



CDE2310
Fundamentals of System Design
G2 Report
AY 2024/2025, SEMESTER 2
Group 2

Group Members	Matric Number
Akash Mukherjee	A0288107Y
Jansen Ken Pegrasio	A0305755B
Lim Ji Yong	A0282723A
Shah Jay	A0288344U
Vanchinathan Sindhu Yazhini	A0309545Y

Table of Contents

Chapter 1: Problem Definition	4
1.1 Stakeholder requirements and project deliverables	4
Chapter 2: Literature Review	5
2.1 Navigation	5
2.1.1 Maze Traversal Algorithms	5
2.1.2 Path-Finding Algorithms	5
2.2. Payload Design	5
2.2.1 Launch Mechanisms	5
2.2.2 Loading Mechanisms	7
2.3 Heat Signature Sensors	9
Chapter 3: Conceptual Design	10
3.1 Software System	10
3.2 Electrical System	10
3.3 Mechanical System	11
Chapter 4: BOGAT	12
4.1 Maze Traversal Algorithms	12
4.2 Path-Finding Algorithms	12
4.3 Heat Sensor Selection	13
4.4 Payload Design Decision	13
Chapter 5: Preliminary Design	14
Chapter 6: System Technical Specifications and User manual	15
6.1 Turtlebot Specifications	15
6.2 Assumptions and Constraints	15
6.3 Acceptable Defect Log	16
6.4 Factory Acceptance Test	16
6.5 Maintenance and Part Replacement Log	17
Chapter 7 : Mechanical Subsystem	18
7.1 Prototyping & Testing	18
7.2 Critical System Design	22
7.2.1 Bill of Materials	22
7.2.2 Final Payload Design	22
The Ramp	22
The Column	23
The Linear Actuator (Servo + Pinion + Rack)	23
The ‘Survivor Detector’	24
The ‘Flare Firing’ Mechanism	24
The “KEN” Shield	25
7.3 Assembly Instructions	25
Layer 1 Assembly Modifications	26
Layer 2 Assembly Modifications	27
Layer 3 Assembly Modifications	30
Layer 4 Assembly Modifications	33
FINAL SYSTEM AFTER ASSEMBLY	36

Chapter 8: Electrical Subsystem	38
8.1 Electrical Components	38
8.1.1 Heat sensors	38
8.1.2 Motors and Motor-driver:	40
8.1.3 SG90 Servo motor:	41
8.2 Critical Design	42
8.2.1 System Finances (Bill of Materials)	42
8.2.2 Electrical Schematic Diagram	43
8.3 Energy Consumption	44
Chapter 9 : Software Subsystem	45
9.1 Autonomous Navigation - attributes class diagram	45
9.2 ROS2 Implementation (RQT)	45
9.3 Algorithms and Control Strategies	46
9.4 Embedded Firmware Structure	46
9.5 Navigation Algorithm	47
9.5.1 Introduction	47
9.5.2 Localization	47
9.5.3 Target Finding	47
9.5.3.1 Initial Approach: Greedy Based on Total Distance	47
9.5.3.2 Revised Approach: Greedy Based on Minimum Distance	48
9.5.3.3 Final Approach: Multi-source Dijkstra	48
9.5.3.4 Minor Improvement	48
9.5.4 Path Finding	49
9.5.4.1 Initial Approach: Plain A Search Algorithm*	49
9.5.4.2 Final Approach: A Search Algorithm with Wall Penalty*	49
9.5.5 Movement Execution	50
9.5.5.1 Initial Approach: Point-to-Point Movement Strategy	50
9.5.5.2 Final Approach: Cluster-to-cluster Movement Strategy	50
9.5.6 Heat Pursuit Movement	50
9.5.7 Map Update Delay	50
9.5.8 Points Storing Mechanism	50
9.5.9 Parameter Descriptions	51
9.5.10 Appendix	53-65
Chapter 10: Safety Protocol	66
10.1 Risks and Mitigation Strategies	66
Chapter 11: Troubleshooting	67
11.1 Mechanical Subsystem	67
11.2 Electrical Subsystem	67
11.3 Software Subsystem	68
Chapter 12: Future Scope of Expansion	69
12.1 Mechanical Aspects	69
12.2 Electrical Aspects	69
12.3 Software Aspects	69
References	71
Datasheets	72

Chapter 1: Problem Definition

The “Earthquake Problem” requires a robot to autonomously navigate through a randomized 3.5 m x 3.5 m maze. The primary mission is to detect two survivors (simulated by incandescent lamps) by detecting heat signals and fire 3 flares (ping-pong balls) per survivor to alert Search and Rescue (S&R) teams for assistance in evacuation.

A bonus mission includes the robot navigating to a ramp (at a known location within the maze) to rescue a third survivor. The entire mission is confined to a maximum of 25 minutes. This project mimics real-world search and rescue scenarios, where the robot assists teams with immediate assistance in an Earthquake Disaster Zone. Further details on the software, electrical, and mechanical design, and stakeholder requirements are elaborated below.

1.1 Stakeholder requirements and project deliverables

Stakeholder requirements	Project deliverables	System Requirements
The robot has to autonomously navigate through the maze and develop a Simultaneous Localization And Mapping (SLAM) map of the area.	The robot must cross the start line, navigate past obstacles, complete the mission and reach the endpoint. The robot has to explore the maze and provide a map using SLAM.	<ul style="list-style-type: none"> • Autonomous TurtleBot operation • 2D LiDAR Sensor
The robot should use heat signals to locate survivors (incandescent lamps).	The robot must randomly transverse through the maze and find bulbs located in unknown regions, and should then send signals by firing ping-pong balls.	Thermal Sensor
The robot must fire a predetermined number of flares in a fixed interval.	The robot must safely carry 5 ping pong balls and when the signals are received we need to shoot balls upwards with the same time interval between them.	Ball Feeding and Launching Mechanisms

Chapter 2: Literature Review

2.1 Navigation

2.1.1 Maze Traversal Algorithms

Two potential ways to navigate the maze are Breadth-First Search (BFS) and Depth-First Search (DFS). In general, BFS works by prioritizing the nodes nearest to the source node (layer by layer), whereas DFS goes to one end and then starts again to traverse the others (path by path).

For our project, we have decided to use frontier-based exploration, a specific implementation technique of BFS, to navigate the unknown maze autonomously. Utilizing SLAM with LiDAR, the robot detects frontiers (boundaries between explored and unexplored areas) and selects the nearest one to explore. It updates the map while moving toward the frontier, ensuring complete exploration without prior maze knowledge.

2.1.2 Path-Finding Algorithms

Two highly efficient ways to find the quickest path from one point to another (known) point are using the A* Search Algorithm and Dijkstra's Algorithm.

- **A* Algorithm**: This algorithm uses a heuristic approach to find the shortest path efficiently. A* will help TurtleBot3 navigate the maze by prioritizing paths with the lowest combined travel cost and estimated distance to the target. The algorithm will use an occupancy grid map generated by SLAM, allowing the robot to make informed decisions while avoiding obstacles.
- **Dijkstra's Algorithm**: This algorithm finds the shortest path in an unweighted graph but is computationally expensive. It can be used as a fallback method if no heuristic data is available, ensuring that TurtleBot3 always finds a path, even in unknown environments. Since it evaluates all possible paths, it may be helpful during the initial mapping phase but is slower than A* in a large maze.

2.2. Payload Design

2.2.1 Launch Mechanisms

- a. **Flywheels**: A possible design is to use 2 flywheels rotating at high speeds to launch the balls into the air. Two flywheels will be spinning in opposing directions, and

positioned on either side of a feed tube. When a ball is fed between them, the wheels impart angular momentum, propelling the ball upward.

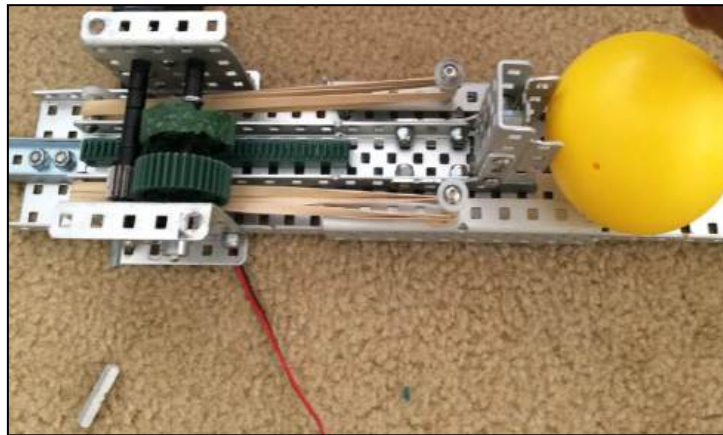


Fig 1: Flywheel Mechanism

- b. Solenoid Plunger: A third launching mechanism is to use a solenoid plunger. A solenoid is an electromagnet with a long, helical coil of wire wound around a magnetic core. When current passes through the solenoid, the metal core or ‘plunger’ is attracted by the electromagnet, compressing the spring within it. When the circuit is opened, or no current is passed through, the electromagnet loses its ability to attract. The spring’s elastic potential energy is converted to kinetic energy as it launches the plunger back to its original position. The impulse generated from this launch can be used to fire the ping-pong balls into the air.



Fig 2: Solenoid Plunger

2.2.2 Loading Mechanisms

- a. The 'Revolver': One of the initial designs included a disc with 5 slots for ping-pong balls to horizontally rotate and 'load' into the firing slot. This is similar to a traditional revolver gun consisting of 6 chambers for 6 bullets with a rotating cylinder to load the bullets for firing. Once a ball is loaded, launch mechanisms such as springs or flywheels can fire the flare. The illustration shows flywheels implemented with the revolver.

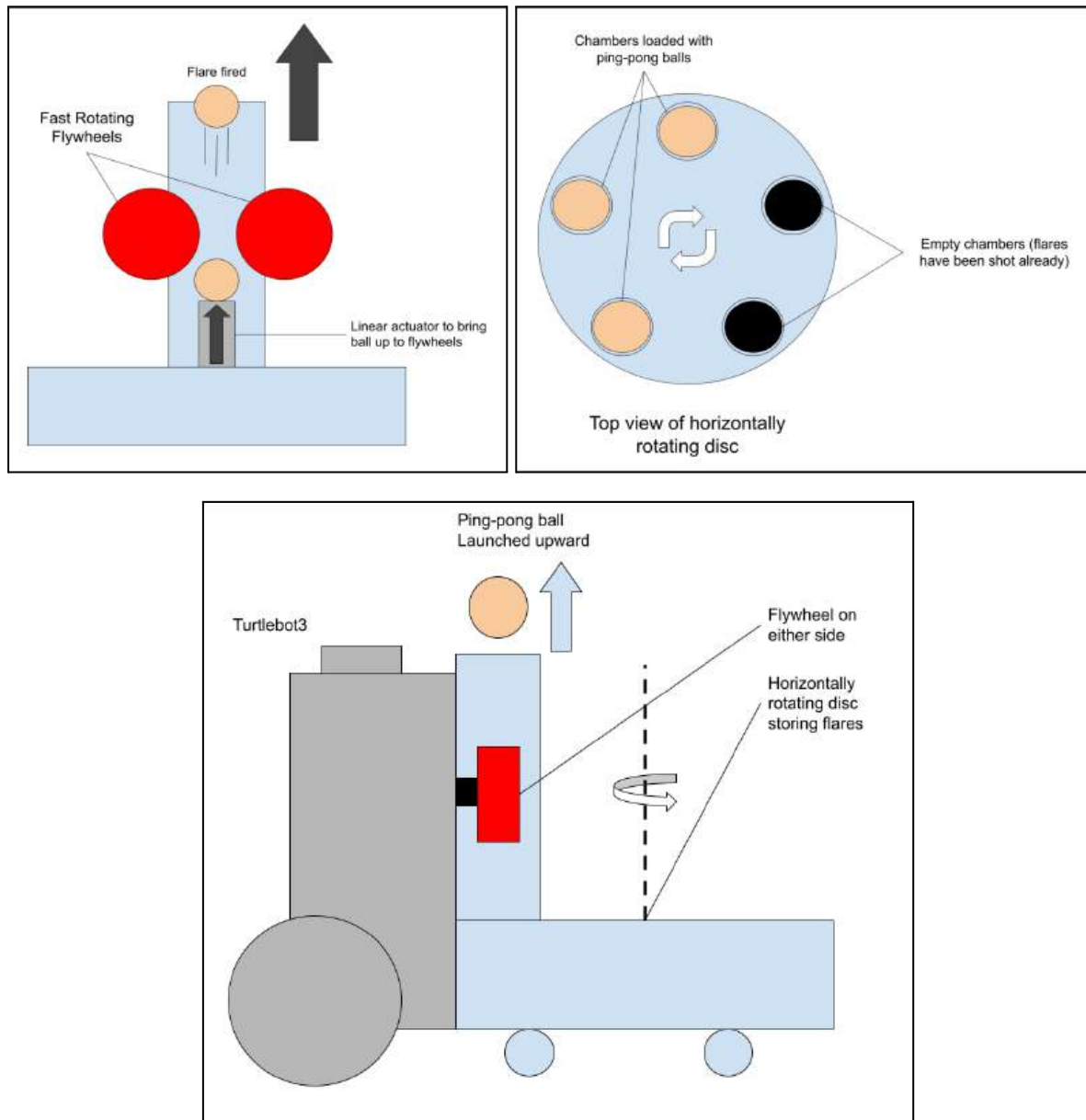


Fig 3a, 3b, 3c: Revolver Diagrams

- b. Impulse Launcher: The ball loading mechanism relies on a simple yet effective gravity-fed inclined ramp with two vertical guiding columns slightly larger than the diameter of the ping pong balls. These columns ensure that the balls remain aligned while rolling down the ramp. At the bottom, an additional incline directs the balls to the bottom right corner, positioning them precisely for launch. This method eliminates the need for external actuators or power sources for ball feeding, as each ball naturally falls into place once the previous one is launched.

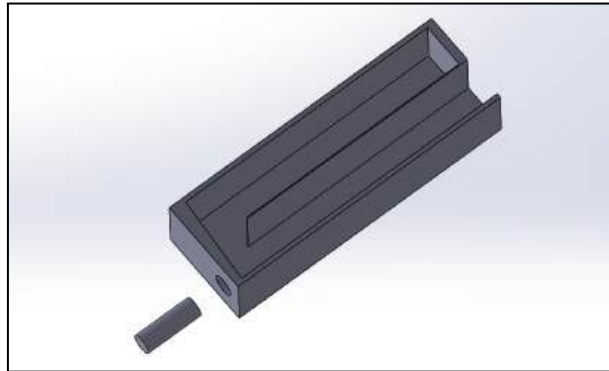


Fig 4: Impulse Launcher Visualization

- c. The Ramp: The primary motivation behind designing the ramp was to loop it around the TurtleBot, utilizing the space surrounding it to store all nine balls while keeping the overall size of the TurtleBot as compact as possible. This configuration ensured that neither the ramp, nor the balls, nor the launching mechanism obstructed the LiDAR in any way. Additionally, by routing the ramp around the TurtleBot, the design maintained the center of gravity close to that of the original, unmodified TurtleBot.

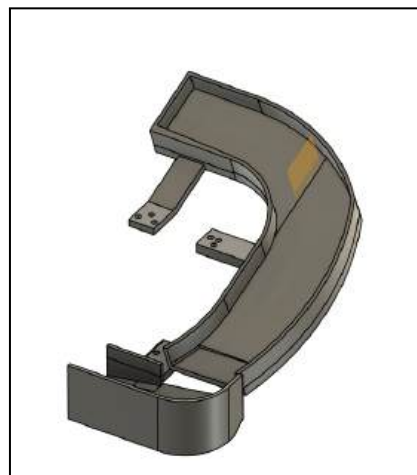


Fig 5: Ramp

2.3 Heat Signature Sensors

As per mission requirements, the TurtleBot must accurately detect survivors using heat signals. This necessitates equipping the robot with precise heat sensors capable of locating incandescent lamps. Key characteristics of the sensors include compatibility with existing python libraries while running it on RPi , sensitivity to heat signals and angular range of frontal scanning, and the price. Based on these criteria, the team identified three potential heat sensors:

- **MLX90640:** A thermal imager array temperature sensor module that can produce a heat map, provided with 55° x 35°. It communicates via I²C, making it compatible with Python. It can detect heat through thin acrylic barriers, though sensitivity may decrease with thicker materials. The price is around \$80 to \$100.
- **AMG8833:** A Grid-Eye thermal sensor that could detect heat sources and generate heat maps. IR thermal sensor array with a 60° field of view. It uses an I²C protocol and is compatible with Python through libraries like Adafruit's CircuitPython. The cost is approximately \$35 to \$50.

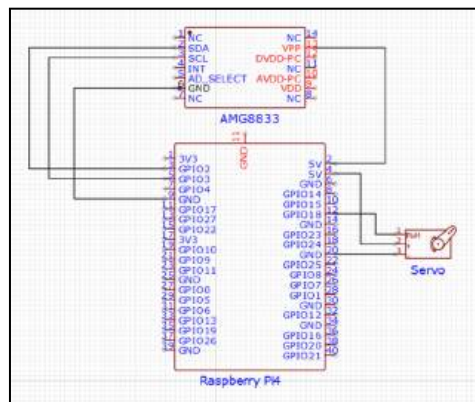


Fig 6: Circuit diagram (Without payload)

- **TMP36:** An analogue temperature sensor operating with a range of -40°C to 125°C is suitable for ambient temperature readings. An analog contact-based temperature sensor that measures temperature through direct thermal conduction.

Chapter 3: Conceptual Design

3.1 Software System

Instead of employing a combined technique of maze mapping followed by pathfinding, our approach is to concurrently explore the area and do the mission. The Raspberry Pi will process the incoming data from the HLS-LFCD2 LIDAR module while ROS will continuously store the map as a 2D grid. Our team chose this strategy after meticulously considering the context of the mission: if survivors are found while exploring the area, the robot will directly approach survivor(s) and fire flares to immediately alert search and rescue teams to assist. It is more reasonable to do this rather than completely mapping the area first before rescuing them.

In the bonus mission, however, the ramp's location is fixed and known for all mazes. In such a case, we can use the saved map and use the A* Search algorithm to find the shortest path to the ramp. The turtlebot will move according to the electrical signals that Raspberry Pi sends to the DYNAMIXEL motors.

3.2 Electrical System

After navigating through the maze using LiDAR-based localisation, there will be a thermal sensor connected to the turtlebot Raspberry Pi to sense the heat signals coming from the incandescent bulb. The data will be processed as a heat map by the RPi and it will send instructions to navigate and activate the payload to shoot two ping-pong balls each.

We will be using the AMG8833 as our thermal sensor and it will be connected directly to the RPi for both power and signals.

GPIO Pin-in and Pin-out:

<u>AMG8833 Pin</u>	<u>Raspberry Pi GPIO</u>
VCC	3.3V Pin1
GND	GND Pin6
SDA	GPIO2 Pin3
SCL	GPIO3 Pin5

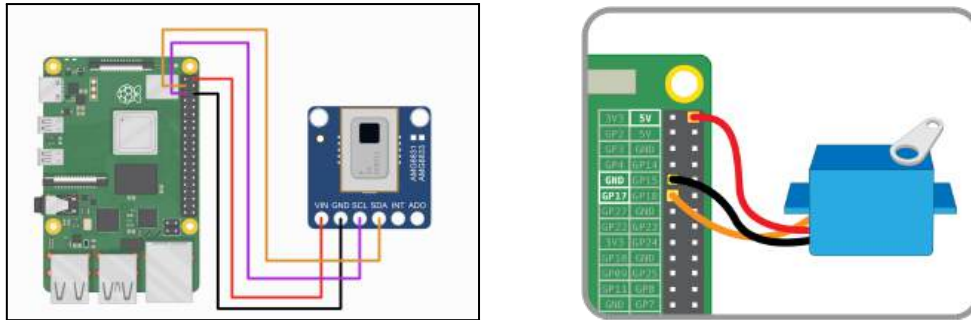


Fig 7a, 7b: RPi connection with AMG8833 and Servo motor

The ball launching mechanism will employ flywheels - for which we will be using DC motors. They will be programmed to spin continuously once the robot is close to the identified heat source. Motor drivers for the respective motors (DC, and stepper) will be integrated.

Lastly, we will be using servo motors with an attachment to push the ball from the feeding mechanism to the feed tube to push the ball to the middle of the two flywheels. The servo motors will be connected directly to the RPi for both power and signals.

3.3 Mechanical System

To launch flares, a flywheel mechanism will be employed. The ball will be stored in a sloped chamber that can roll into the base of the vertical feed tube. A linear actuator will push the ball, one at a time, into the feed tube at the middle of the two flywheels. The feed tube's width will be just slightly wider than the ball's diameter so that consecutive balls will not be caught stuck during actuation.

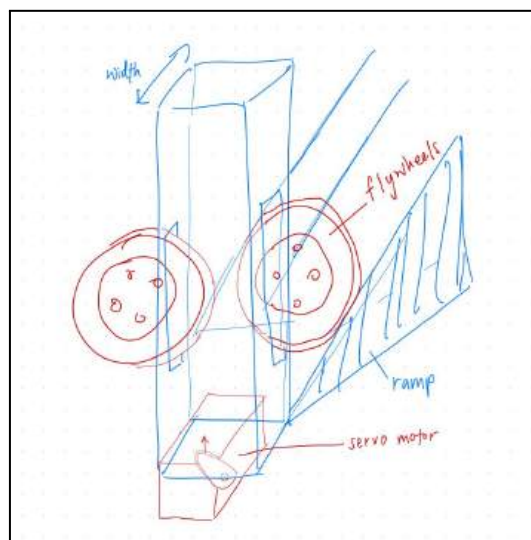


Fig 8: Preliminary Vision of the Payload Mechanism

Chapter 4: BOGAT

The BOGAT section outlines various design options that we explored throughout the project. It contrasts between different ideas and evaluates its advantages and disadvantages and finally, we present our design choice with the rationale behind why we chose it.

4.1 Maze Traversal Algorithms

	Breadth First Search (BFS) – Frontier-based Exploration	Depth First Search (DFS)
Pros	It stores all paths in the maze. It is guaranteed that it will immediately move towards the closest survivor.	Lower memory usage as it only keeps track of the current path.
Cons	It takes high memory consumption and it is more complex due to queue-based processing.	It exhibits unexpected behaviour when traversing loops. It might reach survivors later due to deep exploration in one path.
Final Decision	We decided to use the Breadth First Search (BFS), Frontier-based Exploration, considering the following reasons: <ol style="list-style-type: none"> 1. Maze can contain many looped paths, which is the weakness of DFS. 2. We want to store all possible paths in the maze, which is only achievable when we use BFS 	

4.2 Path-Finding Algorithms

	A* Search Algorithm	Dijkstra's Algorithm
Pros	It uses a heuristic function which makes it faster and more efficient. It finds the shortest path and traverses using it.	It tries to guarantee the shortest path by exploring all possible routes. It works well in a stable environment because it does not have heuristic functions. Suitable for rather static maps.
Cons	It is risky because use of poor heuristic function can lead to suboptimal paths. It requires a bit more computational power than the simple algorithms like BFS or DFS.	It is computationally expensive and slower than A* as it does not prioritize goal-directed movement. It is not ideal for a dynamic environment.
Final Decision	We decided to combine these techniques according to the condition we face. If we haven't explored a path to the ramp, we will use Dijkstra's algorithm to find the shortest path. Otherwise, we will use A* search algorithm considering the heuristic function, which is very efficient.	

4.3 Heat Sensor Selection

	MLX90640	AMG8833	TMP36
Pros	It gives high-resolution thermal imaging, has a wide field of view and is useful for detecting heat at varying angles.	It gives a decent resolution for detecting heat zones, is lightweight and suitable for turtlebot implementation. It has I2C which facilitates interfacing with RPi. It is far less expensive than other sensors.	It is simple and cost-effective, has low power consumption, it is rather compact and lightweight.
Cons	It is expensive for a heat sensor, and requires more computational power and software implementation time.	It gives lower resolution and a limited detection range. It gives slightly less detailed thermal imaging.	It only detects temperature at a single point of contact, rather than over an area. It requires an ADC to output digital signals.
Final Decision	We decided to choose the AMG8833 thermal sensor as it provides an 8×8 thermal grid, and is capable of detecting heat signals even behind acrylic walls, it has a simple integration with RPi via I2C making implementation easy. It is also much less costly than other options.		

4.4 Payload Design Decision

	Flywheel Launching	Impulse Launcher
Pros	High power launching is achievable. No parts moving around, so less possible sources to troubleshoot.	Less power is required. Potentially takes up less space on the robot (no need for flywheels, motor driver and encoder).
Cons	A higher amount of power will be required to rotate the motor quickly and continuously, we will need to buy a power source.	Complicated launch mechanism. To release an elastic impulse, the servo must be attached to a complicated mechanism. Developing it will require lead time for each of the many iterations.
Final Decision	We will go with the flywheel launcher as it is more mechanically reliable than the impulse mechanism.	

Chapter 5: Preliminary Design

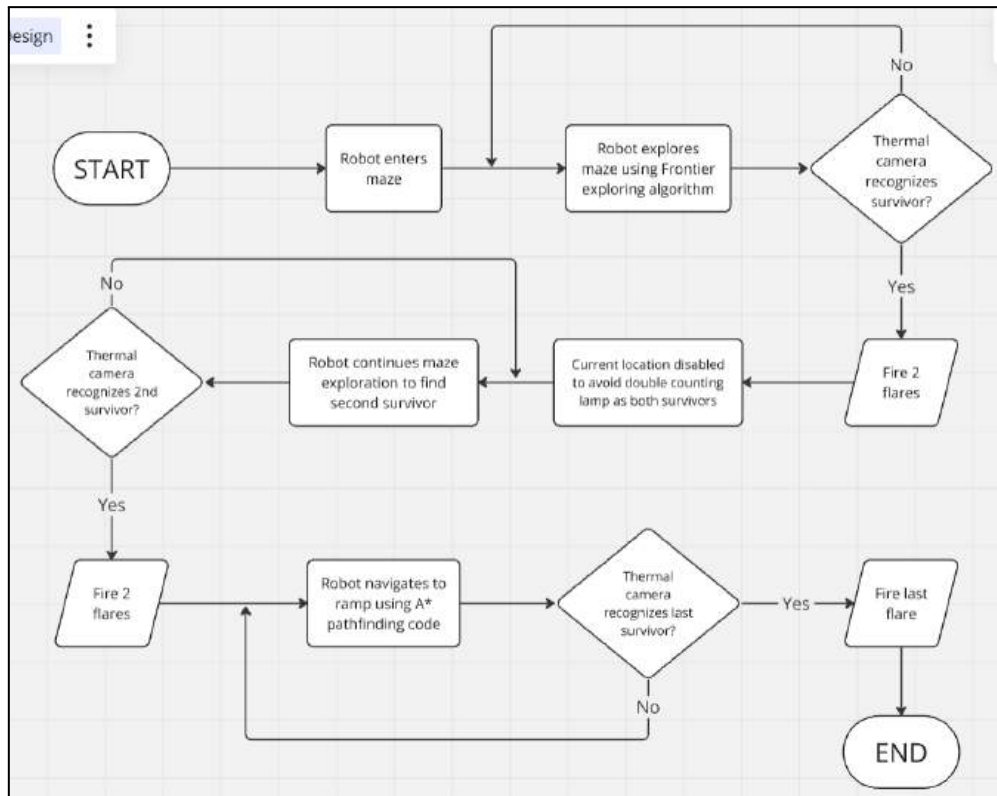


Fig 9: Preliminary Operational Mind Map

Chapter 6: System Technical Specifications and User manual

6.1 Turtlebot Specifications

List	Specifications
Weight (kg)	1504 g
Maximum translational velocity	0.22 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)
Dimensions	268 mm x 254 mm x 205 mm
Payload capacity	9 ping pong balls, 3 per survivor
Power Supply	LiPo Battery (11.1V 1,800mAh)
Expected Operating Time	45 mins
Sensors onboard	1x LDS-01 2D LiDAR Sensor 1x AMG8833 Thermal Camera
Electronics (excluding Dynamixel motors from Turtlebot 3)	1x SG90 Servo motor 1x L298N Motor Driver 2x RS PRO Geared 15 V DC Motor
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
SBC	Raspberry Pi 4B
Battery	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C

6.2 Assumptions and Constraints

Constraints	Assumption
A time limit of 25 minutes is only given to complete the whole mission challenge.	Environmental factors (temperature, humidity, lighting, etc.) are kept constant.
The heat sensors should locate at least 2 heat signals and fire a total of 4 shots.	The maze layout remains unchanged between initial testing and final execution
There should be 0 human interaction with the robot, it should navigate independently and be autonomous.	Every part of the maze is accessible and the heat zones are stationary
Trying to navigate inside the maze you should not use the line-following mechanism.	The battery life is sufficient for the entire mission duration

6.3 Acceptable Defect Log

Defect Description	Defect Classification			Acceptance Criteria
	Critical	Major	Minor	
Robot may vibrate while navigating			X	Balls do not fall out of ramp
Silicone paste on flywheels peels off		X		Ball reaches minimum height when launched
Gap between ramp parts			X	Ball smoothly rolls along ramp parts into column
Temperature varies when identifying survivors.		X		All survivors identified and launch mechanism triggered
Flywheels may not have the same speed			X	Flares launched reach over wall height.
Rack and pinion teeth have poor grip			X	Rack pushes ball upward till flywheels & moves down to load the next ball

6.4 Factory Acceptance Test

Component	Description	Testing
Structural Stability	Components mounted correctly and not loose	Perform shake test
OpenCR	OpenCR powered by the LiPo battery.	<ul style="list-style-type: none"> Green LED lights up Boot up tune plays
RPi	RPi turns on	<ul style="list-style-type: none"> Red light turns on Green light flashes
Motor and flywheel	Flywheels rotate in desired directions	Balls pushed from the middle shoots upwards
Servo, Rack & Pinion	Servo turns pinion in correct gear number	Rack movement range sufficient to load new ball & lifting it to flywheels
sensor	Sensor calibrated to locate survivors.	<ul style="list-style-type: none"> Heat sensor outputs max temperature for each column measured Temperatures measured correlate to ground truth values

	LIDAR or heat sensor should have no obstruction	Make sure that the cameras in the sensors do not have anything placed in front of them.
Motor Driver	The motor driver is powered by OpenCR	Red lights on motor driver are blinking after robot is powered on

6.5 Maintenance and Part Replacement Log

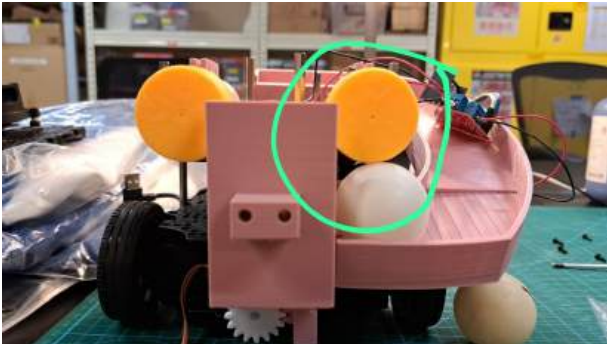
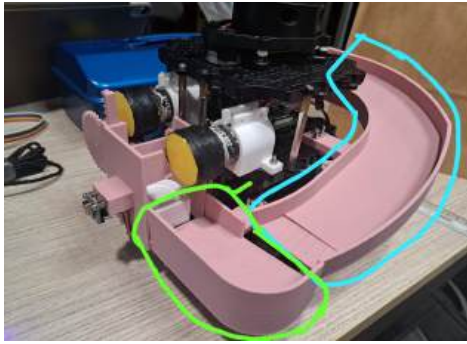
Defect Date	Description	Rectification	Close Date
31/3/2025	L298N Motor driver only powers one pair of output terminals	Loaned a new L298N from the lab	1/4/2025
1/4/2025	Servo motor configuration changed, stopped working.	Replace with backup servo from the kit	2/4/2025
7/4/2025	L298N Motor driver not working, we connected 12 V from power supply hence it got over-powered and it burned.	Replace it with another motor driver from the lab	8/4/2025
8/4/2025	OpenCR reboots after every 1-2 minutes	Power draw from 12V DC motors was too high. Decreased their RPM.	10/4/2025
10/4/2025	RPi 1 cannot boot up. Green light flickers erratically.	Replaced our RPi with the RPi 2	10/4/2025

Chapter 7 : Mechanical Subsystem

7.1 Prototyping & Testing

The chosen payload design was a ramp around the Turtlebot that stores and feeds ping-pong balls automatically (by gravity and sloping shape) into a column, where the balls are then moved up one at a time by a linear actuator (consisting of the servo motor, pinion, and rack) up to the flywheels. During the prototyping and testing phase, the team designed digital models of various parts - such as the ramp, column, and motor mounts - and integrated them into a single TurtleBot CAD assembly file. This was to ensure no interference existed between parts, as well as to determine the mounting points for the components. The parts were then 3D printed and assembled onto the TurtleBot itself to verify the expected operation of mechanisms.

Below are some of the defects the team had found during testing, and the adjustments that were made accordingly to overcome those:

Defects observed during Prototyping & Testing	Solutions Implemented
<p>The payload ramp was initially designed as one continuous piece, with ping-pong balls rolling down from the back to the column at the front. However, the balls ‘waiting’ on the ramp were in contact with the flywheels. The fast rotation of flywheels would cause the balls to vibrate heavily and get ‘knocked off’ the ramp.</p>  <p><i>Fig 10: Ball interference with flywheels</i></p>	<p>To avoid this risk, the ramp was separated into two parts - one that had the same profile as the earlier ramp, but only along the back and side of the robot. The second ramp piece acted as a linkway from the first ramp to the column, and sloped further down so that the balls would rest well below the flywheels.</p> 

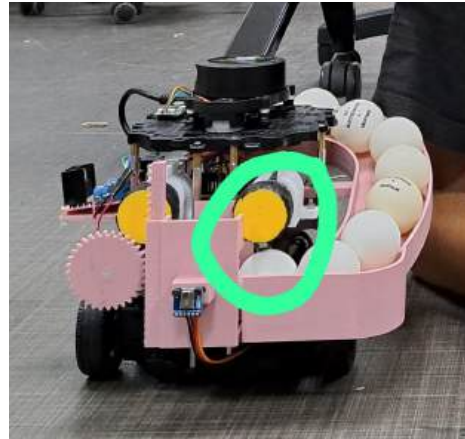


Fig 11a, 11b: New ramp design providing clearance

Another feature in the initial concept design was the linear actuator - the servo, pinion, and rack - being located vertically below the column and flywheel shooting point (as marked in blue in the image of the initial prototype below). This led to a key dilemma - a long rack was required to push the ball from the column level (Layer 2) up to the flywheels (Layer 3). However, since the servo was to be mounted at a low height (between Layers 1 and 2), the rack's length was limited by the ground clearance. Additionally, the rack was inefficient in pushing the ball vertically upwards to between the flywheels, while also preventing the next balls from rolling away.

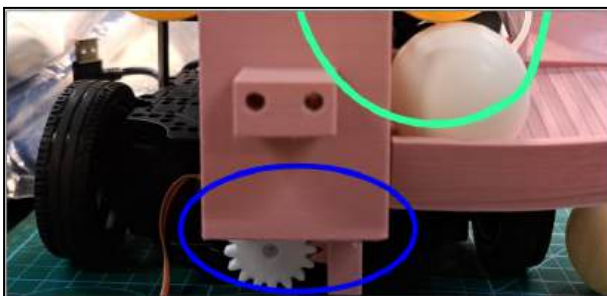
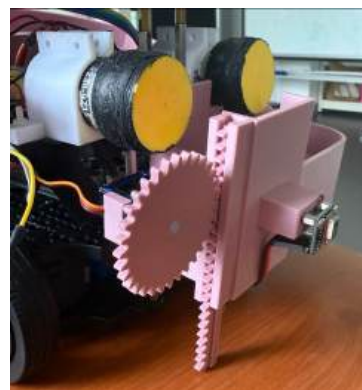
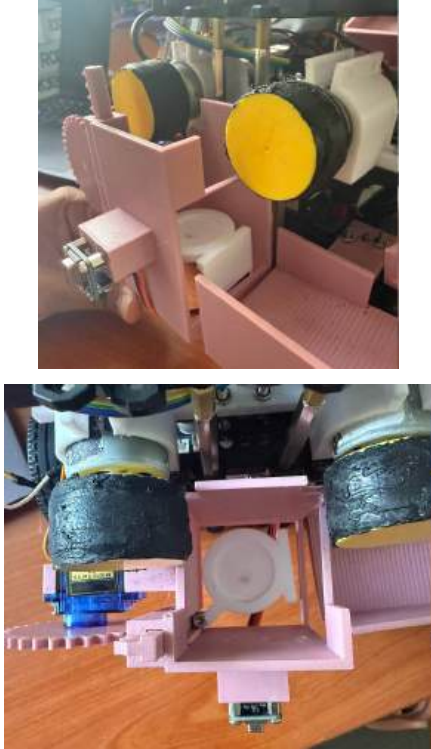


Fig 12: Initial location of the rack and pinion (below ball)

A simple design change resolved the corresponding issue. The linear actuator was shifted to the side of the column from vertically below it. This provided more flexibility in designing a long rack that would fit the requirements and constraints of the system. Additionally, a ball tray with a 'blocker' was designed, that would hold and transport the ball to the flywheels smoothly while blocking the next balls.



	 <p><i>Fig 13a, 13b, 13c: relocation of linear actuator with new ball tray design</i></p>
<p>The design of the base TurtleBot 3 Burger did not provide enough space for payload parts to be mounted.</p>	<p>Various parts such as the Raspberry Pi, the USB2LDS, and the support rods between Layers, were relocated slightly to account for mounting of payload components. Detailed mechanical assembly instructions can be found in the Assembly Instructions section of this document.</p>
<p>Another key issue discovered was metal screws, which were used to mount the payload, were located underneath the OpenCR and Raspberry Pi. Malfunctioning of the electronics during testing was suspected to be due to the metal screws shorting the electronic boards.</p>	<p>The solution implemented was replacement of the metal screws with nylon screws and rivets, which are poor conductors of electricity and would eliminate the risk of shorting.</p>

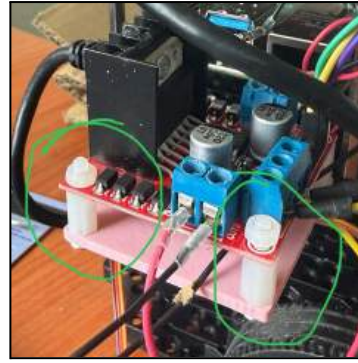


Fig 14: Motor Driver mounted with nylon brackets

During mission tests, it was noticed that the balls launched would have variable trajectories and land at different points. Occasionally, the ball also landed onto the payload ramp itself, nearly knocking the remaining balls off the robot.



Fig 15: Ball landing back on ramp disturbing other loaded balls during test run. No shield was installed back then

As a safety measure, the “KEN” shield was fabricated later and added onto Layer 4 which sheltered the balls on the ramp, and this proved to be effective.

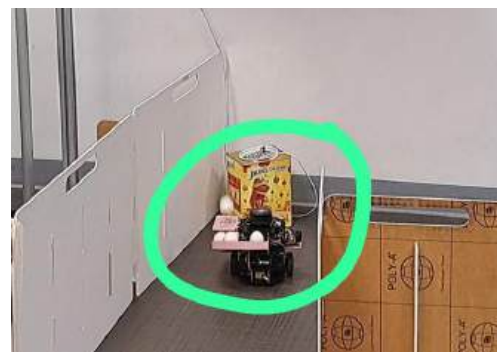
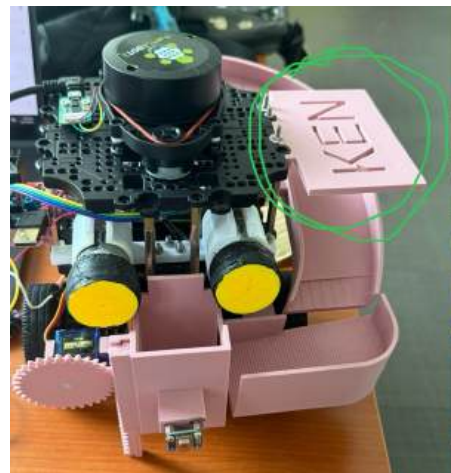


Fig 16a, 16b: “KEN” shield mounted on robot and protecting the loaded balls during a later test run

7.2 Critical System Design

7.2.1 Bill of Materials

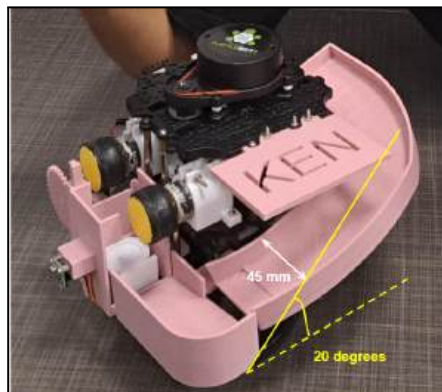
COMPONENTS	SOURCE	UNIT PRICE	UNIT	TOTAL PRICE
M3x10 metal screws	IDP LAB	N.A.	16	N.A.
M3 metal nuts	IDP LAB	N.A.	22	N.A.
M3x20 metal screws + M3 metal nuts	IDP LAB	N.A.	4	N.A.
M1.5x20mm screws + nuts	IDP LAB	N.A.	2	N.A.
M3x10 nylon screws	IDP LAB	N.A.	6	N.A.
M3x10 nylon nuts	IDP LAB	N.A.	3	N.A.
M3 Hex Support Spacers 10 mm (brass)	IDP LAB	N.A.	5	N.A.
M3 Hex Support Spacers 10 mm (nylon)	IDP LAB	N.A.	3	N.A.

7.2.2 Final Payload Design

The Ramp

The final payload design consists of a two-piece ramp inclined at 20 degrees to the horizontal which stores all 9 ping-pong balls required for the mission. The ramp's floor is profiled as a slope which guides the balls to roll down into a column, where they would be lifted one at a time to the flywheels for the 'firing of flares'.

The ramp's width is set to 45 mm, which allows balls (diameter 40 mm) to roll only after one another. Furthermore, the surface texture minimizes the frictional force against balls rolling down, while staying clear of the TurtleBot's wheels and the LiDAR sensor's range. The second ramp piece, connected between the first ramp body and column, ensures that balls vertically below the flywheels have enough clearance gap to directly feed into the launching column.



The Column

From the ramp, balls are guided to a column, which is essentially a hollow cuboid (cross-section of 43 mm x 43 mm) allowing just one ping-pong ball. The ball rolls along the ramp into a ball tray at the base of the column, acting as a ‘movable floor’. The tray is a circular plate with a capacity of just one ball, and is attached to the rack of a linear actuator so that it moves upward to transport the loaded ball to the flywheels for launching, and downward for loading the next ball. The tray is also coupled with a ‘blocker’, whose purpose is to keep the next balls stationary when a ball is being launched. Otherwise, as the tray moves up, the balls waiting on the ramp would roll under it and slip through the column.

The Linear Actuator (Servo + Pinion + Rack)

The linear actuator system, designed for moving the tray vertically, consists of an SG90 servo motor, a pinion press-fitted onto the servo, and a rack gear that interacts with the pinion. The rack is fitted to the right of the pinion, so counterclockwise rotation results in upward linear motion. The distance required for the ball and rack to move up by is about 70 mm, from column base (near Layer 2) to flywheels (between Layers 3 and 4). Since the SG90 servo has a limited rotation angle of just 180 degrees (or π radians), the minimum diameter required for the pinion is:

$$\frac{70 \times 2}{\pi} \approx 44.56 \text{ mm}$$

The pinion and rack have been set to actual dimensions of diameter 48 mm and length 113 mm. The servo was press-fitted into a slot that was built into the column itself, while a cross-section was created throughout the column’s height for the rack to move through. The gear teeth profiles were designed with the same gear module to mesh with each other accurately.

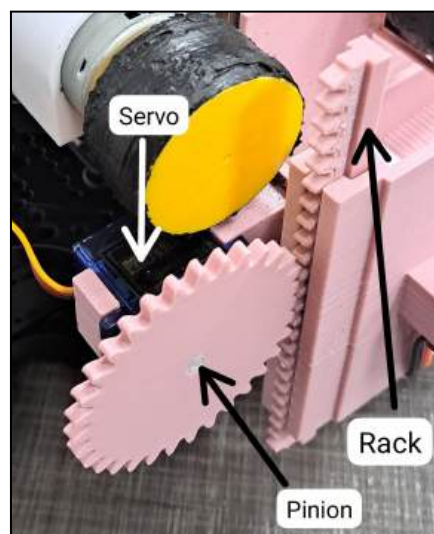


Fig 18: The Linear Actuator

The ‘Survivor Detector’

Another feature on the column was the mount for the AMG8833 Thermal Camera sensor, for detecting heat signatures and identifying target survivors. Two holes (diameter 2.5 mm) were created at the front of the column, aligning with those on the AMG sensor and dedicated for the mounting of the heat sensor. The sensor would read heat signals in front of the TurtleBot up to a range of 60 degrees. Heat inserts were placed into the column holes to create a threading profile.

The above parts - ramp pieces, column, pinion, and rack - were manufactured by 3D printing of PLA (Polylactic Acid). This material has a low density (1.24 g/cm^3). Payload parts are lightweight and do not disturb the centre of gravity of the entire robot significantly, therefore keeping it stable.

The ‘Flare Firing’ Mechanism

DC motors, flywheels, and the motor driver (L298N motor driver) were essential parts of the ball launching system in the TurtleBot. Flywheels were manufactured of TPU (Thermoplastic Polyurethane), unlike the other PLA parts. This is because TPU being relatively more elastic could compress against the ping-pong ball when launching it. The shape of the flywheel was designed such that it would curve around the ball (as shown below), creating a greater surface area of contact and leading to a more effective launch. Silicone paste was applied around the flywheels and allowed to cure. This provided the flywheels a better grip over the ping-pong balls during firing, ensuring the minimum flare height requirement of 1.5 m was satisfied. Two flywheels were placed at the top of the column, one on either side of the ping-pong ball. These were fitted into RS-PRO DC motors, which were powered by the OpenCR. The voltage of the motors was adjusted by the L298N motor driver, such that it provided sufficient rotational speed while not drawing too much current. The casings for the two DC motors were created as a single part, while an extension plate was designed on Layer 3 for the motor driver. Both parts were 3D printed of PLA, like the other payload parts.

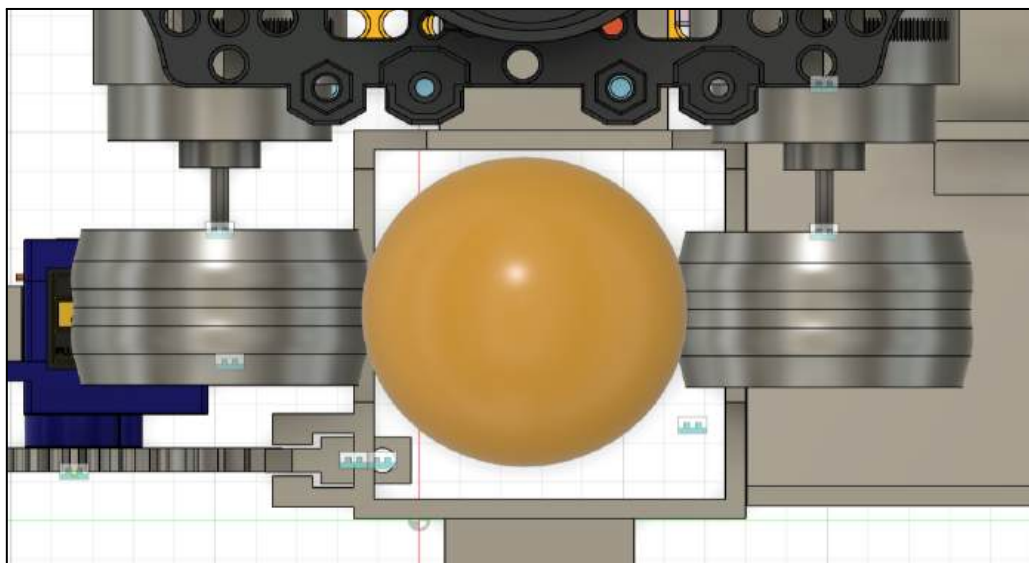


Fig 19: Flywheel Grooves

The “KEN” Shield

Lastly, a ‘shield’ was mounted on Layer 4 to cover part of the ramp. As described in the Prototyping & Testing section of this document, the purpose of the shield was to mitigate the risk of launched balls falling back onto the ramp. The shield’s dimensions were set such that it would sufficiently cover the ramp and not increase the width of the TurtleBot further. Additionally, it was ensured that there was enough clearance between the shield and the balls rolling under it.

Apart from the geometry and dimensions of the various parts, a crucial aspect to be considered were the mounting points of these pieces. In a CAD assembly file of the entire TurtleBot system, payload parts were inserted at their designated position. This allowed for accurate location of Waffle plate holes, as well as accurate projection of the Waffle plate edge profile (for the motor driver plate and shield). The most suitable holes were determined for mounting individual parts using screws and nuts, and created accordingly while 3D printing. A tolerance of 0.5 mm was used. Most parts were fixed in place using metal screws and nuts. However, there were some exceptions like:

- Brackets: the heat sensor at the front and the motor driver at the side were placed on bracket spacers that consist of a threaded profile, eliminating the need for nuts.
- Nylon screws and rivets: mounting holes for the ramp were located below the Raspberry Pi. Use of metal screws led to two key issues - difficulty in assembly, and possible shorting of the Raspberry Pi (as the screws would conduct electricity). Instead, nylon screws, nylon nuts, and rivets (non-metal; obtained from the TurtleBot3 Burger base robot) replaced the metal screws.

3D printed payloads were printed in pink to stay in theme with “KEN” (name of the system), which is also inscribed on the shield.

7.3 Assembly Instructions

The Assembly Manual for the TurtleBot3 Burger (the base robot used to develop the entire system) can be found [here](#). However, there are some modifications required per layer, which have been described below with supporting images:

Layer 1 Assembly Modifications

Change orientation of the battery brackets as demarcated in the image

Assemble with the same rivets/screws as instructed in the manual

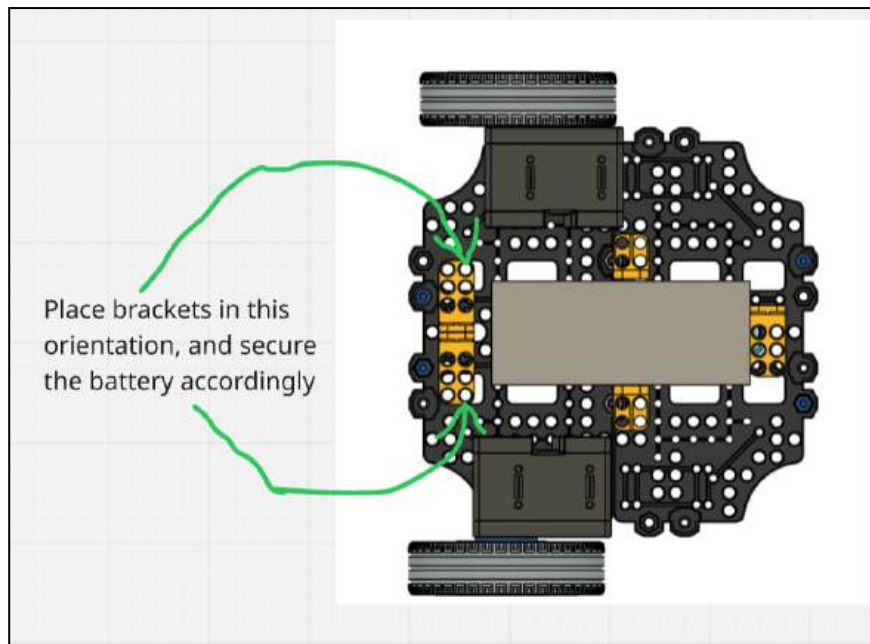


Fig 20: Relocation of Brackets

Move the support rods to their new positions as shown

Assemble with the same rivets/screws as instructed in the manual

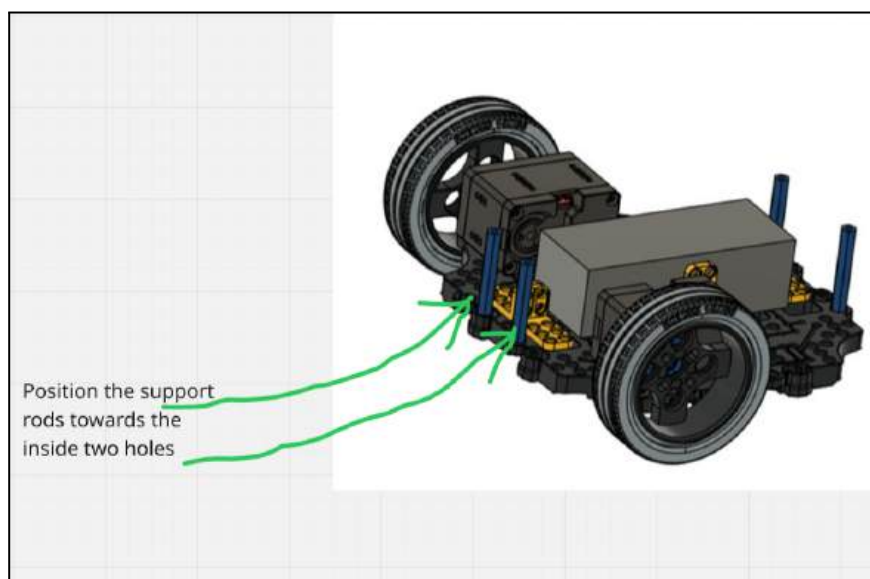


Fig 21: Relocation of Support Rods

Layer 2 Assembly Modifications

Assemble the second layer as per the manual, but change the position of the hex support rods as shown

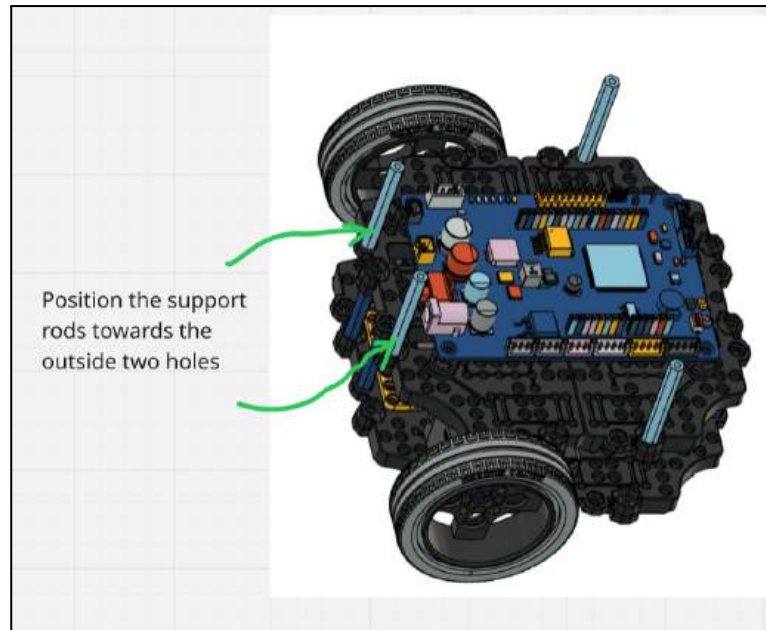
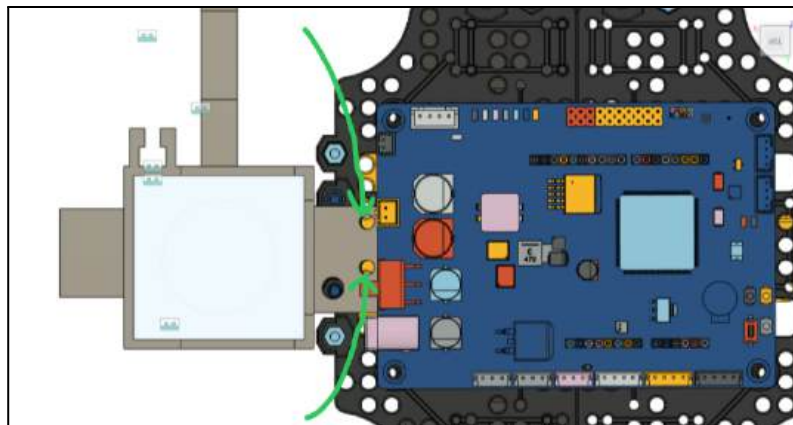


Fig 22: Relocation of Support Rods

Mount the ball launcher column onto the Waffle plate as shown

Use M3x10mm screws and nuts (2x).

Note that there is a third hole through which a screw for one of the support rods is passed. This screw belongs to the TurtleBot base itself.



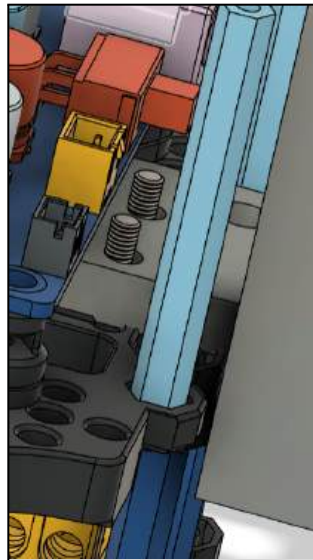
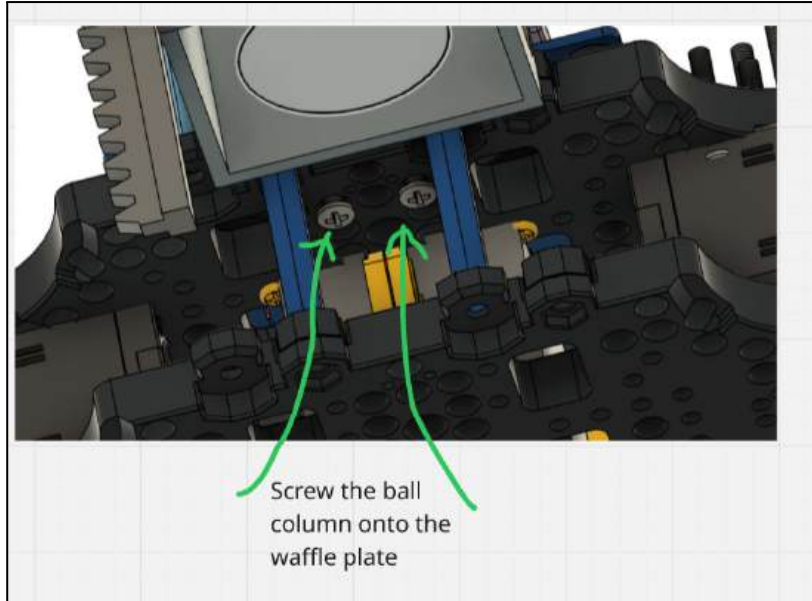


Fig 23a, 23b, 23c: Column Mounting

Fit the SG90 Servo motor into its designated slot within the column as shown. Use M1.5x20mm screws (2x) to fix the servo motor once fitted in position.

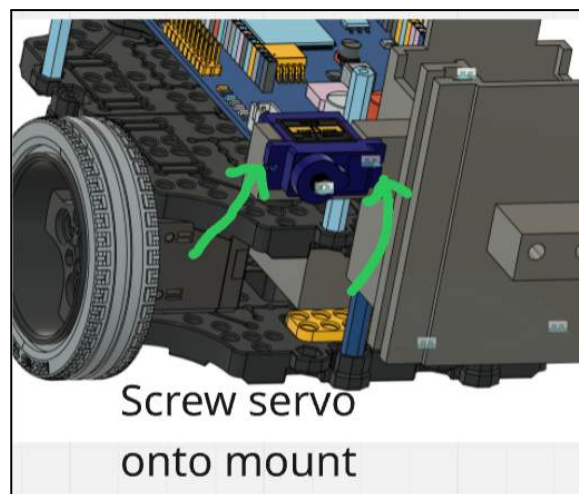


Fig 24: Servo fitting onto column

Press-fit the pinion gear onto the servo motor shaft

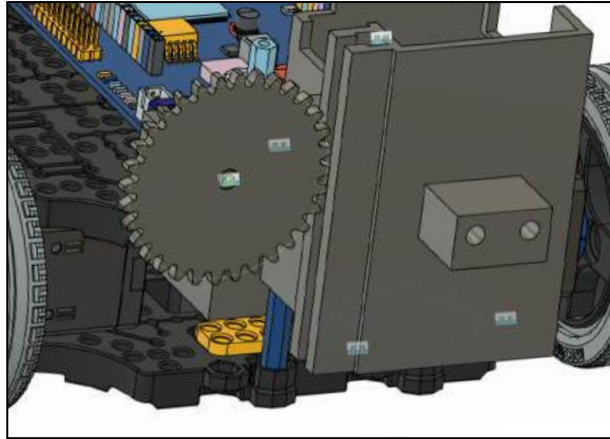


Fig 25: Pinion fitting onto servo

Insert the rack through the dedicated slot within the ball launcher column as shown, and mesh the two gears (pinion and rack) to enable the vertical linear motion

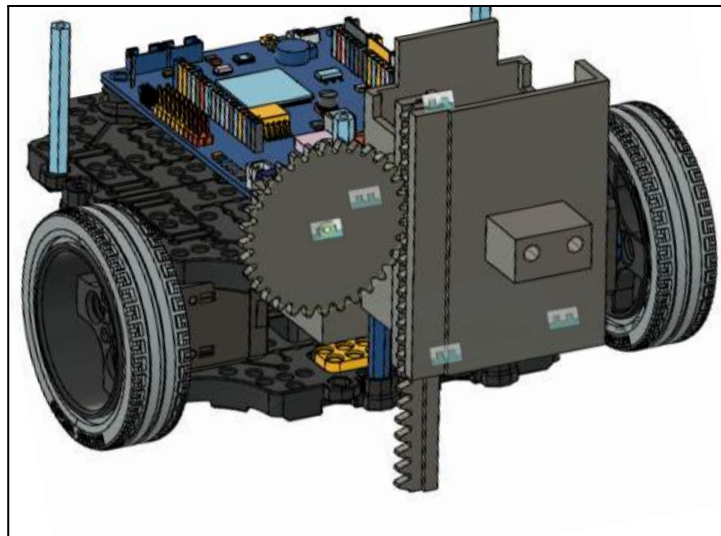
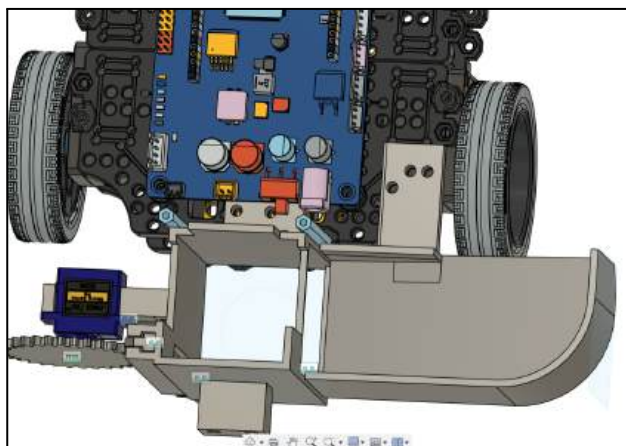


Fig 26: Rack insertion into column

Attach Ramp (Part 1) using M3x10mm screws (3x) and nuts (9x) as shown.

Note that for each of the 3 screws, two nuts are first inserted, followed by the ramp part, and finally a third nut to seal the ramp. The gap of two nuts between the ramp part and the waffle plate allows it to be mounted higher



than the column, so that the ball can smoothly roll into it.

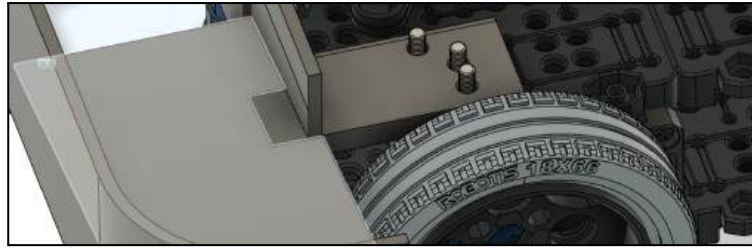


Fig 27a, 27b: Ramp Part 1 Mounting

Layer 3 Assembly Modifications

Assemble third layer as shown in the turtlebot assembly manual, with the edit of removing the one hex rod support as shown in the figure and swapping the front two hex rods as shown

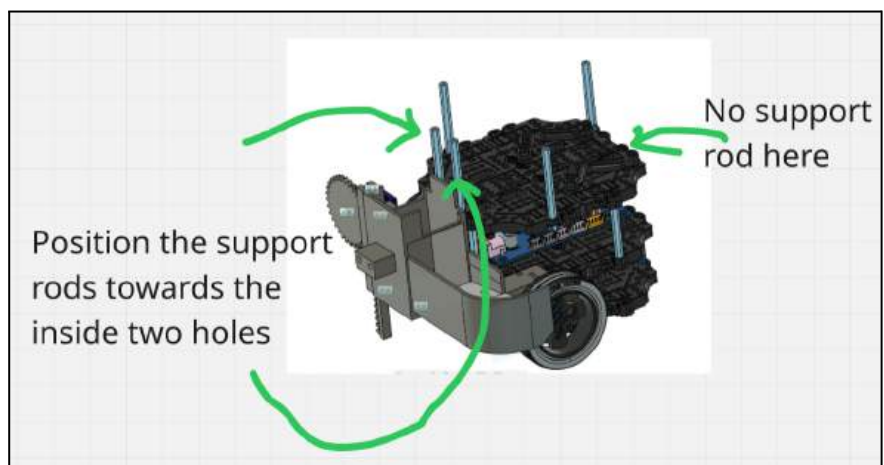
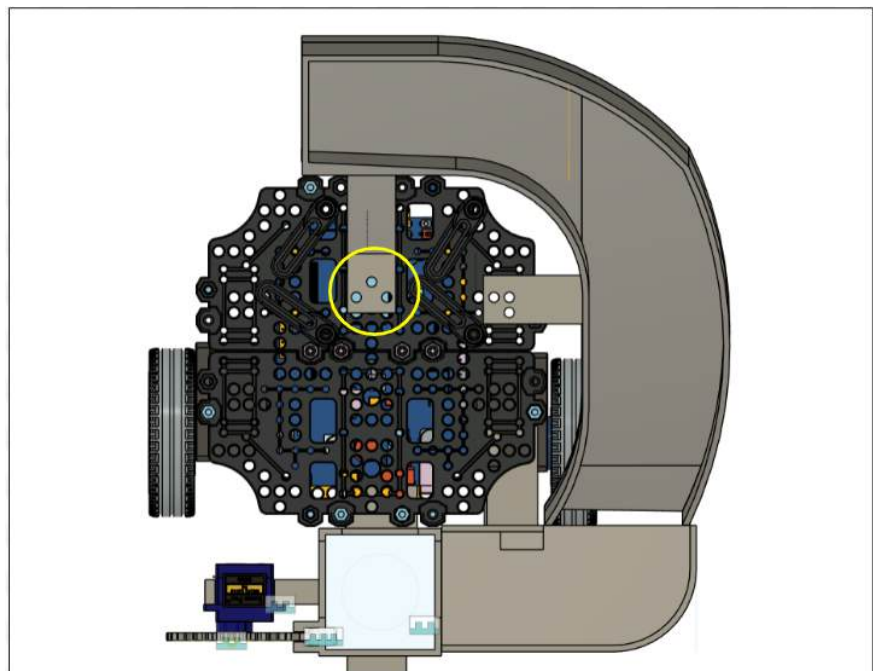


Fig 28: Relocation of Support Rods

Assemble the ramp (Part-2) as shown in the figure

Screw the M3x10mm screws as shown.

Note that nylon screws are used for the marked mounting holes instead of traditional metal screws, to avoid shorting of the Raspberry Pi which would be mounted over them. Alternatively, spare rivets from the TurtleBot3 Burger



robot kit can be used

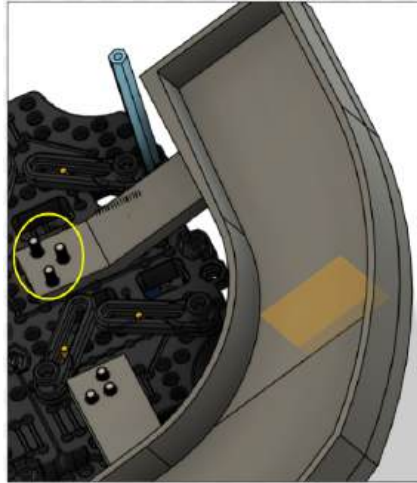


Fig 29a, 29b: Ramp Part 2 Mounting

Assemble the RPi as per manual, but do not assemble the USB2LDS, as this would be relocated onto Layer 4)

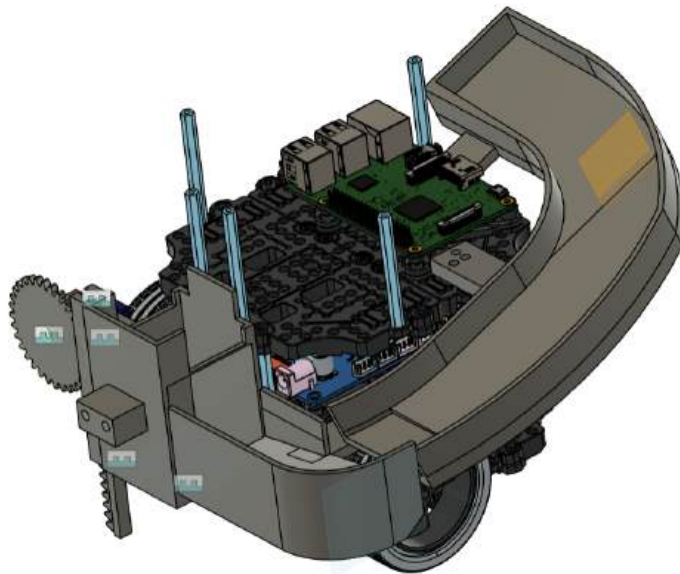


Fig 30: Raspberry Pi Assembly

Add extra 10mm male to female brass M3 hex support spacer on top of the original 45mm support rod given



Fig 31: Assembly of Hex Support Spacers

Attach the motor driver mount with the 10mm M3 hex support as shown in the figure

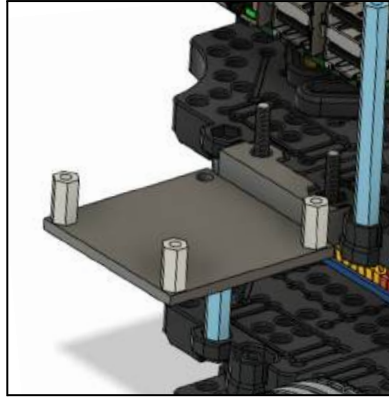


Fig 32: Assembly of Motor Driver Mount

Attach the L289N motor driver as shown

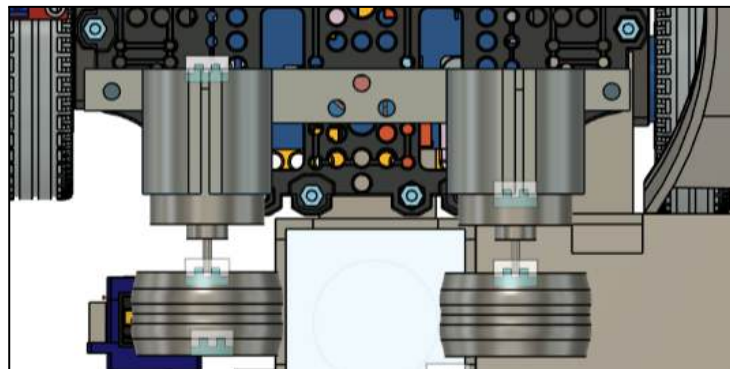


Fig 33: Assembly of L289N Motor Driver

Attach motor casing with RS PRO DC motor and flywheel attached directly as shown

Manually position the location of the flywheel based on the column gap for the flywheel

Use the M3x20mm screws to fasten the motor casing to waffle plate and to clamp the motor to motor casing



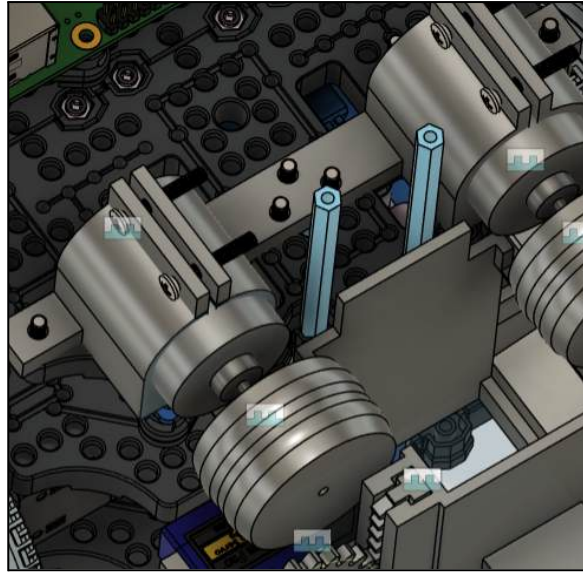


Fig 34a, 34b: Assembly of Flywheel Motors

Layer 4 Assembly Modifications

Assemble Layer 4 as
per TurtleBot manual

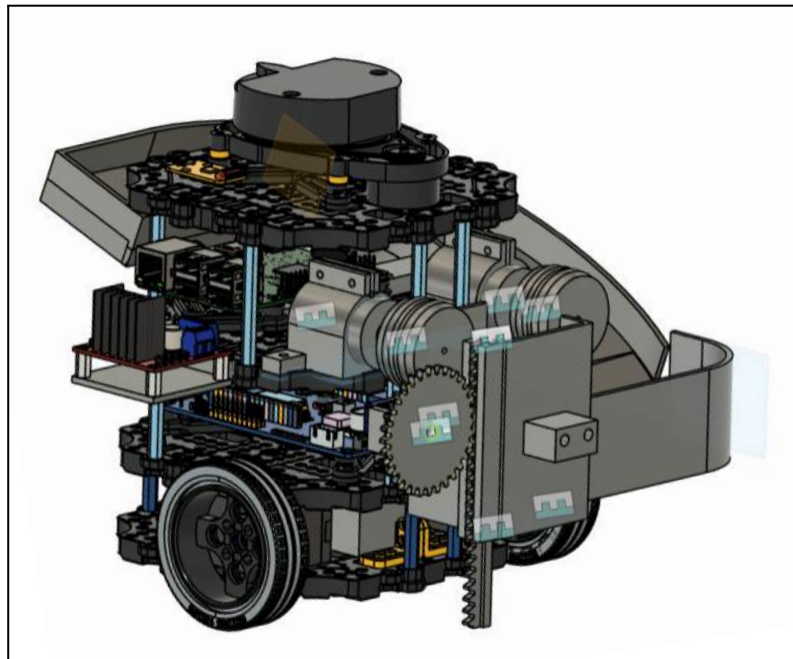


Fig 35: Traditional Assembly of LiDAR sensor

Assemble the USB2LDS as shown with the appropriate wire connection

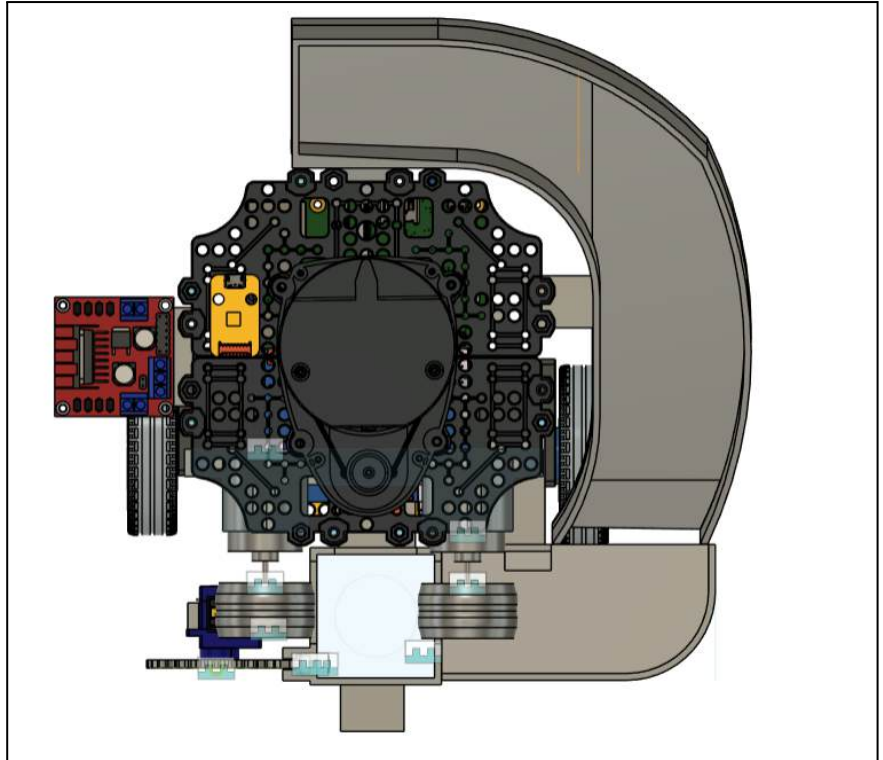
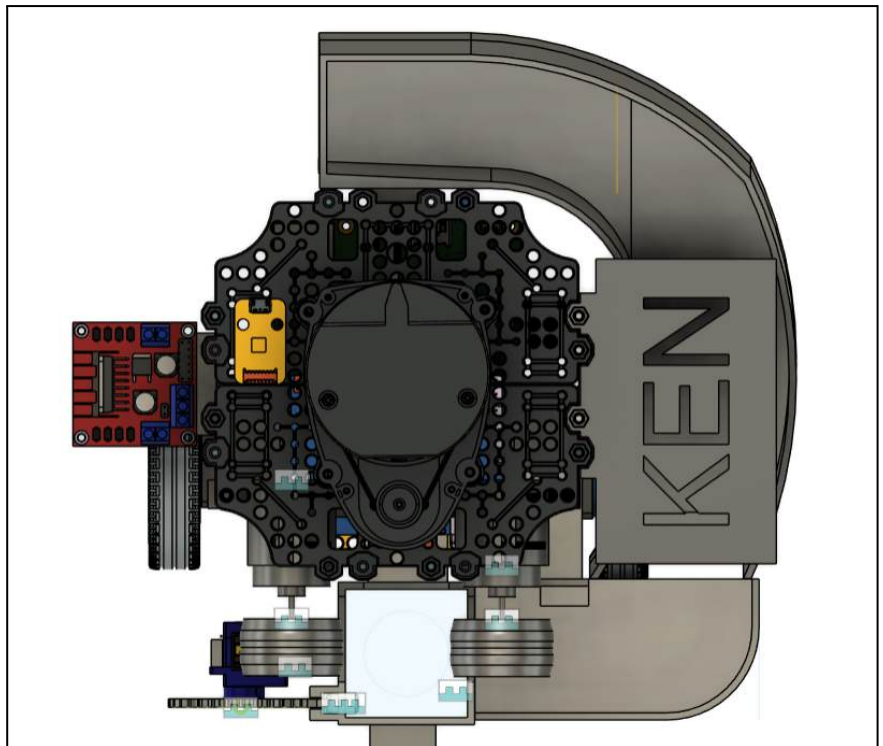


Fig 36: Assembly of USB2LDS

Attach the ball shield as shown

Use the M3x20mm screws and nuts to fasten



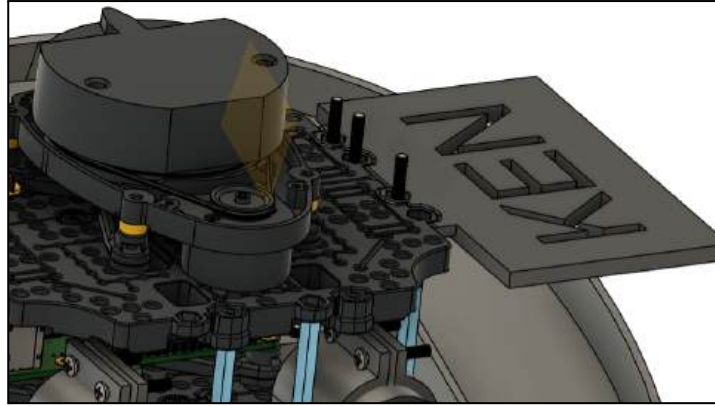


Fig 37a, 37b: Assembly of "KEN" Shield

Mount the AMG8833 heat sensor as shown, using the two M3 5mm hex supports as shown

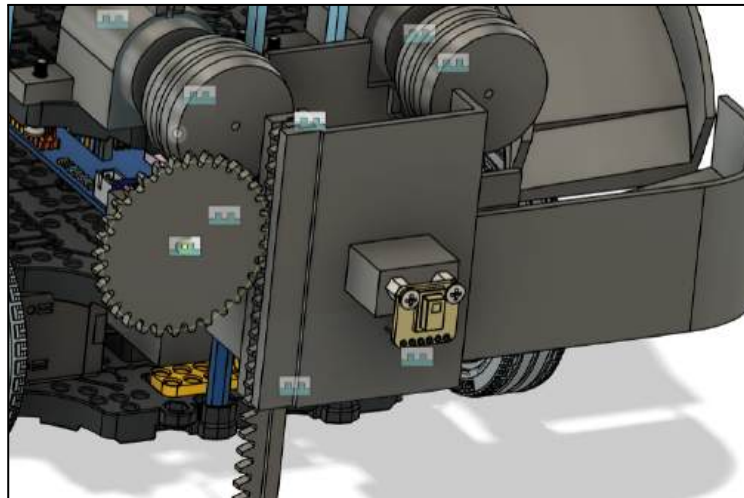


Fig 38: Assembly of AMG8833 Thermal Sensor

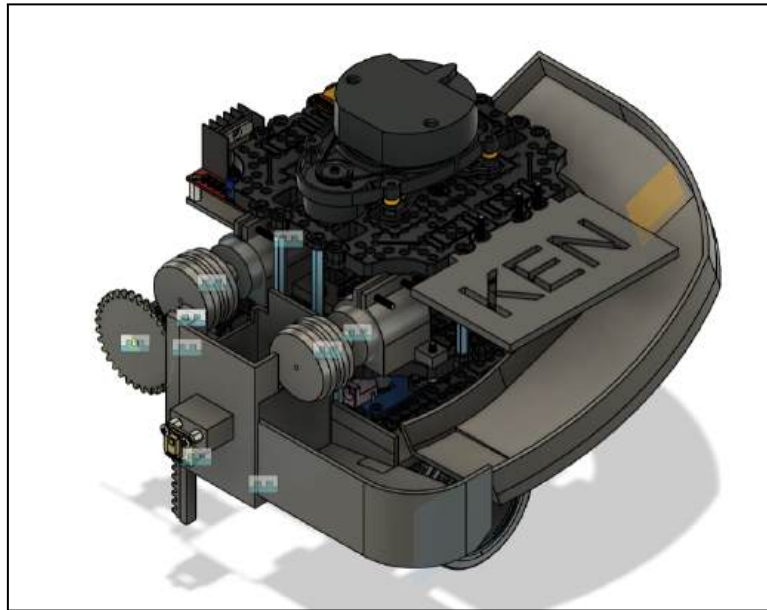
FINAL SYSTEM AFTER ASSEMBLY

Fig 39a: Isometric View of Final System

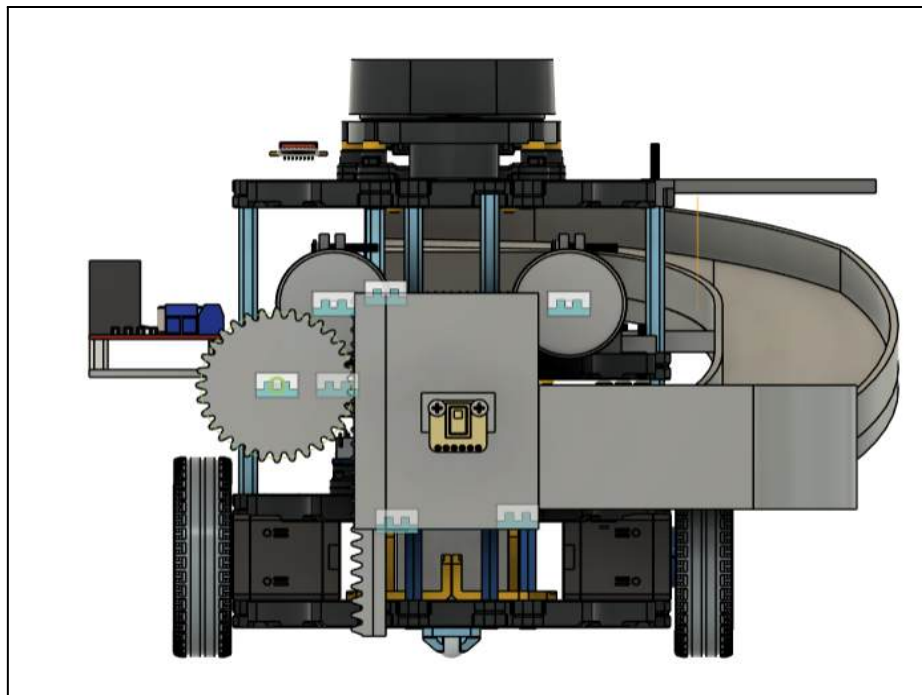


Fig 39b: Front View of Final System

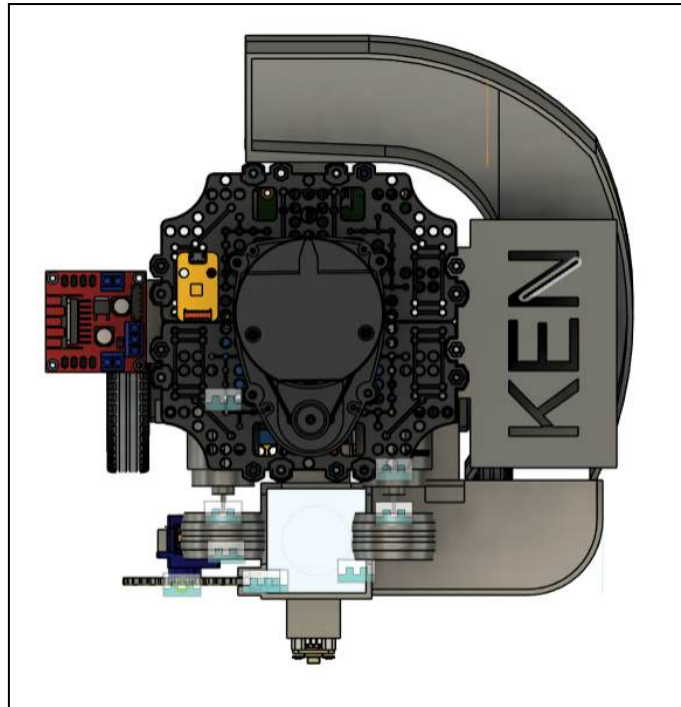


Fig 39c: Top View of Final System

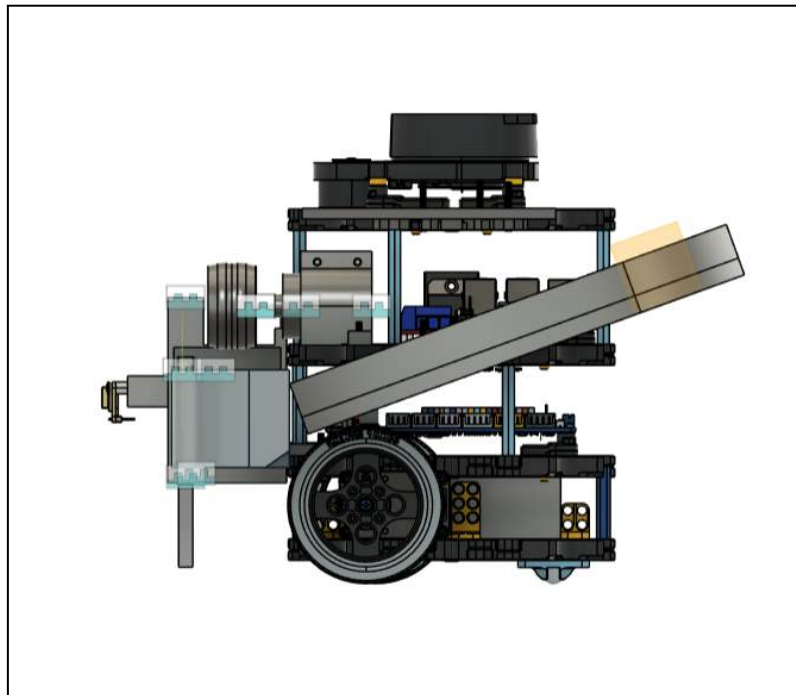


Fig 39d: Right View of Final System

Chapter 8: Electrical Subsystem

8.1 Electrical Components

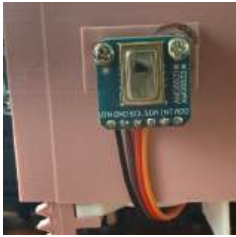
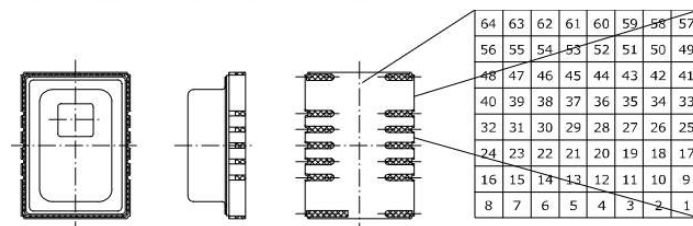
8.1.1 Heat sensors

Principle:

The AMG8833 sends out a temperature number for each pixel, ten times a second. With a 8x8 array of heat sensors, the AMG8833 module sends out 64 temperature readings - over I²C.

(1) Pixel array

Pixel array from 1 to 64 is shown below.



(2) Viewing angle (Typ.)

Sensor viewing angle is shown below.

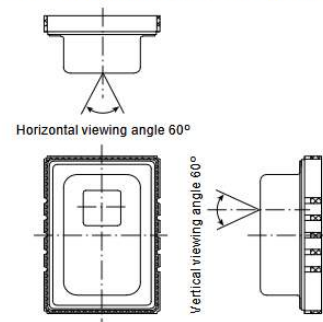


Fig 40: Heat Sensor Datasheet Explanation

Output Example:

- Each value represents the temperature detected by one pixel in the sensor's 8x8 grid.
- A significant rise in values in certain cells indicates a heat source or human presence.

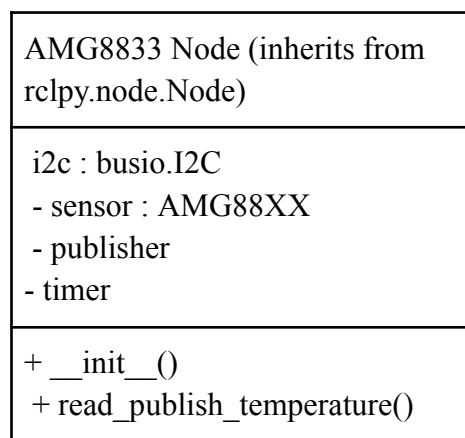
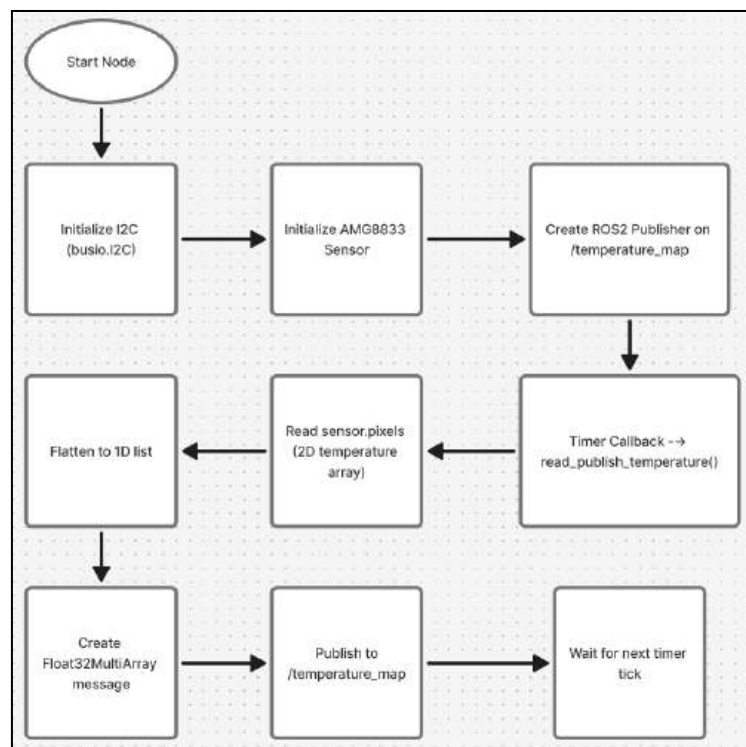
[

```
[22.5, 22.7, 22.9, 23.0, 22.8, 22.7, 22.6, 22.5],
[22.6, 22.9, 23.2, 23.6, 23.1, 22.9, 22.8, 22.7],
[22.7, 23.1, 24.0, 25.2, 24.1, 23.2, 22.9, 22.8],
[22.8, 23.3, 25.1, 28.6, 26.5, 24.0, 23.0, 22.9],
[22.9, 23.5, 25.5, 30.2, 27.0, 24.3, 23.1, 22.9],
[22.8, 23.1, 24.4, 26.0, 25.0, 23.5, 23.0, 22.8],
[22.7, 22.9, 23.5, 24.2, 24.1, 23.0, 22.8, 22.7],
[22.6, 22.7, 22.9, 23.0, 22.9, 22.8, 22.7, 22.6]
```

]

Connection Guide:

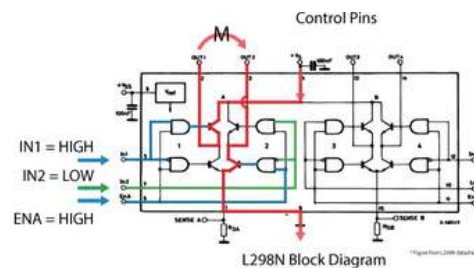
OUTPUT PINS (RPI4)	BCM PINS	AMG8833 PINS
PIN 3	GPIO 2	SDA
PIN 5	GPIO3	SCL
PIN 1	3.3V	VIN
PIN 9	GND	GND

Class diagram:**Code to retrieve heat sensor outputs and logic to identify the heatsource:***Fig 41: Algorithm to retrieve heat sensor outputs*

8.1.2 Motors and Motor-driver:

L298N Motor Driver:

This dual H-bridge motor driver controls DC motors' direction by reversing the voltage polarity. Transistors within the driver act as switches, and activating specific pairs directs current flow to dictate rotation direction. While capable of PWM-based speed control, the payload will only leverage its ability to control direction. PWM control is used but it is fixed at 75 % hardcoded in the integrated code.



Connection Guide:

OUTPUT PINS (RPI4)	BCM PINS	L298N PINS
PIN 31	GPIO 12	IN1
PIN 16	GPIO 23	IN2
PIN 33	GPIO 13	IN3
PIN 22	GND 25	IN4
OPEN CR POWER		12V
		GND
MOTOR 1		OUT1, OUT2
MOTOR 2		OUT3, OUT4

RS PRO 238-9715 Geared DC Motor:

Using kinematic equations to reach 1.5m in height:

$$\text{tangential velocity, } v = \sqrt{2gh} = \sqrt{2(9.81)(1.5)} \approx 5.425 \text{ m/s}$$

Using flywheels of 18mm radius,

$$\omega = \frac{v}{r} = \frac{5.425}{0.018} = 301.39 \text{ rad/s} = 301.39 \times \frac{60}{2\pi} \approx 2878.05 \text{ RPM}$$

Assuming efficiency = 60%, using 2 motors, $\text{RPM} \div 0.6 \div 2$.

We need 2 motors of about 2398.38 RPM.

Ping Pong Ball Mass: ~2.7g (0.0027 kg)

$$\text{Torque} = 2.7g \times 1.8 = 4.86 \text{ g} \cdot \text{cm}$$

The RS PRO 238-9715 is a 51g, 12V brushed DC geared motor that delivers a maximum torque of 64.2 gcm at 8768 rpm. These key characteristics safely exceed our requirements calculated above, and so the motor is suitable for our flywheels.

RPM and Torque Ratings required in the payload:

Code for motors:

```
motor_pins = [23, 25]    enable_pins = [13, 12]
```

Set the motor pins to LOW simulating ground and Enable input pins which are connected to PWM to 75% power the motors to 75% of its maximum speed. We don't need to change direction so it is just coded to spin forward. Note that for the current system, setting motor PWM above 75% will cause power overload, making the OpenCR board reboot whenever the motors are activated.

Motor A: IN1 = 75%, IN2 = LOW

Motor B: IN3 = 75%, IN4 = LOW

8.1.3 SG90 Servo motor:

The **SG90** is a small, lightweight **servo motor** used for precise movement—like turning an arm, a door, or a camera to a specific angle.

- It rotates within a limited range of **0° to 180°**.
- It's controlled using **PWM (Pulse Width Modulation)** signals.

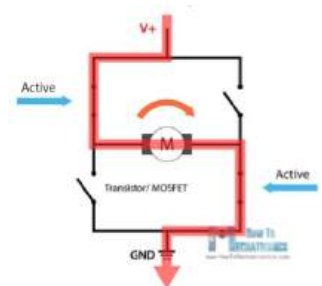


Fig 42 a Motor driver logic

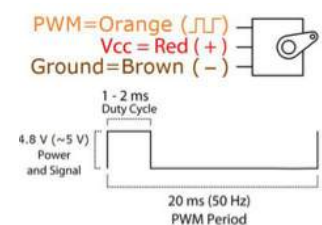


Fig 42 b Servo motor logic

The servo has a built-in sensor (usually a potentiometer) that tells it its current position. This creates a **closed-loop control system**. If the servo isn't at the angle the control pulse is asking for, the internal circuitry will keep adjusting the motor until it reaches the desired position.

The gearbox reduces the high speed of the small internal DC motor and increases its **torque** (rotational force).

Connection Guide:

OUTPUT PINS (RPI4)	BCM PINS	SERVO PINS
PIN 29	GPIO 5	SIGNAL PIN
RPI POWER		5V
		GND

Code for servos:

ChangeDutyCycle(x) sets the **angle**.

The value of x changes the length of the ON pulse.

With a **50 Hz signal**, each cycle is 20 ms:

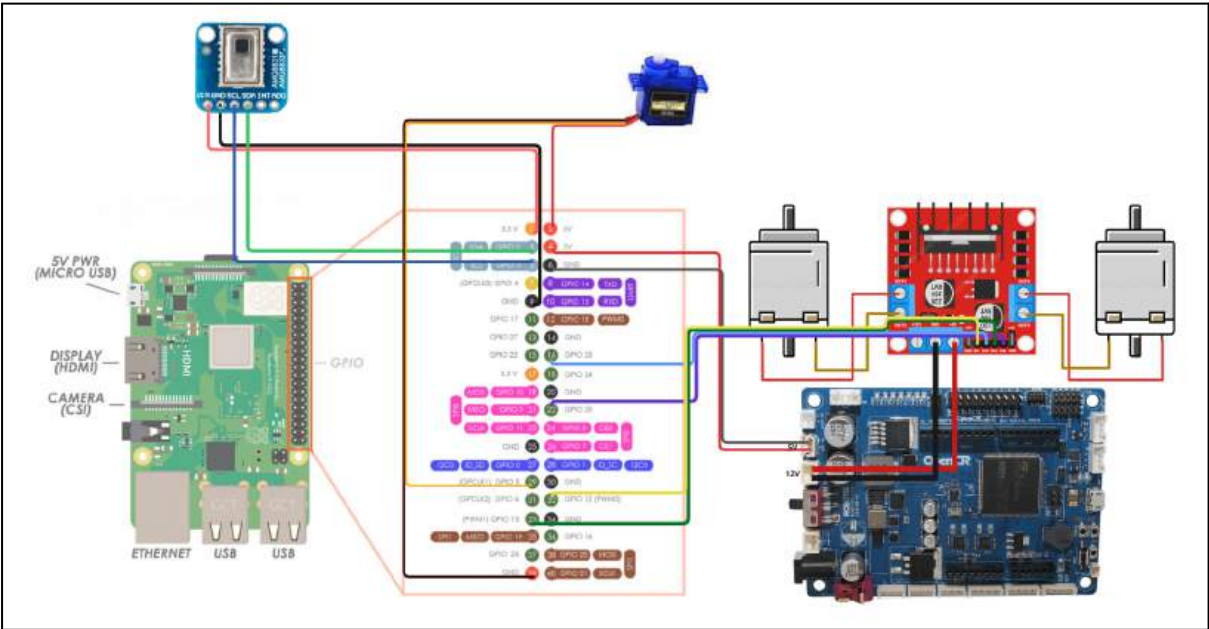
- A **2.5% duty cycle** = 0.5 ms ON → 0°
- A **10% duty cycle** = 2 ms ON → ~180°
- A **7.5% duty cycle** = 1.5 ms ON → 90° (center)

8.2 Critical Design**8.2.1 System Finances (Bill of Materials)**

COMPONENTS	SOURCE	UNIT PRICE	UNIT	TOTAL PRICE
TurtleBot3 Burger (complete list of parts available here)	IDP LAB	N.A.	1	N.A.
Servo Motor SG90	IDP LAB	N.A.	1	N.A.
AMG8833 IR Sensor	IDP LAB	N.A.	1	N.A.
RS BDC Motor	RS PRO	11.06	2	22.12
L298N MOTOR DRIVER	IDP LAB	N.A.	3	N.A.
Screws (M4x15)	IDP LAB	1.72	8	13.76
Standoffs	IDP LAB	0.62	12	7.44
Ramp	3D Print	N.A.	2	N.A.
Column	3D Print	N.A.	1	N.A.

Flywheel	3D Print	N.A.	2	N.A.
Motor Mounts	3D Print	N.A.	2	N.A.
L-brackets	3D Print	N.A.	5	N.A.
Silicon paste	LINK			
Heat inserts	LINK	N.A.	2	N.A.
Total				62.54

8.2.2 Electrical Schematic Diagram



Output pins	Peripherals	GPIO	Particle	Pin #	x	Pin #	Particle	GPIO	Peripherals	Output pins
AMG8833 Power		3.3V		1	x	2		5V		Servo Power
AMG8833 Communication Protocol I2C	I2C	GPIO2	SDA	3	x	4		5V		OpenCR Power
	I2C	GPIO3	SCL	5	x	6		GND		
		GPCLK0	D0	7	x	8	TX	GPIO14		
AMG8833 Ground		GND		9	x	10	RX	GPIO15	UART	
	Digital I/O	GPIO17	D1	11	x	12	D9/A0	GPIO18	PWM 1	
	Digital I/O	GPIO27	D2	13	x	14		GND		
	Digital I/O	GPIO22	D3	15	x	16	D10/A1	GPIO23	Digital I/O	Motor driver Input 1
		3.3V		17	x	18	D11/A2	GPIO24	Digital I/O	
				19	x	20		GND		
				21	x	22	D12/A3	GPIO25	Digital I/O	Motor driver input 4
	SPI	GPIO10	MOSI	23	x	24	CE0	GPIO8		
		GPIO9	MISO	25	x	26	CE1	GPIO7	SPI (chip enable)	
		GPIO11	SCK	27	x	28	DO NOT USE	ID_SC	DO NOT USE	
		GND		29	x	30		GND		
				31	x	32	D13/A4	GPIO12	PWM	Motor driver input 2
Motor driver Input 3		GPCLK2	D5	33	x	34		GND		
	PWM 2	GPIO13	D6	35	x	36	D14/A5	GPIO16	PWM 1	
	SPI	GPIO19	MISO	37	x	38	MOSI	GPIO20		
	Digital I/O	GPIO26	D8	39	x	40	SCLK	GPIO21	SPI	
Servo Ground		GND								

Fig 43a, 43b: Electrical Schematic Diagram

8.3 Energy Consumption

Assuming robot operation at the maximum speed and that all components are always operating. Assume the efficiency of the LiPo battery that of a typical battery 75%:

Energy stored in battery = 19.98Wh

Maximum power consumption during operation:

$$2.568 + 9.70 + 5.75 + 5.75 = 23.76 \text{ Wh}$$

Time of operation of the Turtle bot using a fully charged LiPo battery:

$$19.98 * 0.75 / 23.76 = 1.001\text{h} = 37.84 \text{ min}$$

Component	Voltage (V)	Current (A)	Power (W)
OpenCR+RPi	Boot: 11.1	Boot: 0.790	Boot: 8.769
	Operation: 11.1	Operation: 0.766	Operation: 8.5026
	Standby: 11.1	Standby: 0.519	Standby: 5.7609
Servo Motor	Idle: 4.92	Idle: 0.00074	Idle: 0.000365
	Rotating: 4.93	Rotating: 0.521	Rotating: 2.56853
Turtle bot: LIDAR + Servo + (Peak Load Scenario)	12 (11.1V causes the OpenCR board to beep.)	Operation: 0.809	Operation: 9.708
Left Motor for flywheels	12.0	0.770	5.75 - 9.24
Right Motor for flywheels	12.0	0.770	5.75 - 9.24

Note: "Operation" includes engaging the motors to activate the wheels.

Chapter 9 : Software Subsystem

9.1 Autonomous Navigation - attributes class diagram

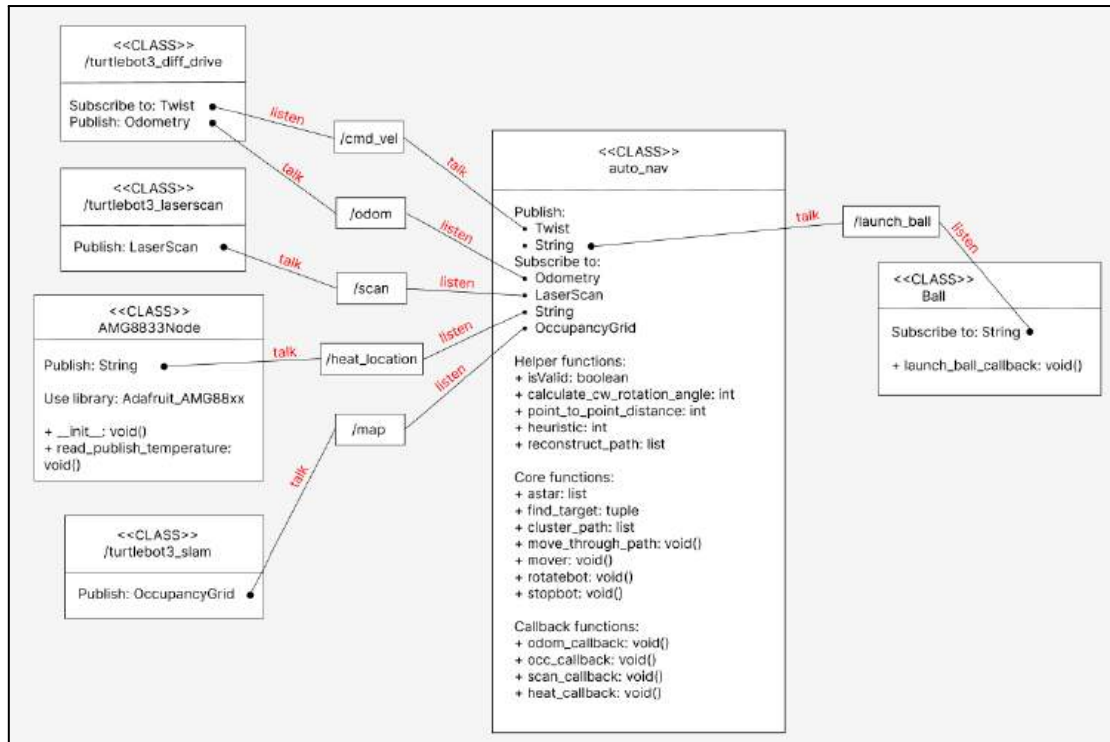
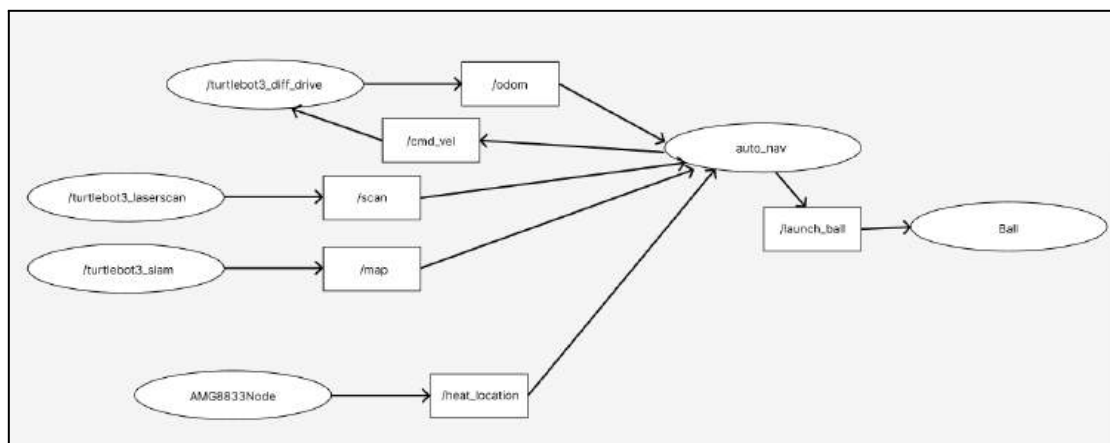


Fig 44: Attributes Class Diagram for Autonomous Navigation

9.2 ROS2 Implementation (RQT)



Note: Circles indicate nodes, while rectangles indicate topics

Fig 45: RQT Diagram

The `AMG8833Node` was made to publish the information about the nearby heat location through the `heat_location` topic. The `auto_nav` node will then be subscribing to this topic to

process it as an interrupt. To launch the ball, a *Ball* node has been made. The *auto_nav* node will publish a message to the *launch_ball* topic when it is time to launch the ball.

9.3 Algorithms and Control Strategies

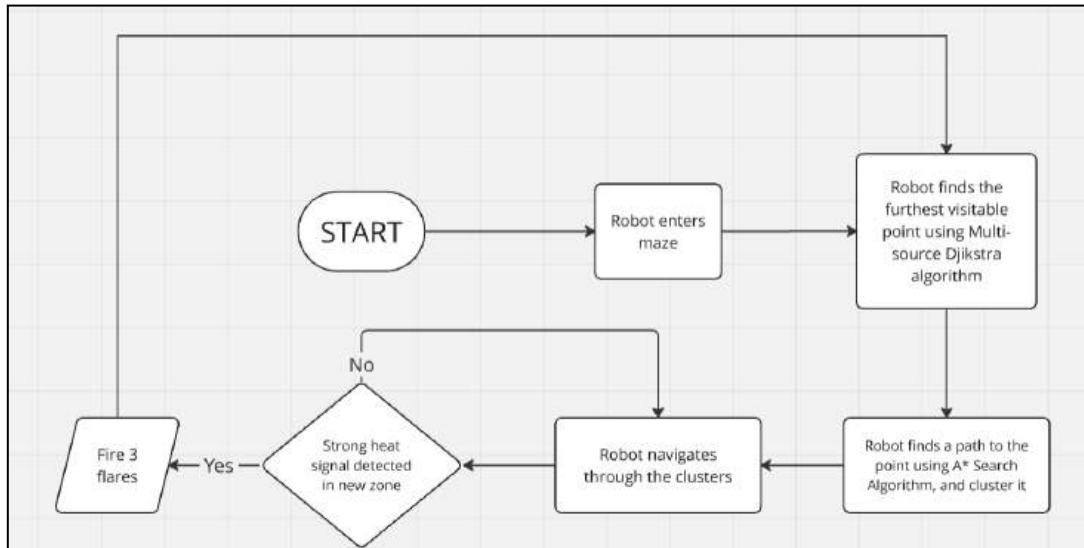


Fig 46: Algorithm Overview

9.4 Embedded Firmware Structure

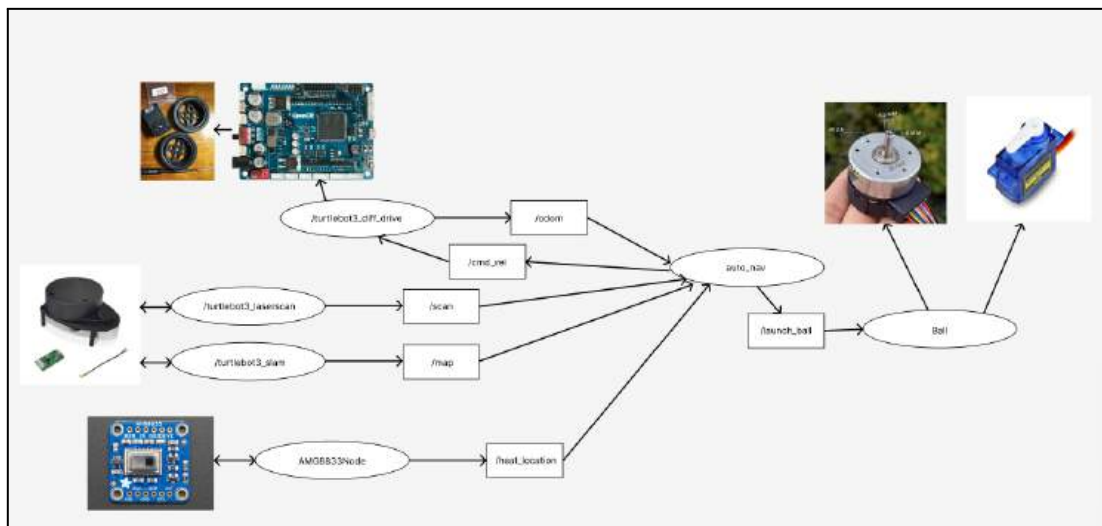


Fig 47: Embedded Firmware Structure

The four main parts of our embedded firmware components are:

1. OpenCR, controlled by *turtlebot3_diff_drive* node to move the wheels
2. LiDAR, communicating with *turtlebot3_laserscan* and *turtlebot3_slam* node
3. Heat sensor (Adafruit AMG8833), communicating with *AMG8833Node* node
4. Servo, controlled by *Ball* node to launch the ball

9.5 Navigation Algorithm

9.5.1 Introduction

The navigation system of our robot is built upon three foundational pillars: locating the target, planning an optimal path, and executing movement to reach the target. Each of these pillars is supported by dedicated algorithmic strategies, which are described in the following sections.

9.5.2 Localization

Before diving into the three foundational pillars, there is a fundamental problem that must be addressed: *How can we determine the robot's location on the map?* To answer this, the team decided to use the `tfBuffer` provided by ROS2. The implementation of this can be found in the [Appendix A](#).

9.5.3 Target Finding

9.5.3.1 Initial Approach: Greedy Based on Total Distance

The team's first approach to target identification applies a greedy heuristic. The heuristic aims to search a point that maximizes the sum of Euclidean distances to all previously visited points.

Formally, let \mathbf{v} be the array of all visited points with size k , and let \mathbf{v}_i denote the i -th visited point where $1 \leq i \leq k$. The goal is to find a point \mathbf{b} that maximizes the expression:

$$\sum_{i=1}^k \text{distance}(\mathbf{b}, \mathbf{v}_i)$$

The implementation of this approach is provided in [Appendix B](#).

However, after several testing, the team realizes edge cases where the greedy heuristic is not efficient. Consider the edge case below!

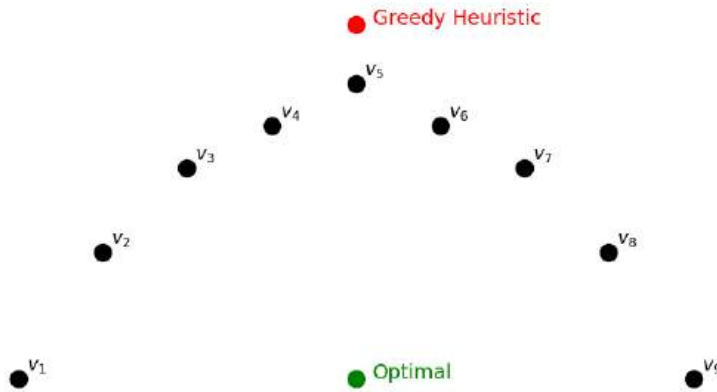


Figure 48: Edge Case for Greedy Based on Total Distance

In this edge case, the greedy heuristic detects the red point as the optimal target even though the optimal point to explore is the green point.

9.5.3.2 Revised Approach: Greedy Based on Minimum Distance

Upon examining the edge cases of the first heuristic, the team refined our approach. The revised strategy focuses on finding a point that maximizes the minimum Euclidean distance to any of the previously visited points.

Formally, with the same notation as above, the objective becomes finding a point \mathbf{b} that maximizes:

$$\min_{i=1}^k \text{distance}(\mathbf{b}, \mathbf{v}_i)$$

The implementation of this approach is provided in [Appendix C](#).

However, after many trials, this greedy approach also fails in several edge cases. Consider the scenario below.

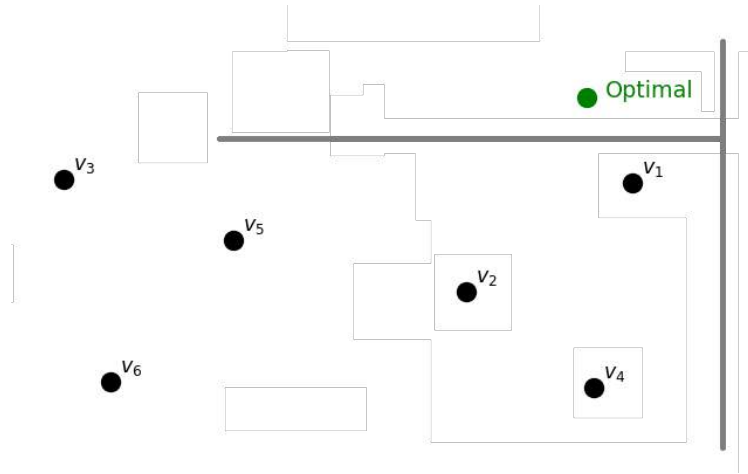


Figure 49: Edge Case for Greedy Based on Minimum Distance

In the Euclidean distance perspective, the green point is located near \mathbf{v}_1 . Therefore, the greedy heuristic will treat these two points as close to each other, even though in reality it does not. This suggests that this greedy heuristic needs to be improved, especially the limitation of using Euclidean distance.

9.5.3.3 Final Approach: Multi-source Dijkstra

To address the limitations of using Euclidean distance in complex terrains, the team's final approach utilizes a cost-based method. Specifically, the team applies the multi-source Dijkstra algorithm, treating all visited points as sources. This results in a cost map that reflects the minimum traversal cost to a point from the closest visited point. In order to make the robot explore as far as possible, the point which has the highest cost will then be chosen.

The implementation of this approach is provided in [Appendix D](#).

9.5.3.4 Minor Improvement

Due to the map update delay discussed in Chapter 7, the robot may occasionally collide with obstacles that have not yet been reflected in the updated map, especially when exploring distant areas. To mitigate this risk, the team introduces a new threshold called `DISTANCE_THRESHOLD`. This ensures that the robot avoids exploring points with a cost higher than `DISTANCE_THRESHOLD`, thereby reducing the likelihood of encountering unupdated or unknown obstacles.

The implementation of the algorithm after this improvement is provided in [Appendix E](#).

9.5.4 Path Finding

9.5.4.1 Initial Approach: Plain A* Search Algorithm

The team's first approach on finding the shortest path to the target is to use the plain A* Search Algorithm. The implementation of this approach can be found in [Appendix F](#) (for helper functions) and [Appendix G](#) (for A* Search implementation).

However, after analyzing the generated path, the team realized that the selected points were too close to the wall, making it likely that the robot would crash if it followed the path.

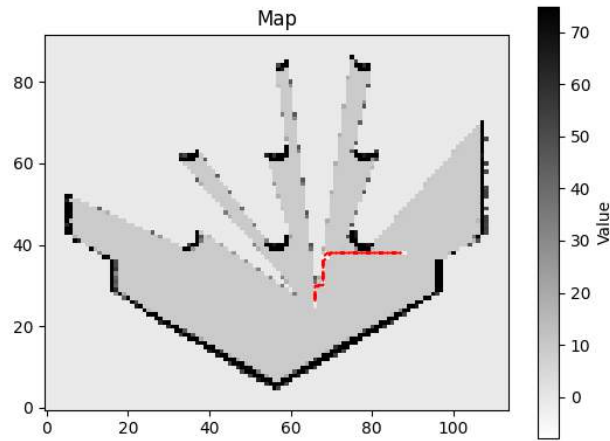


Figure 50: Path produced (in Gazebo Simulation)

9.5.4.2 Final Approach: A* Search Algorithm with Wall Penalty

To avoid potential crashes, the team decided to add a wall penalty mechanism that will add up to the A* heuristic when it is too close to the wall. By this, the algorithm will penalize locations which are too close to the wall, avoiding them to be used. The implementation of this approach can be found in [Appendix F](#) (for helper functions) and [Appendix H](#) (for A* Search implementation).

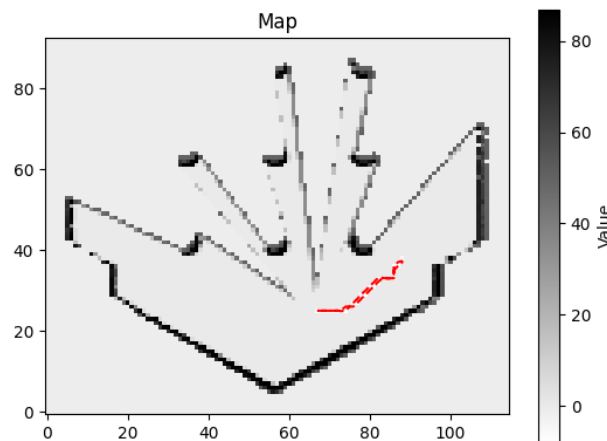


Figure 51: Path produced when wall penalty is applied (in Gazebo Simulation)

9.5.5 Movement Execution

9.5.5.1 Initial Approach: Point-to-Point Movement Strategy

The team's initial approach on movement execution was to make the robot follow the path generated by the A* Search Algorithm, navigating from point to point. The implementation of this method is provided in [Appendix I](#). However, in practice, the robot's movement is inconsistent, almost as if it were "jumping" between locations, due to localization inaccuracies.

9.5.5.2 Final Approach: Cluster-to-cluster Movement Strategy

To address the localization issue, the team proposed an alternative movement strategy: navigating the robots from cluster to cluster. These cluster points are selected from the path generated by the A* Search Algorithm, as discussed in Chapter 4.2. The implementation details of the clustering approach can be found in [Appendix J](#).

Then, to account for localization inaccuracy, the team introduced a tolerance margin, meaning the robot does not need to reach an exact coordinate. As long as it remains within a certain distance of the target point, it is considered a successful arrival. The implementation of this movement strategy can be found in [Appendix K](#) (for helper functions) and [Appendix L](#) (for movement implementation).

9.5.6 Heat Pursuit Movement

When heat is detected and the distance between the current point and the closest shooting area is higher than `SHOOTING_AREA_THRESHOLD`, an interrupt routine is triggered to process the event. In a separate program called `elec_nodes.py` (code can be found by clicking this [link](#)), nearby heat locations are published regularly via a ROS2 topic named `heat_location`. The heat information can be one of five types: `left`, `right`, `forward`, `ok`, and `null`. Based on the received signal, the robot will respond accordingly:

<code>null</code>	: No interrupt is triggered
<code>right</code>	: Stop the robot. Rotate the robot to the right by <code>HEAT_ROTATE_ANGLE</code>
<code>left</code>	: Stop the robot. Rotate the robot to the left by <code>HEAT_ROTATE_ANGLE</code>
<code>forward</code>	: Stop the robot. Move the robot forward
<code>ok</code>	: Stop the robot. Activate the shooting mechanism

The implementation of this interrupt routine can be found in [Appendix O](#). To see in detail where this routine is ran, see the full implementation of `move_through_path` function [here](#).

9.5.7 Map Update Delay

Initially, the team's strategy for updating the map within the callback function was to update it every time the callback was triggered (implementation can be found in [Appendix M](#)). However, this approach led to a problem. During movement execution, the robot heavily relies on row and column indices. As the map expands, these indices also change, causing inconsistencies that affect the robot's movement.

To address this issue, the team introduced a control flag variable called `can_update`. This flag determines whether the map should be updated within the callback function. With this modification, the map is only updated when `can_update` is explicitly enabled, ensuring consistent behavior during movement execution. The value of `can_update` is managed within the `move_through_path` function (can be found in [Appendix L](#)). The implementation of this can be found in [Appendix N](#).

9.5.8 Points Storing Mechanism

Recall the earlier observation: once the map is updated, the previously calculated row and column indices may no longer be valid. This is because the grid layout shifts with changes in the map origin. To address

this, we made a slight modification in how points are stored in the visited points set and the shooting area set.

Consider the code snippet below, which computes the row and column indices:

```
grid_x = round((self.cur_pos.x - self.map_origin.x) / self.map_res) # column in
        numpy
grid_y = round((self.cur_pos.y - self.map_origin.y) / self.map_res) # row in
        numpy

self.currow = self.map_width - 1 - grid_x
self.curcol = self.map_height - 1 - grid_y
```

Notice that while `cur_pos` remains unchanged, the resulting indices (`currow` and `curcol`) vary depending on `map_origin`. Therefore, instead of storing `currow` and `curcol` directly in the visited points and shooting area sets, we store the actual `cur_pos`. This change is also reflected when constrasting the initial ([Appendix M](#)) and final ([Appendix N](#)) implementation of the callback function.

Later, whenever we need to perform operations involving these sets, we convert the stored positions into row and column indices as needed. This approach has already been applied in the `find_target` function for the visited point set (see [Appendix E](#)) and the heat interrupt routine for the shooting area set (see [Appendix O](#)).

9.5.9 Parameter Descriptions

The navigation code relies heavily on these parameters. Tune these parameters to achieve the best performance. Understand what these values are and tune the parameters by making observation during testing.

1. SHOOTING_AREA_THRESHOLD:

Let x represent the distance from the current location to the nearest shooting area. This value is compared against a `SHOOTING_AREA_THRESHOLD` threshold to determine whether the heat interrupt should be processed:

$$\begin{cases} x < \text{SHOOTING_AREA_THRESHOLD} & : \text{Ignore the heat interrupt} \\ x \geq \text{SHOOTING_AREA_THRESHOLD} & : \text{Proceed to check the heat interrupt} \end{cases}$$

2. RESET_VISITED_POINTS_THRESHOLD:

This threshold controls the maximum number of points that can be stored in the visited points set. Once the size of the set exceeds this threshold, the oldest point will be removed.

The main purpose of this variable is to reduce computational complexity when the visited points set becomes too large. However, if the threshold is set too low, the robot may repeatedly explore the same areas, as it will forget previously visited points too quickly.

Suggestion: Observe the chosen path from the map and increase the threshold if the robot keeps exploring the same area too fast.

3. HEAT_ROTATE_ANGLE:

This threshold controls how many degrees to turn left or right when heat interrupt is triggered and heat is detected on left or right.

Suggestion: If the robot turns to the left or right too much when heat interrupt is triggered, lower down the value of this threshold.

4. WALL_THRESHOLD

Let x represent a value from the LiDAR-generated map. This value is compared against a threshold to determine how each point is interpreted:

$$\begin{cases} x < \text{WALL_THRESHOLD} : \text{Empty space} \\ x \geq \text{WALL_THRESHOLD} : \text{Obstacle} \end{cases}$$

The threshold `WALL_THRESHOLD` distinguishes between empty space and obstacles based on LiDAR readings. This threshold is only used in the `isValid` function that can be found in [Appendix F](#).

5. `DISTANCE_THRESHOLD`

This threshold controls the highest cost that can be chosen to be explored. This threshold is only used in the `find_target` function that can be found in [Appendix E](#). Refer to Chapter 3.4 for more detailed explanation.

6. `wall_penalty`

This threshold defines the maximum allowed row and/or column distance for a point to be considered penalized. This threshold is used in the `astar` that can be found in [Appendix H](#) and `find_target` function that can be found in [Appendix E](#).

Alert: This threshold is highly sensitive. If set too high, certain areas of the map may remain unexplored because A* Search is unable to find a path to it. If set too low, the robot may frequently crash into obstacles. Therefore, it is important to tune this value carefully.

7. `cluster_distance`

This threshold defines the minimum distance that separate any two adjacent cluster points. It is used only in the `cluster_path` function, which can be found in [Appendix J](#).

8. `localization_tolerance`: controls the tolerance margin that is described in Chapter 5.2.

9. `rotatechange`: controls the robot's angular speed.

10. `speedchange`: controls the robot's linear speed.

A Localization

```
# Declared in the init function of the class
self.tfBuffer = tf2_ros.Buffer(cache_time=rclpy.duration.Duration(seconds=5.0))
self.tfListener = tf2_ros.TransformListener(self.tfBuffer, self)

# Implemented in the map callback funtion
try:
    trans = self.tfBuffer.lookup_transform(
        'map', 'base_link',
        rclpy.time.Time(),
        timeout=rclpy.duration.Duration(seconds=1)
    )
except (LookupException, ConnectivityException, ExtrapolationException) as e:
    print(e)
    return

self.cur_pos = trans.transform.translation
```

B Greedy Based on Total Distance Implementation

```
def find_target(self):
    dis = -math.inf
    targetrow, targetcol = -1, -1
    num_rows, num_cols = self.cur_map.shape
    for idxrow in range(num_rows):
        for idxcol in range(num_cols):
            too_close_to_wall = False
            for penaltyx in range(-wall_penalty, wall_penalty + 1):
                if too_close_to_wall:
                    break
            for penaltyy in range(-wall_penalty, wall_penalty + 1):
                next_row = idxrow + penaltyx
                next_col = idxcol + penaltyy
                if not self.isValid(next_row, next_col):
                    too_close_to_wall = True
                    break
            if too_close_to_wall:
                continue
            if self.cur_map[idxrow][idxcol] == 0:
                cur_dis = 0
                for x, y in self.visited_points:
                    cur_dis += (idxrow - x) ** 2 + (idxcol - y) ** 2
                if dis < cur_dis:
                    dis = cur_dis
                    targetrow = idxrow
                    targetcol = idxcol
    return (targetrow, targetcol)
```

C Greedy Based on Minimum Distance Implementation

```
def find_target(self):
    dis = -1
    targetrow, targetcol = -1, -1
    num_rows, num_cols = self.cur_map.shape
    self.get_logger().info('Finding Target')
    for idxrow in range(num_rows):
        for idxcol in range(num_cols):
            too_close_to_wall = False
            for penaltyx in range(-wall_penalty, wall_penalty + 1):
                if too_close_to_wall:
                    break
                for penaltyy in range(-wall_penalty, wall_penalty + 1):
                    next_row = idxrow + penaltyx
                    next_col = idxcol + penaltyy
                    if not self.isValid(next_row, next_col):
                        too_close_to_wall = True
                        break
            if too_close_to_wall:
                continue
            if self.cur_map[idxrow][idxcol] == 0:
                min_dis = math.inf
                for x, y in self.visited_points:
                    grid_x = round((x - self.map_origin.x) / self.map_res)
                    grid_y = round(((y - self.map_origin.y) / self.map_res))
                    convertx = self.map_width - 1 - grid_x
                    converty = self.map_height - 1 - grid_y
                    min_dis = min(min_dis, (idxrow - convertx) ** 2 + (idxcol -
                        converty) ** 2)
                if dis < min_dis:
                    dis = min_dis
                    targetrow = idxrow
                    targetcol = idxcol
    return (targetrow, targetcol)
```

D Multi-source Dijkstra Implementation

```
def find_target(self):
    num_rows, num_cols = self.cur_map.shape
    cost_map = [[-1 for _ in range(num_cols)] for _ in range(num_rows)]
    for x, y in self.visited_points:
        grid_x = round((x - self.map_origin.x) / self.map_res)
        grid_y = round((y - self.map_origin.y) / self.map_res)
        convertx = self.map_width - 1 - grid_x
        converty = self.map_height - 1 - grid_y
        cost_map[convertx][converty] = 0
    points = []
    for x, y in self.visited_points:
        grid_x = round((x - self.map_origin.x) / self.map_res)
        grid_y = round((y - self.map_origin.y) / self.map_res)
        convertx = self.map_width - 1 - grid_x
        converty = self.map_height - 1 - grid_y
        heapq.heappush(points, (0, convertx, converty))
    while points:
        curcost, currow, curcol = heapq.heappop(points)
        for idx in range(4):
            nextrow = currow + dr[idx]
            nextcol = curcol + dc[idx]

            if not self.isValid(nextrow, nextcol):
                continue

            too_close_to_wall = False
            for penaltyx in range(-wall_penalty, wall_penalty + 1):
                if too_close_to_wall:
                    break
                for penaltyy in range(-wall_penalty, wall_penalty + 1):
                    next_row = nextrow + penaltyx
                    next_col = nextcol + penaltyy
                    if not self.isValid(next_row, next_col):
                        too_close_to_wall = True
            if too_close_to_wall:
                continue

            nextcost = curcost + MOVE_COST[idx]
            if cost_map[nextrow][nextcol] == -1 or nextcost < cost_map[nextrow][
                nextcol]:
                cost_map[nextrow][nextcol] = nextcost
                heapq.heappush(points, (nextcost, nextrow, nextcol))

    np.savetxt("cost_map.txt", cost_map, fmt="%3d")

    highest_cost = 0
    targetrow, targetcol = -1, -1
    for row in range(num_rows):
        for col in range(num_cols):
            if highest_cost < cost_map[row][col]:
                highest_cost = cost_map[row][col]
                targetrow = row
                targetcol = col

    return (targetrow, targetcol)
```


E Multi-source Dijkstra Implementation with Cost Restriction

```
def find_target(self):
    num_rows, num_cols = self.cur_map.shape
    cost_map = [[-1 for _ in range(num_cols)] for _ in range(num_rows)]
    for x, y in self.visited_points:
        grid_x = round((x - self.map_origin.x) / self.map_res)
        grid_y = round(((y - self.map_origin.y) / self.map_res))
        convertx = self.map_width - 1 - grid_x
        converty = self.map_height - 1 - grid_y
        cost_map[convertx][converty] = 0
    points = []
    for x, y in self.visited_points:
        grid_x = round((x - self.map_origin.x) / self.map_res)
        grid_y = round(((y - self.map_origin.y) / self.map_res))
        convertx = self.map_width - 1 - grid_x
        converty = self.map_height - 1 - grid_y
        heapq.heappush(points, (0, convertx, converty))
    while points:
        curcost, currow, curcol = heapq.heappop(points)
        for idx in range(4):
            nextrow = currow + dr[idx]
            nextcol = curcol + dc[idx]

            if not self.isValid(nextrow, nextcol):
                continue

            too_close_to_wall = False
            for penaltyx in range(-wall_penalty, wall_penalty + 1):
                if too_close_to_wall:
                    break
                for penaltyy in range(-wall_penalty, wall_penalty + 1):
                    next_row = nextrow + penaltyx
                    next_col = nextcol + penaltyy
                    if not self.isValid(next_row, next_col):
                        too_close_to_wall = True
            if too_close_to_wall:
                continue

            nextcost = curcost + MOVE_COST[idx]
            if cost_map[nextrow][nextcol] == -1 or nextcost < cost_map[nextrow][
                nextcol]:
                cost_map[nextrow][nextcol] = nextcost
                heapq.heappush(points, (nextcost, nextrow, nextcol))

    np.savetxt("cost_map.txt", cost_map, fmt="%3d")

    highest_cost = 0
    targetrow, targetcol = -1, -1
    for row in range(num_rows):
        for col in range(num_cols):
            if highest_cost < cost_map[row][col] and cost_map[row][col] <=
                DISTANCE_THRESHOLD:
                highest_cost = cost_map[row][col]
                targetrow = row
                targetcol = col

    return (targetrow, targetcol)
```

F A* Search Algorithm Helper Function

```
def isValid(self, row, col):
    rowsize, colsize = self.cur_map.shape
    return 0 <= row < rowsize and 0 <= col < colsize and self.cur_map[row][col] <
        WALL_THRESHOLD and self.cur_map[row][col] != -1

def heuristic(self, curpoint, targetpoint):
    return (targetpoint[0] - curpoint[0]) ** 2 + (targetpoint[1] - curpoint[1]) **
        2

def reconstruct_path(self, parent_map, start, target):
    path = []
    node = target
    while node != start:
        path.append(node)
        node = parent_map[node]
    path.append(start)
    path.reverse()
    return path
```

G Plain A* Search Algorithm Implementation

```
def astar(self, target_row, target_col):
    start_row = self.currow
    start_col = self.curcol
    # Priority queue: (cost + heuristic, row, col)
    astar = []
    heapq.heappush(astar, (0, start_row, start_col))

    cost_map = { (start_row, start_col): 0 }
    parent_map = { (start_row, start_col): None }

    find_path = False
    while astar:
        cost, currow, curcol = heapq.heappop(astar)

        if (currow, curcol) == (target_row, target_col):
            find_path = True
            break

        for idx in range(4):
            nextrow = currow + dr[idx]
            nextcol = curcol + dc[idx]
            if not self.isValid(nextrow, nextcol):
                continue

            nextcost = cost + MOVE_COST[idx]
            if (nextrow, nextcol) not in cost_map or nextcost < cost_map[(nextrow,
                nextcol)]:
                cost_map[(nextrow, nextcol)] = nextcost
                parent_map[(nextrow, nextcol)] = (currow, curcol)
                priority = nextcost + self.heuristic((nextrow, nextcol), (
                    target_row, target_col))
                heapq.heappush(astar, (priority, nextrow, nextcol))

    if find_path:
        print("A* Search found a path")
        path = self.reconstruct_path(parent_map, (start_row, start_col), (
            target_row, target_col))
        return path
    else:
        return []
```

H A* Search Algorithm with Wall Penalty Implementation

```
def astar(self, target_row, target_col):
    start_row = self.currow
    start_col = self.curcol
    # Priority queue: (cost + heuristic, row, col)
    astar = []
    heapq.heappush(astar, (0, start_row, start_col))

    cost_map = { (start_row, start_col): 0 }
    parent_map = { (start_row, start_col): None }

    find_path = False
    while astar:
        cost, currow, curcol = heapq.heappop(astar)

        if (currow, curcol) == (target_row, target_col):
            find_path = True
            break

        for idx in range(4):
            nextrow = currow + dr[idx]
            nextcol = curcol + dc[idx]
            if not self.isValid(nextrow, nextcol):
                continue

            cnt = 0
            for penaltyx in range(-wall_penalty, wall_penalty + 1):
                for penaltyy in range(-wall_penalty, wall_penalty + 1):
                    next_row = nextrow + penaltyx
                    next_col = nextcol + penaltyy
                    if not self.isValid(next_row, next_col):
                        cnt += 1

            nextcost = cost + MOVE_COST[idx] + cnt * 200
            if (nextrow, nextcol) not in cost_map or nextcost < cost_map[(nextrow,
                nextcol)]:
                cost_map[(nextrow, nextcol)] = nextcost
                parent_map[(nextrow, nextcol)] = (currow, curcol)
                priority = nextcost + self.heuristic((nextrow, nextcol), (
                    target_row, target_col))
                heapq.heappush(astar, (priority, nextrow, nextcol))

    if find_path:
        print("A* Search found a path")
        path = self.reconstruct_path(parent_map, (start_row, start_col), (
            target_row, target_col))
        return path
    else:
        return []
```

I Point-to-point Movement Strategy Implementation

```
def move_through_path(self, points):
    # Debugging purposes
    for point in points:
        self.get_logger().info(f'Path: {point[0]}, {point[1]}')

    twist = Twist()
    for row, col in points:
        self.get_logger().info(f'Moving to {row}, {col}')
        while self.currow != row or self.curcol != col:
            # Triggering all callbacks
            for _ in range(3):
                rclpy.spin_once(self)
                time.sleep(0.1)

            self.get_logger().info(f'Current row col: {self.currow}, {self.curcol}')
            self.get_logger().info(f'Target row col: {row}, {col}')
            self.get_logger().info(f'Current Angle: {self.initial_angle}')
            self.visited_points.add((self.currow, self.curcol))

            rotate_angle = 0
            speed = speedchange
            if self.curcol < col:
                rotate_angle = 360 - self.initial_angle
            elif self.curcol > col:
                rotate_angle = 360 - self.initial_angle
                speed = -speed
            elif self.currow < row:
                rotate_angle = 450 - self.initial_angle
            elif self.currow > row:
                rotate_angle = 450 - self.initial_angle
                speed = -speed

            self.get_logger().info(f'Rotate the bot ccw {rotate_angle} degree')
            if rotate_angle != 360:
                self.rotatebot(rotate_angle)
                self.initial_angle = (self.initial_angle + rotate_angle) % 360
            twist.linear.x = speed
            self.publisher_.publish(twist)
    self.stopbot()
```

J Path Clustering Implementation

```
def cluster_path(self, find_path):
    if find_path == []:
        self.get_logger().info('No path is found!')
        return
    clustered_path = []
    clustered_path.append(find_path[0])
    for idx in range(1, len(find_path)):
        point = find_path[idx]
        if self.point_to_point_distance(point, clustered_path[-1]) <=
            cluster_distance ** 2:
            continue
        clustered_path.append(point)
    return clustered_path
```

K Cluster-to-cluster Movement Strategy Helper Functions

```
def point_to_point_distance(self, a, b):
    return (a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2

def calculate_cw_rotation_angle(self, F, T):
    currow, curcol = F
    targetrow, targetcol = T
    coldiff = abs(curcol - targetcol)
    rowdiff = abs(currow - targetrow)
    if currow == targetrow and curcol == targetcol:
        return 0
    if currow > targetrow and curcol == targetcol:
        return 0
    if currow > targetrow and curcol < targetcol:
        return np.degrees(np.arctan(coldiff / rowdiff))
    if currow == targetrow and curcol < targetcol:
        return 90
    if currow < targetrow and curcol < targetcol:
        return 90 + np.degrees(np.arctan(rowdiff / coldiff))
    if currow < targetrow and curcol == targetcol:
        return 180
    if currow < targetrow and curcol > targetcol:
        return 180 + np.degrees(np.arctan(coldiff / rowdiff))
    if currow == targetrow and curcol > targetcol:
        return 270
    if currow > targetrow and curcol > targetcol:
        return 270 + np.degrees(np.arctan(rowdiff / coldiff))
```

L Cluster-to-cluster Movement Strategy Implementation

```
def move_through_path(self, points):
    if points == None:
        return
    twist = Twist()
    self.can_update = False
    self.get_logger().info(f'Path produced by A* Search')
    for row, col in points[1:]:
        self.get_logger().info(f'Row Col: {row}, {col}')
    please_redirect = False
    for row, col in points[1:]:
        if please_redirect:
            self.get_logger().info('Redirecting...')
            break
        self.get_logger().info(f'Moving to {row}, {col}')
        distance = self.point_to_point_distance((self.currow, self.curcol), (row, col))

        while distance > localization_tolerance ** 2:
            for _ in range(5):
                rclpy.spin_once(self)
                time.sleep(0.1)

            self.get_logger().info(f'Current row col: {self.currow}, {self.curcol}')

            distance = self.point_to_point_distance((self.currow, self.curcol), (row, col))

            self.visited_points.append((self.cur_pos.x, self.cur_pos.y))
            while len(self.visited_points) > RESET_VISITED_POINTS_THRESHOLD:
                self.visited_points.pop(0)

            # Calculate the angle between (self.currow, self.curcol) and (row, col)
            map_angle = self.calculate_cw_rotation_angle((self.currow, self.curcol), (row, col))
            map_angle = 360 - map_angle
            self.get_logger().info(f'Head up to the angle of {map_angle}')

            # Rotate the robot to that angle
            current_angle = math.degrees(self.yaw) + 180
            self.get_logger().info(f'Current Angle: {current_angle}')
            relative_angle = (current_angle - self.initial_angle + 360) % 360
            angle_to_rotate = (map_angle - relative_angle + 360) % 360
            self.rotatebot(angle_to_rotate)
            self.get_logger().info(f'Current Angle: {math.degrees(self.yaw)}')

            # Publish the twist
            twist.linear.x = speedchange
            self.publisher_.publish(twist)

        self.stopbot()
    self.can_update = True
    self.get_logger().info(f'Path traversed successfully')
```


M Callback for Updating Map (Initial Approach)

```
def occ_callback(self, msg):
    # find transform to obtain base_link coordinates in the map frame
    # lookup_transform(target_frame, source_frame, time)
    try:
        trans = self.tfBuffer.lookup_transform(
            'map', 'base_link',
            rclpy.time.Time(),
            timeout=rclpy.duration.Duration(seconds=0.1)
        )
    except (LookupException, ConnectivityException, ExtrapolationException) as e:
        print(e)
        return

    cur_pos = trans.transform.translation

    # get map resolution and map origin
    map_res = msg.info.resolution
    map_origin = msg.info.origin.position

    # get map grid positions for x, y position
    grid_x = round((cur_pos.x - map_origin.x) / map_res) # column in numpy
    grid_y = round(((cur_pos.y - map_origin.y) / map_res)) # row in numpy

    received_map = np.array(msg.data).reshape((msg.info.height, msg.info.width))
    adjusted_map = np.fliplr(np.rot90(received_map))

    self.currow = msg.info.width - 1 - grid_x
    self.curcol = msg.info.height - 1 - grid_y
    self.visited_points.add((self.currow, self.curcol))
    self.cur_map = adjusted_map
```

N Callback for Updating Map with Delay Flag

```
def occ_callback(self, msg):
    # find transform to obtain base_link coordinates in the map frame
    # lookup_transform(target_frame, source_frame, time)
    try:
        trans = self.tfBuffer.lookup_transform(
            'map', 'base_link',
            rclpy.time.Time(),
            timeout=rclpy.duration.Duration(seconds=1)
        )
    except (LookupException, ConnectivityException, ExtrapolationException) as e:
        print(e)
        return

    self.cur_pos = trans.transform.translation

    if self.can_update:
        # get map resolution and map origin
        self.map_res = msg.info.resolution
        self.map_origin = msg.info.origin.position
        self.map_width = msg.info.width
        self.map_height = msg.info.height
        received_map = np.array(msg.data).reshape((msg.info.height, msg.info.width))
        adjusted_map = np.fliplr(np.rot90(received_map))
        self.cur_map = adjusted_map
        self.get_logger().info(f'Updating Map!')

    # get map grid positions for x, y position
    grid_x = round((self.cur_pos.x - self.map_origin.x) / self.map_res) # column
    in numpy
    grid_y = round(((self.cur_pos.y - self.map_origin.y) / self.map_res)) # row in
    numpy

    self.currow = self.map_width - 1 - grid_x
    self.curcol = self.map_height - 1 - grid_y
    self.visited_points.append((self.cur_pos.x, self.cur_pos.y))
    while len(self.visited_points) > RESET_VISITED_POINTS_THRESHOLD:
        self.visited_points.pop(0)
```

O Heat Interrupt Routine

```
# Calculate the shortest distance to the shooting areas
shortest_to_shooting_area = math.inf
for x, y in self.shooting_area:
    grid_x = round((x - self.map_origin.x) / self.map_res)
    grid_y = round((y - self.map_origin.y) / self.map_res)
    convertx = self.map_width - 1 - grid_x
    converty = self.map_height - 1 - grid_y
    shortest_to_shooting_area = min(shortest_to_shooting_area, self.
        point_to_point_distance((convertx, converty), (row, col)))
print(f'Shortest distance to shooting area: {shortest_to_shooting_area}')

# If heat is detected and it is not one of the shooting area before, shoot
if self.heat_location != None and shortest_to_shooting_area >
    SHOOTING_AREA_THRESHOLD:
    while self.heat_location == 'right' or self.heat_location == 'left' or self.
        heat_location == 'forward':
        for _ in range(5):
            rclpy.spin_once(self)
            time.sleep(0.1)
        if self.heat_location == 'right':
            self.get_logger().info('Detecting heat in the right')
            self.stopbot()
            self.rotatebot(-HEAT_ROTATE_ANGLE)
        elif self.heat_location == 'left':
            self.get_logger().info('Detecting heat in the left')
            self.stopbot()
            self.rotatebot(HEAT_ROTATE_ANGLE)
        elif self.heat_location == 'forward':
            self.get_logger().info('Detecting heat in front')
            self.stopbot()
            twist.linear.x = speedchange
            self.publisher_.publish(twist)

    if self.heat_location == 'ok' and shortest_to_shooting_area >
        SHOOTING_AREA_THRESHOLD:
        self.get_logger().info('SHOOTTTTT!!! Stop for 9 seconds')
        self.stopbot()
        self.launch_ball_publisher.publish(String(data='ok'))
        self.shooting_area.add((self.cur_pos.x, self.cur_pos.y))
        please_redirect = True
        time.sleep(9)
    break
```

Chapter 10: Safety Protocol

Every component headers and power lugs were checked for connection integrity and then secured with heat-shrink to prevent any possible short circuits when in use. Loose connections with jumper wires were securely taped together. Ensure that no bare wiring is exposed before powering the robot.

In case of testing with an external power source, isolate the component of interest for testing first. For instance, do not externally supply power to the L298N driver's power output pins. Instead, test the isolated motors first, then test the isolated driver+motor circuit.

10.1 Risks and Mitigation Strategies

Risk	Impact	Mitigation Strategy
Heat sensor false positives	Incorrect flare deployment	Implement filtering algorithms or try multi-sensor fusions.
Navigation failure	Robot stuck in the maze	Loop detection & re-routing, Failsafe Mechanisms if there are errors detected trigger a recovery sequence.
Launcher jamming	Missed flare deployment	Retry mechanism & manual reset
Ramp climbing failure	Incomplete mission	Optimize speed control & fallback
Battery depletion	Premature shutdown	Power management & monitoring
System failure	Communication loss leads to abortion of the mission	Create a feedback system and pre-run testing and have backups for all software and hardware appliances.

Chapter 11: Troubleshooting

This section describes some problems faced when putting this project together, and how the team resolved these issues.

11.1 Mechanical Subsystem

The majority of the issues corresponding to the mechanical aspect of the system were loosening of screws and nuts causing stationary parts to move slightly. These were minor and were resolved easily by regular re-tightening of screws and nuts using screwdrivers and pliers.

One major error that had occurred was the ramp piece not being tall enough to allow the ball to roll into the column. This was overcome by adding nuts which acted as 'spacers' to mount the ramp on a slightly higher altitude than the column's base.

11.2 Electrical Subsystem

Early on, the motor pairs didn't spin or could only spin at full speed. There was difficulty in controlling the motor speed because the motor driver enable pins were used instead of the input pins to control the RPM via PWM.

After component testing for the servo, flywheels, and heat sensor, the team tried to integrate the test code into a singular compiled node, but could not compile the code. The team realised that the pin-numbering convention (BOARD and BCM) must be fixed. Since one of the libraries being used had already declared BCM convention, the team updated the schematic and pin-definition header to BCM mapping.

In the case where the robot reboots when activating flywheels, the motors are likely drawing too much power from the OpenCR board. Reduce the flywheel speed to under 75% and try again.

Whenever any component is not working in the integrated system, the team tries to connect it separately to power and check to see if it is a hardware problem or a software problem.

11.3 Software Subsystem

Most of the navigation issues stem from improperly tuned parameter values. Among these, the **wall penalty** parameter is the most sensitive. If set too high, the robot may avoid certain areas of the map entirely, as it cannot find a viable path. On the other hand, if the value is too low, the robot may crash into walls.

For further debugging, the team often resorted to outputting relevant variables, like the scanned map, predestined path and the robot's position. When the robot was behaving unexpectedly, it was helpful to visualize the mapping and path generated by the algorithm, and check how the robot was following the intended path generated.

Chapter 12: Future Scope of Expansion

12.1 Mechanical Aspects

- Exploration of other storage and launching mechanisms, such as a compact ‘revolver’ storage system and the solenoid plunger launcher (as described earlier in [Section 2.2](#)). If designed well while adhering to mission constraints, they could improve the stability of the TurtleBot with a lower centre of gravity, and decrease the size of the system as well to ensure it does not collide with maze boundaries.
- Additionally, since the team faced challenges in assembling the system and inserting screws in places difficult to access, components could instead be ‘designed for assembly’ for more convenience when mounting onto the robot.

12.2 Electrical Aspects

- Implement the use of Inertial measurement unit (IMU) in the OpenCR board to enable knowledge on its location and orientation relative to the ramp. This would help the robot to conduct the mission in the ideal “ramp last” order of survivor rescue. It will also facilitate climbing and descending the ramp, and enable the robot’s environment mapping in 3D.
- Upgrade the robot battery (higher C-rate and peak discharge current) to cater to the high power draw by the flywheels. It can also be good to get a battery of higher capacity, for extending the running time of the robot if needed.
- Make a custom PCB to make wires compact, centralised, and fast to troubleshoot connections. Try to replace the jumper wires with soldered paths to avoid loose connections.

12.3 Software Aspects

- Train a deep learning or reinforcement learning model to accurately tune navigation parameters. The model can initially be trained in a simulated environment, where it evaluates parameter adjustments based on the robot’s performance. For instance, if the robot crashes, the system can penalize the parameters that likely caused the failure, gradually learning more effective configurations. Once sufficiently trained, the model can be deployed in real-world scenarios, where it will continue running and

dynamically tuning the parameters. This will also allow the robot to adapt and improve its navigation accuracy over time based on actual operating conditions.

- Integrating more heat sensors to capture thermal signatures from multiple angles - to help ensure that no victims are missed due to limited sensor angle coverage.
- Implement a robust emergency stop mechanism that immediately halts the robot when a potential collision or unsafe situation is detected. For a realistic application, add a manual override feature that allows human operators to take control in such scenarios. In essence, build a comprehensive safety system to ensure human intervention is possible whenever the robot behaves unpredictably.
- Leverage computer vision techniques to improve the detection of earthquake victims in complex and cluttered environments in addition to using heat sensing.

References

Datasheets

MOTORS:

<https://docs.rs-online.com/5307/A700000008719335.pdf>

SERVO:

http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf

AMG8833 HEAT SENSORS:

<https://cdn-learn.adafruit.com/downloads/pdf/adafruit-amg8833-8x8-thermal-camera-sensor.pdf>

<https://industrial.panasonic.com/cdbs/www-data/pdf/ADI8000/ADI8000C66.pdf>

Other references:

1. Navigation:

<https://www.geeksforgeeks.org/a-search-algorithm/>

<https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/>

2. Electrical system:

<https://makersportal.com/blog/thermal-camera-analysis-with-raspberry-pi-amg8833?srltid=AfmBOorFDGJ3zT1wy2NTNb0NzRN8hEgu7Amaab3JhcH3OeEOSt958FBO>

3. Sensor datasheet:

<https://cdn-learn.adafruit.com/downloads/pdf/adafruit-amg8833-8x8-thermal-camera-sensor.pdf>

4. Flywheel Launch Mechanism

<https://forums.adafruit.com/viewtopic.php?t=17638&utm>

<https://www.youtube.com/watch?v=UJjsCS4e370>

<https://www.amazon.sg/Uxcell-Plunger-Solenoid-Electromagnet-Actuator/dp/B0173B72OK>