simple and easy -Almost guaranteed to find destination as it checks everywhere	-Memory intensive -Might end up taking a lot of time to find the destination

Table 6.1.2: Pathing Algorithms

Dijkstra's algorithm is chosen for its simplicity over the A* algorithm while also being able to find the shortest path to the destination better than the Breadth-First Search algorithm.

The A* algorithm, while highly accurate due to its use of heuristics, is more complex to implement and requires careful selection of a good heuristic function. Dijkstra's algorithm, on the other hand, is easier to implement and still guarantees finding the shortest path, making it a more practical choice for our navigation needs.

The Breadth-First Search algorithm, although straightforward, may take an excessive amount of time to find the destination, as it exhaustively checks all possible paths. While it is almost guaranteed to find the destination eventually, the potential time inefficiency makes it a less desirable option compared to Dijkstra's algorithm, especially when considering the 20 minute time allocated for the entire mission including the dropping of ping pong balls.

6.2 Communication Protocols

OpenBurger sends a request to the ESP32 server to open the elevator door. The protocol used is HTTP, and the two devices are connected to a local Wi-Fi network. The following table compares the different versions of the HTTP protocol for the delivery robot to receive order information, along with their respective pros and cons:

	Pros	Cons
HTTP 1.1	-Most popular library for sending HTTP requests on Python -Stable and reliable	-Less efficiency and more constraints
HTTP 2.0	-New technologies such as multiplexing and header compression -Faster and more efficient	-Not supported by the Requests library, thus requires other libraries such as Hyper or HTTPX

Table 6.2: Communication Protocols

HTTP 1.1 is chosen as the JSON string we are receiving is actually lightweight and doesn't require high efficiency upon the receive action. It is a stable and reliable protocol, with the Requests library being the most popular choice for sending HTTP requests in Python. Since the order information we need to receive is a lightweight JSON string, the efficiency gains of HTTP 2.0 are not crucial. HTTP 2.0 also requires other libraries like Hyper or HTTPX, which are less well-maintained or are still in development (beta stage). Therefore, the simplicity and widespread support of HTTP 1.1 make it the more suitable choice for our application.

6.3 Navigating to the Bucket

6.3.1 Identifying the Lift

The following table compares the different methods considered for identifying the lift location, along with their respective pros and cons:

	Pros	Cons
NFC Tags	-Easier to implement -Minimal resources consumption required -Cheap and reliable	-Limited functionality but able to do its job of transmitting information -Have to put 2 NFC tags at the lifts
OpenCV	-Offers visual data processing -Can be used for other purposes like helping in navigation	-Computationally and resource heavy -Hard to setup and configure Opencv -Depends on camera quality

Table 6.3.1: Identifying the lift

NFC tags is our final decision as it is able to perform the identifying process and, at the same time, is easier to be implemented. While OpenCV offers visual data processing capabilities and the potential for additional uses like assisting in navigation, it is computationally and resource-heavy. Setting up and configuring OpenCV can also be challenging, and the accuracy of the system depends heavily on the quality of the camera hardware.

In addition, NFC tags are a simpler and more straightforward approach. It requires minimal resources and is relatively cheap and reliable. Although NFC tags have limited functionality and would require placing two tags at the lift locations, they can still perform the essential task of transmitting information about the lift's presence and location.

6.3.2 Identifying the Bucket

The following table compares the different methods considered for identifying the bucket before launching the ping pong balls, along with their respective pros and cons:

	Pros	Cons
OpenCV	- Easy colour recognition - Able to obtain bucket’s	- High cost of camera - High processing power requirement

	distance and orientation - No need for trackers	
Bluetooth Low Energy (BLE) positioning	- Low cost - Positioning is possible by trilateration	- Lower accuracy - Requires 3 or more active fixed nodes for trilateration
Radio Frequency Identification (RFID)	- Low cost - Passive tags do not require power or circuitry	- Positioning is not possible - Short maximum range
Ultra Wideband Positioning (UWB)	- More accurate - Positioning is possible by trilateration	- Higher cost compared to BLE - Requires 3 or more active fixed nodes for trilateration

Table 6.3.2: Identifying the bucket

The decision to use OpenCV for identifying the bucket for the ping pong ball launcher is based on several factors. While the other methods like Ultra Wideband (UWB) Positioning, Radio Frequency Identification (RFID), and Bluetooth Low Energy (BLE) Positioning have their own advantages, OpenCV provides a more straightforward and cost-effective solution for our specific requirements.

One of the main advantages of OpenCV is its ability to perform easy colour recognition. By training the system to recognize a specific colour, in this case the red colour of the bucket, we can reliably identify the bucket's location without the need for additional tracking devices or infrastructure.

Furthermore, OpenCV can also provide information about the bucket's distance and orientation relative to the TurtleBot's camera. This information can be useful for accurately aiming and launching the ping pong balls toward the bucket.

Another advantage of using OpenCV is the flexibility it provides. If needed, the same camera and image processing pipeline could potentially be extended for other tasks, such as assisting in navigation or object avoidance, without the need for additional hardware.

6.4 Ping Pong Ball Shooting Mechanism

The following table includes the possible shooting mechanisms for the ping pong ball launcher, along with their respective pros and cons:

	Pros	Cons
Flywheels Launching Mechanism	-Can fit 5 ping pong balls in the PVC pipe -Accessible and cost-effective parts -Easy mechanism	-Uses electricity for the flywheels motor -Servo motor might not dispense 1 ping pong ball at once
Catapult Launching Mechanism	-Operates without using much power -More consistent -Does not have to compute the projectile launch	-Harder to design a reload system for catapult -The catapult might block the LIDAR sensor
Spring-Loaded Launching Mechanism	-Adjustable tension of the spring to adjust the launching distance -Straightforward to operate	-Potential for injuries when handling the spring system -Excessive force may destabilise the TurtleBot

Table 6.4: Ping Pong ball shooting mechanism

Our final decision will be the Flywheels Launching Mechanism as it is the relatively easiest and cheapest way to provide a stable launching process. The amount of electrical energy used is affordable by the TurtleBot's battery after examination.

The Flywheels Launching Mechanism involves using a small motor to spin a set of wheels or flywheels. When the ping pong balls are fed into the spinning flywheels, they will be launched out at a good velocity. This design leverages the accessible PVC pipes and motor components, making it a cost-effective solution.

While a servo motor might have issues dispensing one ping pong ball at a time, the advantage of being able to store multiple balls in the PVC pipe outweighs this concern. The Flywheels Mechanism is also relatively easy to implement and can fit within the space constraints of the TurtleBot.

In comparison, the Catapult Launching Mechanism, although operating without much power and being more consistent, would be harder to design a reload system for and could potentially block the LIDAR sensor, which is crucial for navigation.

The Spring-Loaded Launching Mechanism, while straightforward to operate and offering adjustable tension for controlling the launch distance, poses potential safety risks when handling the spring system. Additionally, excessive force from the spring could destabilise the TurtleBot, which is undesirable.

Chapter 7 | Testing & Validation (Conceptual Design → Final Design)

This section provides justifications for the decisions made regarding the adoption or rejection of strategies mentioned in Chapter 6. Testing and validation processes were conducted to ensure that the final strategy chosen for each subsection is the most suitable option, considering factors such as cost-effectiveness, energy consumption, consistency, and ease of control.

7.1 Autonomous Exploration Through Maze

In the end, we decided to switch from autonomous navigation using pathing algorithms to coloured line following. This was because of several reasons:

1. Our team members had limited knowledge in the domain of using Dijkstra's algorithm for autonomous navigation.
2. There were more online resources regarding line following using OpenCV compared to autonomous navigation using Dijkstra's algorithm.
3. Line following allows for a faster mission completion time compared to using Dijkstra's algorithm, hence being advantageous for competitive scoring.

7.1.1 Line Following, Initial Version

The initial conceptual design was under the impression that we could use line following for mapping. The expected scenario was to map out the whole maze following the coloured line, and then proceed to complete the mission.

The robot would first follow the blue coloured tape, and when it reaches a dead end, there will be a special yellow signature marker. This yellow marker signals to the robot to turn 180 degrees, and then looks for the next coloured line to follow. In the example maze below, it would be red. The purpose of changing the coloured line is to ensure that in the case where there are 2 lines laid out beside each other, the robot would know which is the correct line to follow. The maze below is a demonstration of how the line would be laid out.

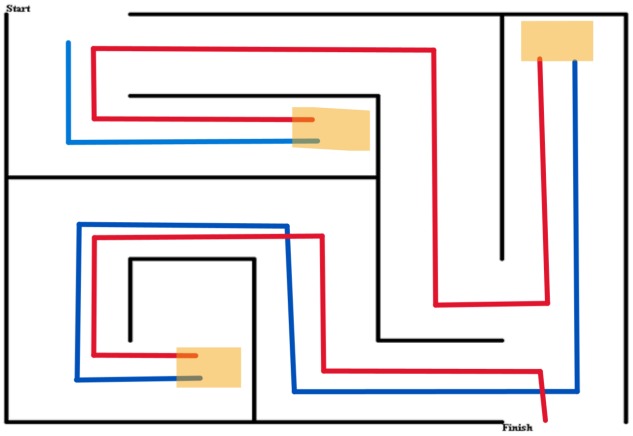


Figure 7.1.1: Layout of line follower strategy

For the movement behaviour, we had 2 fixed variables, *speedchange*, and *rotatechange*. The robot would always move at a fixed linear speed of *speedchange*, and for the rotation speed, we set a 45-degree threshold where if we see that the line is < 45 degrees or > 135 degrees, the robot would turn for a fixed period of 0.25s at the speed of *rotatechange*.
7.1.2 Line Following, Final Version After discovering that line following couldn't be utilised for mapping purposes, we eliminated the distinct yellow marker and standardised line following to a single colour. Additionally, we introduced an autonomous navigation algorithm as detailed in section 2.3.3. Recognizing inaccuracies in our previous movement behaviour algorithm, particularly during turns, we opted to overhaul the movement dynamics for improved precision. Details of this updated movement behaviour are also provided in section 2.3.3.
7.1.3 Autonomous Exploration after Mission In the conceptual design phase, the OpenBurger will terminate and complete its mission after dropping the ping pong balls. But since line following cannot be used for mapping purposes, OpenBurger will traverse the maze one more time with autonomous exploration. Once the mission ends, the OpenBurger will proceed to backtrack and make a 180° turn and follow the original line back to the yellow mark set for HTTP call. For the second time the OpenBurger detects this mark, it will trigger it to run the autonomous exploration function. The autonomous exploration function is made based on the r2autonav.py file that has been provided. We calibrated the speed of rotation, angle for detection, and also the distance for the robot to stop. Another additional function is to add a counter which detects the not changing of the occupied points in the occupancy grid map and try to navigate itself out of that area which trapped it. The details and the code are provided in section 2.3.3.

Table 7.1: Autonomous Exploration Through Maze

7.2 Communication Protocols

Our project continued to utilise the HTTP 1.1 library for sending and receiving HTTP POST requests due to its reliability and compatibility with our system architecture.

7.3 Navigating to the Bucket

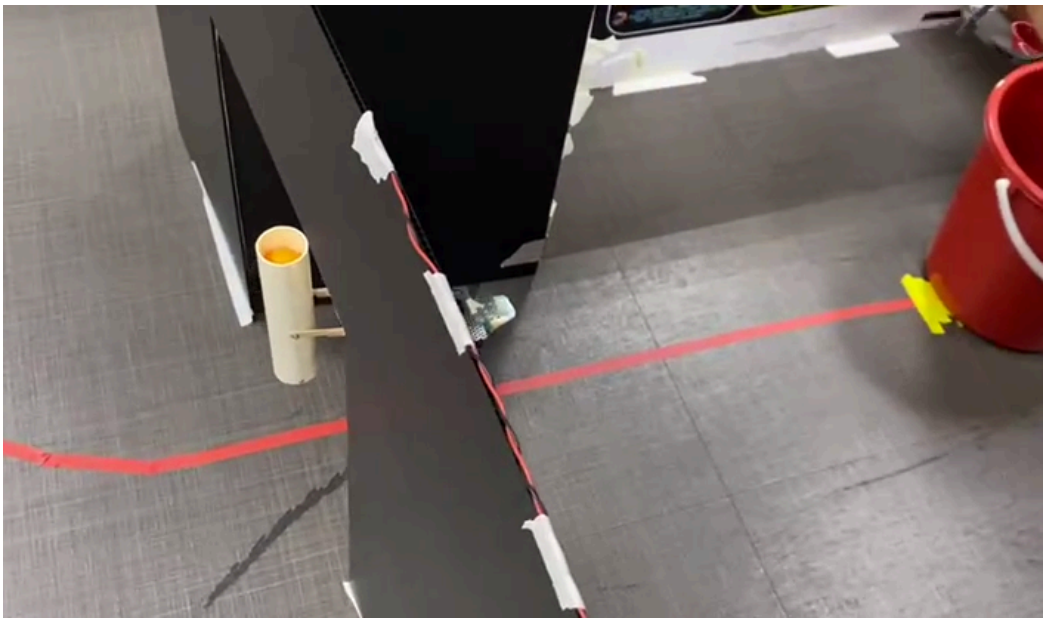


Figure 7.3.2: Yellow marker signifying the location of bucket

7.3.1 Identifying the Lift

The initial plan was to use NFC tags to identify the location of the lift. However, this idea was forfeited as using our line-following method, there is no need to identify the location of the lift, as the robot can simply follow the line that will lead it to the lift. To identify and differentiate between the two lifts, we split up into two paths leading to a lift each, characterised by red and blue coloured lines.

7.3.2 Identifying the Bucket

Initially, we intended to rely on OpenCV colour recognition to identify the bucket before launching the ping pong balls. However, we ultimately opted to use yellow tape markers positioned on the floor instead of directly identifying the bucket. Upon detecting the yellow marker, the line following algorithm was terminated, and the balls were dropped into the bucket. By using yellow tape markers, we would have better precision in controlling the distance between the robot and the bucket, facilitating precise activation of the shooting mechanism at the moment we want.

Table 7.3: Navigating to the bucket

7.4 Ping Pong Ball Delivery Mechanism

7.4.1 Flywheel Mechanism, Initial Version

Figure 7.4.1: Flywheel Mechanism

We initially opted for a flywheel ball launcher mechanism as it offered the advantage of continuously launching ping pong balls with a single uninterrupted motion. However, after thorough testing and validation, we decided to discard the flywheel ball launcher mechanism due to its inconsistency, stemming from both mechanical and electrical factors.

Mechanical

We encountered challenges in properly securing the flywheels onto the platform due to the longer-than-expected DC motors we purchased, which did not fit snugly into the motor holder we 3D printed. This resulted in instability of the motors. Additionally, upon contact with the ping pong ball, the DC motor tended to veer outward, reducing friction and contact with the ball, leading to suboptimal launching distances.

Furthermore, the tall mechanical structure of our flywheel launcher prototype significantly increased the centre of gravity of our robot. To avoid obstructing the LIDAR sensor, we could only position the launcher on the top layer of the Turtlebot3 burger. However, this placement contributed to the robot's lack of stability during movement and sudden stops, increasing the risk of toppling over, particularly at high speeds or abrupt halts.

Electrical

During testing, we encountered significant challenges in controlling the speed of the motor. By manipulating the current drawn by both motors, we hoped to achieve consistent performance, but this approach proved unreliable. We estimated an 80% motor efficiency to account for power loss, but without detailed knowledge of specific losses—such as copper loss or rotational loss—it was difficult to maintain a consistent RPM for the DC motor.

This lack of precision, coupled with the fact that battery voltage variations influenced the DC motor's speed, led to inconsistent shooting distances. Ultimately, we abandoned this approach because it became clear that maintaining a constant motor speed was nearly impossible with the given setup. The variable speed affected the flywheel's shooting consistency, making it unreliable for our intended use.

Table 7.4.1: Flywheel Mechanism, Initial Version

7.4.2 Dropping Mechanism, Final Version

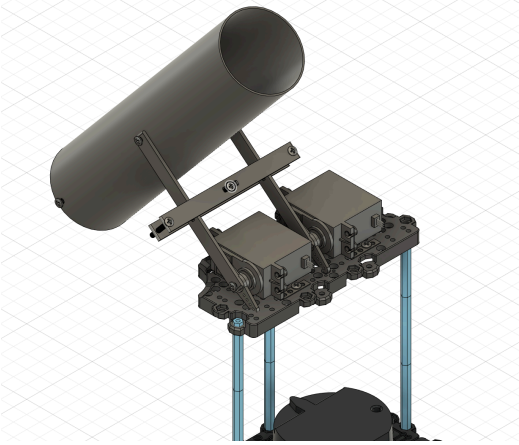


Figure 7.4.2: Dropper Mechanism

We transitioned to a simpler ping pong ball delivery mechanism that is more consistent and easier to operate. This mechanism employs two servo motors to rotate the mechanical arms which are connected to a ping pong ball holder (PVC pipe) by 90 degrees, effectively dropping the ping pong balls into the bucket.

Mechanical

The ball dropper system boasts a lighter weight compared to the previous flywheel launcher system. Despite utilising only one piece of the waffle plate, positioned at the rear of the robot, it achieves balance with the Li-Po battery weight placed on the opposite (front) side. Additionally, the centre of gravity is significantly reduced, contributing to overall structural stability. The robust servo motors effectively maintain the mechanical structure in a stable position when powered.

Furthermore, the system is meticulously adjusted to tilt the angle of the tube after the 90-degree rotation of the servo motors. This configuration allows the ping pong balls to be released one by one due to gravitational pull. Simultaneously, precautions are taken to ensure that the PVC pipe does not make contact with the bucket, thus avoiding penalties during the mission.

Electrical

Initially, the SG90 servo motor was used to release balls into a bucket. However, we quickly discovered that it lacked the torque necessary to lift the tube and drop the balls. Various attempts to increase its performance, such as adjusting the pulse-width modulation (PWM)

signal and connecting the motor to an external 5V power source, were unsuccessful. To overcome this limitation, we replaced the SG90 with the more powerful MG 995 servo motor. This servo motor provided enough torque to lift the arm and drop the balls into the bucket, but it exhibited a degree of jiggling that occasionally caused balls to fall off the tube.

To mitigate this jiggling, we attempted to adjust the PWM signal to stabilise the servo's movement, but these efforts were not successful. Further investigation led us to implement the pigpio library, allowing us to more accurately control the PWM and fine-tune the turning angle of the servo. This approach reduced the jiggling and improved the servo's overall performance, ensuring that balls were dropped reliably into the bucket.

Table 7.4.2: Dropping Mechanism, Final Version

Chapter 8 | Conclusion & Areas for Improvement

8.1 Key Findings

The table below summarises the key findings obtained throughout this project, categorised into mechanical, electrical, and software aspects:

Mechanical ● Calibrating the flywheel launcher system proves challenging due to the inconsistent contact of the ping pong balls with the flywheels. ● Maintaining a low centre of gravity for the robot while ensuring its ability to effectively launch ping pong balls into the tall bucket poses difficulty. ● The ping pong ball holder (PVC pipe) is not securely fixed and may tilt when the servo motors exhibit jiggling behaviours.
Electrical ● The default RPI.GPIO library caused the servo to exhibit jiggling behaviours ● The inconsistent power supply to the Dynamixel motor resulted in the robot's inability to turn to the desired angles. ● The Raspberry Pi has only one 5V pin to supply the power to 2 MG995 servo motors and one LED as another 5V pin is used to power up the Raspberry Pi by OpenCR 5V socket.
Software ● Under specific lighting conditions, colour detection accuracy was compromised, occasionally resulting in premature detection of the special action marker (yellow) and premature execution of actions. ● The autonomous navigation algorithm occasionally misinterpreted directions, causing the robot to become trapped moving between two corners of a maze segment. ● The navigation speed was notably slower in comparison to other teams.

Table 8.1: Key findings

8.2 Discussions

The table below summarises the success of our project in addressing the encountered challenges:

Mechanical ● Implemented the ping pong ball dropper strategy for its improved consistency and cost-effectiveness. ● Reduced the weight of the payload system to lower the centre of gravity. ● Utilised round-hole aluminium plates to securely fasten the servo motors, LED light, and Raspberry Pi camera onto the Turtlebot.
Electrical ● Import pigpio library to control the PWM and turing angle precisely and reduce the jiggling of the servo motor despite in Rosbu or non-Rosbu condition ● We solder the 5V pin and Ground Pin on a vero board which increase the number of 5V pin and Ground Pin of the Raspberry Pi but the current is lower as it is distributed into each branches
Software ● The robot was able to successfully follow the line, send the HTTP request, and activate the launching mechanism without fail in a single try, under normal lighting circumstances.

Table 8.2: Discussions

8.3 Areas for further improvements

The table below summarises areas for further improvement across the mechanical, electrical, and software aspects of the project:

Mechanical ● Implement the use of proper fastener techniques such as using lock nuts to further strengthen the loose joints ● Customise and 3D print the L-brackets used to secure the servo motors onto the waffle plates instead of using round-holes aluminium plate, to ensure better fit. ● Develop a stronger mechanical arm structure that has 3 points of contact with the PVC pipe to avoid tilting of the PVC pipe.
Electrical ● Implement the use of Gyroscope or Inertial measurement unit (IMU) to ensure that the robot turn in the desired direction and accelerate and decelerate accurately ● Implementing an external power supply such as AA battery to the component used such as the 5V pin of the MG995 servo motor can be connected to 2 double A batteries which provide 6V to the servo motor to ensure enough energy is supplied to the component in the long run. ● Try to implement better cameras such as Rpi Camera V3 in the future which are able to detect the accurate colour type and contour of the colour even in low ambient light.
Software ● In the future, we plan to create a ROS2 node for our OpenCV. We could integrate the usage of cv_bridge to convert between ROS messages and OpenCV images. This will allow us to receive a live video feed from the RPI. which will help greatly in the debugging process. ● We also plan to further calibrate our HSV values so that the colour detection will be more accurate throughout different lighting conditions. ● Our line following navigation algorithm could be further calibrated using different values to achieve a faster speed. ● Our autonomous navigation algorithm could potentially utilise path-finding algorithms to increase the success rate and efficiency.

Table 8.3: Areas for further improvements

Chapter 9 | References

TurtleBot3 (robotis.com). (2024). Retrieved from https://emanual.robotis.com/docs/en/platform/turtlebot3/hardware_setup/

ESP32 Set Up Manual retrieve from https://github.com/edic-nus/EG2310/blob/main/src/main.ino

Mason, M. T. (2001). Mechanics of Robotic Manipulation. In *The MIT Press eBooks*. The MIT Press. https://doi.org/10.7551/mitpress/4527.001.0001

Riisgaard, S., & Blas, M. (n.d.). *SLAM for Dummies A Tutorial Approach to Simultaneous Localization and Mapping By the "dummies."* Retrieved April 12, 2020, from https://dspace.mit.edu/bitstream/handle/1721.1/36832/16-412JSpring2004/NR/rdonlyres/Aeronautics-and-Astronautics/16-412JSpring2004/A3C5517F-C092-4554-AA43-232DC74609B3/0/1Aslam_blas_report.pdf

Cyrill Stachniss. (2009). Robotic Mapping and Exploration. In Springer tracts in advanced robotics. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-01097-2

Abiy, T., Pang, H., & Williams, C. (2016). Dijkstra's Shortest Path Algorithm | Brilliant Math & Science Wiki. Brilliant.org. https://brilliant.org/wiki/dijkstras-short-path-finder/

A* Search | Brilliant Math & Science Wiki. (2016). Brilliant.org. https://brilliant.org/wiki/a-star-search/

Breadth-First Search (BFS) | Brilliant Math & Science Wiki. (n.d.). Brilliant.org. https://brilliant.org/wiki/breadth-first-search-bfs/

ABCOM. (2019, January 30). HTTP/1.1 vs HTTP/2: What's the Difference? Digitalocean.com; DigitalOcean. https://www.digitalocean.com/community/tutorials/http-1-1-vs-http-2-what-s-the-difference

The HTTP protocol. (n.d.). Www.ibm.com. https://www.ibm.com/docs/en/cics-ts/5.3?topic=concepts-http-protocol

Jones, K. (2023, December 26). Object Detection with Python using OpenCV: Introduction to computer vision. Medium. https://medium.com/@kylejones_47003/object-detection-with-python-using-open-cv-introduction-to-computer-vision-74508c39191d

Chandler, N. (2012, March 14). What's an NFC Tag? HowStuffWorks. https://electronics.howstuffworks.com/nfc-tag.htm

By. (2021, June 9). What Is Ultra Wideband? Hackaday. https://hackaday.com/2021/06/09/what-is-ultra-wideband/#more-481084

Ultra-wideband (UWB) for precise indoor location tracking | RFiD Discovery. (n.d.). Www.rfiddiscovery.com. Retrieved January 30, 2024, from https://www.rfiddiscovery.com/en/content/ultra-wide-band-uwb-location-tracking

Adaptation of Ultra Wide Band positioning system for Adaptive Monte Carlo Localization | IEEE Conference Publication | IEEE Xplore. (n.d.). Ieeexplore.ieee.org. Retrieved January 30, 2024, from https://ieeexplore.ieee.org/document/9874314

Proctor, B. (n.d.). Active Vs. Passive RFID For Location Tracking | Link Labs. Www.link-Labs.com. https://www.link-labs.com/blog/active-vs-passive-rfid

Kingatua, A. (2020, March 26). Bluetooth Indoor Positioning and Asset Tracking Solutions. Medium. https://medium.com/supplyframe-hardware/bluetooth-indoor-positioning-and-asset-tracking-solutions-8c78cae0a03

python-hyper/hyper. (2024, January 29). GitHub. https://github.com/python-hyper/hyper

encode/httpx. (2024, January 30). GitHub. https://github.com/encode/httpx/

franciscodias.net, X. site: (n.d.). Arduino Controlled Ping Pong Balls Launcher. Instructables. https://www.instructables.com/Arduino-controlled-Ping-Pong-Balls-Launcher/

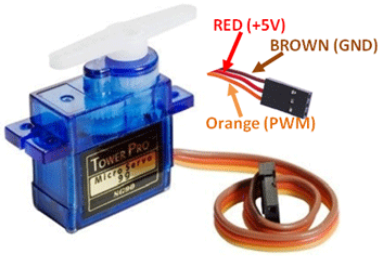
Chapter 10 | Appendices

10.1 List of Tables

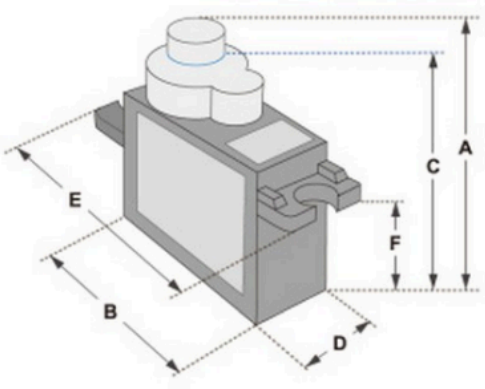
10.2 List of Figures

10.3 List of Datasheet

10.3.1 SG90 Servo motor

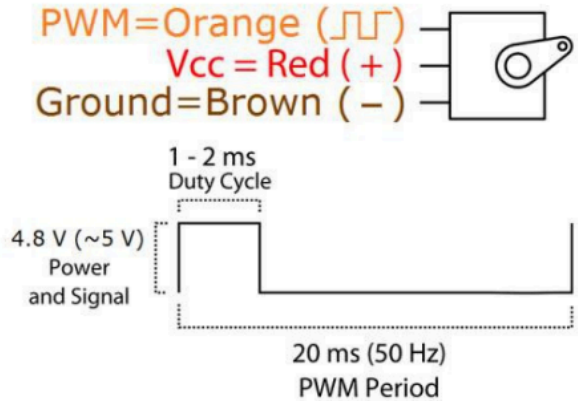


Tiny and lightweight with high output power. Servo can rotate approximately 180 degrees (90 in each direction), and works just just like the standard kinds but smaller. You can use any servo code, hardware or library to control these servos.



Dimensions & Specifications
A (mm) : 32
B (mm) : 23
C (mm) : 28.5
D (mm) : 12
E (mm) : 32
F (mm) : 19.5
Speed (sec) : 0.1
Torque (kg-cm) : 2.5
Weight (g) : 14.7
Voltage : 4.8 - 6

Position "0" (1.5 ms pulse) is middle, "90" (~2ms pulse) is middle, is all the way to the right, "-90" (~1ms pulse) is all the way to the left.

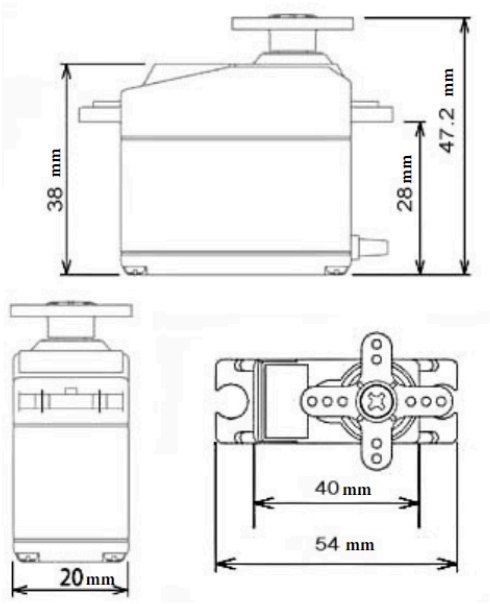


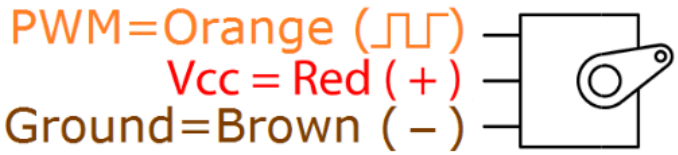
10.3.2 MG995 High Speed Servo Actuator

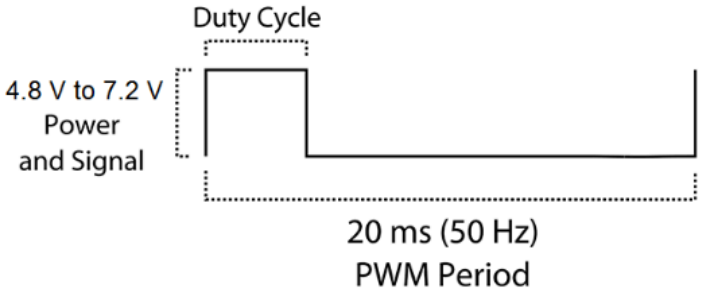
The unit comes complete with colour coded 30cm wire leads with a 3 X 1 pin 0.1" Pitch type female header connector that matches most receivers, including Futaba, JR, GWS, Cirrus, Blue Bird, Blue Arrow, Corona, Berg, Spektrum and Hitec. This high-speed servo actuator is not code dependent; You can use any servo code, hardware or library to control them.

Specifications

- Weight: 55 g
- Dimension: 40.7 x 19.7 x 42.9 mm approx.
- Stall torque: 8.5 kgf·cm (4.8 V), 10 kgf·cm (6 V)
- Rotation Angle: 120deg. (+- 60 from center)
- Operating speed: 0.2 s/60º (4.8 V), 0.16 s/60º (6 V)
- Operating voltage: 4.8 V to 7.2 V
- Dead band width: 5 µs
- Stable and shock proof double ball bearing design
- Metal Gears for longer life
- Temperature range: 0 ºC – 55 ºC







10.3.2 Raspberry Pi Camera V2

Raspberry Pi Camera v2

***Part number:** RPI 8MP CAMERA BOARD*

- 8 megapixel camera capable of taking photographs of 3280 x 2464 pixels
- Capture video at 1080p30, 720p60 and 640x480p90 resolutions
- All software is supported within the latest version of Raspbian Operating System

The Camera v2 is the new official camera board released by the Raspberry Pi foundation.

The Raspberry Pi Camera Module v2 is a high quality 8 megapixel Sony IMX219 image sensor custom designed add-on board for Raspberry Pi, featuring a fixed focus lens. It's capable of 3280 x 2464 pixel static images, and also supports 1080p30, 720p60 and 640x480p60/90 video. It attaches to Pi by way of one of the small sockets on the board upper surface and uses the dedicated CSi interface, designed especially for interfacing to cameras.

- 8 megapixel native resolution sensor-capable of 3280 x 2464 pixel static images
- Supports 1080p30, 720p60 and 640x480p90 video
- Camera is supported in the latest version of Raspbian, Raspberry Pi's preferred operating system

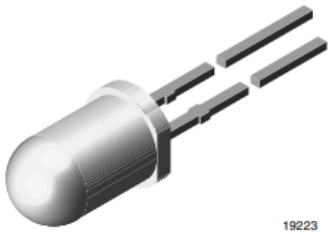
The board itself is tiny, at around 25mm x 23mm x 9mm. It also weighs just over 3g, making it perfect for mobile or other applications where size and weight are important. It connects to Raspberry Pi by way of a short ribbon cable.
The high quality Sony IMX219 image sensor itself has a native resolution of 8 megapixel, and has a fixed focus lens on-board. In terms of still images, the camera is capable of 3280 x 2464 pixel static images, and also supports 1080p30, 720p60 and 640x480p90 video.

Specification

- Net price: $25
- Size: Around 25 × 24 × 9 mm
- Weight: 3g
- Still resolution: 8 Megapixels
- Video modes: 1080p47, 1640 × 1232p41, and 640 × 480p206
- Sensor: Sony IMX219
- Sensor resolution: 3280 × 2464 pixels
- Sensor image area: 3.68 x 2.76 mm (4.6 mm diagonal)
- Pixel size: 1.12 µm x 1.12 µm
- Optical size: 1/4"
- Focus: Adjustable
- Depth of field: Approx 10 cm to ∞
- Focal length: 3.04 mm
- Horizontal Field of View (FoV): 62.2 degrees
- Vertical Field of View (FoV): 48.8 degrees
- Focal ratio (F-Stop): F2.0
- Maximum exposure times (seconds): 11.76
- Lens Mount: N/A
- NoIR version available?: Yes

10.3.3 Ultra Bright LED White

Ultrabright White LED, Ø 5 mm Untinted Non-Diffused Package



FEATURES

- Untinted non-diffused lens
- Utilizing ultrabright InGaN technology
- High luminous intensity
- Luminous intensity and color categorized for each packing unit
- ESD-withstand voltage: up to 4 kV according to JESD22-A114-B
- Circuit protection by Zener diode
- Material categorization: for definitions of compliance please see www.vishay.com/doc?99912

Pb-free

e3

RoHS COMPLIANT

HALOGEN FREE

GREEN (5-2008)

DESCRIPTION

The VLHW5100 is a clear, non-diffused 5 mm LED for high end applications where supreme luminous intensity required.

These lamps with clear untinted plastic case utilize the highly developed ultrabright InGaN technologies.

The lens and the viewing angle is optimized to achieve best performance of light output and visibility.

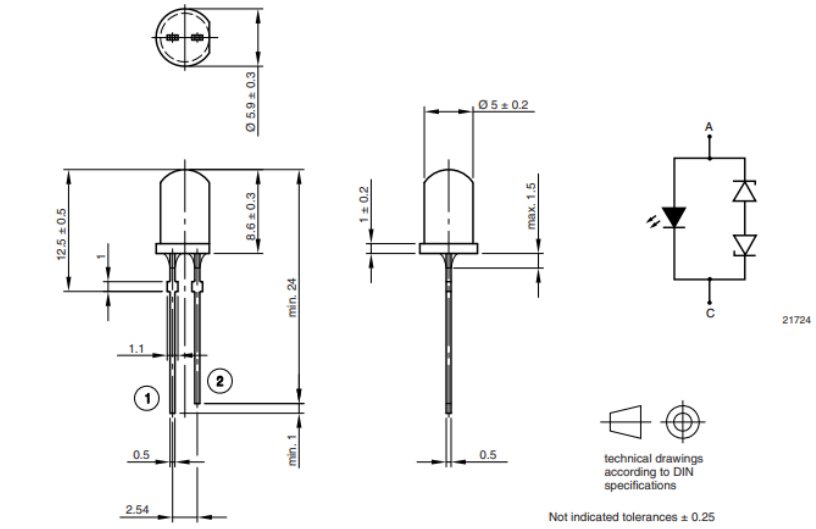
PRODUCT GROUP AND PACKAGE DATA

- Product group: LED
- Package: 5 mm
- Product series: standard
- Angle of half intensity: ± 10°

APPLICATIONS

- Interior and exterior lighting
- Outdoor LED panels
- Instrumentation and front panel indicators
- Replaces incandescent lamps
- Light guide compatible

PACKAGE DIMENSIONS in millimeters



PARTS TABLE														
PART	COLOR	LUMINOUS INTENSITY (mcd)			at I_F (mA)	COORDINATE (x, y)			at I_F (mA)	FORWARD VOLTAGE (V)			at I_F (mA)	TECHNOLOGY
		MIN.	TYP.	MAX.		MIN.	TYP.	MAX.		MIN.	TYP.	MAX.		
VLHW5100	White	5600	-	11 200	20	-	0.33, 0.33	-	20	2.8	-	3.6	20	InGaN and converter

ABSOLUTE MAXIMUM RATINGS (T_{amb} = 25 °C, unless otherwise specified) VLHW5100				
PARAMETER	TEST CONDITION	SYMBOL	VALUE	UNIT
Reverse voltage		V_R	5	V
DC forward current		I_F	30	mA
Peak forward current	at 1 kHz, t_p/T = 0.1	I_{FSM}	0.1	A
Power dissipation		P_V	100	mW
Zener reverse current		I_Z	100	mA
Junction temperature		T_j	100	°C
Operating temperature range		T_{amb}	-40 to +100	°C
Storage temperature range		T_{stg}	-40 to +100	°C
Soldering temperature	t ≤ 5 s	T_{sd}	260	°C
Thermal resistance junction-to-ambient		R_{thJA}	400	K/W

TYPICAL CHARACTERISTICS (T_{amb} = 25 °C, unless otherwise specified)

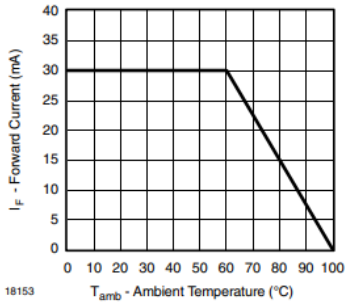


Fig. 1 - Forward Current vs. Ambient Temperature

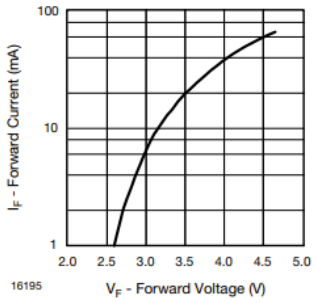


Fig. 4 - Forward Current vs. Forward Voltage

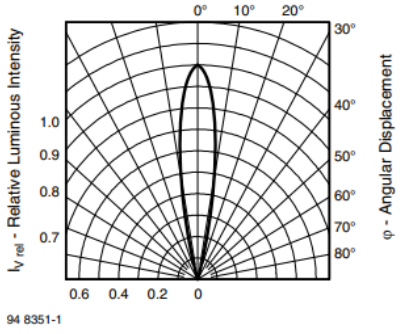


Fig. 2 - Relative Luminous Intensity vs. Angular Displacement

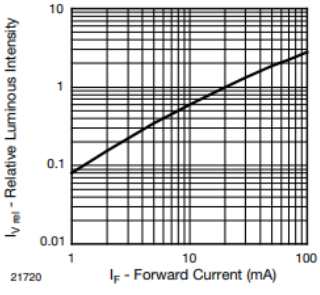


Fig. 5 - Relative Luminous Flux vs. Forward Current

10.4 G1 Documentation (Chapter 2,3,5)

Chapter 2 | Literature Review

2.1 Navigating through the maze

In order for the robot to avoid obstacles and find its way to the lift, we will need to employ mapping techniques so that the robot will know its surroundings, and then utilise pathing algorithms to get the robot to reach its destination.

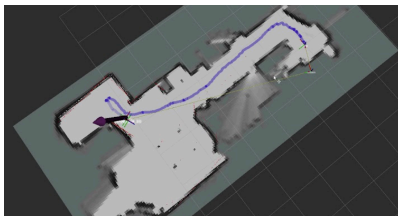
2.1.1 Mapping

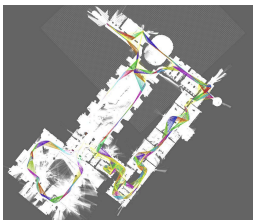
➢ GMapping

GMAPPING leverages Rao-Blackwellized particle filters, requiring a laser range finder (LIDAR) to efficiently perform SLAM in primarily indoor environments, making it an ideal choice for autonomous navigation in spaces like homes and warehouses where computational resources are limited.

➢ Cartographer SLAM

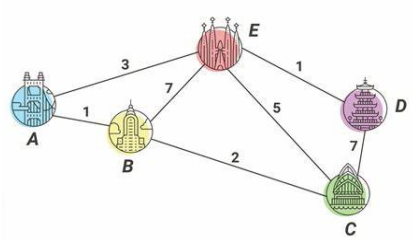
Cartographer SLAM, integrates real-time loop closure and advanced optimization for precise 2D and 3D mapping, widely used in complex applications such as autonomous vehicle navigation, large-scale environmental mapping, and augmented reality projects, where its ability to handle diverse, dynamic environments and produce highly accurate maps is essential.





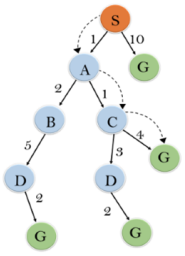
2.1.2 Pathing algorithms

➢ Dijkstra's Algorithm



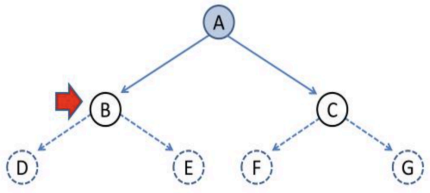
Dijkstra's algorithm is a weighted graph search algorithm used to find the shortest path from a single source node to all other nodes in a graph, where the edge weights represent the cost of traversing between nodes.

➢ A* Algorithm



A* Algorithm combines the benefits of Dijkstra's algorithm and heuristic search, making it more efficient and capable. It uses a heuristic function to estimate the cost to reach the goal from a node, prioritising nodes that lead to a more optimal path.

➢ Breadth-First-Search



BFS is an algorithm that explores all neighbour nodes at the present depth before moving on to nodes at the next depth level, ensuring a comprehensive search of the nearest paths. BFS is useful in simple and constrained environments, where the objective is to explore the vicinity thoroughly.

2.2 Sending out HTTP request

After clearing the maze, our robot would face two lifts and it is required to identify which lift will be opened. Our robot would need to send a HTTP request and receive a JSON string which indicates which door will be opened and identify the opened door

2.2.1 HTTP request and receive

A HTTP request consists of 3 elements: The HTTP method, request target, and the HTTP version.

The HTTP method indicates the desired action of the HTTP request. Most commonly used are the GET, POST and PUT methods. Since, we need to retrieve JSON containing information about the door to be opened, we will be using the GET method.

The request target is a URL to the server that will be responding to our HTTP request.

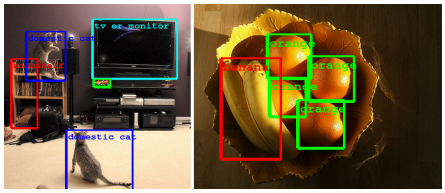
As for the HTTP version, there are two popular versions to choose from, HTTP/1.1 and HTTP/2.

- **HTTP/1.1**
  This protocol was introduced in 1997, with more functions compared to HTTP/1.0. The Requests library for Python, which is the most popular library for sending HTTP requests on Python (dependeed by >1 million Github repositories), uses this protocol. Hence, usage of HTTP/1.1 on Python is stable and reliable. However, less efficiency and more constraints compared with the newer generation, HTTP/2.
- **HTTP/2**
  This is a more modern protocol introduced in 2015. Due to new technologies such as multiplexing and header compression, HTTP/2 is faster and more efficient than HTTP/1.1. However, usage in Python requires other libraries such as Hyper or HTTPX, as it is not supported by the Requests library. Hyper is no longer being maintained, while HTTPX is in beta.

2.2.2 Identifying the Lift

After a Lift has been “opened”, our robot will need a way to differentiate which is the opened lift and which is the closed one(lift 1 and lift 2)

- **OpenCV**
  OpenCV is a library widely used for tasks like image and video analysis, object detection, and machine learning. When used with a Raspberry Pi camera module, OpenCV enables the Raspberry Pi to capture and process images and videos in real-time. This allows the robot to be able to see and differentiate the 2 different lifts .



➢ Near-field communication (NFC) Tags

NFC tags can be utilised to differentiate between lifts by placing NFC tags to each elevator. The robot can then read the unique data from the NFC tags using NFC reader hardware, allowing it to identify the 2 different lifts. This method provides an easy way to distinguish between different lifts.

2.3 PingPong Ball Launching

After solving the lift problem, our robot is now required to enter the room and locate the bucket. Then, it is required to launch 5 standard size ping pong balls accurately into an opened bucket.

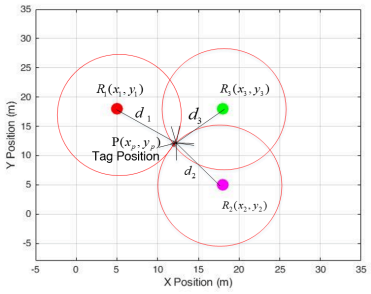
2.3.1 Identifying the bucket

➢ Ultra wideband (UWB) positioning

UWB is a short-range RF technology operating in the 3.1-10.6 GHz frequency range with at least 500MHz of spectrum. UWB relies on Time Difference of Arrival (TDOA) for positioning, providing distance measurements with centimetre-level accuracy (within 30 centimetres) in real-time. Getting the position of the TurtleBot relative to the buckets can be done through trilateration. Trilateration requires 3 or more fixed points (such as nodes placed in the buckets) to calculate the position of the Turtlebot in 2D space, and it works by finding the common intersection point of the circular wave of signals from the points.

➢ Radio Frequency Identification (RFID)

Passive RFID tags, when attached to buckets, are inexpensive and don't require batteries. The TurtleBot uses an RFID reader to wake up these tags with powerful low-frequency signals. The tags then send information like location back to the reader, but they can't be used for TurtleBot positioning. Their range is limited to about a metre due to their passive nature.

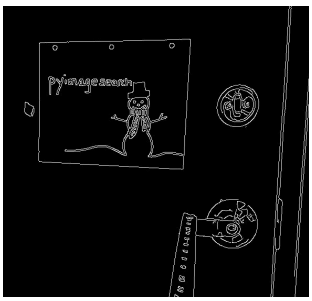


➢ Bluetooth Low Energy (BLE) positioning

BLE operates at 2.4 GHz and uses RSSI instead of TDOA for distance and direction estimation between the TurtleBot and buckets. To estimate object position, static beacons with known coordinates and RSSI data are used. For meter-level accuracy trilateration, at least 3 fixed beacons (excluding the TurtleBot) are required.

➢ Computer Vision

OpenCV, a mature Python library, can be used to process camera inputs on the TurtleBot. To identify a bucket, methods like edge, colour, or keypoint detection can be applied. Once identified, the bucket's distance from the TurtleBot can be calculated using triangle similarity:



$$D' = (W \times F) / P,$$

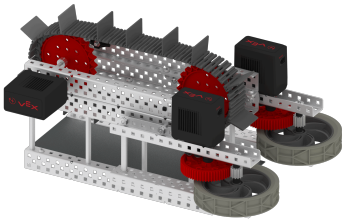
where D’ is the distance of the bucket to the camera, W is the width of the bucket, F is the focal length of the camera (to be calculated beforehand), and P is the perceived width in pixels.

TAdditionally, the bucket's angle from the TurtleBot can be determined based on its position in the image frame. This information can help the TurtleBot move into an optimal position for shooting ping pong balls.

2.3.2 Shooting Mechanism

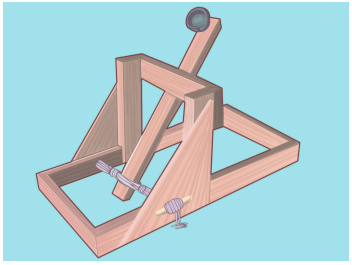
➢ Flywheels Launching Mechanism

Utilise a PVC pipe to neatly store five ping pong balls. Employ a servo motor to precisely dispense the balls into the flywheel area, where two motors power wheels for speed and launching. Incorporate a microcontroller such as Arduino or Raspberry Pi for precise control of the dispenser's movements.



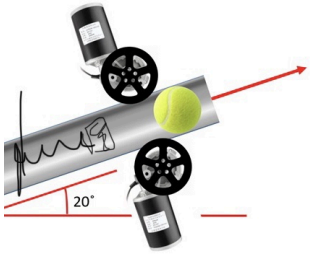
➢ Catapult Launching Mechanism

A ping pong ball is placed within a basket positioned at one end of a lever, with the lever being activated by a servo motor to launch the ball. The servo motor's rotation generates the force needed to propel the ping pong ball into a designated bucket. To enable continuous launching, a dispenser system is incorporated to automatically reload the catapult, allowing for the sequential throwing of multiple ping pong balls into the bucket.

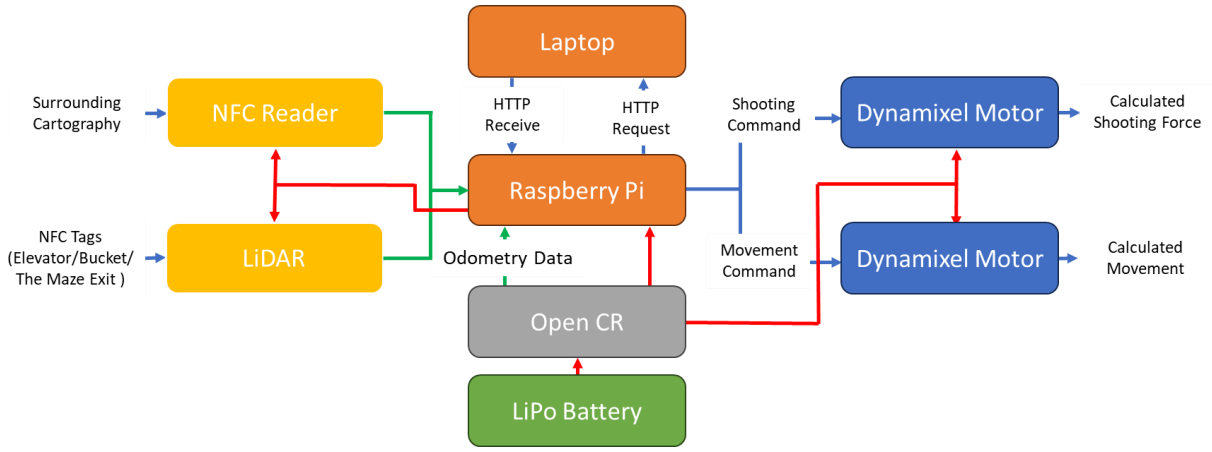


➢ Spring-Loaded Launching Mechanism

Position a ping pong ball inside a PVC pipe, employing a servo motor to unleash the compressed spring. The spring imparts the necessary force, propelling the ball into a designated bucket. For seamless and continuous launching, integrate a dispenser system, similar to the two other mechanisms, to reload the launcher efficiently.



Chapter 3 | Conceptual Design



3.1 Navigating through the maze

The primary objective is to enable the robot to create a detailed map of an unknown environment using Cartographer SLAM (Simultaneous Localization and Mapping) and then navigate efficiently through the environment using Dijkstra's algorithm.

3.1.1 Mapping process

The TurtleBot 3 Burger is equipped with a LIDAR and a Raspberry pi, making it suitable for SLAM and autonomous navigation tasks. The Raspberry pi will be used for data processing and control, allowing TurtleBot 3 to navigate through the environment. The LIDAR will then collect data on its environment and Cartographer SLAM will process this data to construct and update a 2D map of the environment in real-time.

3.1.2 Navigation process

Once the map is created, Dijkstra's algorithm will be implemented to determine the shortest path to take to the destination for the TurtleBot 3.

3.2 Identifying the lift and Sending out HTTP request

NFC tags will be placed at the lift so that when the robot reaches it, it will be able to scan the NFC tag, allowing it to identify whether the lift it is in front of is lift 1 or lift 2. It will then proceed to send a HTTP request, to open the lift door. Once the HTTP request sends back a response, the robot will move into the lift that is opened.

```
# Define function to open the door and identify the opened door
FUNCTION openDoorAndIdentify():
    # Set the URL for opening the door
    doorEndpoint = "https://example.com/open-door"
```

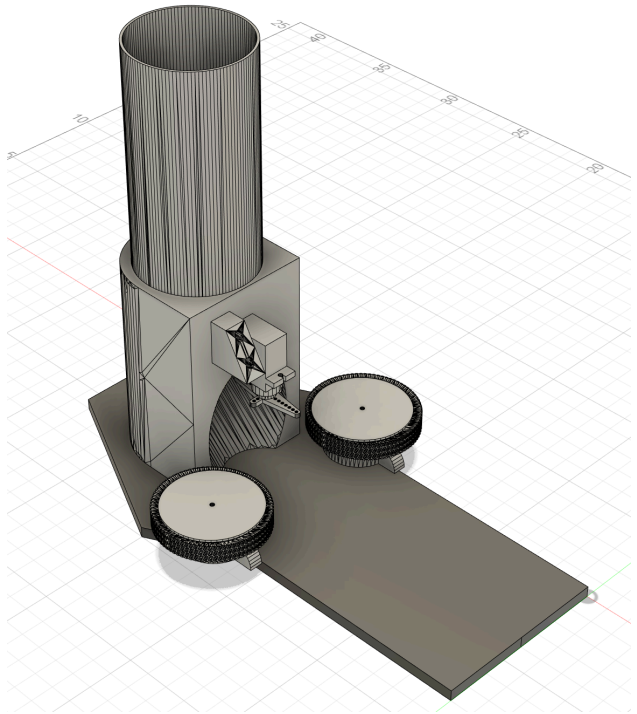
```
    # Try to send an HTTP GET request to open the door
    TRY:
        response = HTTP_GET(doorEndpoint)

        # Check if the request was successful (status code 200)
        IF response.status_code == 200 THEN
            # Parse the JSON to extract info about the opened door
            doorData = PARSE_JSON(response.body)

            # Identify the opened door from the JSON data
            openedDoor = doorData["opened_door"]

            # Return the identification of the opened door
            RETURN openedDoor
        ELSE
            # Print error message if the request was not successful
            PRINT "Error: Cannot open door: " + response.status_code
            RETURN NULL
        ENDIF

    # Handle exceptions, such as network errors
    EXCEPT Exception AS e:
        PRINT "Error: " + e
        RETURN NULL

# Example usage
openedDoor = openDoorAndIdentify()

# Print the result
IF openedDoor != NULL THEN
    PRINT "The opened door is: " + openedDoor
ELSE
    PRINT "Failed to open the door."
ENDIF
```

3.3 Ping Pong Ball Launching

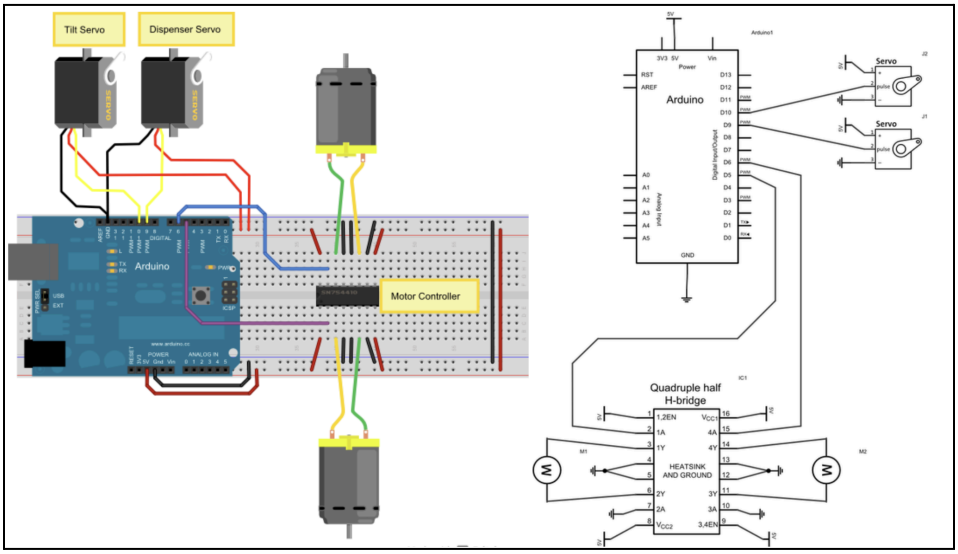
3.3.1 Identifying the bucket

When the bucket enters the lift, OpenCV will be utilised to process the video frame-by-frame to detect the red bucket, using colour detection method. Once the red bucket is detected, the LIDAR will be used to determine the distance from the bucket. The robot will then re-adjust itself, moving to an optimal distance and orientation (facing directly at the bucket) for shooting the ping pong balls into the bucket.

3.3.2 Shooting Mechanism

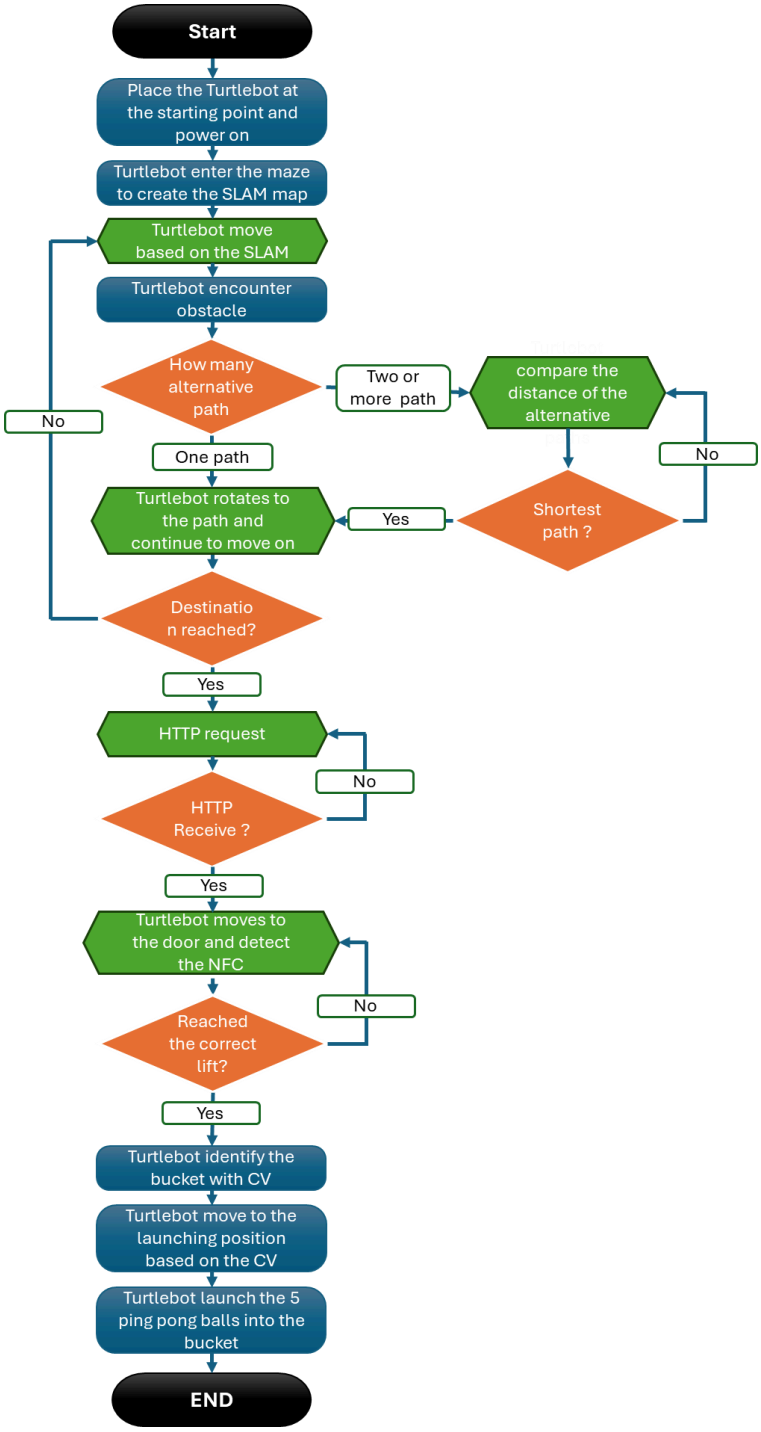


The Flywheels Launcher is a user-friendly mechanism, driven by a Raspberry Pi control system, designed to catapult ping pong balls with precision and speed. This launcher utilises a servo for controlled ball dispensing and two strategically positioned motors with flywheels. The 3D-printable components, coupled with a CNC or laser-cut base, facilitate a straightforward fabrication process. The servo, meticulously programmed by Raspberry Pi, releases ping pong balls from the hopper, while the flywheels provide the necessary rotational energy to launch them with accuracy. The launcher will be installed at the front of our robot without blocking the LIDAR sensor and the launching angle could be adjusted by tilting the system as a whole.

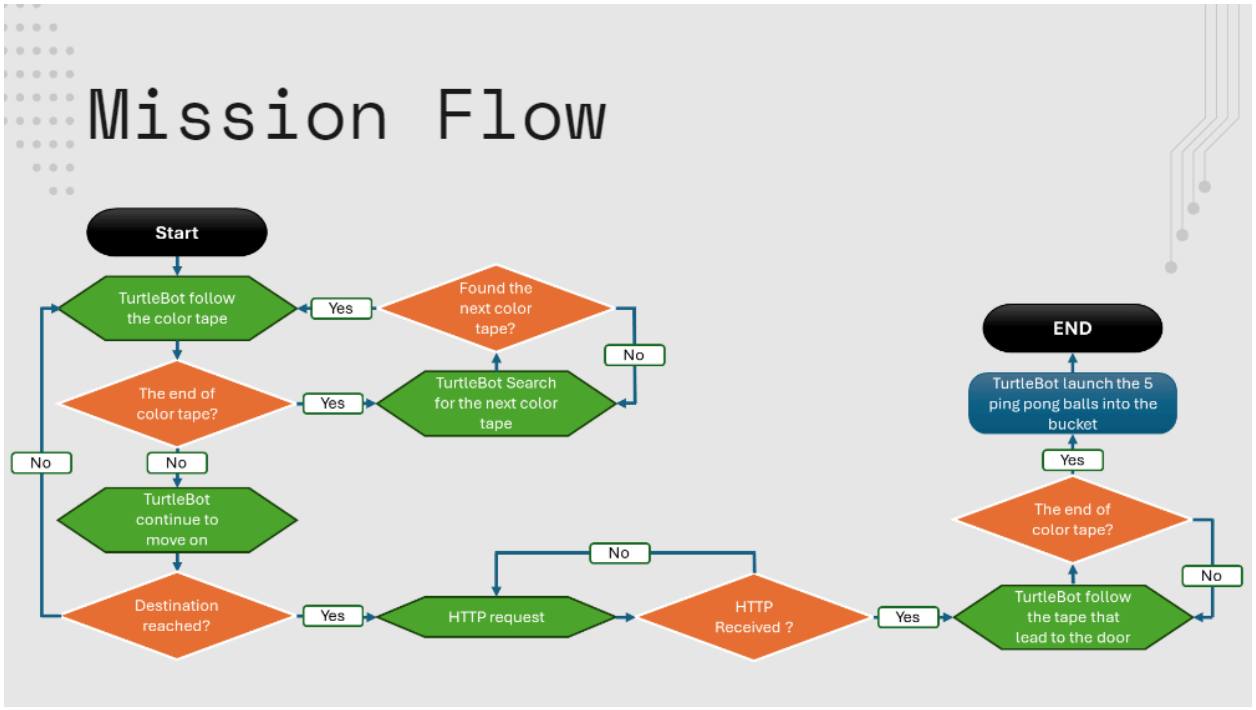


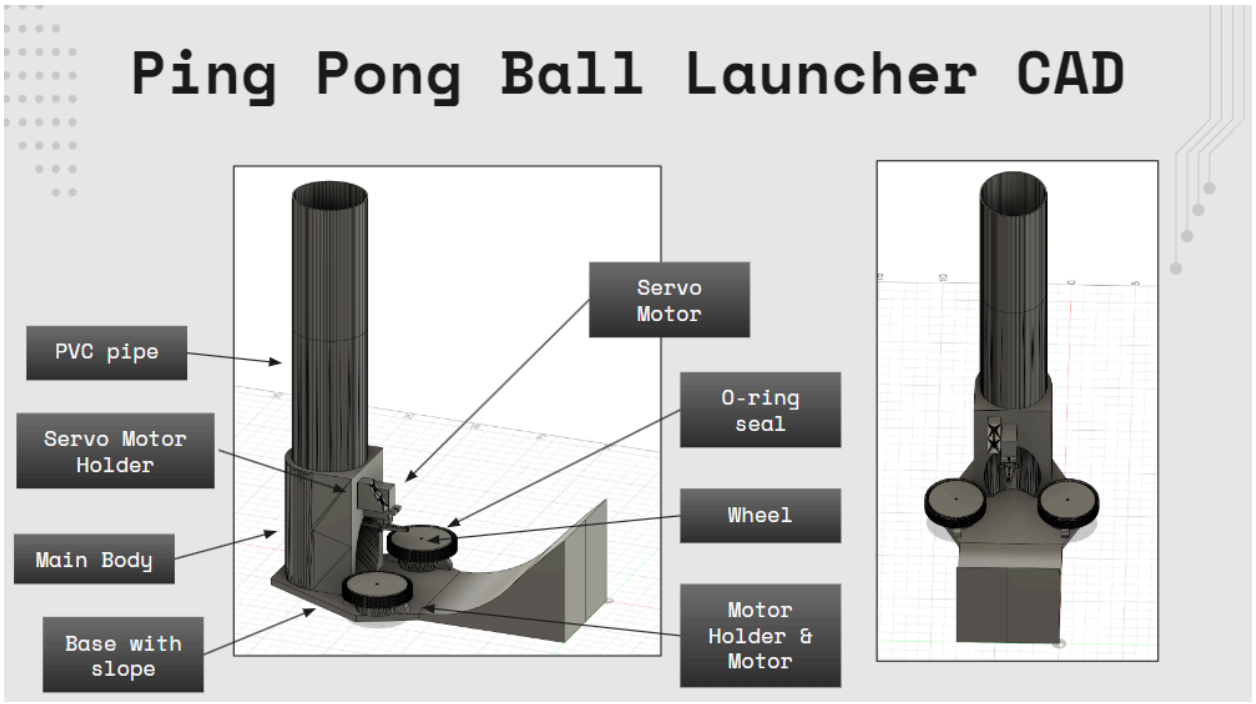
Chapter 5 | Preliminary Design

5.1 Mission Plan



10.5 Preliminary Design Review





Fabrication Plan

3D Printing

Materials = ABS

3D Printer = TwoTrees SP-5

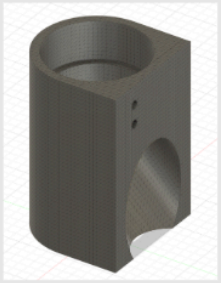
List of Parts
1. Main body x1
2. Servo Support x1
3. Motor Support x2
4. Wheel x2
5. Base x1

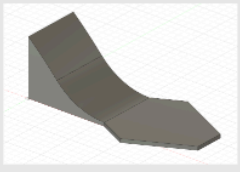
Purchasing

List of Parts
1. Servo Motor
2. Flywheel Motors
3. O-ring seals
4. PVC pipe

3D Printing

Parts Specifications



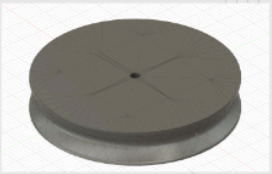


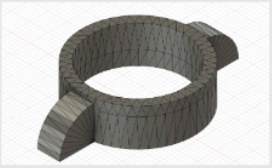
	Inner diameter (cm)	Max Width (cm)	Max Length (cm)	Max Height (cm)	Solid Volume (cm^3)	Mass (g)
Main body	5.00	6.00	6.00	9.00	156.373	164.19
Servo support	-	1.00	2.00	2.00	2.447	2.57
Motor support	2.50	3.00	4.80	1.00	2.861	3.00
Wheel	0.20	4.30	4.30	0.80	10.733	11.27
Base	-	12.00	23.00	7.80	322.73	338.87

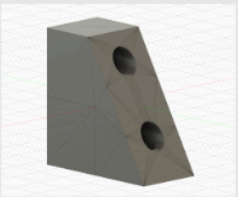
*All Widths, Lengths, and Heights are within the 3D-printable range as specified in the M4 document

(Density of ABS = 1.05 g/cm^3)

Printable Size L(mm) x B(mm) x H(mm)	TwoTrees
	300 x 300 x 330







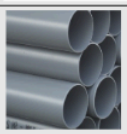
"Why are we using ABS as materials?"

Comparison between ABS and PLA:

Properties	ABS	PLA	Remarks
Tensile Strength	27 MPA	37 MPA	PLA stiffer
Density	1.05 g/cm^3	1.43 g/cm^3	ABS lighter
Flexural Modulus	2.1-7.6 GPa	4 GPa	ABS more flexible
Elongation	3.5% - 50%	6%	ABS more flexible
Glass Transition Temperature	105 C	60 C	ABS more heat resistant
Price (1kg, 1.75mm, black)	$USD21.99	$USD 22.99	ABS slightly cheaper

*The glass transition temperature (Tg) is the temperature at which an amorphous solid material transforms from a hard, brittle state to a softer, more malleable state that resembles a supercooled liquid

Choice of Components & Justifications

Component	Choice	Justification
Flywheel Motor	Kitronik 2546 DC Motor Low Inertia 1820 RPM 0.5-3V 	-Lower RPM but able to provide enough torque, shorter projectile (suitable for small map) -Low Cost: $2.59 each -Low Power Consumption
Servo Motor	Analog Micro Servo 9g (3V-6V) 	-Operating Angles: 180 degrees (able to dispense the ping pong balls) -Low Cost: $4.29 each -Low Power Consumption
O-ring Seal	LAPP KABEL 53102050 Skindicht®, NBR (Nitrile Butadiene Rubber), Black, M40 x 2 mm 	-Internal diameter: 36mm (able to fit on our wheel) -Rubber material (provide frictional force for the flywheels)
PVC pipe	PVC Pipe 2" (50MM) Class O x 5.8M 	-Internal diameter: 50mm (able to fit ping pong ball - diameter 40mm)

"Can our launcher hit the target?"

Launcher Calculation :

Motor used : Kitronik Low Inertia DC Motor (1820 RPM, torque = 3.03g·cm)

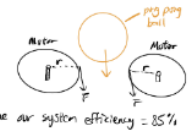
Calculating torque from angular velocity :

radius of wheel, $r = 2.15 \times 10^{-2}$ m

Rated torque, $\tau = 3.03$ g·cm $= 2.97 \times 10^{-3}$ Nm

$\tau = F \times r \Rightarrow F = \frac{2.97 \times 10^{-3}}{2.15 \times 10^{-2}} = 0.014$N

Total Force $= 0.014 \times 2 = 0.028$ N

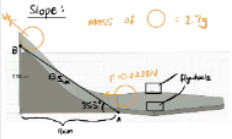


Assume our system efficiency = 85%

Force exerted on ping pong ball $= 0.028 \times 0.85$
$= 0.0238$ N

Slope :

mass of ○ = 2.7g



$F = ma$

$a = \frac{0.0238}{2.7 \times 10^{-3}} = 8.81\ ms^{-2}$

Assume that velocity of ball before contact with flywheels, $u = 0.2\ ms^{-1}$

" after contact at point A, $v_A = ?$

" time of contact, $t = 0.3$ s

*Assume constant acceleration

$v_A = u + at$
$= 0.2 + (8.81)(0.3)$
$= 2.843\ ms^{-1}$

Calculating velocity of ball at point B :

Material of slope : ABS, Coefficient of friction btwn ABS & P.P ball ≈ 0.2

Assuming the ping pong ball have contact with only half the slope along A to B

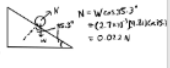
S_{AB} (curve) $= \frac{13.45 \sin 35.3°}{2} = 0.07$m

Energy$_A$ = Energy$_B$ + F_{loss}

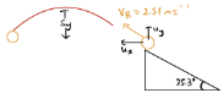
$\frac{1}{2}mv_A^2 + mgh_A = \frac{1}{2}mv_B^2 + mgh_B + M_{loss}N s_{AB}$

$\frac{1}{2}(2.7\times10^{-3})(2.843)^2 = \frac{1}{2}(2.7\times10^{-3})v_B^2 + (2.7\times10^{-3})(9.81)(2\times10^{-2}) + (0.2)(0.0238)(0.07)$

$v_B^2 = 6.32 \Rightarrow v_B = 2.51\ ms^{-1}$



$N = W\cos 35.3° + (2.7\times10^{-3})(9.81)\cos 35.3° = 0.023$ N



$u_x = 2.51\cos 35.3° = 2.05\ ms^{-1}$

$u_y = 2.51 \sin 35.3° = 1.45\ ms^{-1}$

Finding maximum height :

$v_y = 0$, $u_y = 1.45\ ms^{-1}$, $a_y = -9.81\ ms^{-2}$, $s_y = ?$

$v^2 = u^2 + 2as$
$0 = 1.45^2 + 2(-9.81)s_y$
$s_y = 0.107$ m
≈ 10.7 cm

Finding x-displacement :

$s_y = 0$, $u_y = 2.05\ ms^{-1}$, $a_y = -9.81\ ms^{-2}$, $t = ?$

$s = ut + \frac{1}{2}at^2$
$0 = 2.05t + \frac{1}{2}(-9.81)t^2$
$4.905t^2 - 2.05t = 0$
$t(4.905t - 2.05) = 0$
$t = 0$, $t = 0.42$ s

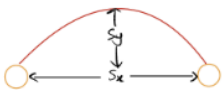
$s_x = ?$, $a_x = 0$, $t = 0.42$ s, $u_x = 2.05\ ms^{-1}$

$s = ut + \frac{1}{2}at^2$
$= (2.05)(0.42)$
$= 0.861$ m
≈ 86.1 cm

* All values are subjected to 10% deviations due to the large amount of assumptions made during calculation.

"Can our launcher hit the target?"

Conclusion :

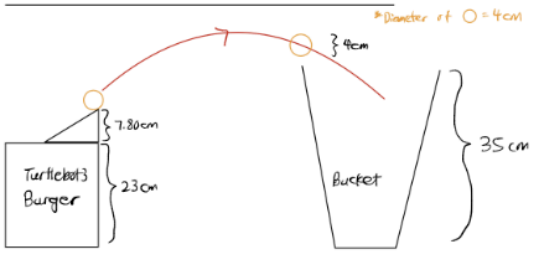


Projectile range :

$S_x \approx (77.5 - 94.7)$ cm

$S_y \approx (9.6 - 11.8)$ cm

can we launch the balls into the bucket ?



Height to achieve = 4 + 35 = 39cm

Height of ping pong ball projectile = 23 + 7.8 + 10.7 = 41.5cm

✓ Pass

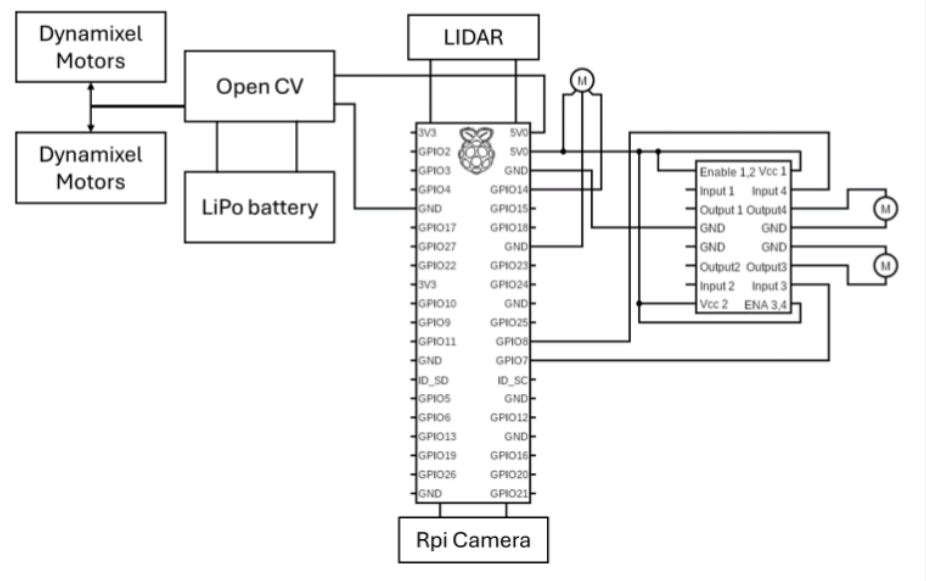
"What could possibly go wrong with this system?"

1. The **center of gravity** is excessively **high**, which could lead to **imbalance** in the robot, causing a potential risk of toppling when pushing through the elevator door.
2. **Inconsistent launching mechanism**
   a. The ping pong ball may not follow a straight path when launched, potentially deviating due to collisions and varying contact timing with the two flywheels.
   b. Due to the strong elastic characteristics of the ping pong ball, there is a possibility of rebound when it hits the slope.
   c. There is a risk that the ping pong ball may not be launched to a sufficient height.
   d. There is a potential for the ping pong ball to be launched too far or too near the intended target.
3. The dispenser (servo motor) might encounter **difficulties in dispensing the ping pong balls individually**, possibly resulting in collisions between them.

"What do we do if it goes wrong?"

1. Attach an additional waffle plate to the TurtleBot's backside and affix a ball caster for improved movement, effectively **increasing the base area** and lowering the center of gravity.
2. Possible solutions:
   a. **Remove one** of the **flywheels** and **replace it with a wall** to ensure the ping pong ball is launched upon contact with the remaining flywheel, while the wall guides its direction.
   b. **Angle the launcher** and decrease the slope angle to enhance the trajectory of the ping pong ball.
   c. **Utilize a single motor** instead of two, providing more power and voltage to enhance motor performance.
3. **Implement** a more intricate dispenser system involving **two servo motors** that alternate turns, ensuring the individual dispensing of ping pong balls.

Schematic Diagram



Power Budget (Total Power Consumption)

Component	Quantity	voltage (V)	current (A)	power consumption (W)
Servo motor	1	3.3	0.143	0.4719
TurtleBot Burger (Boot up)	1	11.1	0.784	8.7024
TurtleBot Burger (During Operation)	1	11.1	0.848	9.4128
Raspberry Pi Camera	1	2.8	0.038	0.1064
DC motor (Start up)	2	0.5	0.03	0.03
DC motor (During operation)	2	3	0.03	0.18
Total Power consumption				18.9035

Power Budget (Battery Sizing)

Battery sizing and durability

ASSUMPTIONS

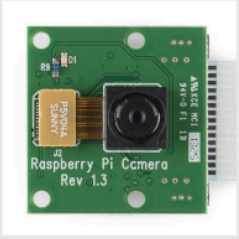
- voltage of lipo battery will decrease significantly when 90% capacity is used, thus total capacity used = 1800mAH x 0.9 =1620mAH
- System conversion rate = 80%
- Boot up time needed to be 3 seconds for turtlebot

Total power that can be used on turtlebot = Voltage x capacity x system conversion rate
= 11.1 x (1620 x 10^{-3}) x 0.8
= 14.3856 Wh

Operation time = 14.3856 / 18.9035
= 0.7597 hrs
= 45.58 min
= 2735.05 s

BATTERY SPECIFICATIONS
1800mAH, 5c
Max safe current = 9A
Voltage rating = 11.1V

Choice of Components & Justifications

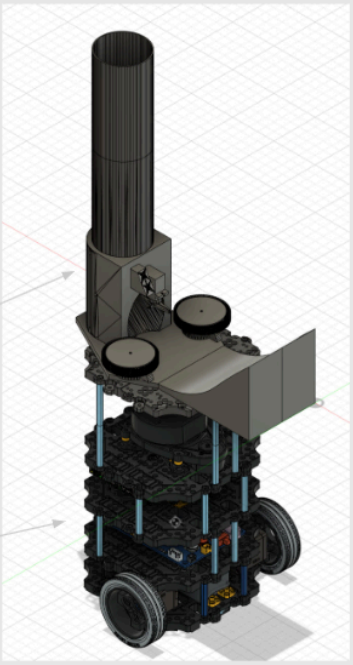
Component	Choice	Justification
Camera Sensor	Raspberry Pi Camera v1.3 	-High resolution of 5 megapixels, more than enough for line detection -RGB image sensor allows us to do colour detection -Uses the CSI interface on the Pi (instead of USB), which reduces CPU usage and improves frame rate -Low Cost: $3.62 each including case

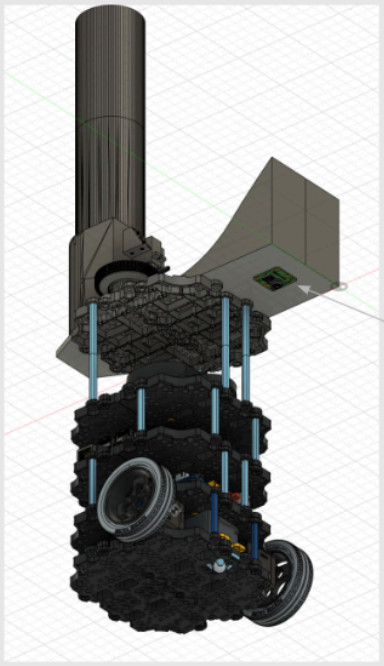
“What could possibly go wrong?”

- Back EMF (Electromotive Force): When a motor is turned off, the spinning rotor can generate a voltage in the opposite direction, causing a voltage spike. If the motor driver is not protected against this back EMF, it can lead to damage the L293D motor driver
- Motor might drawn too much current and lead to overheating and eventually damage. This can happen if the load on the motor is too high or if there is a mechanical blockage in the motor system.
- Color detection issues may arise with the RPi camera, potentially hindering its ability to accurately identify the color of the tape. This problem could be attributed to challenges in brightness conditions, leading to erratic behavior in the TurtleBot.

Ping Pong Ball Launcher

TurtleBot3 Burger





Raspberry Pi Camera

Centre of Gravity of Integrated System

EG2310 Parts weights

Part	Mass(g)	X-position(cm	Mass x X-position	Y-position(cm)	Mass x Y-position	Z-position(cm)	Mass x Z-position
Dynamixel motor (Right)	57	5	285	-2	-114	2.5	142.5
Dynamixel motor (Left)	56	-5	-280	-2	-112	2.5	140
RPI 4	45	0	0	3	135	11	495
OpenCR1.0	62	0	0	0	0	6	372
Lidar	129	0	0	0	0	17.5	2257.5
Battery	139	0	0	0	0	2.5	347.5
PCB support x 4 (2nd layer)	9.5	0	0	0	0	5	47.5
PCB support x 4 (3rd layer)	9.5	0	0	3	28.5	10	95
PCB support x 4 (4th layer)	9.5	0	0	0	0	15	142.5
Wheel + tyre (Right)	42	8	336	-3.5	-147	2	84
Wheel + tyre (Left)	42	-8	-336	-3.5	-147	2	84
Waffles plates (1st layer)	35.2	0	0	0	0	0.4	14.08
Waffles plates (2nd layer)	35.2	0	0	0	0	4	140.8
Waffles plates (3rd layer)	35.2	0	0	0	0	8.9	313.28
Waffles plates (4th layer)	35.2	0	0	0	0	13.7	482.24
Support M3x45 (1st to 2nd layer)	6.6	2.5	16.5	6.5	42.9	2	13.2
Support M3x45 (1st to 2nd layer)	6.6	2.5	16.5	-6.5	-42.9	2	13.2
Support M3x45 (1st to 2nd layer)	6.6	2.5	16.5	6.5	42.9	2	13.2
Support M3x45 (1st to 2nd layer)	6.6	2.5	16.5	6.5	42.9	2	13.2
Support M3x45 (2nd to 3rd layer)	6.6	6.4	42.24	2.4	15.84	6.9	45.54
Support M3x45 (2nd to 3rd layer)	6.6	6.4	42.24	2.4	15.84	6.9	45.54
Support M3x45 (2nd to 3rd layer)	6.6	1.25	8.25	6.5	42.9	6.9	45.54
Support M3x45 (2nd to 3rd layer)	6.6	1.25	8.25	-6.5	-42.9	6.9	45.54
Support M3x45 (3rd to 4th layer)	6.6	2.5	16.5	6.5	42.9	11.8	77.88
Support M3x45 (3rd to 4th layer)	6.6	2.5	16.5	-6.5	-42.9	11.8	77.88
Support M3x45 (3rd to 4th layer)	6.6	6.5	42.9	-2.5	-16.5	11.8	77.88
Support M3x45 (3rd to 4th layer)	6.6	2.5	16.5	6.5	42.9	11.8	77.88
Support M3x45 (3rd to 4th layer)	6.6	2.5	16.5	6.5	42.9	11.8	77.88
Bracket 1	3.75	0.9	3.375	-5.5	-20.625	1.4	5.25
Bracket 2	3.75	-0.9	-3.375	-5.5	-20.625	1.4	5.25
Bracket 3	3.75	2.5	9.375	0.7	2.625	1.6	6
Bracket 4	3.75	-2.5	-9.375	0.7	2.625	1.6	6
Support M3x45 (4th to 5th layer)	6.6	-6.4	-42.24	2.4	15.84	17	112.2
Support M3x45 (4th to 5th layer)	6.6	6.4	42.24	2.4	15.84	17	112.2
Support M3x45 (4th to 5th layer)	6.6	-1.25	-8.25	-6.5	-42.9	17	112.2
Support M3x45 (4th to 5th layer)	6.6	1.25	8.25	-6.5	-42.9	17	112.2
Waffles plates (5th layer)	35.2	0	0	0	0	22.3	784.96
Payload Base	338.9	0	0	-6.1	-2067.29	23.2	7862.48
Motor + Motor Support	66	0	0	0.7	46.2	24.7	1630.2
Main Body + Servo Support + Servo	175.76	0	0	6.4	1124.864	26.9	4727.944
PVC pipe + 5 Ping pong balls	66.81	0	0	6.4	427.584	42.6	2846.106
Total	1557.77		5		-1002.382		24183.13
Centre of gravity		X-position(cm	0.003209716	Y-position(cm)	-0.6434724	Z-position(cm)	15.52419805

Conclusion: (Centre of Gravity of Turtlebot3 Burger)
(x,y,z) = (0, -0.5, 6.9) - in cm for Turtlebot3 Burger only
(x,y,z) = (0, -0.6, 15.5) - in cm for Turtlebot3 Burger + Payload

Total mass of the system = approx 1557.77g
Note: Turtlebot3 Burger = approx 1000g

Bill of Materials

No.	Type of Component	Model Name	Quantity	Unit Price (S$)	Total Cost (S$)
1	Flywheel Motor	Kitronik 2546	2	2.59	5.18
2	O Ring Seal	Lapp Kabel 53102050	1	8.31	8.31
3	Motor Driver	ST L293D	2	0.50	1
4	Servo Motor	TS90A Micro Servo	1	4.29	4.29
5	Camera	Raspberry Pi Camera v1.3 w/ Case	2	3.62	7.24
6	PVC Pipe	PVC Pipe 2" (50MM) Class O x 5.8M	1	7.73	7.73
7	3D Printing	-	10	5.00 / hour	50.00
8	Colored Tape	-	5	0.46	2.30
					85.05

Prototyping Progress

Software:
- Writing code for OpenCV line detection, simulation with laptop webcam
- Using inputs to control the actuators i.e. motors and servo

Elec:
- Selected the camera sensor, motor and servo to be used for software and mechanical aspects

Mech:
- Built CAD model of prototype
- Determined how to fabricate it

Upcoming challenges (general)

Finish build phase **in time** (2 weeks)
- Wait for sourcing and fabrication of parts.
- Group members facing quizzes and midterms.

Operating the **fabrication tools** e.g. 3D printer
- We will need someone from teaching department to advise us on printing our CAD files at a reasonable cost and quality.

Modifying ROS2 packages to fit our project purpose
- We hope to **create new Python scripts** in the r2_autonav package to run our modified code
- We anticipate the need for someone to clarify on this topic