CDE2310 G2 Final Report

Group 6

Name	Matriculation No.
Chong Kai Jie	A0306749U
Diwakar Vidya Adhavan	A0312132B
Li Zhuangda	A0308781X
Toh Ee Sen, Izen	A0309181H
Zhang Yuening	A0301319R

https://github.com/cde2310grp6

CDE2310 24/25 Sem 2 Group 6

Special Thanks

Mr Chew Wanlong, Nicholas

Mr Ee Wei Han, Eugene

Mr Royston Shieh Teng Wei

Mr Soh Eng Keng

Teaching Assistants of CDE2310

1.0: Introduction

1.1: Problem Definition

This report is a summary of the system design process of our (Group 6's) robot, developed to meet the mission requirements of the NUS CDE2310 module, taught during Semester 2 of Academic Year 2024/2025.

The mission outline, meant to simulate a search and rescue mission in an earthquake-affected zone, is as follows:

Develop a system capable of autonomously navigating a maze, identifying two randomly placed heat sources (survivors) within the maze, and fire three ping pong balls (flares) at intervals of 2-4-2 seconds. A third heat source, featuring a ramp (in a known location in the maze), offers additional points for teams that manage to scale it using the provided 2D lidar, however, this is not mission-critical. Teams have 25 minutes to complete the mission. Line-following navigation is prohibited.

Below is a table listing the requirements we planned to achieve throughout the mission.

Feature	Requirement
1. Fully autonomous robot	A. Using LiDAR sensor to map the surroundings. B. Able to detect heat signals. C. It must not collide with any obstacle. D. Robot needs to approach the heat signal proximity. E. Does not navigate to the same heat signal and fire flares. F. The robot needs to be able to be fully autonomous and not remote controlled.
2. Fire flares to signal S&R teams for immediate survivor assistance	A. Launching mechanism must be able to fire 3 flares at set time interval B. Flares need to be self reloadable as it is fully autonomous C. Flares need to be fired high enough for S&R teams to be able to see.
3. Power system (battery, rasp pi & other electronic components)	A. Battery needs to last long enough for the whole mission. B. Needs to power the launching mechanism, rasp pi, motors, LiDAR and OpenCR. C. Motors require a sufficient amount of battery remaining or it will stop working.
4. Mechanical structure	A. Robot needs to have a relatively low and central center of gravity such that it can navigate uneven areas (such as ramps) without toppling over. B. Robot needs to be not bulky as it needs to navigate through tight spaces. C. Robot needs to be mounted with a launching mechanism with a magazine for flare firing.

1.2: Mission Overview

In view of the mission objectives, our system is required to have the following features:

- Autonomously navigate an unknown maze using an appropriate navigation algorithm implemented in the Robot Operating System 2 (ROS2).
- An effective payload delivery mechanism to launch ping-pong balls at the appropriate places and at the correct time intervals.
- A thermal camera in order to relay infrared (IR) data to the on-board Raspberry Pi 4B and detect the heat sources (survivors).
- Be able to seamlessly integrate the above systems to work together to accomplish the mission objectives.

Our system design process is a single iteration of the CDE2310 V-Model of Systems Design, illustrated below:

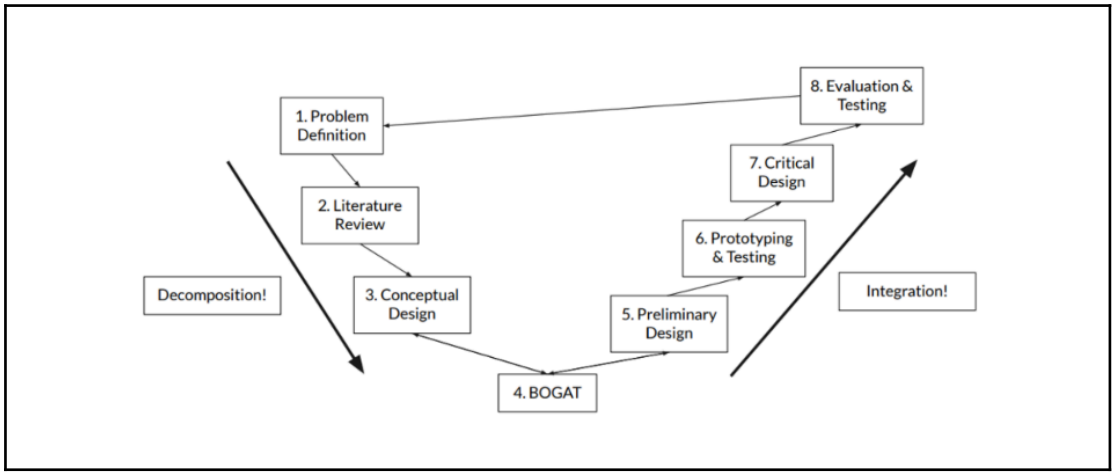


Fig. 1.2: NUS CDE2310 System Design V-Model

1.3: Evaluation Criteria

1.3.1: Maze Mapping and Heat Source Detection

No.	Scoring Item	Max Score
1	Robot leaves maze start zone.	10
2	Robot able to detect the first heat source.	5
3	Robot successfully fires 'flares' in 2-4-2 second delay pattern above maze walls: (1.5 pts for each ping pong ball firing with 3 successes given a total 5)	5
4	Robot achieves firing of first flare sequence of up till 1.5m target: (1.5 pts for each ping pong ball firing with 3 successes given a total 5)	5
5	Robot able to progress and correctly identify second heat source.	5

6	Robot successfully fires 'flares' in a 2-4-2 second delay pattern. Robot achieves firing of flare sequence of up till 1.5m target (5 points each with similar breakdown as above)	10
7	Successfully complete mission on first attempt. (No reattempts)	10

1.3.2: Hardware

No.	Scoring Item	Max Score
1	Robot has sufficient battery power for entire mission (No “Low Battery” warning or “battery dead” situation)	5
2	System is mechanically stable during operation (proper weight balancing)	5
3	Structure and components are secured during the operation (no loose parts or dropping parts including payload or components)	5
4	System is well assembled (logical sequence and no missing fasteners)	5

1.3.3: Bonus

No.	Scoring Item	Max Score
1	Robot able to traverse ramp with no collisions	5
2	Robot able to detect 3rd heat source and initiate firing sequence	10
3	Robot able to fire all 3 “flares” with no jamming. Projectile height does not matter here. (1.5 pts for each ping pong ball firing with 3 successes given a total 5)	5

1.3.4 Competitive Scoring

No.	Scoring Item	Max Score
1	Mission Complete on Time	30

1.3.5: Penalty

No.	Penalty Item	Max Score
1	Any part of the system (i.e. Robot, markers) damaged or displaces maze element. 5 points per displaced element, 50 points for damages.	Up to -100
2	Mission Overtime. -5 points per minute	Up to -100

2.0: Literature Review

2.1: Maze Exploration Algorithms

Maze exploration algorithms are methods used to explore a maze. These algorithms can be broadly categorized into algorithms that work with unknown mazes and those that work with known mazes.

- **Random Mouse Algorithm:** A simple method in which the bot will make random decisions for direction at each junction. This algorithm does not require any memory and will eventually find the solution. Theoretically, this algorithm will find the exit to a maze if given infinite time. However, this is extremely inefficient for maze solving, as there is a tendency to backtrack as there is no memory [1].

- **Hand On Wall Rule:** also known as the left-hand/ right-hand rule. The robot will keep moving in the same direction if there are walls detected on the left/right. This method ensures that the robot will not get lost. However, it may not be suitable for this course as the maze is not fully connected [1].

As our task is to map the environment and complete tasks within, traditional maze exploration algorithms

2.2: Coverage Path Planning Methods

Coverage path planning (CPP) algorithms are used to cover an area to ensure that every point has been visited. It is highly relevant in this use case, to autonomously map the entire environment using SLAM.

Cell decomposition:

- **Boustrophedon Decomposition:** Divides the environment into simple cells that can be covered with back-and-forth motions [2].

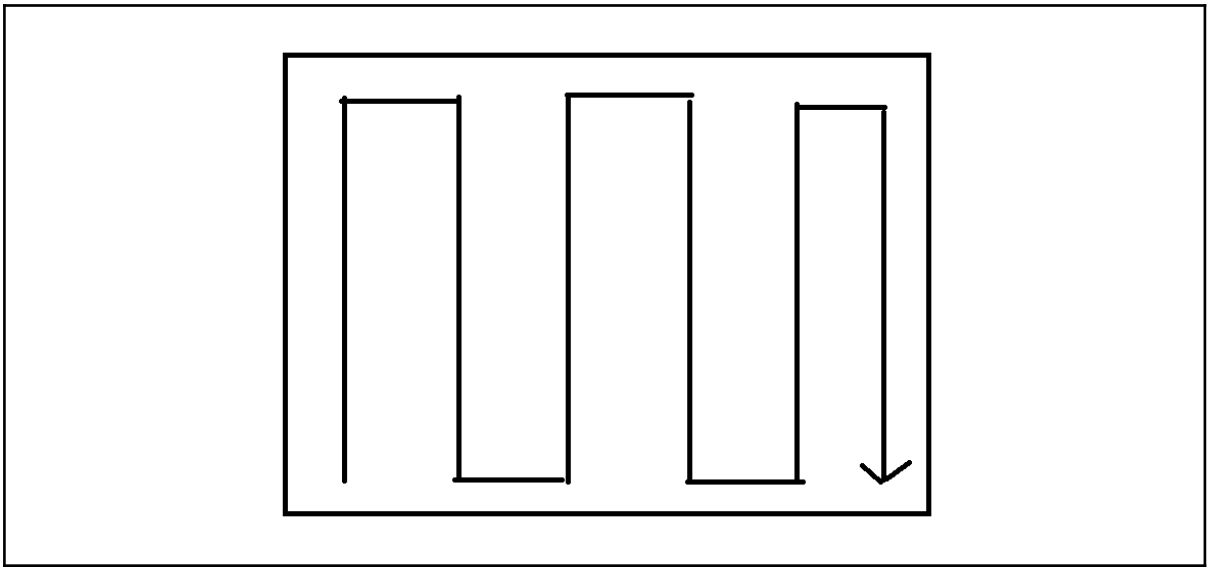


Fig 2.2a: Boustrophedon Decomposition in a rectangular environment

- **Trapezoidal Decomposition:** Partitions the area into trapezoidal sections for systematic coverage [2].

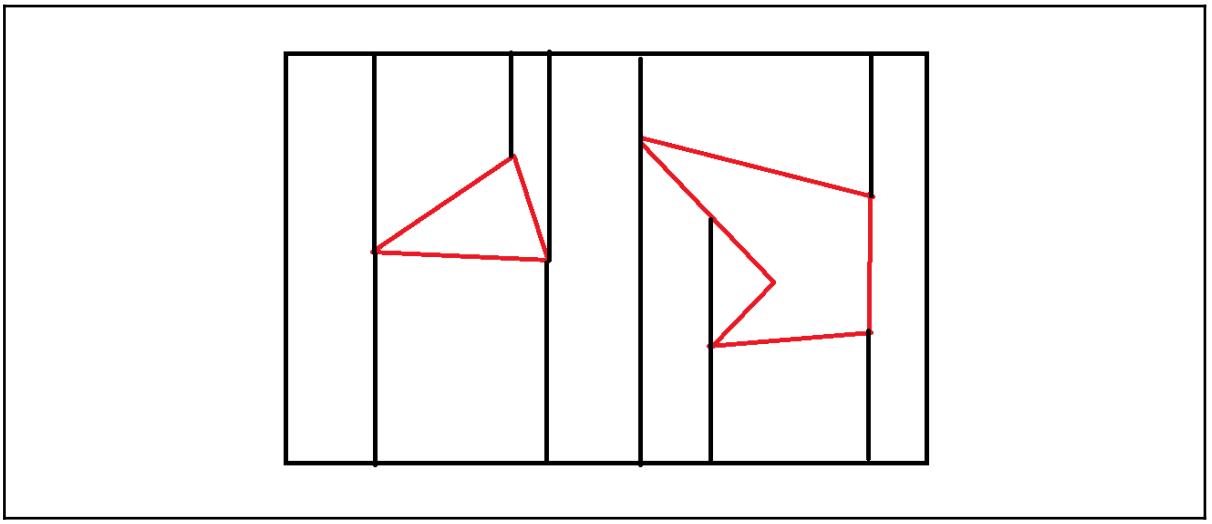


Fig 2.2b: Trapezoidal Decomposition in an environment with obstacles

Graph-based methods:

- **Spanning Tree Coverage (STC):** Constructs a spanning tree over the area and generates a path that covers all nodes [2].

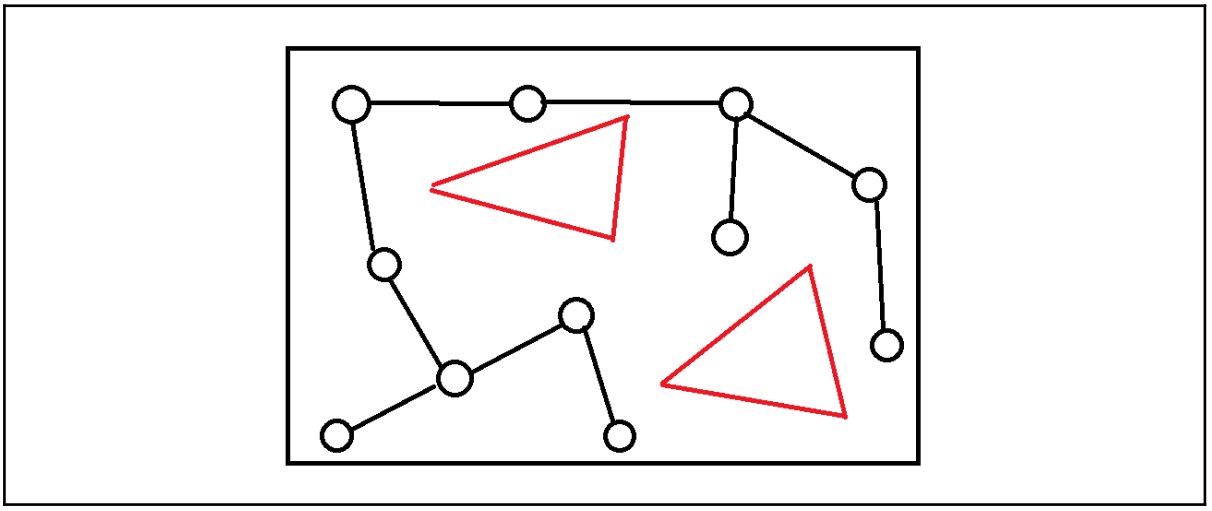


Fig 2.2c: STC in an environment with obstacles

- **Turn-Minimizing STC:** An enhancement of STC that focuses on reducing the number of turns, improving efficiency in multi-robot scenarios [3].

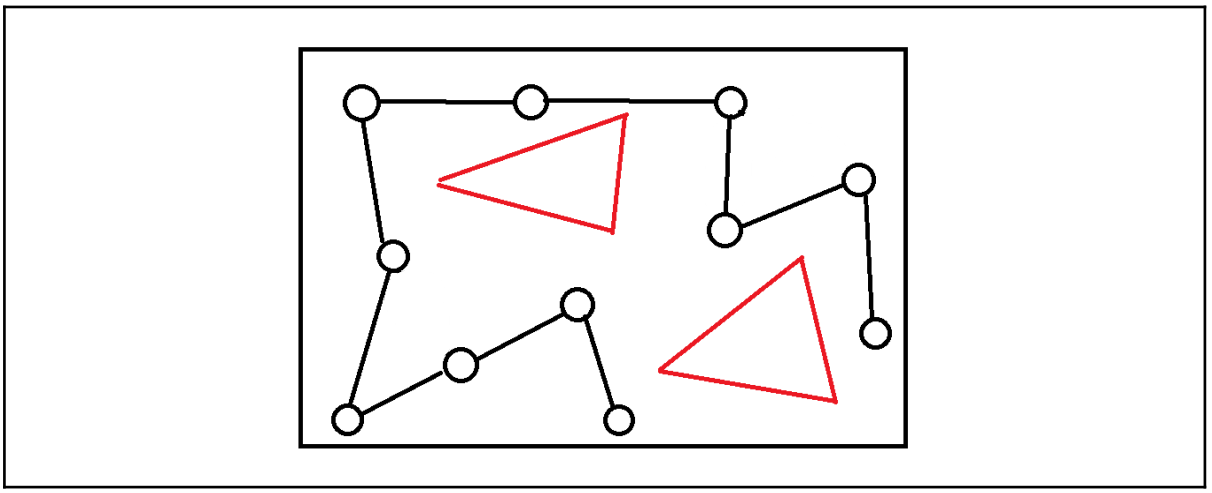


Fig 2.2d: Turn-Minimising STC with optimisations from the previous figure.

Sampling-based algorithms:

- **Probabilistic Roadmap (PRM):** Samples random points in the environment to build a roadmap for path planning [4].

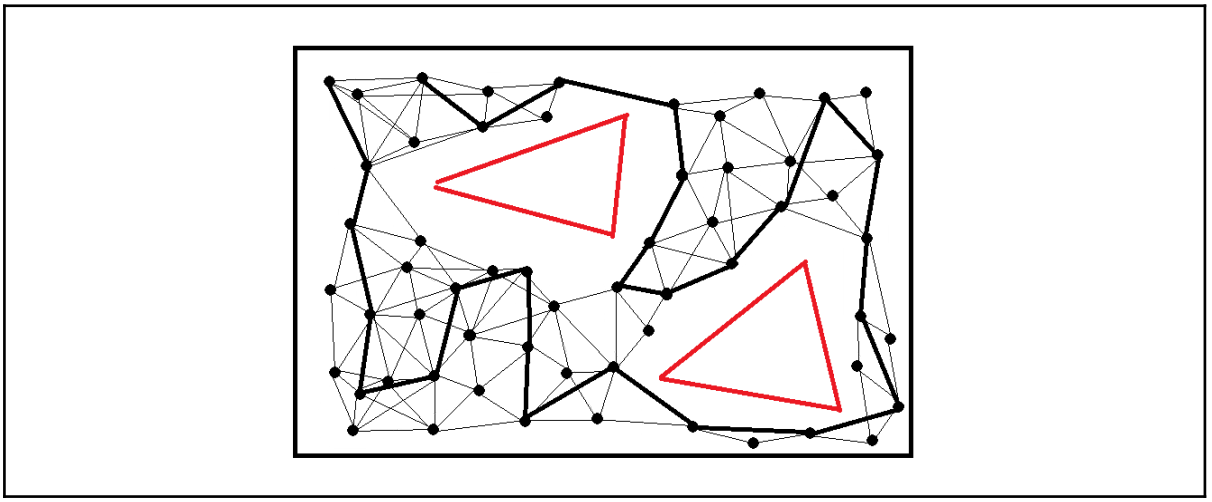


Fig 2.2e: PRM with path optimised for coverage in an environment with obstacles.

- **Rapidly-Exploring Random Trees (RRT):** Efficient and fast algorithm that searches high-dimensional spaces by building a tree that explores the space randomly [5].

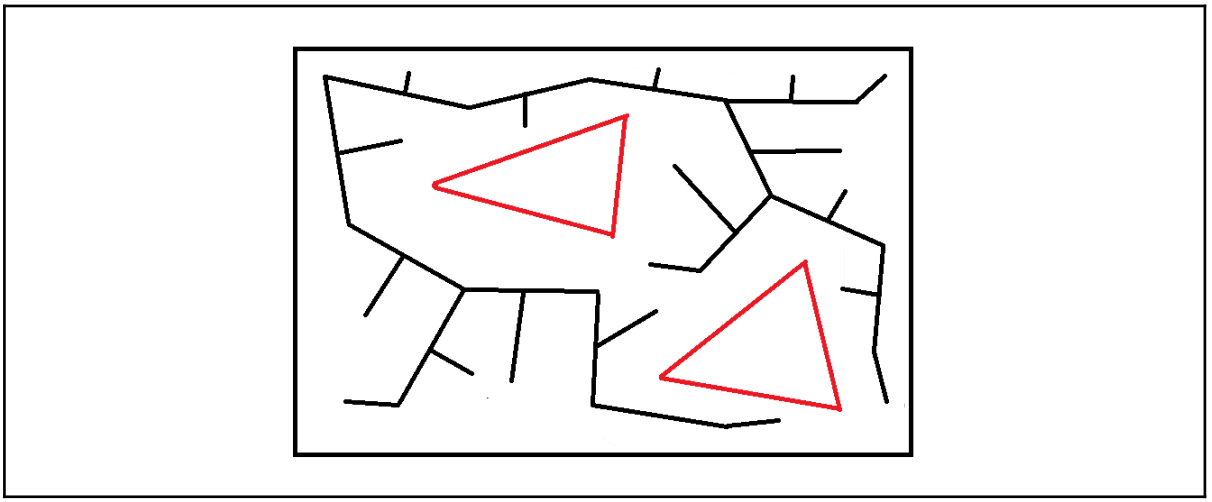


Fig 2.2f: RRT with 2 iterations in an environment with obstacles

The aforementioned methods of CPP are more applicable to known environments. For example, by a robot measuring ground temperature in a known environment with static obstacles. In our case, we require an expiration algorithm that helps explore and map an unknown environment.

Frontier-based exploration:

Focuses on exploring the boundary between known and unknown areas, guiding the robot to new, unexplored areas. Frontiers are chosen to maximise information gain while weighing against factors such as distance or reachability as cost [6].

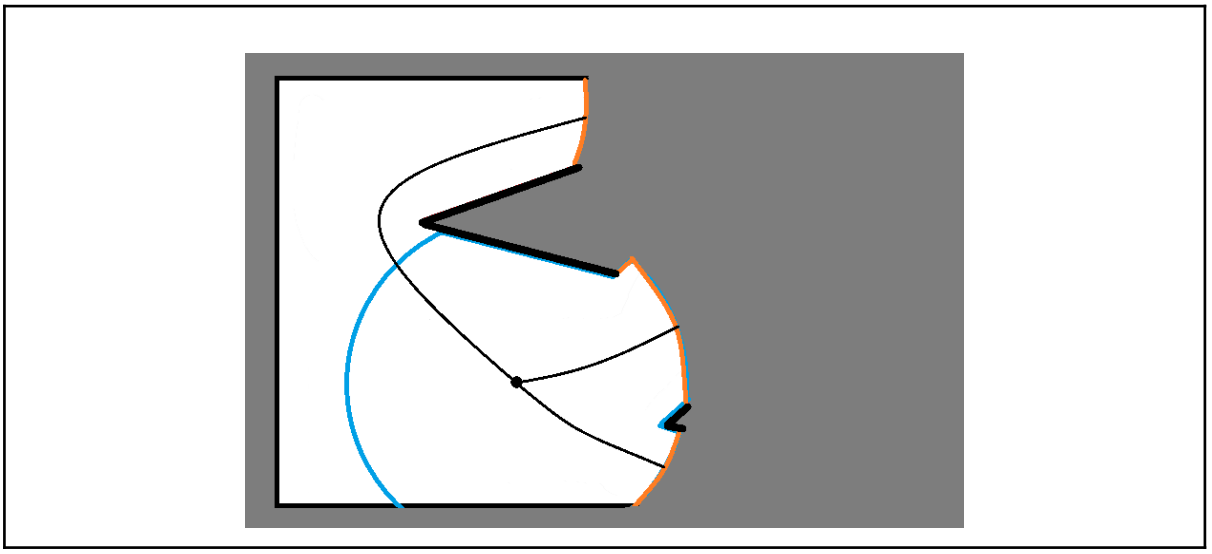


Fig 2.2g: Frontier-based exploration, where orange regions represent the frontiers to be explored, and thin black lines represent potential paths to take

Frontier-based exploration is most applicable to our use case in exploring an unknown environment, to ensure all obstacles and free space are properly mapped. Thus we decided to make our exploration phase frontier-based.

2.3: Thermal Mapping Methods

Thermal mapping methods used to measure and record temperature data across a map by combining thermal sensor readings with the robot's position.

- **Grid-Based Thermal Mapping:** Divide the map into a 2D grid and record the temperature values in each coordinate using the sensor readings as the robot moves. This method is not used as we chose a sensor that's able to detect heat sources in one direction instead of the point of the sensor to increase efficiency.
- **Ray-Casting Triangulation:** Whenever a heat source is detected, record a straight line that connects the heat source and the sensor. Heat source is located if more than 3 lines intersect at one point.
- **Simple Ray-Casting:** The robot will explore the map first using other exploration algorithms mentioned above. It then uses its current position and yaw to cast a ray and apply a temperature value to occupied cells on the occupancy grid to form a heat map.

2.4: Payload Delivery Mechanisms

Payload delivery mechanisms are methods used to transport and accurately release or place objects at target locations, which in this case firing ping pong balls.

- **Spring-loaded launcher:** a compressed spring is used to store mechanical energy, which is released at the instance to eject the payload. It's simple and fast, usually triggered by a latch controlled by a servo or solenoid. However, reloading the spring is needed between every shot, which is not as desired for short-interval launching.

- **Flywheel launcher:** one or two fast-spinning wheels to grip and propel the payload forward through friction. This allows continuous launching and speed control by adjusting motor RPM.

- **Compressed air:** Compressed air is stored and released behind the ball, forcing it out of the barrel. This method allows for very fast and powerful projectiles. In this case, however, compressing and storing air for the mechanism would take up too much space and be too heavy. It would also be time consuming to ensure air-tightness of the system.

2.5: System Integration

System integration in robotics involves connecting various hardware and software components to function cohesively. This includes integrating sensors, actuators, localization systems, navigation algorithms, and control mechanisms to enable the robot to operate effectively in real-time environments.

TurtleBot3 Platform: TurtleBot3 is an open-source, low-cost mobile robot designed for education and research. It features a modular design, allowing users to customize hardware components such as the single-board computer (e.g., Raspberry Pi or NVIDIA Jetson Nano), sensors like 2D LiDAR, and actuators. The platform supports ROS2, facilitating the development and integration of various robotic applications.

ROS2: ROS2 is a set of software libraries and tools that help build robot applications. It offers improved performance over its predecessor, ROS1, including real-time capabilities and better support for multi-robot systems. ROS2 uses a publish-subscribe communication model, where nodes can publish messages to topics or subscribe to receive messages, enabling modular and scalable system integration [7].

Sensor Integration: Integrating sensors is crucial for a robot's perception of its environment. Common sensors used with TurtleBot3 include:

- **LiDAR**: Used for mapping and obstacle detection. The LDS-02 included with the turtlebot3 is compatible with ROS2 through packages like ld08_driver [8].

- **Cameras**: For visual perception tasks, cameras can be integrated with turtlebot3 and ROS2. The Autorace package written for the turtlebot3 makes use of cameras and computer vision to autonomously control the turtlebot [9].

- **IMU and Wheel Encoders**: Provide data for odometry and localization. These sensors can be integrated using appropriate ROS2 drivers and packages.

Navigation Stack Integration: The Navigation2 (Nav2) stack in ROS2 enables autonomous navigation capabilities, including path planning, obstacle avoidance, and localization. Integrating Nav2 with TurtleBot3 involves configuring the robot's sensors and actuators to work with the navigation algorithms. Detailed tutorials are available to guide users through this process .

3.0: Conceptual Design

Below is an overview of the concept design.

Outcome	**Technologies/Methods Required**
Integration of heat sensors with robot	1. Sensor fusion with LiDAR point cloud 2. Integration with cartographer to produce a map with walls coloured using thermal data
Ability to map environment autonomously	1. Frontier-based exploration and Monte-Carlo Localisation 2. LiDAR point cloud, IMU data and Odometry
Ping-pong ball launcher	1. Flywheel-based ball launcher 2. Loading mechanism (spring/gravity fed)

3.1: Mechanical Concept Design

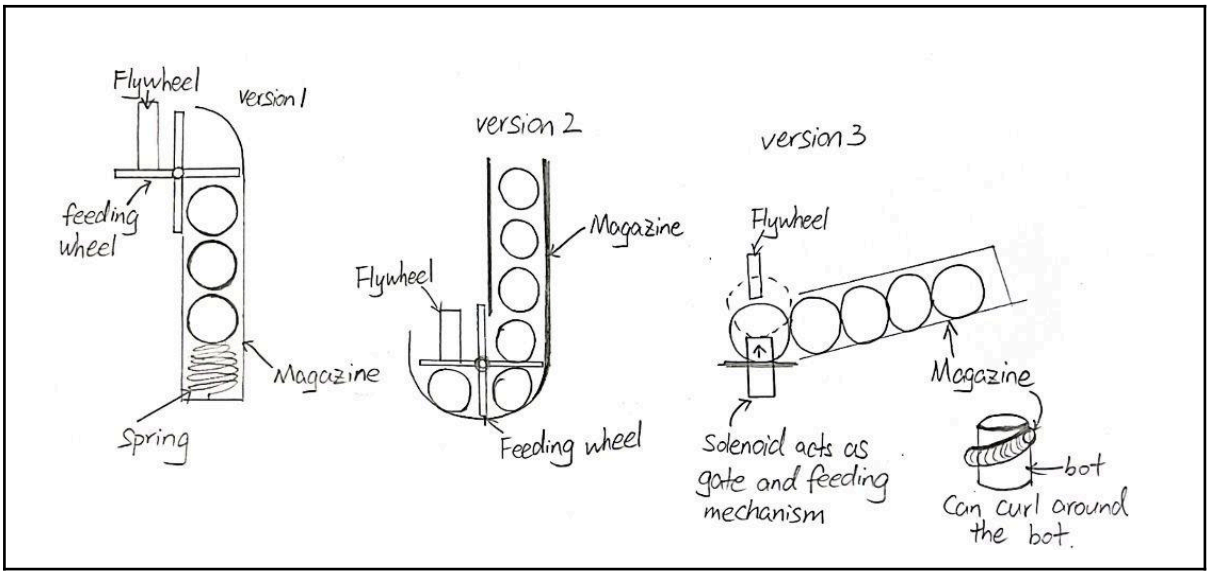


Fig. 3.1a: Sketches of Ping-pong Ball Loading Mechanism

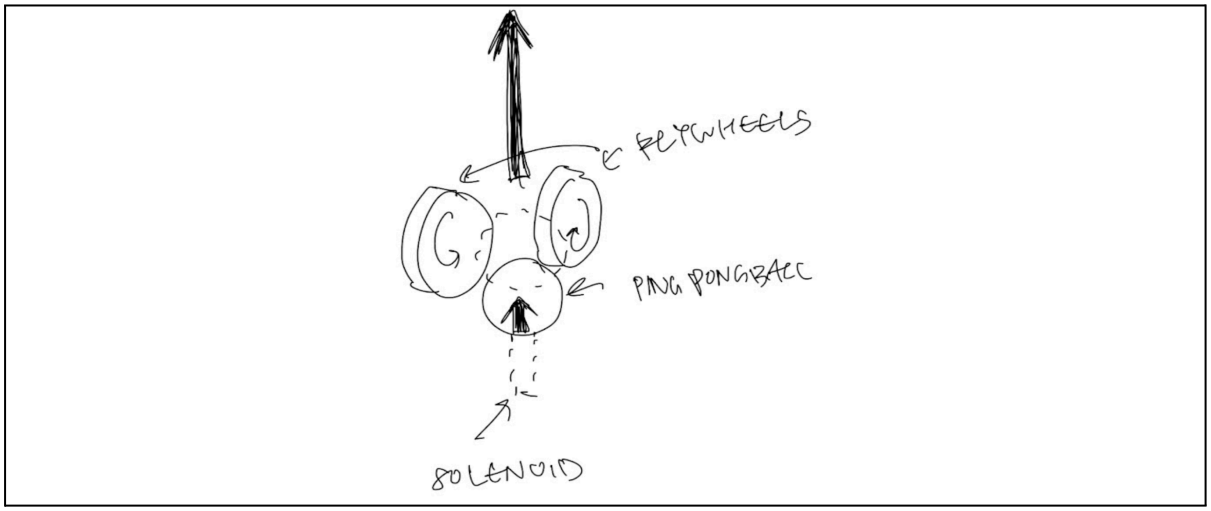


Fig. 3.1b: Sketch of Flywheel Mechanism

For carrying the ping-pong balls, we intended to 3D print a tube-shaped magazine (see Fig. 3.1a) that will wrap around the TurtleBot, ensuring that the ping-pong balls are carried securely while also not interfering with any of the TurtleBot's other systems (e.g LiDAR sensor). The launching mechanism will comprise of two flywheels, inspired by the work of Engineering Dads [10], with more detailed design references developed by Butler N [11]. Finally, a solenoid actuator located at the intersection of the holder and the flywheels will be programmed to feed the ping-pong balls into the flywheels when the TurtleBot is in the correct position and at the correct time intervals. We chose flywheels because they were the simplest to implement, with fewest moving parts and not requiring high levels of precision.

3.2: Electrical Concept Design

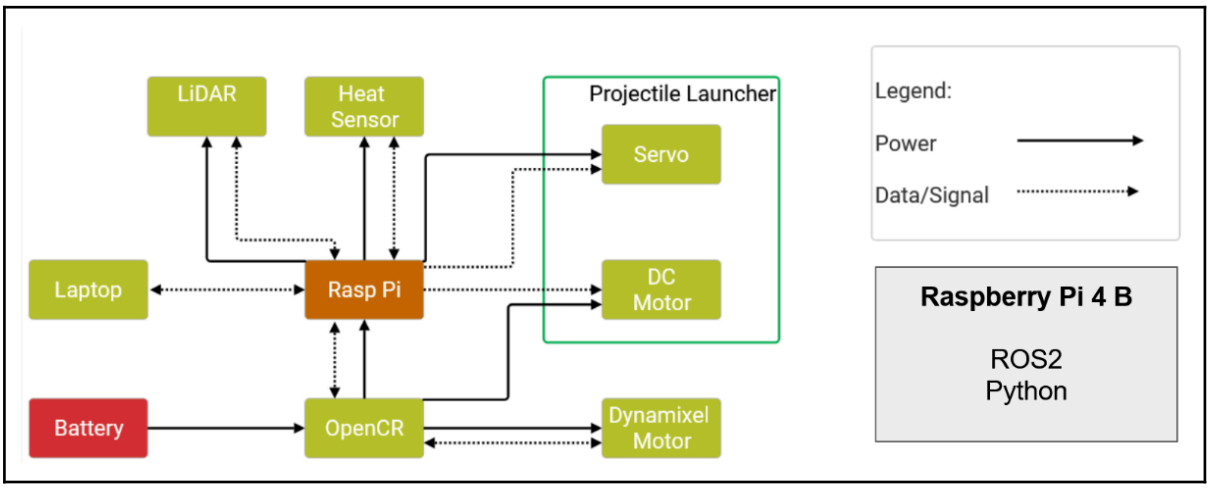


Fig. 3.2: Electrical Architecture

Apart from the original TurtleBot, we added 2 DC Motors, 1 Servo and 1 Heat Sensor for the projectile launcher subsystem. The Servo is controlled by PWM from the Raspberry Pi and only needs 5V to operate so it will be connected directly to the pins of the Raspberry Pi. The DC Motor is also controlled by PWM from the Raspberry Pi but requires 12V to power so it will be connected to the OpenCR for power and Raspberry Pi for signal. Lastly, the heat sensor is only connected to the Raspberry Pi since it is giving data directly to the Raspberry Pi and is connected to the 3.3V output of the Raspberry Pi. A circuit involving the 2 DC Motors will be soldered as we intend to use a MOSFET to help control the motors.

3.3: Software Concept Design

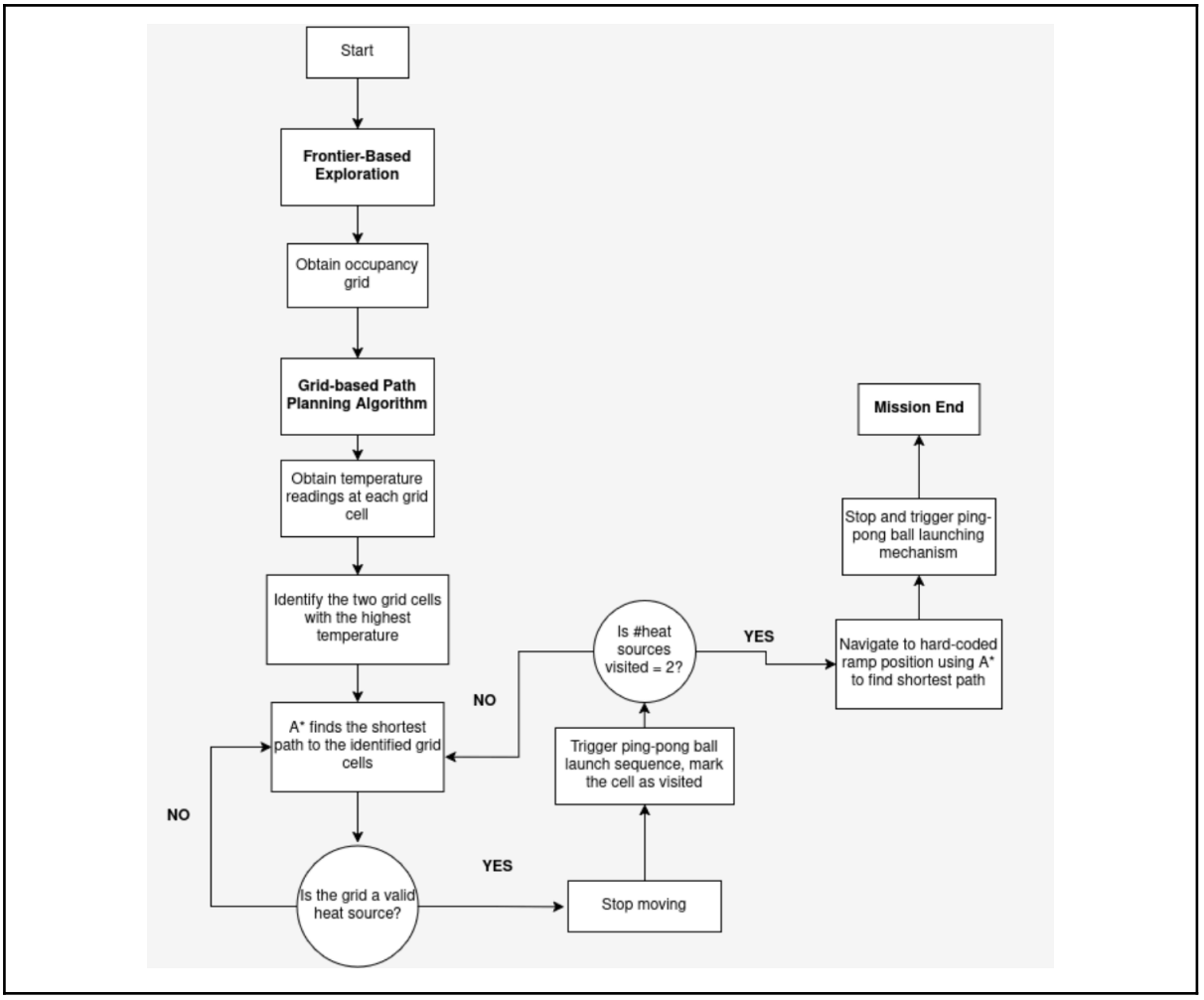


Fig 3.3.1: Flowchart showing the software concept

The mission concept is shown in the figure above. Firstly, the robot conducts a frontier-based exploration to obtain the occupancy grid of the environment. After that is obtained, the robot will use a grid-based path planning algorithm to travel the maze and gather temperature readings to populate the occupancy grid as a heat map. The system will then find the 2 cells with the highest temperature reading and then plan a path to travel to the heat source. Finally, it will travel to a hard-coded location where the ramp is and carry out the bonus objective.

4.0: Preliminary Design

This section details our preliminary design sketches and plans. It is intentionally structured to mirror 6.0 Final Design in order to highlight the key improvements and lessons learnt from Preliminary Design to Final Design.

4.1: Mission Control

Overall, our robot will proceed with the mission in 3 phases. 1. Search 2. Rescue 3. Final Task. This is detailed in the following table:

Phase	Activity
1. Search	Frontier-based exploration on the TurtleBot3 architecture [6] is well researched. This is crucial for the robot to obtain the full layout of the disaster area, taking notes of the positions of the casualties for later. The exploration algorithms are further discussed below in section 4.4.
2. Rescue	The robot navigates towards each of its objectives in a systematic matter, choosing its path based on the A* path finding algorithm [12]. Upon reaching each task, it will fire the flares (ping-pong balls), following the predetermined interval as per S&R SOPs.
3. Final Task	The robot navigates towards the exfiltration point (ramp) and scans for the final casualty, before firing its final flare and completing the mission.

4.2: CAD Concepts and Sub-assemblies

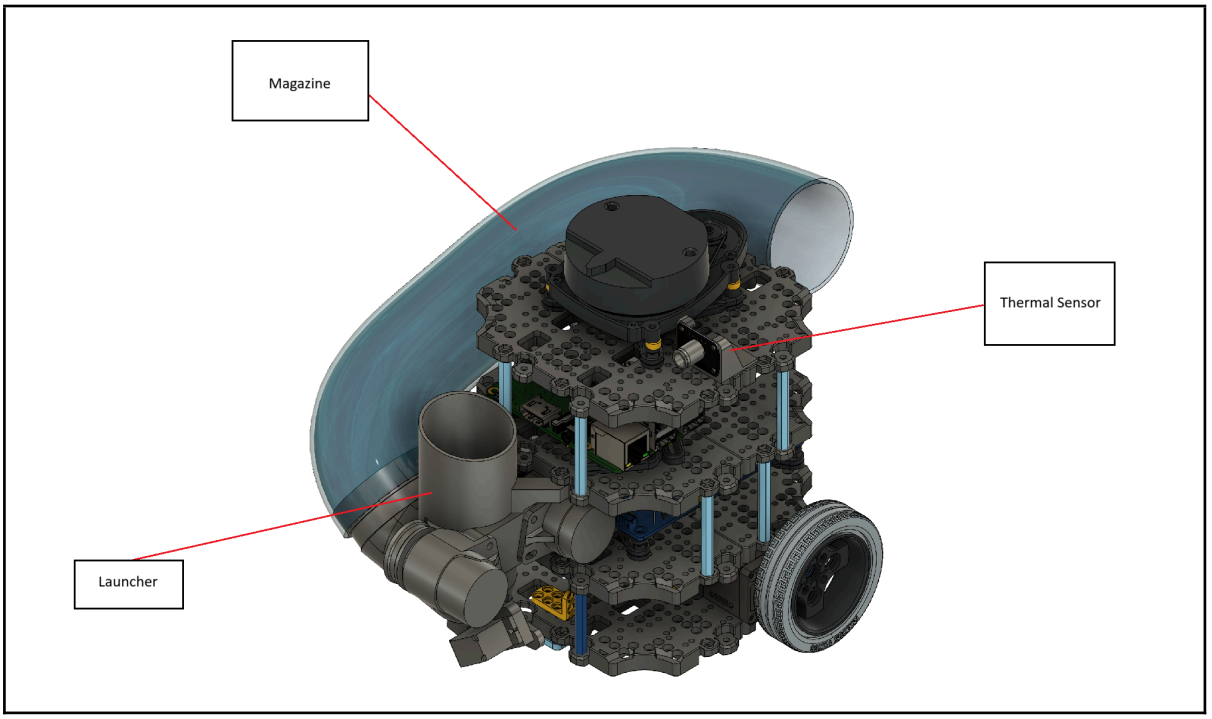


Fig 4.2: Concept submitted for Preliminary Design Report (PDR). The names of labelled components will be how they will be referred to in this report.

The launcher was placed at the back to cause as little impact on the position of the center of mass as possible, with the magazine wrapping around the bot. The thermal sensor is shown to be facing backwards because at this point in time, our dynamixel motors were swapped. As such, the bot travelled backwards and we assumed that that was the direction of its front. These are shown in the figure 4.2.1 above.

4.2.1: Launcher

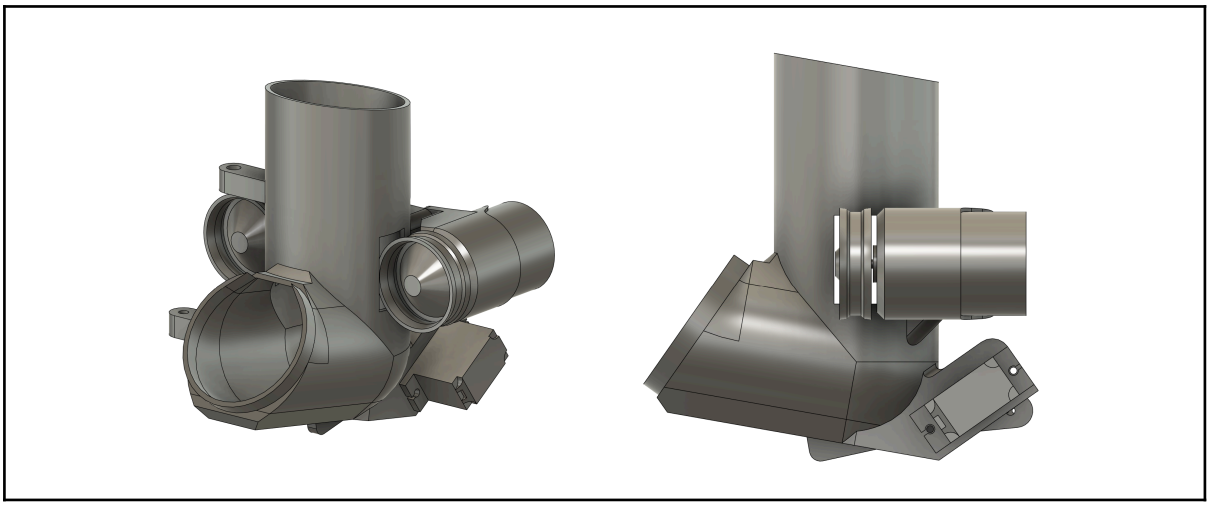


Fig 4.2.1a: Launcher sub-assembly. Three-quarter view, and side view.

Slots for distance adjustment for the flywheels were added at this stage, so that the contact between the flywheels and the ball could be tuned. (Figure 4.2.1a)

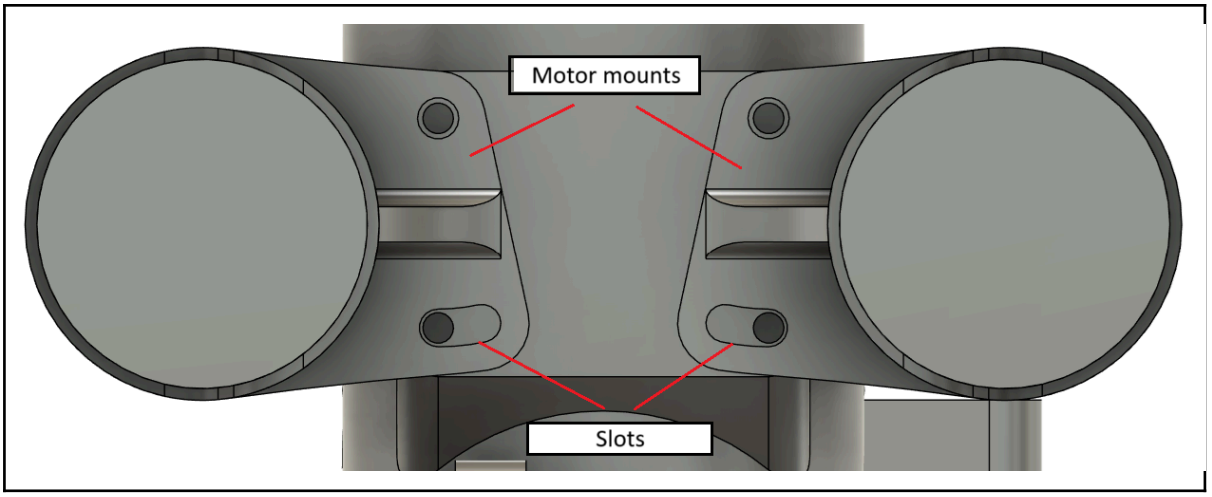


Fig 4.2.1b: Screenshot showing the distance adjustment slots on the motor mounts

In the conceptual design section, it was mentioned that we would use a solenoid to push the ball up into the flywheels. From our testing, we found out that solenoids do not have much pushing force and can be easily overcome. In addition, solenoids are bulky and would reduce the ground clearance of the bot significantly if we were to use one. Thus, we chose to use a servo to push a ball up into the flywheels, as well as stop the remaining balls from rolling onwards. The servo and arm is shown in the figure below.

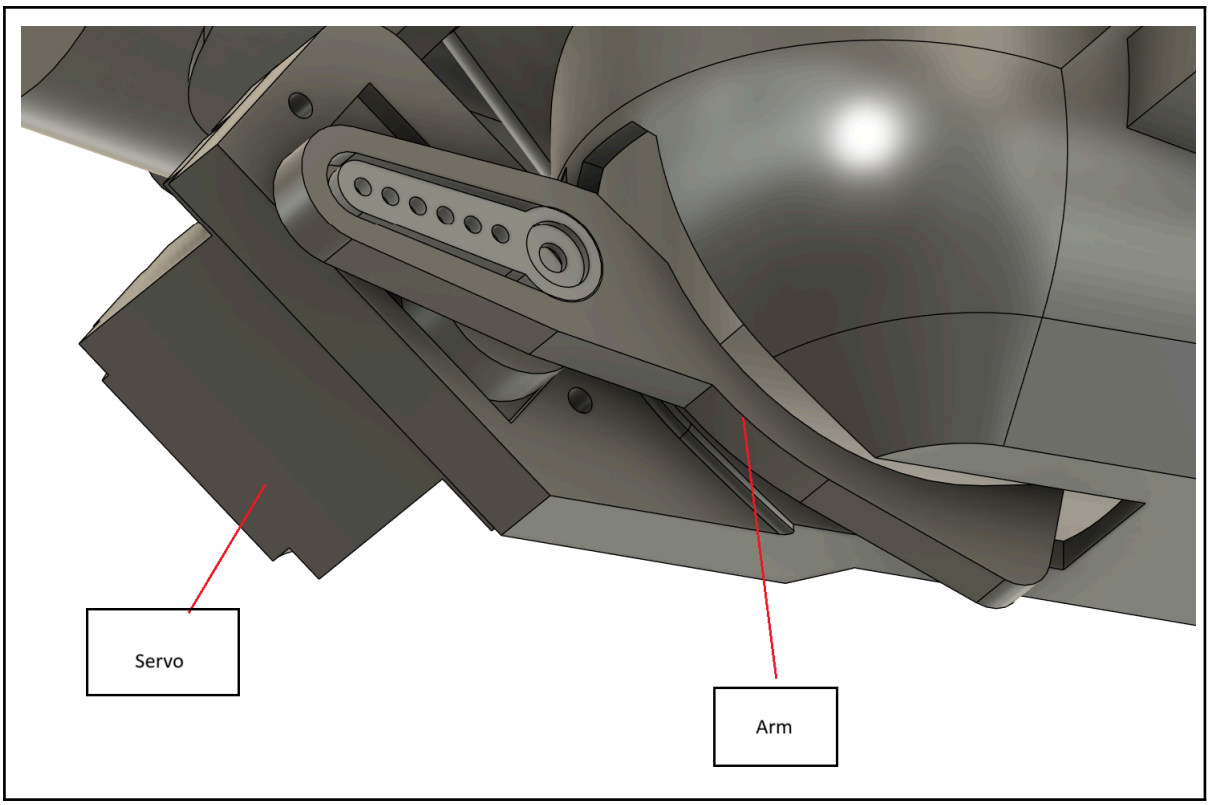


Fig 4.2.1c: Labelled diagram showing the locations of the servo and the arm.

Servo arm: The servo arm was specifically designed to be able to push a single ball up into the flywheel while preventing the next ball from rolling onwards. Its function is shown in the figure below.

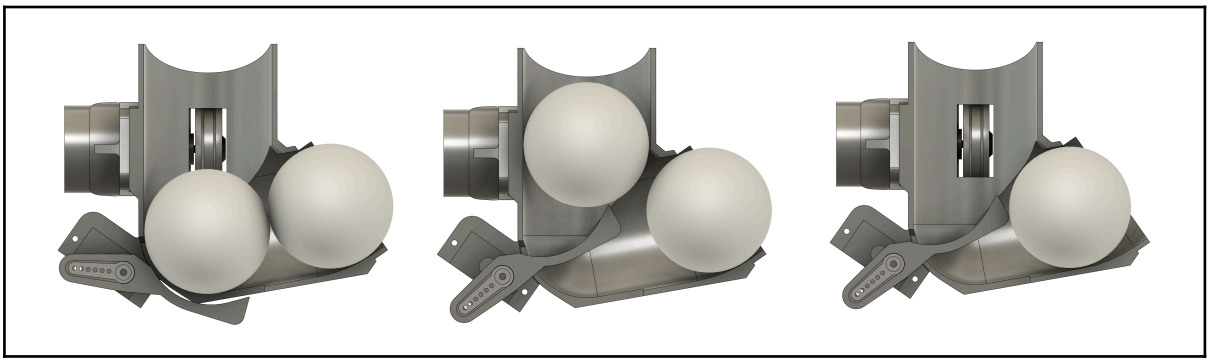


Fig 4.2.1d: Cross sections showing the movement of the servo arm. Note: the launcher shown here is from the final version, but the servo arm design for both the PDR and final versions is the same

Tilted base: An early design (pre-PDR) had a potential design flaw where it required a minimum of 2 balls to fire, as it had a flat base where the final ball would likely not roll down to the servo arm from its own gravity. The bottom of the launcher is thus tilted 10 degrees, and required being 3D printed on an angle, shown in the figure below.

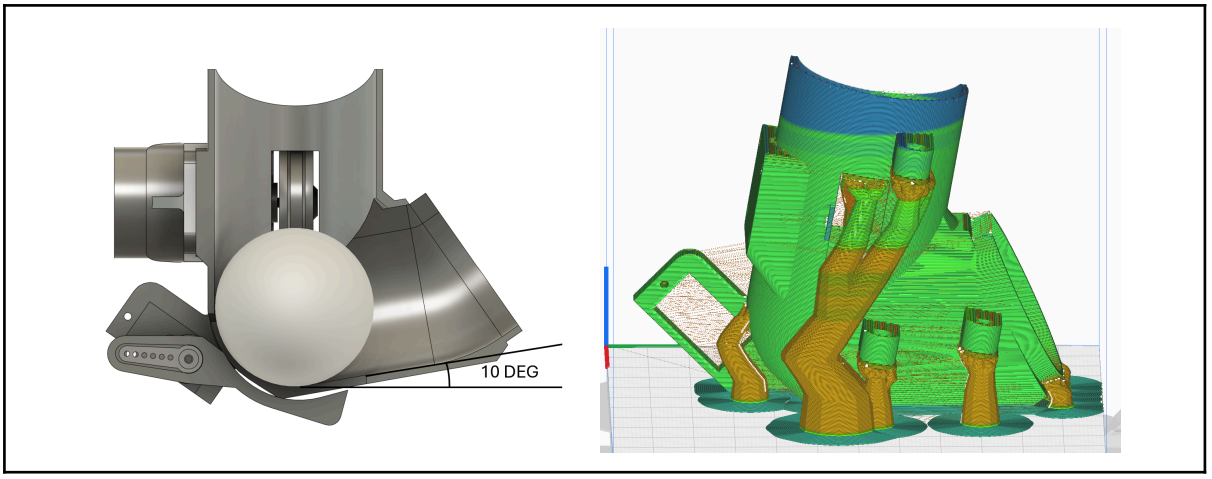


Fig 4.2.1c: The angle of the tilt in the launcher, and the slicer preview of the part Note: the launcher shown here is from the final version, but tilt for both the PDR and final versions is the same

Motor and Flywheel Calculation:

In order to create the most compact launcher mechanism, put as many components in the spaces inside the robot. The gap between the waffle plates is around 40mm. Thus the motor and flywheel must have a diameter smaller than 40mm.

Target = 1.5m

Assuming 50% of kinetic energy wasted by air resistance, calculate using h = 3m

Mass of ball = 2.7g

Flywheel radius = 15mm (to fit between waffle layers with space for adjustment)

mgh = $\frac{1}{2}mv^2$

v = $\sqrt{2gh} = \sqrt{2 \times 9.81 \times 3} = 7.67ms^{-1}$

Circumference = 0.03π

Motor rotational rate =$7.67 \div 0.03\pi$ = 81.4 rps = **4883 rpm**

Required torque = $0.015 \times 0.0027 \times 9.81$ = **0.397E mNm**

130 Motor
Voltage Range: Dc 3V~6V
Motor Size: 15.5 * 20.3 mm
Motor Height: 25mm
Output Shaft Diameter: 2.0 mm
Output Shaft Length: 8mm
Rated Voltage: 6V
Rated Current: 0.14 A

Test Data:
Voltage: 3V No Load Speed: 5500RPM No Load Current: 100mA
Voltage: 6V No Load Speed: 10000RPM No Load Current: 140mA
Weight: 18 g

Fig 4.2.1d: Generic motor sizes and their specs

From figure 4.2.1d, generic 130 size DC brushed motors are able to exceed 4883rpm easily. Thus, any motor larger than 130 size would be suitable.

We borrowed RS380 size motors as they were the only pair of matching motors the lab had that was rated at 12V, and were larger than 130 size motors. We chose motors with a 12V rating for more headroom in terms of power; we can simply use PWM if the motors spin too fast at 12V. Overvolting a brushed motor rated for lower voltages can burn out the brushes and cause damage. The relevant data of generic RS380 motors are shown in the figure below. As can be seen, the maximum torque of the motor far exceeds the requirement calculated.

General Specification:

- Motor Type: RS380SA.
- Shaft Diameter: Ø2.3 mm.
- Shaft Type: Round.
- Shaft Length: 15mm.
- Input Voltage: 3~7.2VDC.
- Operating Current: 0.5A@6V.
- No Load Speed: 12,500RPM @ 6V.
- Rated Torque: 8mNm.
- Stall Torque: 50mNm.
- Output Power: 9W.
- Dimensions: Ø28 x 38 mm
- Electrical Connection: Terminal.
- Weight: 70g.

Fig 4.2.1e: Generic RS380 specifications

4.2.2: Magazine

To avoid having too many 3d printed parts, we decided to look for off-the-shelf solutions for holding the balls. Our criteria for selection was:

- Having a stated inner diameter suitable for carrying the balls—at least 42mm (±1mm)
- Flexible to bend around the bot
- Affordable—less than 15 dollars.

In the end, we settled for a 50mm inner diameter suction hose. It would be sturdy enough to keep its shape, as well as flexible enough to bend to our requirements.

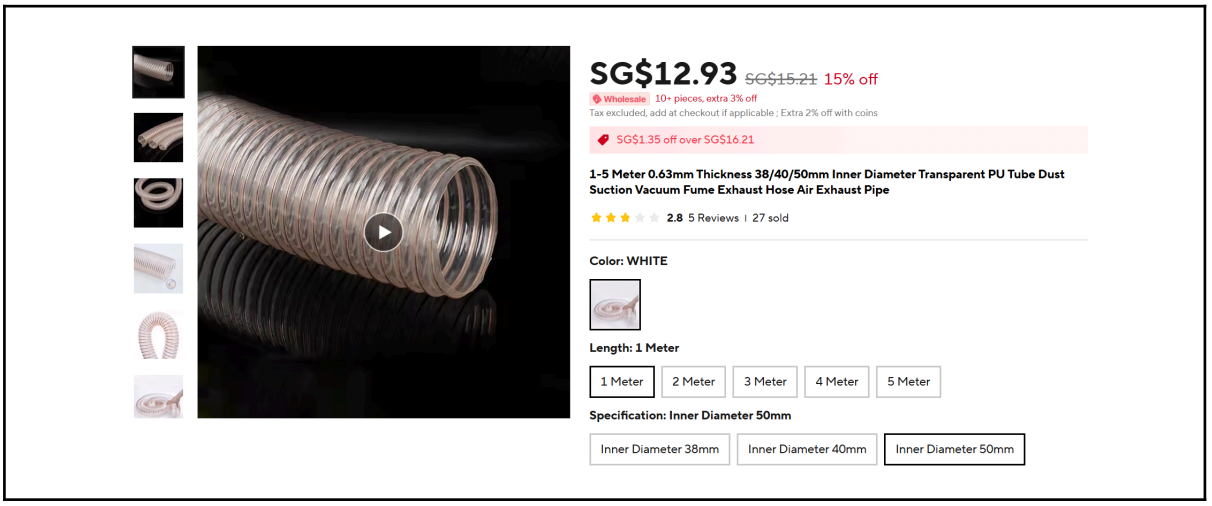


Fig 4.2.2a: Aliexpress listing of the hose we selected

The pipe would interface with the launcher using a 3d printed adapter, sized to fit the inner diameter of the pipe. The adapter would be attached to the pipe using an adhesive like superglue. It would be friction fit against a lip on the launcher. The adapter and interface is shown in the figure below.

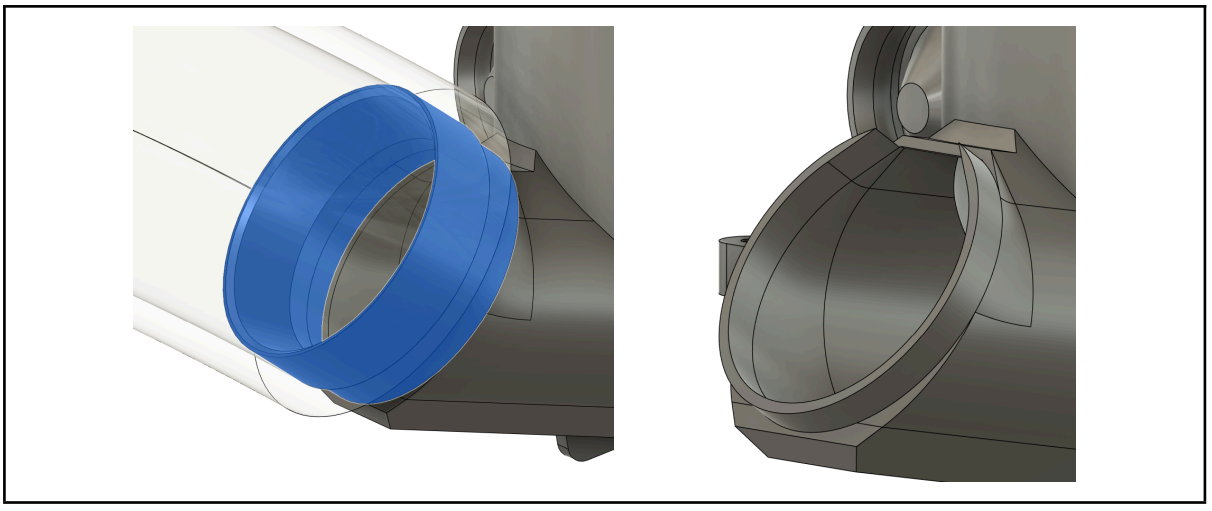


Fig 4.2.2b: Adapter and the lip that it interfaces with

The hose and adapter would form the sub-assembly referred to as the magazine.

4.2.3: Thermal Sensor Mount

The thermal sensor mount was designed to be mounted temporarily with 3M VHB tape on the top layer of the robot. We had planned to make a more permanent fixture for the sensor further along the line, thus we chose to use a strong but removable adhesive. The sensor mount is shown in the figure below.

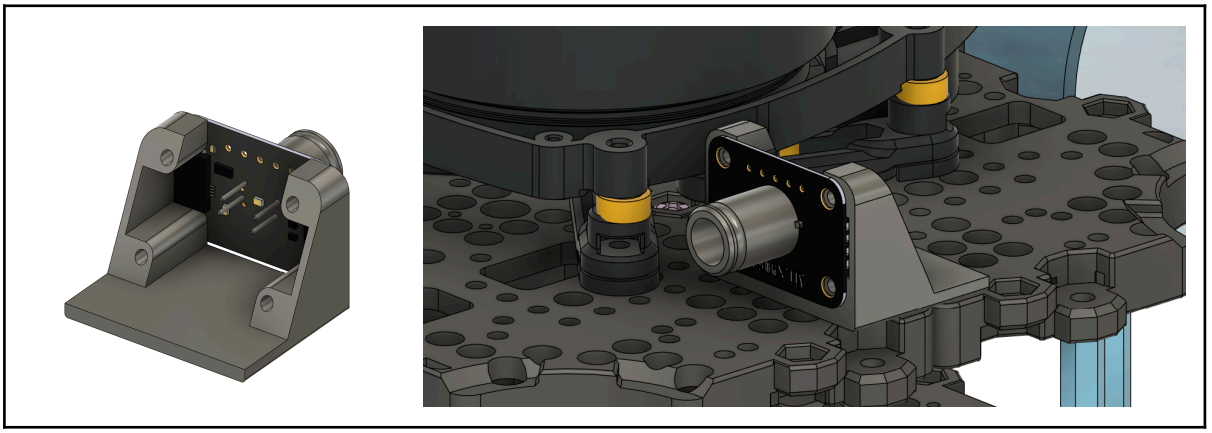


Fig 4.2.3: Thermal sensor and its mount, and potential mounting location.

4.3: Initial Electrical Wiring Design and Power Calculations

4.3.1: The OpenCR

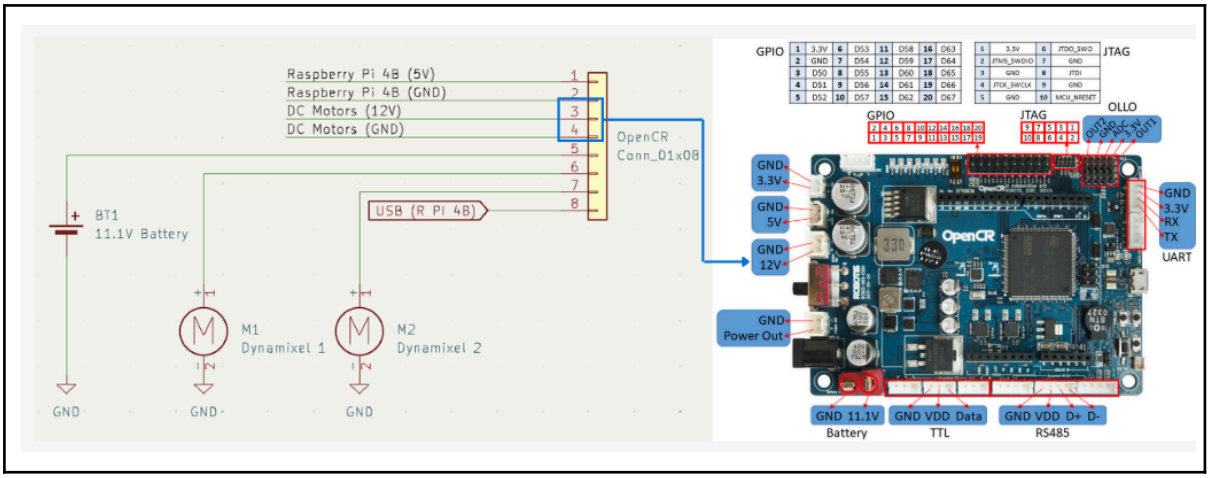


Fig. 4.3.1: Wiring Schematics (OpenCR)

The OpenCR is connected to a 11.1V Li-Po battery, 2 dynamixel motors and Raspberry Pi. The OpenCR has an IMU (Inertial Measurement Unit) that measures the TurtleBot's acceleration, angular rate and sometimes magnetic field to provide data about the TurtleBot's movement, orientation and velocity. For our project, since the Raspberry Pi does not have a 12V output, the flywheels of the projectile launcher are powered by the 12V connector from the OpenCR.

4.3.2: Raspberry Pi (RPi)

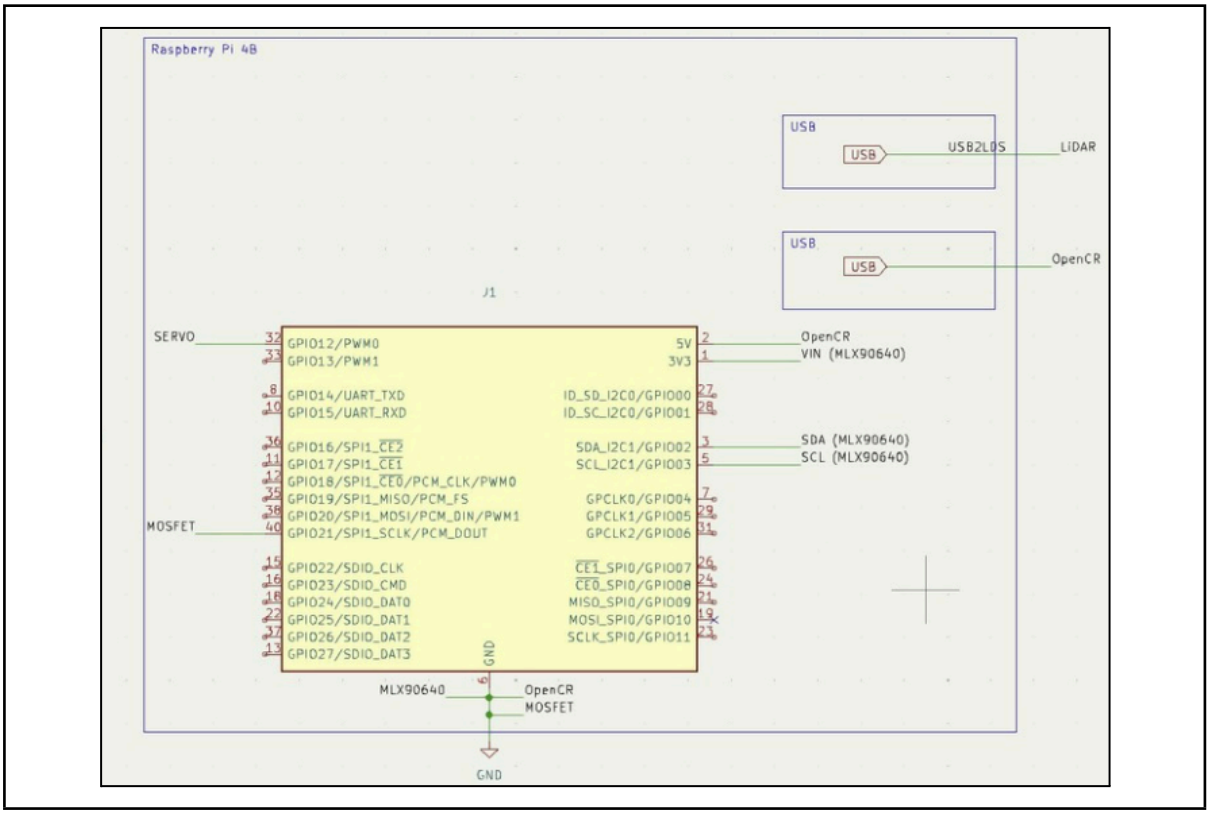
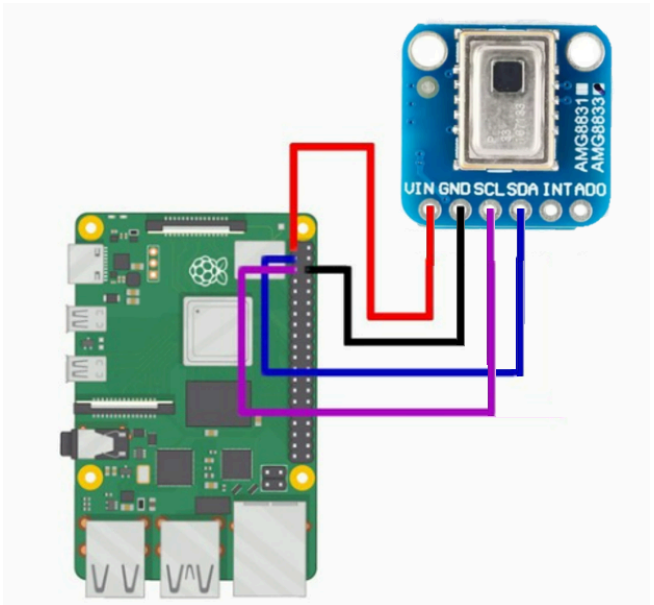


Fig. 4.3.2: Wiring Schematics (Raspberry Pi 4B)

The Raspberry Pi is the centre hub of Turtlebot3, it is receiving range data from LiDAR, heat data from the heat sensor, various data from IMU to help with the localisation of the bot and goal setting for its mission. Hence, LiDAR and OpenCR will be directly connected to the USB port of the Raspberry Pi as per user manual, and the thermal sensor will be connected directly to the SCL and SDA pin of Raspberry Pi for it to read. For signal and power outputs, the Raspberry Pi will power the heat sensor with 3.3V pin and it will control the servo of the projectile launcher with GPIO12 a PWM pin, and the MOSFET switch with GPIO21 a PCM pin.

4.3.3: The Thermal Sensor (AMG8833)

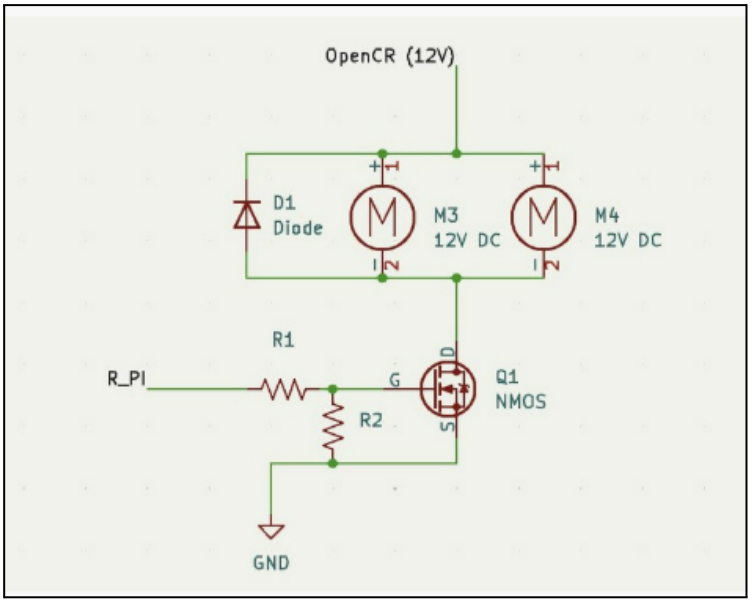


RPI	AMG8833A
3.3V	VIN
GND	GND
SDA (Pin 3)	SDA
SCL (Pin 5)	SCL

Fig 4.3.3: Connections between AMG8833 sensor breakout board and RPi.

Connection of the heat sensor is with its respective pins on the Raspberry Pi.

4.3.4: Projectile Launcher



4.3.4: Wiring schematics for projectile launcher subsystem

For the projectile launcher circuit, it will be powered by OpenCR 12V pin, 2 12V DC motors will be used as our flywheel with a flyback diode to protect the circuit. A MOSFET (IRL3803) will be used as a switch for our flywheels for its low turn on resistance and it allows rapid switching, the Raspberry Pi will output the switching signal. Resistor is placed between the gate and source of the MOSFET to protect the circuit.

4.3.5: Power Calculations

	Min Power (W)	Max Power (W)	Average Power(W)
Turtlebot3	7.55 (Only ROS2 running)	9.28 (ROS2 running with dynamixel max velocity)	8.42
Projectile Launcher Motors (Both on)	4.25	4.76	4.51
AMG8833	-	-	0.0149

Above is the power budget table for our bot. Power consumption of Turtlebot3 with ROS2 running and dynamixels turned on are tested 3 times and recorded average. Power consumption of Projectile Launcher is recorded 2 times and recorded average min and max power. Power consumption of AMG8833, the heat sensor we chose, is calculated with the data provided by the data sheet, it has an operating voltage of 3.3V and an operating current of 4.5mA, which results in 0.0149W.

Assumptions:

1. Duration of boot-up and servo switching are too short compared to the total duration of the mission, hence they are not considered in the power budget.
2. Turtlebot will have ROS2 running at all times, power consumption of turtlebot without ROS2 running will not be considered.
3. Power consumption of AMG8833 is too small compared to the rest of the systems hence will not be considered.
4. Worst case scenario for projectile launcher motors is to be on at all times.
5. Worst case scenario for Turtlebot is assuming ROS2 running + max travel speed at all times.

Calculations:

Battery Capacity: $11.1V \times 1800mAh = 19.98Wh$
Total Power (Worst Case): $9.28 + 4.76 = 14.04W$
Turtlebot3 operable duration (Worst Case): $19.98 \div 14.04 = 1.42hrs$

Since the mission is about 25 minutes long maximum and the Turtlebot3 can last up to 1.42 hours in the worst case scenario, the battery is sufficient for this mission.

4.4 Planned Software Functions

4.4.1 Planned Mission Logic Flow

Phase	Activity
Exploration	The turtlebot will navigate to frontiers and progressively map and explore the environment to generate a full occupancy grid of the entire maze.
Heatmap generation	The turtlebot will use the occupancy grid to determine where heat sources are on the map by collecting thermal data and raycasting them onto a grid that stores the occupancy grid and the thermal data using the robot's pose and yaw.
Casualty saving	The heatmap generated by the previous phase will then be used to determine the locations of the heat sources. The robot will then navigate to the heat sources and carry out the task of firing ping pong balls.

The names of sub-functions in these phases are detailed in the graph below.

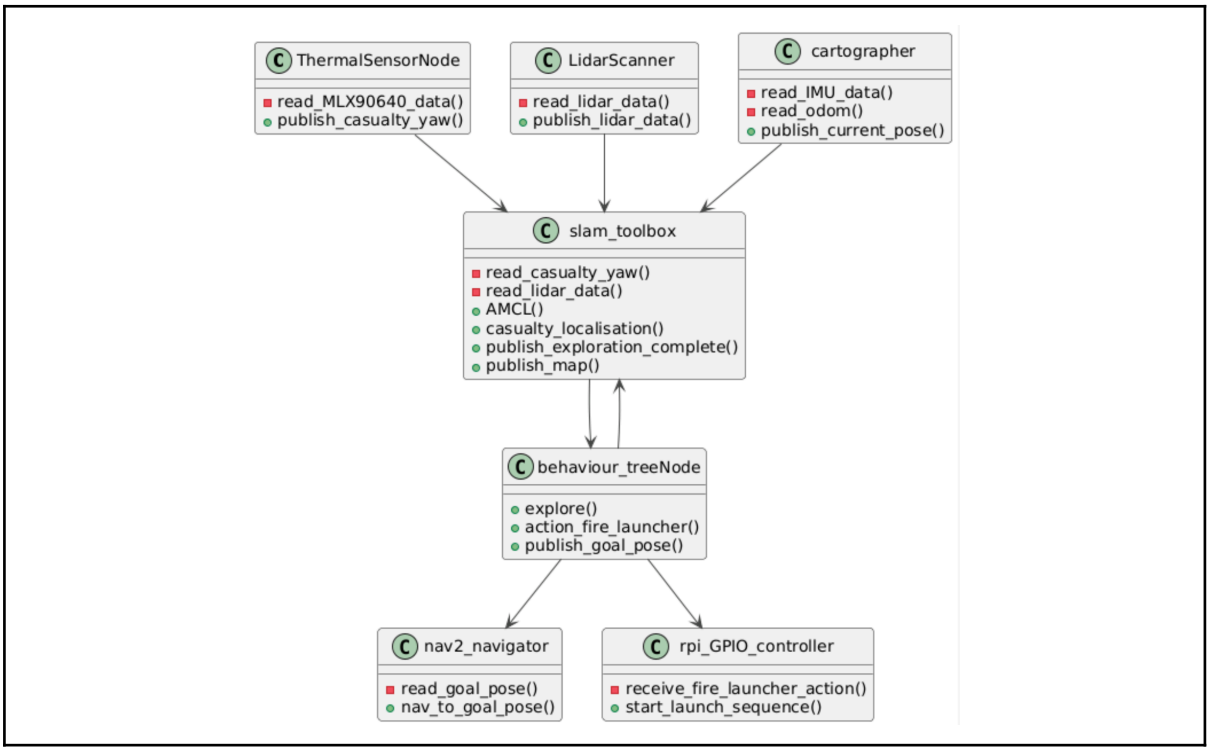


Fig 4.4.2a: Planned logical flow of the system

4.4.2 Planned Frontier Exploration Logic

The logical flow of the frontier exploration is detailed in the figures below:

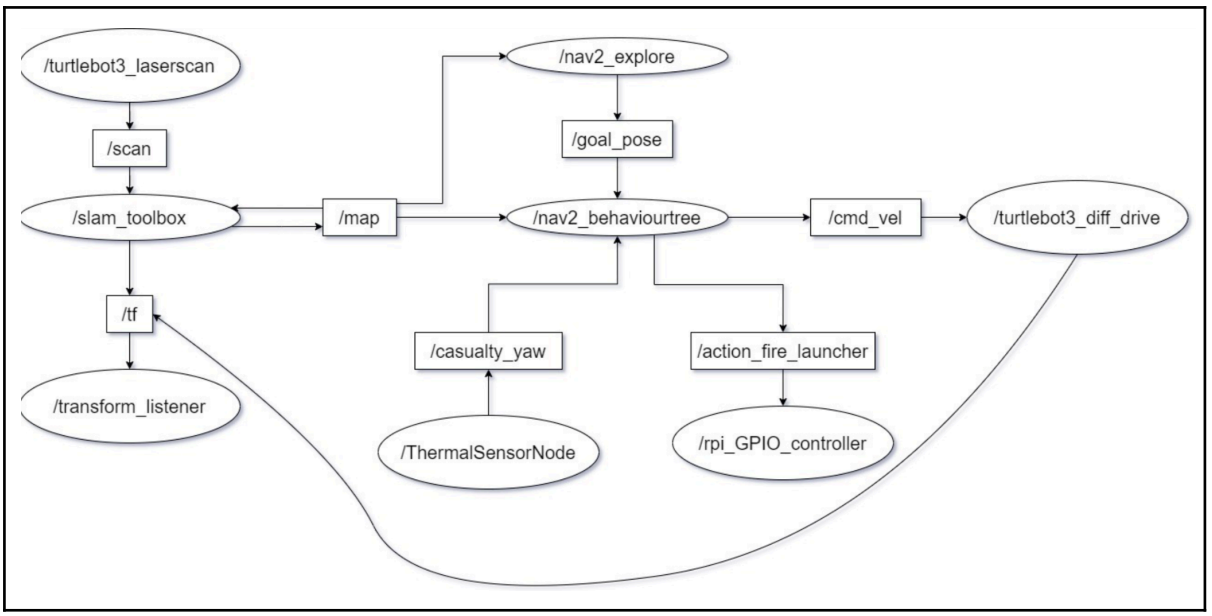


Fig 4.4.1b: Manually constructed rqt_graph of frontier exploration

The robot will read the occupancy grid from /map, and determine the frontiers to travel to. Frontiers will be scored based on 2 factors, namely frontier size and distance. Distance will score negatively, to increase the cost of travelling to further frontiers, while frontier size will score positively, increasing the score for larger frontiers. This is to maximise information gain while reducing cost. This logic is shown in the figure below.

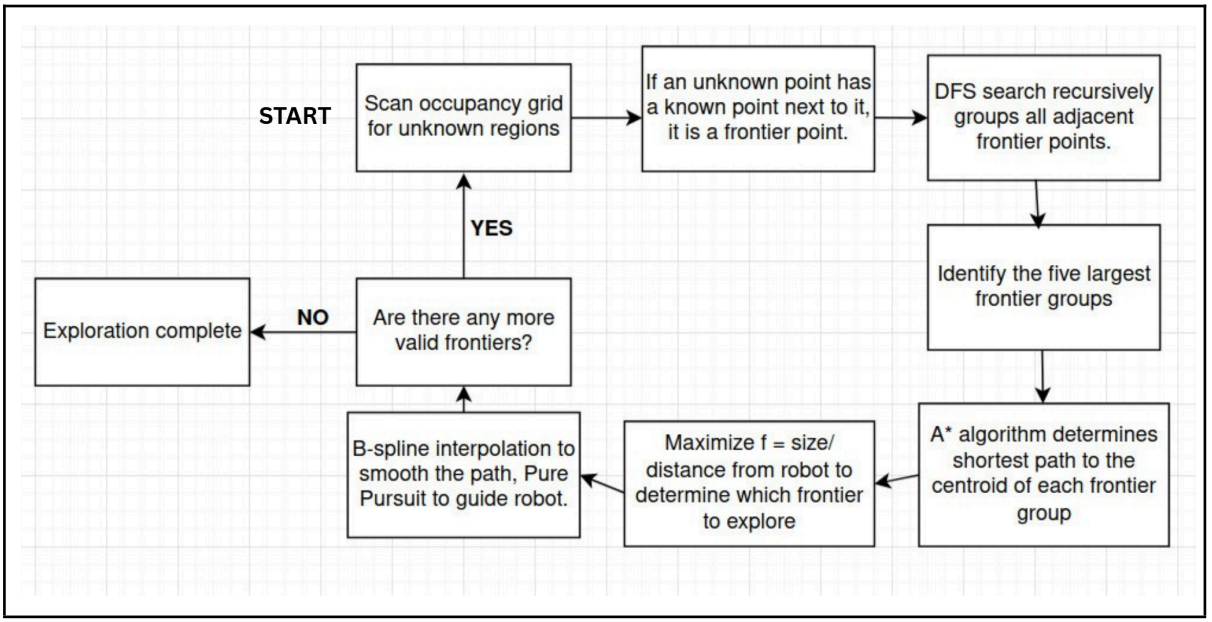


Fig 4.4.1c: Logic flowchart showing frontier choosing logic in the exploration node.

The exploration node will then use generate a path using the A* algorithm, and use pure pursuit to guide the robot to the goal frontier.

5.0: Prototyping & Testing

5.1 Hardware Testing

5.1.1: Launcher

To test the concept of the flywheel based launcher before committing to the rest of the design, we produced an initial prototype meant to simply test the launcher concept in isolation. This prototype design is shown in the figure below. (Note the dummy flywheels on the assembled CAD on the left.)

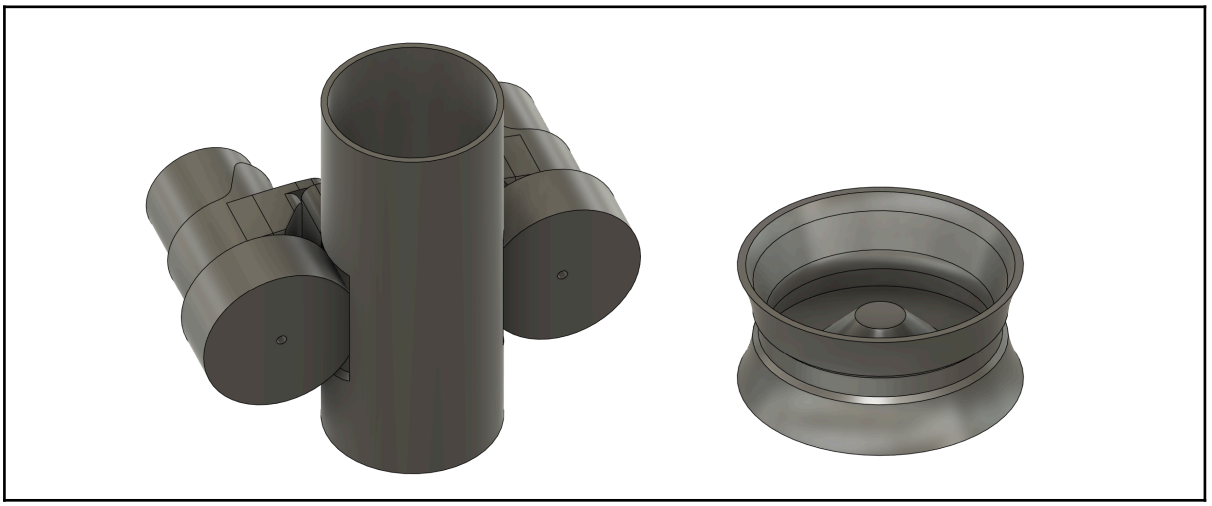


Fig 5.1.1a: CAD model of initial prototype. Flywheel design is in the image on the right.

As the o-rings had yet to arrive at this stage, we used rubber bands, which worked fairly well. The prototype was tested by lowering it onto the ping pong ball, to simulate the ball being pushed upwards into the flywheels. This prototype, with the motors running 12v DC, was able to launch a ping-pong ball quite violently into the ceiling. Screengrabs of the video can be seen in the figure below.

Fig 5.1.1b: 3D printed launcher prototype in action. Link to view a video of the launcher in action: Video link

5.1.2: Launcher Placement

During testing, we found that in the original layout where the launcher is on the back of the robot, the bend in the magazine tube was too great, which caused balls to get stuck. The CAD model of the magazine tube is shown in the figure below.

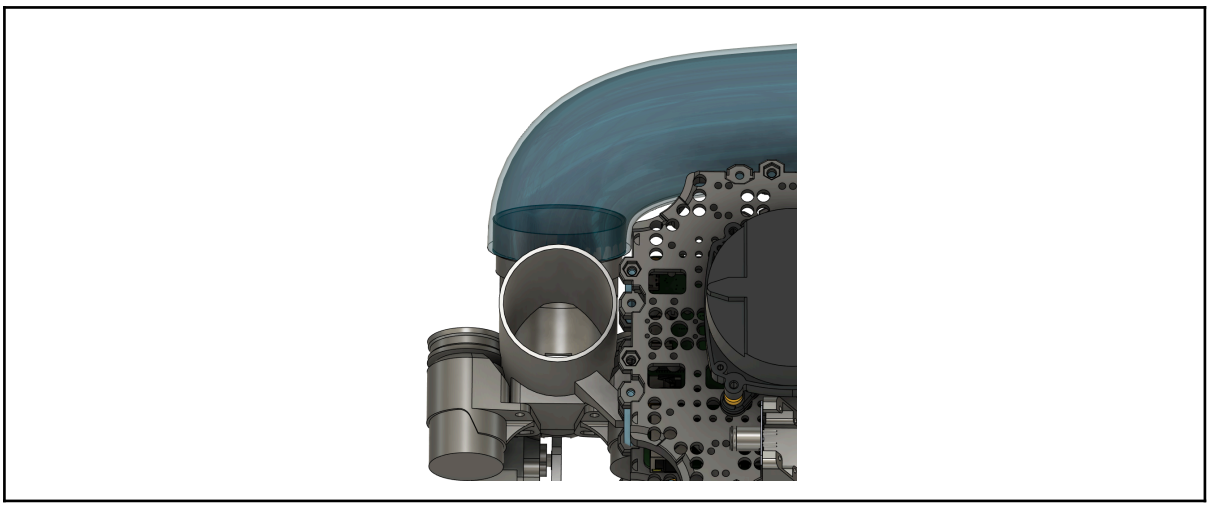


Fig 5.1.2a: Sharp ~80 degree bend in magazine tube

To remedy the problem, we planned to shift the launcher further to the robot's right, to allow a more natural bend in the magazine tube. This is shown with the translucent overlay of the launcher and magazine assembly on the bot on the left, in the figure below. (The forward travel direction of the bot is shown using the arrow.)

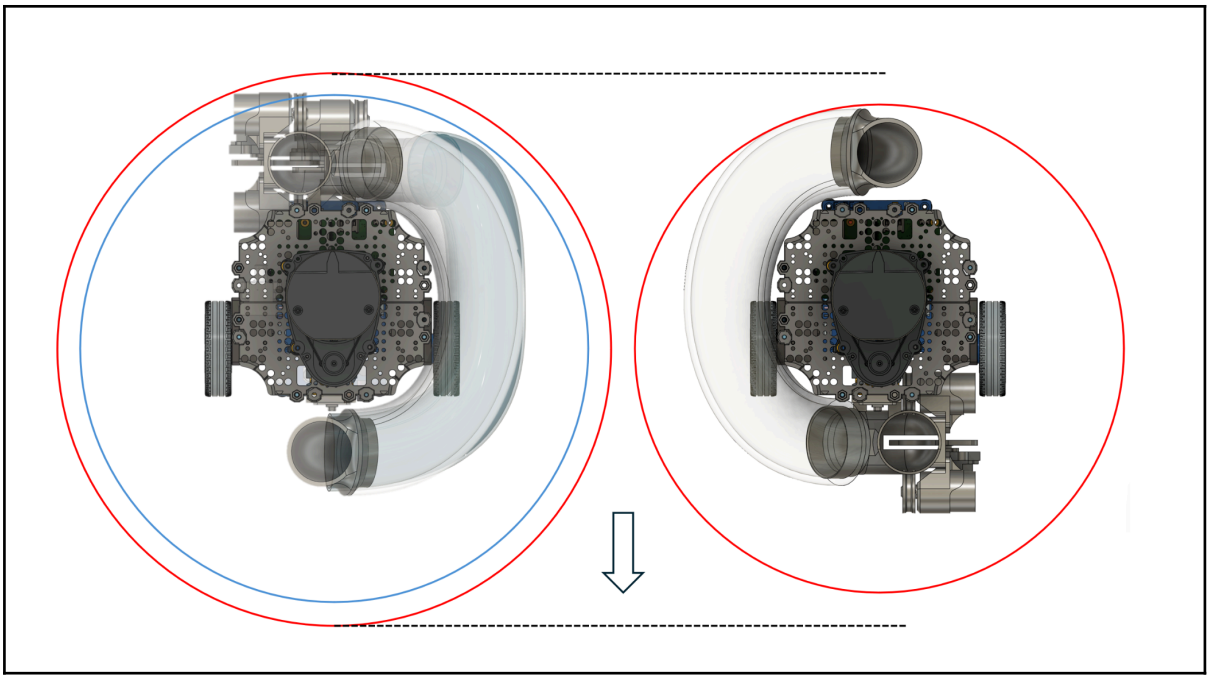


Fig 5.1.2b: Layout options for launcher and their effect on the robot's effective radii

However, such a change on its own would cause a large increase in the robot's effective radius, shown with the blue and red circles on the bot on the left in the figure above. Thus, we decided to flip the launcher and magazine, and mount the launcher on the front of the bot instead. This significantly reduced the effective radius of the bot by around 2 cm from what it would have been if it was mounted at the back, allowing it to navigate tighter corridors with lesser risk of colliding with obstacles.

5.1.3: Thermal Sensor Mount

The initial design for the thermal sensor was for it to be temporary and moveable. Thus it was to be used with a strong but removable double sided adhesive like 3M VHB tape. We found that the Y-offset of the sensor in relation to the robot was crucial in ensuring that yaw values calculated from the thermal camera were accurate and usable in adjusting the heading of the robot.

Having determined the rough location and direction of the thermal sensor, a more permanent mount was designed. The constraints involved in deciding its location were:

- The path of the launcher's projectiles
- Field of view (FOV) of the thermal sensor (60 degrees for AMG8833)
- Lidar height

As such, its current location was decided to be the most ideal, and required no other modification to the location of the rest of the subsystems. The placement of the thermal sensor is shown in the figure below.

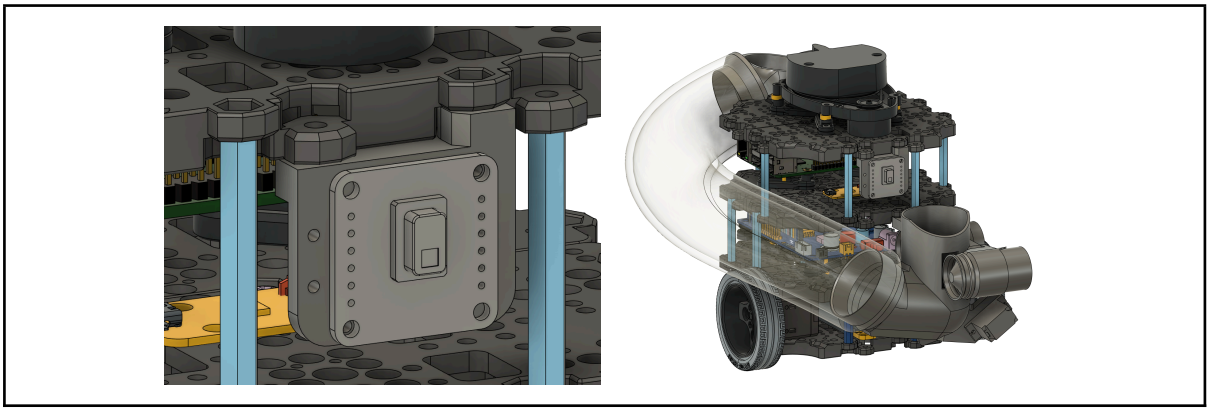


Fig 5.1.3a: Zoomed in view of the thermal sensor mount, and an overview of the turtlebot.

The location of the thermal sensor created more constraints for the launcher, which had to have modifications made to it to ensure it would not obstruct the thermal sensor's FOV. The launcher had to have a cut made to its barrel for this use. This is shown in the figure below.

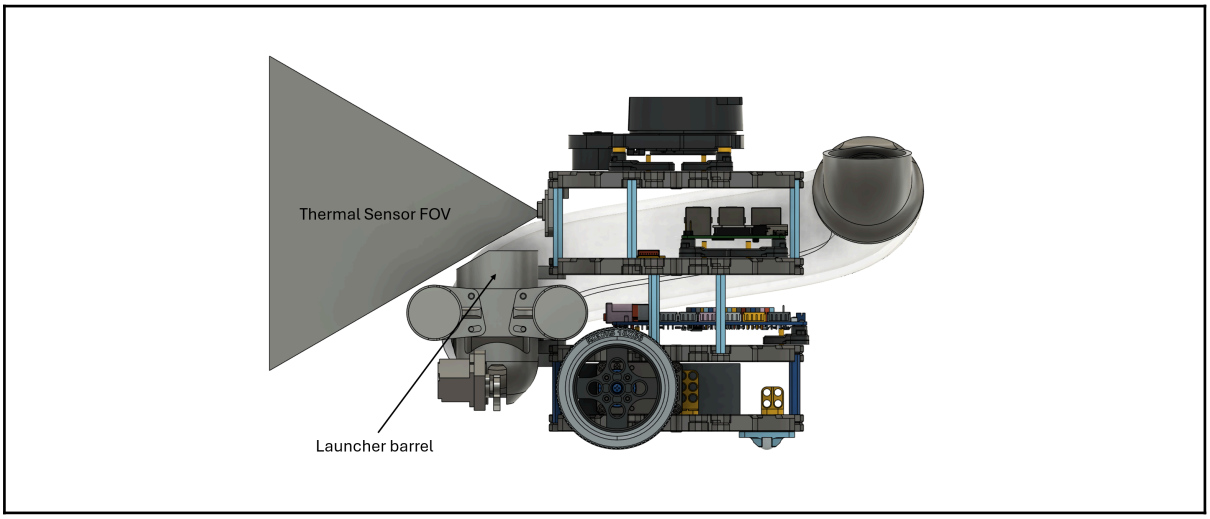


Fig 5.1.3b: Side view showing the cut made to the launcher to accommodate the Thermal Sensor

5.1.4: Magazine

The tube we ordered for the magazine turned out to be too stiff to simply wrap around the bot and be held in place. Thus, we used wire ties to hold the tube in a bent position that was suitable for the launcher. This is shown in the figure below.

Fig 5.1.4a: Image of the turtlebot showing the magazine being held in its bent position with white wire ties

From testing, we found that the incline of the magazine tube was not enough to prevent balls from falling out when the robot was moving, or on an incline. Thus, a stopper at the end was required. To facilitate easy reloading of the magazine, we opted for an angled opening, which acted as a hopper as well as a stopper for the balls.

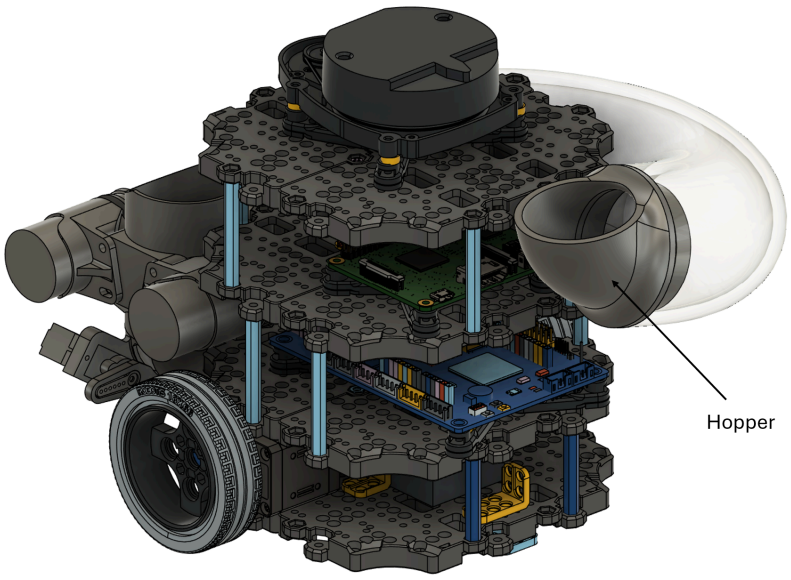


Fig 5.1.4b: Hopper to prevent balls from rolling out of the magazine

5.2 Software Testing

5.2.1: Modular Packages

To facilitate effective unit testing, our scripts are grouped into packages which each have a main purpose. For example, we have a package that controls the flow of events called mission_control, and another package that handles AMG8833 sensor reading and data handling called casualty_location. Casualty_location is further broken down into nodes that handle a specific task. This modularity made it much more manageable to allocate tasks such that there were no two members working on the same node.

5.2.2: Frontier Exploration

To save time and effort, we first looked for Open-source implementations of frontier exploration for the turtlebot on the internet. After testing a few, we narrowed down the choice to 2 packages, which we will dub “AniArka” [13] and “SeanReg” [14], after their creators.

AniArka: Initially, this package did not perform as well as we hoped. It was quite slow in generating frontiers, and was slow in exploration. However, the codebase was extremely readable, which we used to learn how ROS2 packages worked. We added some optimisations to improve its efficiency.

SeanReg: This package was found at a later stage, after optimisations were made to AniArka’s package. Initial tests in Gazebo proved promising, with the turtlebot being able to explore the full Turtlebot World in around 1 minute. It was already faster than AniArka’s package, so we decided to use it. However, the package was coded like a black box, and all of us gave up on trying to understand it. We did manage to implement an optimisation, such as a counter to check if the same frontier is being visited repeatedly. This counter counts up to a limit before ending exploration. This was especially useful in ending exploration when the node chooses an unreachable frontier repeatedly. This almost always happened when slam_toolbox was misbehaving, or when exploration was almost finished.

Thus, we chose to work with the SeanReg frontier exploration, as it was much more efficient even without the optimisation. Our casualty_location node was written using the knowledge gained from AniArka’s package.

5.2.3: MLX90640 and AMG8833 Testing

AMG8833 Testing: After correctly connecting the necessary ports to access the AMG8833 (AMG) via I2C, we used an existing amg8833 driver [15] to observe the data output by the camera. We found that the capture rate of the AMG was around 10 fps, which was around what we expected. Using ourselves as heat sources, we determined that the effective range of the raw output was no more than 1m, while using interpolation boosted this value to around 2m.

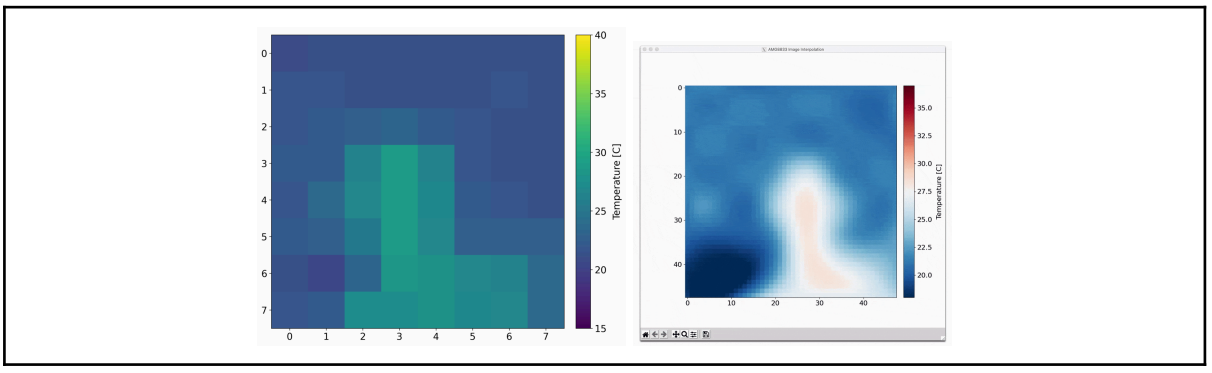


Fig. 5.2.3a: Viewing the Camera Feed, Raw (left), and Interpolated (right).

We wrote a node, pubCentroid, that takes the IR data, interpolates it to a 32x32 grid and performs thresholding on the temperature values to determine whether cells would count as belonging to the heat source. A cell is above the threshold if its value is *temperature_offset* greater than the median value of the cells, or if it's greater than a static temperature threshold. It then uses OpenCV (cv2) functions to determine the centroids of the heat blobs.

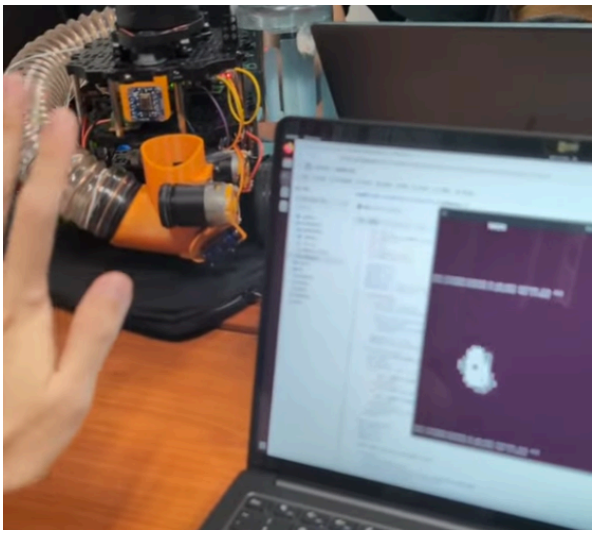


Fig. 5.2.3b: pubCentroid node running. Link to view a video of the node in action: Video link

Computing Useful Data: At first, we attempted to use cv2 to detect yaws of blobs of heat sources to determine the direction of objects relative to the AMG. We constructed a setup using a piece of paper and string in order to measure the yaw versus the horizontal value of the heat blobs, as seen in Figure below.

Fig. 5.2.3c: Yaw Measuring Setup

Simplifying the Computation: Through repeated testing, we eventually realised that it was unnecessary to detect blobs on the 2d plane of the camera output, since we only needed the yaw values of the heat sources in our raycasting algorithm. As such, we simplified our IR publisher node to output a 1d array, containing the maximum temperature detected in each column of the raw camera output. Our casualty_location node then takes this data and interpolates it to 120 values. These values are then applied to the occupancy grid from the /map topic, to assign a temperature value to the obstacles of the environment.

MLX90640 Testing: We originally intended to use the MLX90640 (MLX) due to its higher resolution, which would allow us to generate the thermal map with greater precision and detect heat sources from further distances, intending to allow more speedy mapping as the Turtlebot would not have to travel such a great distance. However, we realised that, despite all the optimisations, we could only receive thermal imaging from the MLX at 1fps. Due to the much lower rate of thermal data from the MLX compared to the AMG which outputs data at 10fps, we decided to sacrifice precision for greater update rate and use the AMG.

5.2.4: Raycasting Algorithm (casualty_location)

To generate a heat map of the environment, the occupancy grid from the /map topic is converted to a 2d array and modified by casualty_location to store thermal data from the thermal sensor.

The rays are cast by looping through an array of interpolated yaw values, starting from the left bound of the sensor's FOV, and ending at the right bound. The gradient of the ray is calculated using its yaw value. It then iterates step by step, calculating the index of the cell that the ray lands on using the gradient multiplied by the number of steps, basically tracing a line from the robot's position. The code snippet that does this function is shown in the figure below.

```
bot_col = int((self.robot_cache[self.pose_index].x - self.initial_origin[0]) / self.resolution)
bot_row = int((self.robot_cache[self.pose_index].y - self.initial_origin[1]) / self.resolution)
rows, cols = len(self.grid), len(self.grid[0])

start_angle = math.degrees(self.robot_cache[self.pose_index].yaw) + SENSOR_FOV / 2
end_angle = math.degrees(self.robot_cache[self.pose_index].yaw) - SENSOR_FOV / 2
num_rays = int(SENSOR_FOV * 2)

data = self.latest_ir_data
x = np.linspace(0, len(data) - 1, len(data))  # Indices of the input data
interpolator = interp1d(x, data, kind='linear', fill_value='extrapolate')  # Linear interpolation
interpolated_data = interpolator(np.linspace(0, len(data) - 1, num_rays))  # Interpolated data
angles = np.linspace(start_angle, end_angle, num_rays)

for idx, angle in enumerate(angles):
    rad_angle = math.radians(angle)
    dx = math.cos(rad_angle)
    dy = math.sin(rad_angle)

    for step in range(1,40):
        row = int(round(bot_row + step * dy))
        col = int(round(bot_col + step * dx))

        if 0 <= row < rows and 0 <= col < cols:
            if self.grid[row][col] > 10:
                if self.grid[row][col] > 90 or interpolated_data[idx] > self.grid[row][col]:
                    self.grid[row][col] = interpolated_data[idx]
                # Check if the next cell is also a wall (100) and hasn't been thermally scanned yet
                row = int(round(bot_row + (step+1) * dy))
                col = int(round(bot_col + (step+1) * dx))
                if 0 <= row < rows and 0 <= col < cols:
                    if self.grid[row][col] > 10 and (self.grid[row][col] > 90 or interpolated_data[idx] > self.grid[row][col]):
                        self.grid[row][col] = interpolated_data[idx]
                break
        else:
            break
```

Fig 5.2.4a: Code snippet showing most recent commit of the raycasting function in casualty_location

The inner loop breaks when either the ray is out of bounds, or when the ray hits an occupied cell. In the case of the latter, the thermal value associated with the ray is allocated to the cell in the grid if it is higher than the cell's existing value. The ray is then allowed to continue one more step. Just like before, the inner loop breaks if the ray is out of bounds or hits an occupied cell. Once again, in the latter case, the thermal value associated with the ray is allocated to the cell if it is higher than the existing value. The inner loop then breaks for good, iterating to the next ray.

Standalone testing: In the early stages, in order to test a proof of concept, a standalone python script was made. It would raytrace a coloured map for "thermal data", then raycast it onto a occupancy grid with unassigned thermal values. The coloured map and occupancy grid was made on microsoft paint, and the Pillow library was used to convert the images into arrays to read from. A screengrab of the script's matplot display is shown in the figure below.

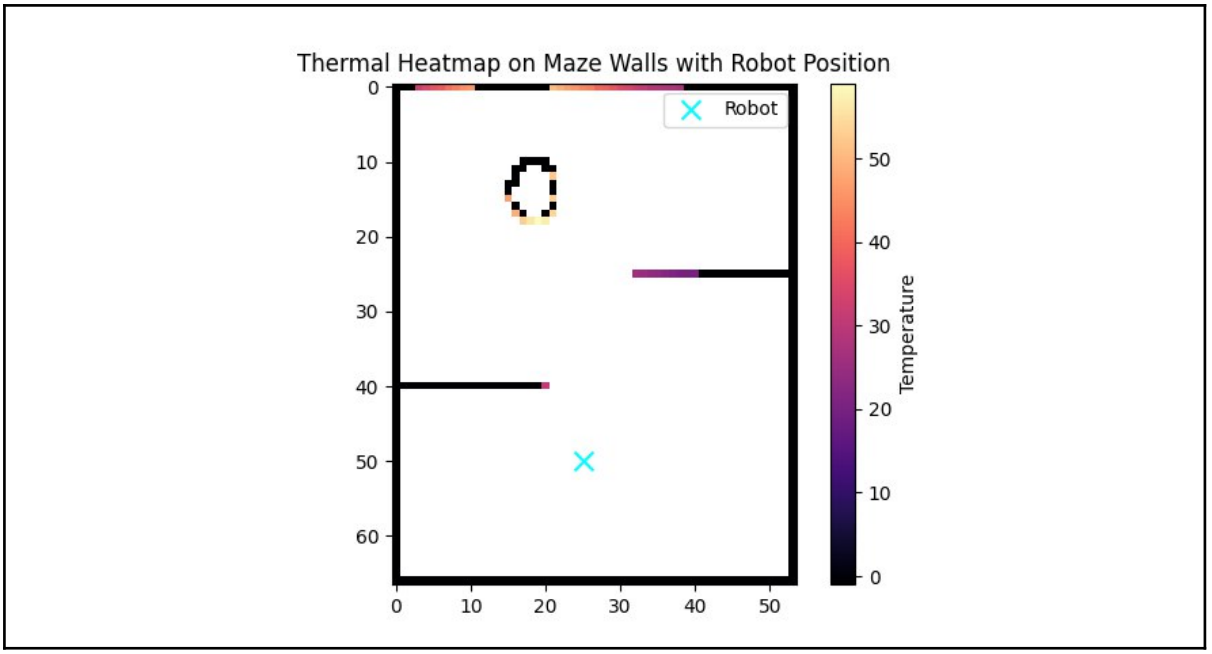


Fig 5.2.4b: Matplot of the grid after raycasting dummy thermal values onto a test environment

Having proven the feasibility of the concept, we decided to carry on with the software implementation discussed in the Preliminary Design Report (PDR). We created a node that took thermal readings from the IR publisher node mentioned in section 5.2.3 and raycast them onto the occupancy grid obtained from the /map topic.

Ensuring coverage: We still needed a way to ensure complete coverage of obstacles. Thus, we used a frontier based exploration method for gathering thermal data of the environment. We defined frontiers in this node as connected regions of obstacles with unassigned thermal data, and determined each cell's score using the formula below.

$$\frac{Region\ size - 10}{Distance\ from\ robot\ to\ cell + 1}$$

Region size is set to the actual region size or 0 if the region size is smaller than 10. This gives all frontiers a score greater than or equal to zero. To visualise the scores and thermal value of cells, we used matplotlib. These plots can be seen in the figure below.

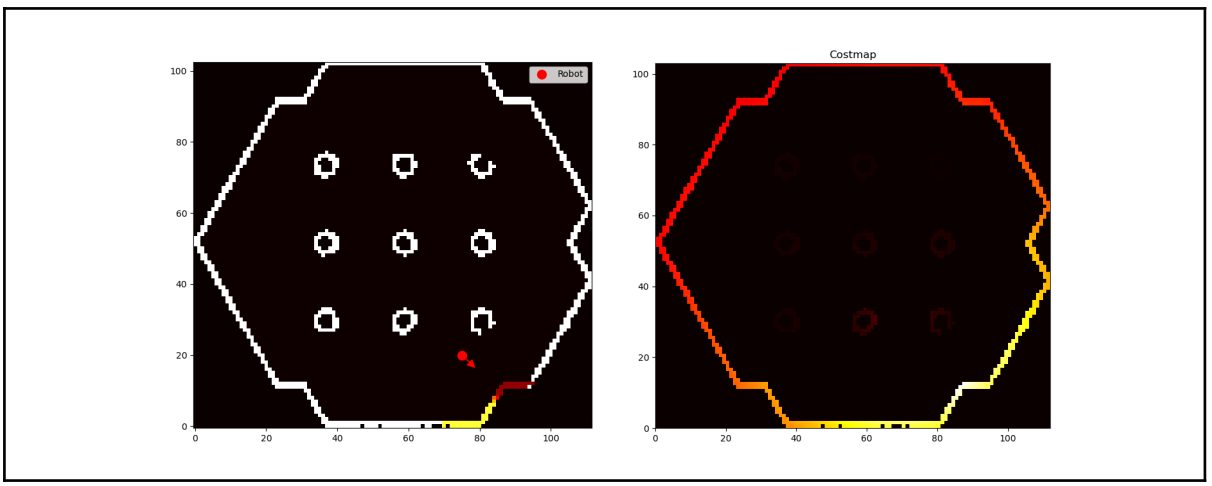


Fig 5.2.4c: Plots showing the grid storing thermal data, as well as the grid storing calculated scores

Choosing Frontiers: Frontiers were chosen based on having the highest score that was greater than zero. This formula causes the node to ignore regions smaller than 10 cells. We also added a function to make the turtlebot spin a full round after each successful nav goal, which greatly sped up the thermal data gathering process. We found that the node had a tendency to select frontiers repeatedly, and discovered that the reason why it happened was because the chosen cell was unreachable by the casted rays, which sent the system into an endless loop. To fix this problem, cells chosen as the frontier would have their score set to 0 permanently, and the score of cells within the radius of 15 cells of the last visited frontier would be set to -1 (for visualisation as a black circle) temporarily to prevent the node from requesting the same pose repeatedly. This is shown in the figure below.

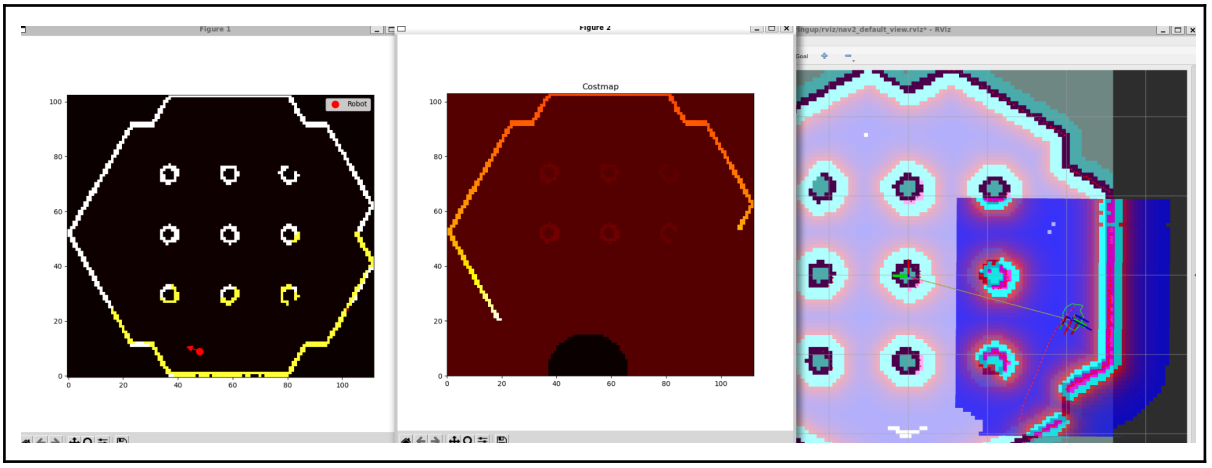


Fig 5.2.4d: matplot windows showing the thermal map on the left, score grid in the middle with ignore radius around the robot's last chosen frontier, and RViz on the right, showing the path to the next chosen frontier.

Testing: We were able to test this in gazebo by setting a static dummy array for the raycasting (paint_wall) function to use. We were also able to test whether IR values were correctly applied left to right by defining the dummy array as [20,20,20,20,80,80,80,80], and seeing if they were applied to the map correctly. (The effect of the dummy array can be seen in figure 5.2.4c, on the left. The red cells would indicate a value of 20, while the yellow cells indicate a value of 80.) We did have some early issues that were caused by the pair of yaw values pertaining to the left and right bounds of the thermal sensor's FOV being flipped. This caused the rays casted by the function to apply values to the wrong cells, which led to multiple attempts to try to get a more accurate thermal map before we narrowed down the cause to being the flipped yaw values.

Some attempts included:

- Keeping only the latest value—this was extremely inaccurate, as the latest, still wrong value would simply overwrite the previous
- Keeping a rolling average—this was also quite inaccurate as the cells would also get overwritten as the robot spun.
- Keeping the highest value—this worked surprisingly well in gathering the general area of the heat source's location.

After fixing the bug caused by the swapped yaw values, we stuck with the method of keeping the highest value. This was due to the falloff in thermal readings with distance. Since the radiation received by the sensor at a distance follows the inverse square law, the most accurate values are when the sensor is closest to the source. Thus, the higher the value is, the more likely it is to be accurate. Thus, we kept the approach of keeping the highest value.

In addition to the problems listed above, we ran into some other issues significant enough to warrant their own sections such as a delay in the thermal sensor reading values (detailed in section 5.2.5), as well as odom drift (detailed in section 5.2.6).

5.2.5: Thermal Camera Delay

From our testing of the raycasting in casualty_location, we found that there was a delay between the appearance of a heat source and the publishing of the data. This delay led to thermal values being inaccurately assigned to cells slightly too far ahead of where the robot actually was. To find this delay, we recorded a video that showed the logs of the pubCentroid node, as well as a member of the team removing their hand from the view of the camera. This is shown in the figure below.

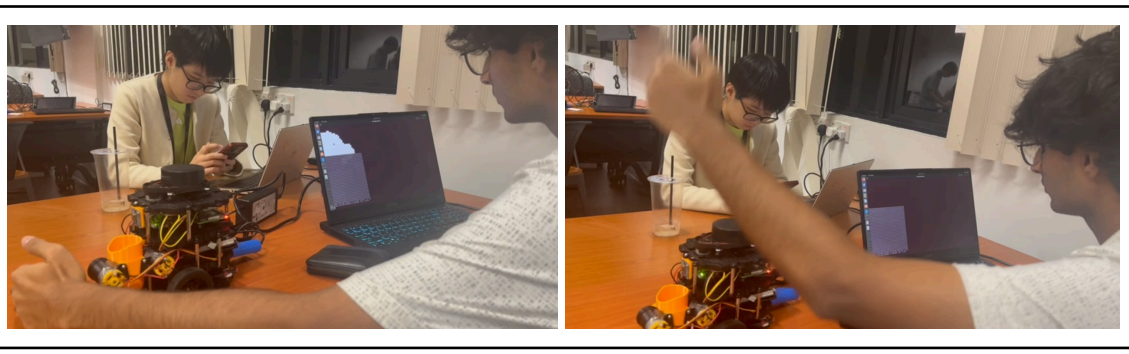


Fig 5.2.5a: Screengrabs from the video used to determine the delay in the IR data publishing.

We determined the delay by taking the time difference between the group member removing his hand and the publisher logs displaying an empty grid. We found this delay to be around 0.2s. To use this delay, casualty_location stores an array of robot poses corrected for offset (detailed in section 5.2.6). The cache is maintained by appending the latest pose and yaw as a tuple, and popping the first index when it reaches the parameterised cache size. Raycasting only takes place when the cache is full. The code snippet that does this is included in the figure below.

```
def odom_callback(self, msg):
    def clean_pose_cache(self):
        # Remove old poses from the cache
        if len(self.robot_cache) > POSE_CACHE_SIZE:
            self.robot_cache.pop(0)
            self.pose_cache_full = True

    x = msg.pose.pose.position.x
    y = msg.pose.pose.position.y
    self.robot_odom_position = (x, y)
    self.robot_position = (x + self.offset[0], y + self.offset[1])

    orientation_q = msg.pose.pose.orientation
    quaternion = (orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w)
    _, _, yaw = euler_from_quaternion(quaternion)
    self.robot_yaw = yaw

    self.pose_received = True
    pose = BotPose(x, y, yaw)
    self.robot_cache.append(pose)
    #self.get_logger().info(f"Robot Pose: {pose}")
    clean_pose_cache(self)
```

Fig 5.2.5b: Code snippet for storing pose and yaw cache

This method stores the most recent pose and yaw data in the last index. The list is then accessed from the last index subtracted by the product of IR_DELAY and ODOM_RATE to find the robot pose and yaw that matches the IR data.

5.2.6: Topic Fusion of Odom and (AMCL) Pose

In our casualty_location node, we use a raycasting algorithm that extrapolates from the position of the robot in the direction of the thermal camera to generate a heatmap. This algorithm thus requires the current position of the Turtlebot3.

Odom Drift: Our raycasting algorithm generates a matplotlib map that is a copy of the OccupancyGrid but coloured with the heatmap values. It also displays the current position and direction of the bot while it is painting. Through this, we realised that, over time, there was a growing mismatch in position between the RViz display and our matplotlib heatmap, which severely hindered accurate heatmap generation.

Alternative Options for Reading Robot Pose: Other than /odom, there was also /pose which uses Adaptive Monte Carlo Localisation (AMCL), as well as transform (tf) tree. /pose produces a more accurate position compared to /odom due to the constant update in localisation by the AMCL. However, we found that it publishes at 1 Hz on the RPi, which was much too slow for use when creating the heatmap. The most ideal would be tf tree, which combines both /odom and /pose to produce a more accurate position estimate. However, tf tree was found to be published even more inconsistently than /pose.

```
def pose_callback(self, msg):
    self.offset.x = self.odom_pose.x - msg.pose.pose.position.x
    self.offset.y = self.odom_pose.y - msg.pose.pose.position.y
    orientation_q = msg.pose.pose.orientation
    quaternion = (orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w)
    _, _, amcl_yaw = euler_from_quaternion(quaternion)
    self.offset.yaw = self.odom_pose.yaw - amcl_yaw
```

Fig. 5.2.6: Code snippet for finding and storing pose and yaw offset

Topic Fusion of /odom and /pose: With /odom publishing at 20 Hz and /pose at 1 Hz, we realised it was possible to update the offset between the two everytime /pose is received, thereby using it to constantly recalibrate the actual robot position being used. The code for this can be seen in Fig. above. This resulted in significantly reduced odometry drift over longer periods of time. This offset would be applied to the pose data gathered from the /odom topic, and the result is stored.

5.2.7: Updated Robot Footprint

From testing, we found that simply using robot_radius in nav2 was insufficient in trajectory planning. Our robot was quite asymmetrical, thus using a radius would be too conservative, and cause the robot to be unable to pass through relatively tighter spaces. The updated robot footprint is shown in the figure below.

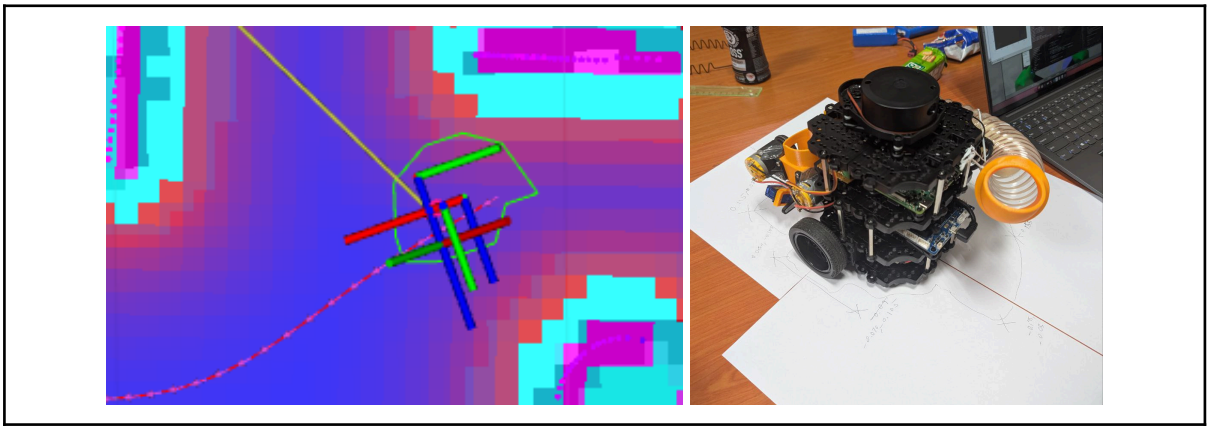


Fig 5.2.7: Screengrab of RViz showing the updated robot footprint, as well as the setup used to measure it

In order for the navigation 2 (nav2) controller to properly avoid collisions, we measured the robot's footprint and input it as a parameter in the nav2 config. The updated robot footprint was crucial in ensuring that collision detection was accurate.

5.3 Integration Testing

5.3.1: Quality of Life (in Debugging)

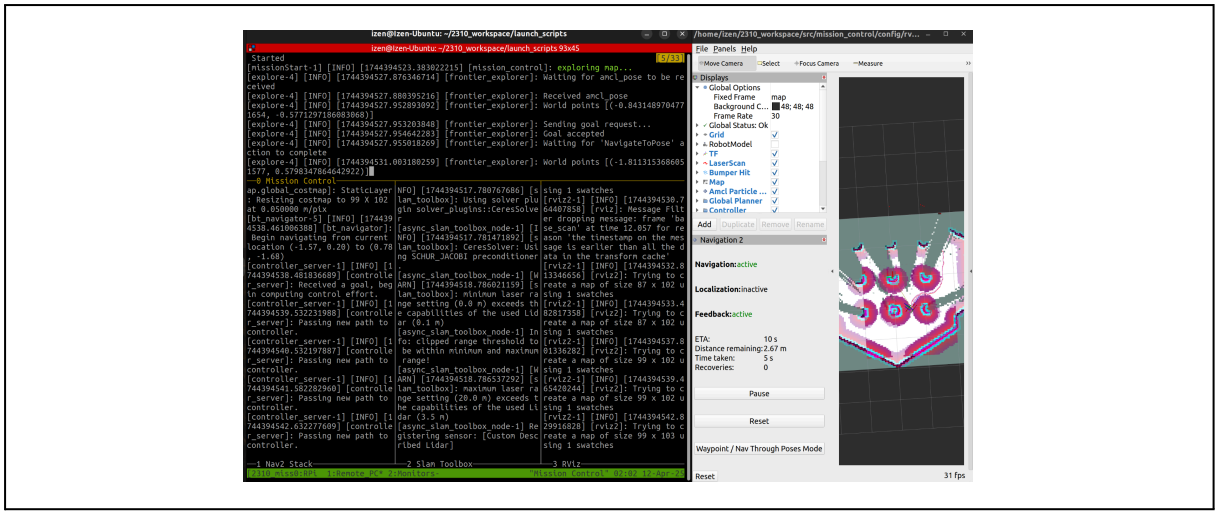


Fig. 5.3.1: Tmuxinator script used for final run

TMux Launch Script: tmuxinator launch scripts allowed us to launch all necessary nodes for testing at once, reducing setup time and avoiding errors such as copy pasting the wrong command. Simultaneously, the easily configurable UI allowed us to monitor several processes at once, such as the Nav2 stack bringup and our mission_control script, enabling easier debugging. Ultimately, during the final run, the script made setup a breeze and even made it user friendly for our end user.

rqt_reconfigure: rqt_reconfigure is a real-time parameter monitor and editor. This was crucial for calibrating the Nav2 navigation parameters as will be discussed below. The discovery of this tool accelerated our testing as it provided immediate feedback on parameter changes.

Map Markers: when integrating casualty_location with the other nodes, it was very useful being able to visualise the casualty locations on RViz. There were many times that this custom helper script that we wrote enabled us to discover bugs such as failing to properly compare between OccupancyGrid coordinates versus World coordinates; and using the right coordinate frame is a crucial part of navigation.

5.3.2: Node Structures - Service Call, Topic Callback

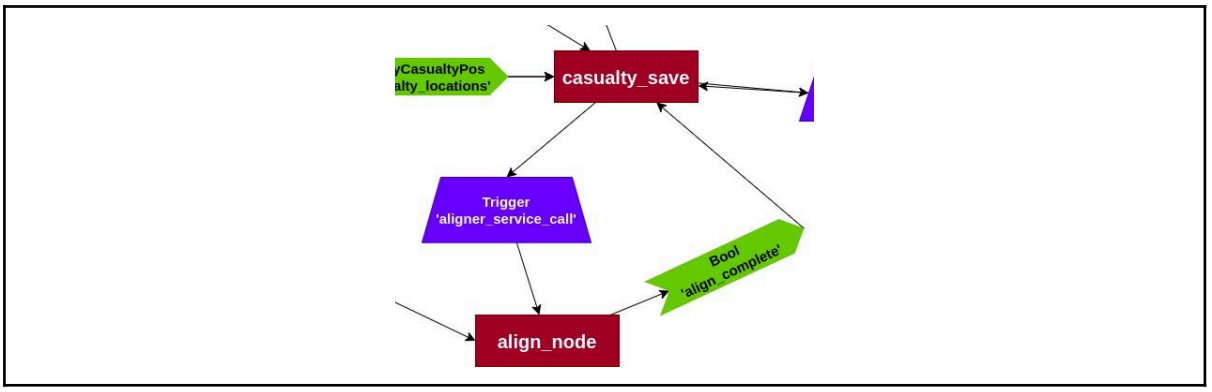


Fig. 5.3.2a: Communication Between casualty_save and align_node

To integrate actions between different nodes, we had to establish a consistent means of communication between them. This is when we discovered the utility of using a Service call and a topic callback. 2 specific examples of this can be seen in 6.4.1 Overview (of Software Stack): between mission_control and casualty_locate, as well as between casualty_save and align_node. In this example casualty_save needs to call align_node service, and can only continue when align_node service has completed its necessary actions. The simplest way to achieve this, we found, was to let align_node publish to a given topic when it has completed its task, thus, we coin this the service call, topic callback structure. If given the time and resources to improve on this, we would certainly try implementing ROS2 Actions, which would be the technically correct way to structure this type of relationship but is harder to understand and implement.

Fig. 5.3.2b: turtlebot_launcher Repository readme.md

Having each node be called via a service was very useful for integration and debugging purposes. For integration, as explained above, using a service call to start the operation of another node functions as a consistent way to ensure that the node has truly been started (as opposed to continuously publishing to a topic and waiting for a callback). This allowed mission_control to easily be integrated with the other scripts. Simultaneously, if there was a need to test the individual function of a script, we could call that one script via a service call. For example, our turtlebot_launcher script is called via a service, which can also be called via command line for debugging purposes (see Fig. 2.3.3 above)

5.3.1: Finite State Machine

In order to ensure logical flow of events, a mission_controller node was implemented with a FSM. This allowed us to easily integrate different nodes into one sequence of events. Another benefit of this was easily being able to add new events into the state machine in case added complexity was needed.

Nav2 Behaviour Trees (BTs): We considered using BTs in our preliminary design review. However upon further research and experimentation, we found that given the complexity of our mission, it would be unnecessary to implement an excessively robust (not to mention difficult to implement) algorithm for our system. We therefore decided to go with the lower cost option of FSM.

5.4 Key Iterations and Fixes

5.4.1: nav2_params.yaml and Porting Nav2 Launch Files

Nav2 works by first generating a global costmap and a local costmap. The global costmap is generated using occupancy grid data from /map, while the local costmap is generated using lidar data from the /scan topic, and is limited to a small area around the robot. The path planner then plans a path between the robot and the goal pose using the global and local costmap. The path is regularly re-evaluated within an interval using the local and global costmap. This path is then passed to the controller that determines the robot's next trajectory. The goal checker then determines if the goal is reached, then marks the action as completed. The action server then returns a result, whether successful or not [16].

In our testing, we found that the stock parameters and plugins selected by Robotis and nav2 itself were inadequate in ensuring consistent navigation behaviour for our use case, which is a robot that navigates relatively tight spaces.

Thus, in order to customise our parameters specific to this project we ported nav2_params.yaml and the Nav2 bringup launch file to our own repository (refer to the mission_control/config repository). This most importantly allowed us to synchronise the parameters between different PCs for testing, which would not be possible with the default Nav2 bringup launch configuration.

Controllers tested:

- **Dynamic Window Based (DWB) Controller:** As the stock controller option, we assumed that it would function the best in most scenarios. Thus, we spent a large amount of time tuning the controller parameters to get it to function well. A major issue we had with DWB controller was its tendency to create trajectories with wide arcs instead of rotating to its path heading. We thus implemented Rotation Shim Controller, detailed below. Another issue we kept running into was the controller scoring trajectories through narrow spaces too low, causing the robot to oscillate in front of a narrow opening. Tuning for obstacle avoidance impacted the controller's ability to follow paths tightly. Thus, we switched to the Regulated Pure Pursuit (RPP) controller.
- **Rotation Shim Controller:** A controller plugin that passes the path to the primary controller, such as DWB or RPP after it has rotated the robot to the goal heading. This allowed for much smoother and predictable behaviour when moving off to a new goal pose.
- **Regulated Pure Pursuit (RPP) Controller:** This controller is good for exact path following. It seemed promising until we discovered that its lookahead behaviour was causing it to round sharp corners, triggering the collision checker constantly near narrow openings and being unable to correct itself. Thus, we switched to Model Predictive Path Integral controller.
- **Model Predictive Path Integral (MPPI) Controller:** This controller worked well out of the box with stock parameters. An interesting and amusing behaviour it had was that it would twirl at the beginning and end of paths to face the desired heading. It never caused any

collisions, and only did so when space was ample. Thus, we decided not to prevent that behaviour. After tuning the path following critics a minor amount, We arrived at the most consistently working controller iteration we had, and kept it for the final run.

Planners tested:

- **NavFn Planner:** This is the default planner plugin. No matter how much tuning was made to the plugin parameters to create paths that kept away from obstacles, the planner prioritised shortest distance paths too much, leading to the robot getting too close to obstacles and colliding.

- **Theta Star Planner:** Just like NavFn, we were unable to get it to follow strictly in the middle of the non-zero cost areas. It created paths that were not in the middle of obstacles, which was the required behaviour for navigating the ramp in the final evaluation.

- **Smac Lattice Planner:** We were unable to get it to use a custom generated lattice, due to lack of documentation of the filepath parameter. When using the stock lattice, it created paths that were too wide, as the turning radius was set too high in the stock lattice configuration.

- **Smac 2D Planner:** This plugin was the last remaining planner that was suitable for differential drive robots. It created quite promising paths out of the box, specifically in the way that paths it created weighed obstacle cost much more than the previous planners we tested. This resulted in paths that stayed much more centered between obstacles even during turns. We used this planner in the final run.

Local costmap tuning:

- **cost_scaling factor:** We increased it from 3.0 to 7.0 to allow a much more gradual gradient in the cost around obstacles.

- **inflation_radius:** We increased it from 0.3 to 0.7 to ensure that there were no zero-cost areas in the costmap. This was to ensure that the planner created smooth paths as close to the exact middle of the obstacles as possible [17].

- **robot_footprint:** We used the robot's actual footprint that was traced and measured. This is detailed in section 5.2.6.

Global costmap tuning:

- **cost_scaling factor and inflation_radius:** Set to the same as in local costmap.

- **robot_radius:** This was set to 0.16, which is the effective radius of our robot. We used robot radius rather than footprint as Smac planner is unable to use robot footprint in its path planning. We used the effective radius to increase cost through narrow spaces, so as to "force" the planner to choose the widest opening to travel through, and to ensure that paths selected by the planner were guaranteed to be able to fit the robot.

The config file was ported to our mission_control package to be used by the launch file included in the same package when launching nav2 stack. This allowed for much easier portability and testing of parameters on different members' computers. We based our launch file off the launch files included in the source code of the nav2 package.

5.4.4: Detection of Bad Goal Pose

Due to the nature of our casualty_locate operations, the code by default chooses a goal pose that is inside a wall. This conflicts with the way Nav2's NavigateToPose works, hence needed fixing. Our quick fix for this was to do a radial search from the chosen goal pose, until a point that was far enough away from the wall was found. Not only was this inconsistent, but as can be seen in our final run on

the third try, this code still had a bug where it would choose a point far away from the wall, but completely outside the maze, which was even worse. Given more time, we would have improved on this process by using the costmap generated by Nav2, which by default encodes for zones which the Turtlebot3 is allowed to navigate to, eliminating this issue.

```
# add waypoint to visited list
x = self.waypoints[0].pose.position.x
y = self.waypoints[0].pose.position.y

self.info_msg(f'visited_frontiers:{visited_frontiers}')

if x in visited_frontiers and visited_frontiers[x] == y:
    self.info_msg(f'Visited {x}{visited_frontiers[x]} already!')
    self.get_logger().warn(f'visited same frontier too many times! aborting!')
    msg = MapExplored()
    msg.explore_complete = True
    self.explored_pub.publish(msg)
    self.info_msg('No More Frontiers')

    # kill the thread
    self.kill_now = True
    return
else:
    visited_frontiers[x] = y
```

Fig. 5.4.4: Code for Detection of 'Bad' Goal Pose

Furthermore, our wfd_nav2 frontier exploration node and casualty_locate node repeatedly encountered bugs with navigation to certain goal poses. For certain problematic goalposes, these nodes would call the ROS2 action NavigateToPose for a goalpose very near to their current pose, this would cause the NavigateToPose to immediately feedback that goalpose is reached, then the node would somehow call the exact same goalpose again, and the cycle continues.

With much experimenting, we eventually found that this was caused by a synchronisation issue between our slam_toolbox node, and the casualty_locate node. The surefire way to fix this was to restart both nodes. But during the final run, we would not be allowed such a solution as manually killing the program would require a restart of the entire run. Hence, our superficial solution (see Fig. 2.3.4 above) to this was to detect whether the same goalpose was being requested too many times, in which case we would force the wfd_nav2 node to choose a different goalpose.

5.4.5: align_node and World Coordinates

Initially, we relied on calculating angles from the Turtlebot3's world coordinates to ensure the correct heading of the turtlebot after the navigation. This was done in the casualty_saver node by calculating the target yaw using the robot's current pose as well as the position of the target. We calculate the difference between the robot's current yaw and the target yaw and send a spin goal to nav2 to rotate that amount. In Gazebo, this approach worked consistently. However, when testing on a real map, this function no longer worked.

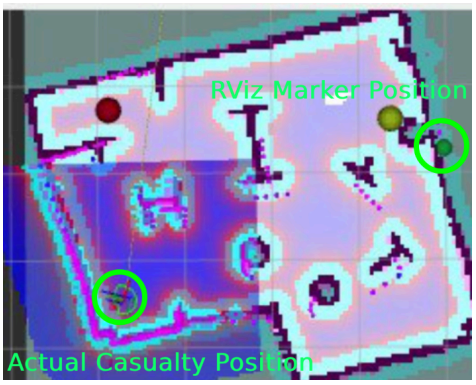


Fig 5.4.5a: RViz monitor showing incorrect position of markers

This taught us how crucial it was to understand the different coordinate systems (transform frames) in doing robotics in general. It seemed mismatched coordinate frames were not only a major source of error, it was difficult to detect, difficult to keep track of and difficult to resolve bugs that are caused by it. As seen in Fig 5.4.5 above, our robot was actually going to the correct coordinates for the casualty, yet the RViz marker we placed was far away.

We were ultimately unable to debug the above issue due to limited time, hence we resorted to a "naive" approach to the problem:

align_node: This node primarily functions in a loop. It subscribes to ir_data, which publishes a string of 8 values associated with the temperature data from the thermal sensor. It takes that string, converts it to an array of floats and finds the index of the highest value. The node instructs the robot to rotate until it senses a heat source. When it does, it rotates the robot to the heat source, and approaches it.

Dynamic threshold: In testing, we discovered that having a static threshold was not enough to determine the validity of a heat source. As the thermal radiation received by the sensor falls off with distance, the effective range we could get with a static threshold was around 0.5 m. Thus, we implemented a dynamic threshold that compared the highest valued entry with the mean. If it was greater than the mean by a certain offset, or if it was greater than a static threshold, the node would consider that index to be associated to the heading of a heat source, and rotate to it. This significantly improved the performance of the node, being able to detect heat sources more than 1.0 m away.

```
goal_msg.pose.position.x = x * self.map_data.info.resolution + self.map_data.info.origin.position.x
goal_msg.pose.position.y = y * self.map_data.info.resolution + self.map_data.info.origin.position.y
```

Fig 5.4.5b: Formula for calculating World Coordinates from OccupancyGrid coordinates

Eliminating Mismatched Transform Frames: Given more time, we would definitely come up with a more robust system for keeping track of coordinate frames. As seen above in Fig. 5.4.6, these conversions were done haphazardly and were not readable. In hindsight, we should have standardised for example that all coordinates be transmitted in World Frame.

6.0: Final Design

Incorporating our key findings from prototyping and continuous iterations, the final product detailed here are the results of rigorous testing and thoughtful analysis. This section, when compared to 4.0 Preliminary Design, truly highlights the collective growth and the innovative solutions we have crafted together.

6.1: System Specifications

Dimension	280mm x 205mm x 190mm (with launcher)
Weight	Approx. 1.5 kg
Center of Gravity	Near the geometric center, slightly rear-biased due to battery placement
Actuators	2 x L430-W250 Dynamixel Motors (For TurtleBot movement) Servo Motors (For loading ping pong ball for launcher operation)
Battery Capacity	11.1V, 1800mAh Li-Po battery
Estimated Battery Life	1.42 hours (under worst-case power consumption)
Payload Capacity	Ability to carry and launch 9 ping-pong balls

6.2: Mission Control

As detailed in 4.1, our original plans for mission control turned out to be too simplistic. The original ‘Search’ phase has now been split into ‘exploration’ and ‘casualty_location’, which reflects the mode of operation of our code: to fully explore the map, before generating a heatmap. The following mission control neatly splits each subtask into its respective ROS2 node:

Phase	Subphase	Activity
exploration	—	Utilise frontier-based exploration in order to generate a full SLAM occupancy grid of the maze.
casualty_location	thermal_mapping	Utilise frontier-based exploration to generate a heatmap of the walls of the maze, assigning a temperature to every wall of the maze
	casualty_identification	Scan through previously generated heatmap for hotspots and identify the positions of 3 casualties
casualty_save	navigate_to_casualty	Navigate towards each casualty, pointing in the general direction of the casualty as determined earlier.
	align_node	Utilise real-time data from the AMG8833 to align the Turtlebot3 to the casualty and close the distance between Turtlebot3 and casualty.
	launcher_service	Receives a service call and fires 3 ping pong balls in sequence as stated by mission requirements.

6.3: Final Hardware CAD

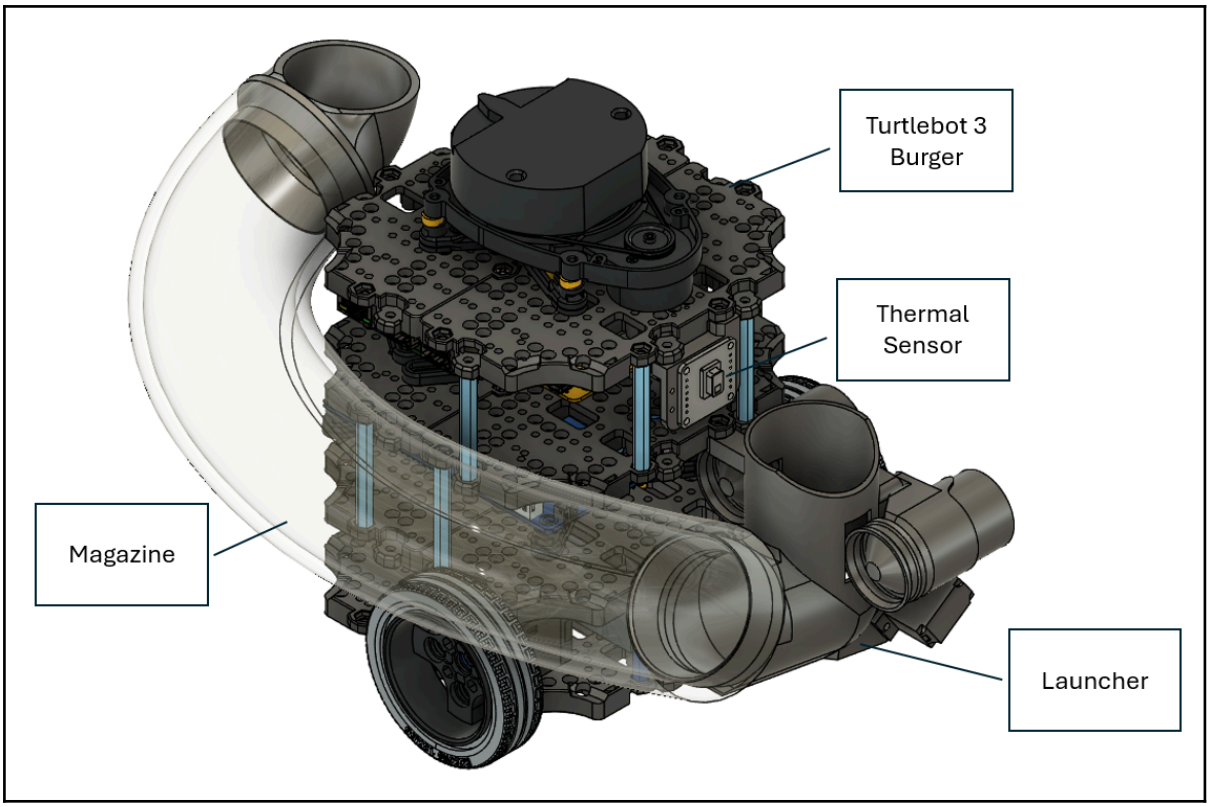


Fig 6.3: Overview of final bot design

Photos and labelled diagrams of parts and subassemblies are shown in Appendix A, while assembly is shown in Appendix B.

6.4: Final Software Stack

In this section, we highlight the main software related integrations and innovations that we have made that were crucial to the success of the project.

6.4.1: Overview

Our original rqt_graph (Fig 4.4.1c) was surprisingly accurate, it focused on our custom nodes' communications with the Nav2 stack and slam_toolbox and such. The following rqt_graph omits the Nav2 nodes and details our custom nodes which focus on mission exploration and mission logic.

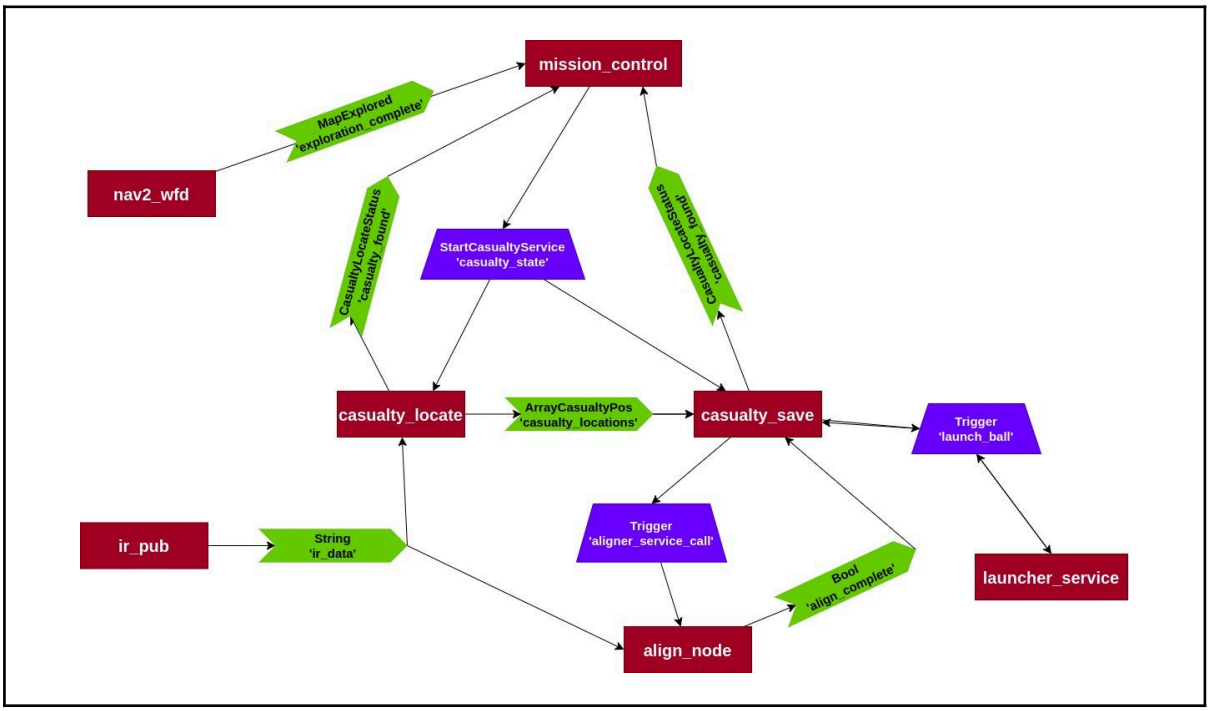


Fig 6.4.1: Simplified rqt_graph showing information flow between nodes

6.4.2: Nodes

The following subsections are of the format: <*package name*>/<*node name*>

With each node, we list some of the potential improvements that have been discussed throughout this report.

One overall area for improvement would be to place all the parameters listed below in a centralised .yaml file, which would make for easier debugging, and would even make it possible to use rqt_reconfigure to debug these parameters on the fly.

6.4.2.1: nav2wfd/explore

Parameter	Description
OCC_THRESHOLD = 10	Cost threshold for determining valid frontiers
MIN_FRONTIER_SIZE = 5	Minimum size of frontier to be considered as valid
REPEAT_LIMIT = 5	Number of times a frontier can be chosen repeatedly before ending exploration

SeanReg's [14] Frontier-Based algorithm that we modified to be compatible with ROS2 Humble. Additional optimizations were added to improve speed of exploration. This, in combination with tuning of Nav2 parameters, yielded a promising exploration algorithm.

Potential Improvements: Better frontier selection. This issue is discussed in section 7.1, where the node would select frontiers that were relatively far away from where the robot was, when there were closer frontiers in the heading of the robot. Most importantly, we could rewrite our own implementation of frontier exploration, which we could then spend less time optimising.

6.4.2.2: casualty_location/casualty_location

Parameter	Description
SENSOR_FOV = 60.0	Field of view of thermal sensor in degrees
POSE_CACHE_SIZE = 10	Number of robot pose data to keep in the cache for raycasting
ODOM_RATE = 20.0	Rate of odometry updates in Hz
DELAY_IR = 0.2	Delay in seconds for IR data processing
CASUALTY_COUNT = 3	Number of casualties to find
DIST_TO_CASUALTY = 3.9	Parameter used to find a valid waypoint DIST_TO_CASUALTY away from goal

This node stores a cache of the robot's corrected pose data from the fusion of /odom and /pose topics. It then uses DELAY_IR (Mentioned in Section 5.2.5) and ODOM_RATE to determine how far back into the cache the node must look to match the data stream from the thermal sensor with odometry.

The casualty_location node locates casualties by using filtered IR data from the on-board AMG8833 with a ray casting algorithm to generate a heat map of the environment. It then finds the grid coordinates on the heat map that have the highest average wall temperature in a 5x5 box. The region

with the highest temperature is then chosen as a heat source. The region around it is suppressed, and the next region with the highest average temperature is identified. This is repeated till 3 heat sources are found. Once the heat sources are identified, it passes the coordinates of the heat sources to casualty_saver. How exactly it creates the heat map is detailed in section 5.2.4.

Potential Improvements: Simultaneous heatmap generation during the frontier exploration phase. We were reluctant to implement this as it would have increased the complexity of the casualty_location code, and would open our code to more errors due the OccupancyGrid constantly resizing during frontier exploration.

6.4.2.3: casualty_location/casualty_saver

Parameter	Description
DIST_TO_CASUALTY = 3.5	Parameter used to find a valid waypoint DIST_TO_CASUALTY away from goal

This node receives the coordinates from casualty_location, then sends goal poses to the TurtleBot3 to move to the heat sources, and then triggers the aligner node, followed by the launcher service.

Potential Improvements: Include robust error handling capability. Lack of error handling led to a rather critical failure during our final run, detailed in section 7.1. In addition, we could use costmap in selection of goalposes for casualty, as explained in section 5.4.4.

6.4.2.4: aligher-node/aligner_node

The aligner_node instructs the TurtleBot3 to orient itself to face the direction of the heatsource. It is called after the Goal to Pose sent by casualty_saver is achieved by the TurtleBot3, before the launcher_service is called and the launcher sequence is triggered.

Potential Improvements: Increase the distance at which the approach to the heat source is stopped. The distance chosen led to the turtlebot ending up in the lethal area of nav2's costmap, leading it to drop the final navigation goal. This is detailed in section 7.1.

6.4.2.5: custom_msg_srv/custom_msg_srv

A node containing custom messages and services used in our Finite State Machine for system integration.

Potential Improvements: Remove unnecessary custom_msgs, just use std_msgs more (detailed in the custom_msg_srv GitHub repository)

6.4.2.6: mission_control/mission_control

Node that integrates all of the nodes running on the system as a finite state machine. Uses service calls and passes messages to the other nodes to control the robot's behaviour. Its package contains launch scripts and navigation parameters for use with the nav2 stack.

Potential Improvements: Include robust error handling capability. Lack of error handling led to a rather critical failure during our final run, detailed in section 7.1.

6.4.2.6: casualty_location/ir_pub

Run on TurtleBot3 SBC. Ignores top four rows of the AMG8833 8x8 output grid, as heat sources are on ground-level. Further filters data by taking the maximum temperature in each column, and publishes a 1d array of floats.

Potential Improvements: Instead of publishing strings, configure it to publish an array of values. This would remove the conversion required for subscriber nodes.

6.4.2.7: turtlebot_launcher/launcher_service

Run on TurtleBot3 SBC. Service to trigger the launcher sequence. Fires in a

fire (wait 4 s) fire (wait 2 s) fire

sequence.

Potential Improvements: Optimise the firing sequence. Though the delays between shots were longer than the required amount to minimise risk of misfiring, they were too long.

6.5: Fulfilment of Requirements

Below are the requirements we planned to fulfil that we set in section 1.1.

Feature	Requirement
1. Fully autonomous robot	A. Using LiDAR sensor to map the surroundings. B. Able to detect heat signals. C. It must not collide with any obstacle. D. Robot needs to approach the heat signal proximity. E. Does not navigate to the same heat signal and fire flares. F. The robot needs to be able to be fully autonomous and not remote controlled.
2. Fire flares to signal S&R teams for immediate survivor assistance	A. Launching mechanism must be able to fire 3 flares at set time interval B. Flares need to be self reloadable as it is fully autonomous C. Flares need to be fired high enough for S&R teams to be able to see.
3. Power system (battery, rasp pi & other electronic components)	A. Battery needs to last long enough for the whole mission. B. Needs to power the launching mechanism, rasp pi, motors, LiDAR and OpenCR. C. Motors require a sufficient amount of battery remaining or it will stop working.
4. Mechanical structure	A. Robot needs to have a relatively low and central center of gravity such that it can navigate uneven areas (such as ramps) without toppling over. B. Robot needs to be not bulky as it needs to navigate through tight spaces. C. Robot needs to be mounted with a launching mechanism with a magazine for flare firing.

Our system has fulfilled all of the above requirements.

6.6: Bill of Materials

No.	Part(s)	QTY	Unit Cost	Total Cost
1	TurtleBot3 and all its accessories	1	Provided by lab	NA
2	Raspberry Pi	1		
3	OpenCR	1		
4	Motor	2		
5	Wheel	2		
6	LiDAR	1		
7	Lipo Battery	1		
8	Ping Pong Ball	10		
9	AMG8833 Thermal Sensor	1		
10	SG90 Servo motor	1		
11	Motors for Flywheels	2		
12	Jumper Wires	NA		
13	M3x4x5 Heat Set Insert	3		
14	M3 Nut	1		
15	M3x10 BHCS	5		
16	M3x6 BHCS	4		
17	M2x6 SHCS	2		
18	Servo Self-Tapping Screw	3		
19	Servo Spline Screw	1		
20	MLX90640BAA	1	35.07	35.07
21	Magazine Pipe	1	13.72	13.72
22	O-rings for flywheels	20	0.053	1.06
23	3D Printed parts (Listed in Appendix B)	NA	8.72	8.72
Total Cost				58.57

$58.57 was spent out of the $80 budget.

7.0: Conclusion

7.1: Lessons Learnt

In conclusion, after having gone through the CDE2310 journey, there are a number of learnings and reflections that we would like to include in this report. They are as follows:

1. **Importance of Documentation:** The complexity of this project taught us the importance of documenting the engineering process. With three main subsystems in our project - electrical, mechanical and software, it was key that team members working on each subsystem knew what the others were developing, in order to avoid compatibility issues when integrating the whole system. Hence, maintaining clear documentation as well as regularly updating it is important for good systems design.

2. **Coding our algorithm from scratch:** When we began working on the frontier navigation code for our system, the rather complex and cumbersome syntax of Python scripts to be run on ROS2 threw us off writing our own frontier-based code from scratch. However, when using readily available algorithms, we found that we were often stuck in a "black box" - simply understanding the code took hours of reading and testing, and optimizing was often difficult because we were working with a codebase that was still rather unfamiliar.

3. **Importance of real-world testing:** A major trend we found throughout the build focus phase in CDE2310 was that a piece of code running on the Gazebo simulation did not imply that it would run perfectly in real life. This was one place where we could have significantly improved - most of our testing was done on Gazebo, and we did not have sufficient time to sort out the host of problems that we encountered once we transitioned to testing in real life.

During our final run, this was especially evident. Our second attempt, which was our most successful, was ultimately disappointing as our robot failed to navigate to the second target due to the lack of error handling in our nodes. The navigation planner had successfully planned a route, but after the robot had approached the first heat source, the robot was now in the lethal cost area. Thus, the controller abandoned the path repeatedly till the navigation failed. This was an issue that we had rarely ever encountered in all of our test runs, and when we did, it was during the exploration phase when the exploration node would simply choose another goal pose, giving the nav2 stack more time to recover, thus "self-correcting" the error. Thus, we did not implement error handling in regards to this issue. We put such a strong emphasis on having the nav2 stack work perfectly, spending such a large portion of our time tuning parameters that we did not put much emphasis at all on error handling. We held the wrong assumption that a system could work perfectly, if all subsystems worked perfectly too. Had we put a stronger focus on error handling and having the mindset of working with an imperfect system, we would probably have finished all the mission requirements in that run.

It was also during our final run that we discovered the inefficiency of our Frontier Based navigation algorithm as well. The Turtlebot3 kept toggling back and forth between chosen frontiers on opposite sides of the map, seemingly conducting a breadth-first search. This was highly inefficient compared to other possible algorithms, such as favouring closer frontiers to reduce travel time. Relating this back to real-world testing, we should have tested the Turtlebot3 in more complex environments (similar to the final run one) to sieve out such inefficiencies.

Additionally, the Dynamixel motors of the TurtleBot3 stopped functioning around 35-40% battery charge, which gave us a rather narrow battery window when we could conduct testing, after which we had to charge the batteries again. This made testing extremely inefficient, with long down-times due to the need to charge the batteries once again.

7.2: Next Steps

If given more time, or that we had to go through the entire process again, there are some things that we would do differently. Firstly, as mentioned in the previous point, we would put a greater focus on real-life testing, to sort out problems much earlier than we did. Additionally, with regards to the software, we would have taken a ground-up approach, taking time to learn how to code effectively with ROS2 and placing emphasis on writing our own code. Even in our experience, we found it much easier to optimize and edit the complementary nodes that we wrote when compared to modifying the adapted frontier navigation algorithm. Hence, in hindsight, the time and resources spent in learning how to code from the ground-up would have been worth it when you compare it to the time spent debugging and optimising.

8.0 References

1. Reddy, N. S. M. (2024). A Comparative Study of Maze-Solving Algorithms: Performance, Complexity, and Practical Applications in AI and Robotics. *International Journal of Advance Research and Innovative Ideas in Education*, *Vol-10*(Issue-5), 1892-1914. https://ijariie.com/FormDetails.aspx?MenuScriptId=234046

2. Fevgas, G., Lagkas, T., Argyriou, V., and Sarigiannidis, P. (2022). Coverage Path Planning Methods Focusing on Energy Efficient and Cooperative Strategies for Unmanned Aerial Vehicles. *Sensors,* 22(3), 1235. https://doi.org/10.3390/s22031235

3. J. Lu, B. Zeng, J. Tang, T. L. Lam and J. Wen. (2023). "TMSTC*: A Path Planning Algorithm for Minimizing Turns in Multi-Robot Coverage," in *IEEE Robotics and Automation Letters*, vol. 8, no. 8, pp. 5275-5282, Aug. 2023, doi: 10.1109/LRA.2023.3293319. https://ieeexplore.ieee.org/document/10175546

4. L. E. Kavraki, P. Svestka, J. . -C. Latombe and M. H. Overmars. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation,* vol. 12, no. 4, pp. 566-580, Aug. 1996, doi: 10.1109/70.508439. https://ieeexplore.ieee.org/abstract/document/508439

5. Connell D and Manh La H. (2018). Extended rapidly exploring random tree–based dynamic path planning and replanning for mobile robots. *International Journal of Advanced Robotic Systems.* 2018;15(3). doi:10.1177/1729881418773874 https://ieeexplore.ieee.org/abstract/document/508439

6. Topiwala, A., Inani, P., & Kathpal, A. (2018, June 10). *Frontier based exploration for Autonomous Robot.* arXiv.org. https://arxiv.org/abs/1806.03581

7. *Interfaces—ROS 2 Documentation: Humble.* (n.d.). https://docs.ros.org/en/humble/Concepts/Basic/About-Interfaces.html

8. Robotis-Git. (n.d.). *GitHub - ROBOTIS-GIT/ld08_driver: ROS package for TurtleBot3 LD08 Lidar.* GitHub. https://github.com/ROBOTIS-GIT/ld08_driver

9. *Autonomous Driving—ROBOTIS e-Manual.* (n.d.). https://emanual.robotis.com/docs/en/platform/turtlebot3/autonomous_driving/

10. Engineering Dads. (2023, June 6). *Powerful Ping-Pong launcher* [Video]. YouTube. https://www.youtube.com/watch?v=MX3i9wQe3_g

11. Butler, N. (2022, January 7). Automatic Ping-Pong Launcher - Nollaig Butler - Medium. *Medium*. https://nollaigengineering.medium.com/automatic-ping-pong-launcher-c77dc140d33b

12. HaiderAbasi. (n.d.). *GitHub - HaiderAbasi/ROS2-Path-Planning-and-Maze-Solving: Developing a maze solving robot in ROS2 that leverages information from a drone or Satellite's camera using OpenCV algorithms to find its path to the goal and solve the maze. :).* GitHub. https://github.com/HaiderAbasi/ROS2-Path-Planning-and-Maze-Solving

13. AniArka. (n.d.). *GitHub - AniArka/Autonomous-Explorer-and-Mapper-ros2-nav2: An autonomous exploration package for ROS 2 Humble using Nav2 and SLAM Toolbox. It enables robots to explore unknown environments, build maps, and recover from localization failures, solving the kidnapped robot problem effectively.* GitHub. https://github.com/AniArka/Autonomous-Explorer-and-Mapper-ros2-nav2

14. SeanReg. (n.d.). *GitHub - SeanReg/nav2_wavefront_frontier_exploration*. GitHub. https://github.com/SeanReg/nav2_wavefront_frontier_exploration

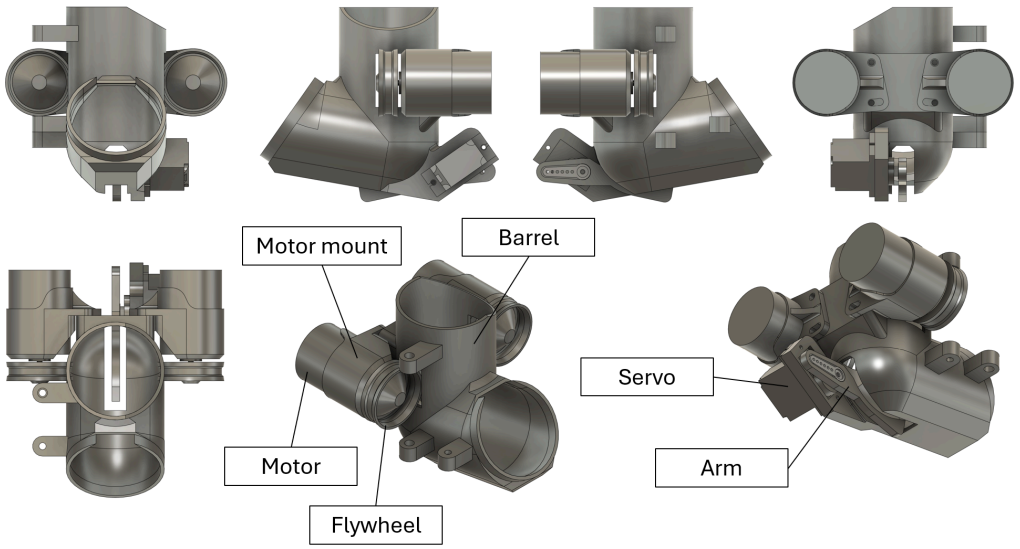
15. Makerportal. (n.d.). *GitHub - makerportal/AMG8833_IR_cam: Python codes for development of a real-time thermal camera using a Raspberry Pi computer and AMG8833 infrared array*. GitHub. https://github.com/makerportal/AMG8833_IR_cam

16. *Navigation Concepts—Nav2 1.0.0 documentation*. (n.d.). https://docs.nav2.org/concepts/index.html

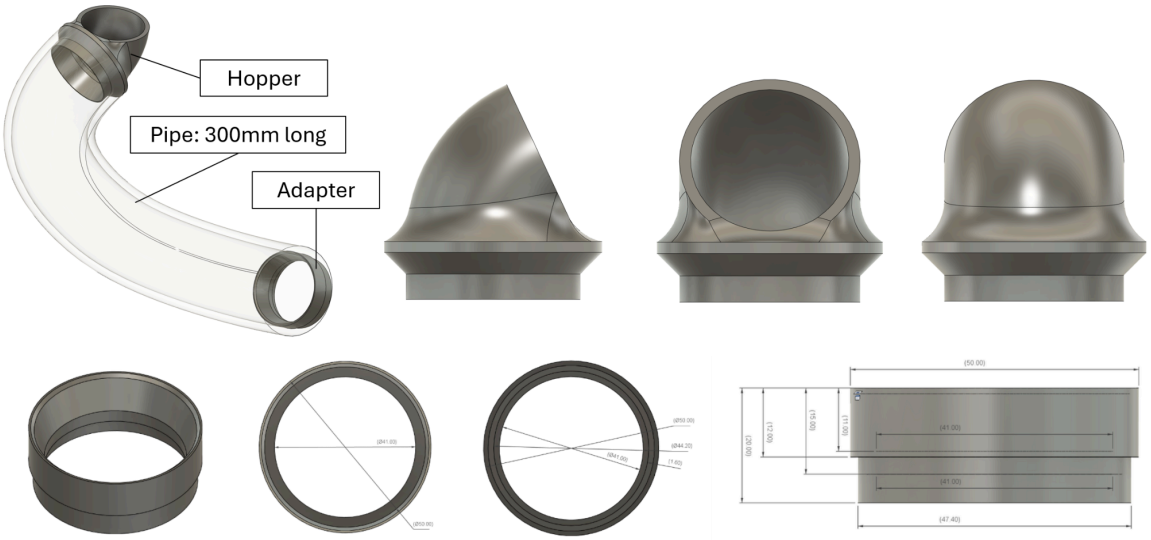
17. *Tuning Guide—Nav2 1.0.0 documentation*. (n.d.). https://docs.nav2.org/tuning/index.html#planner-plugin-selection

Appendix A

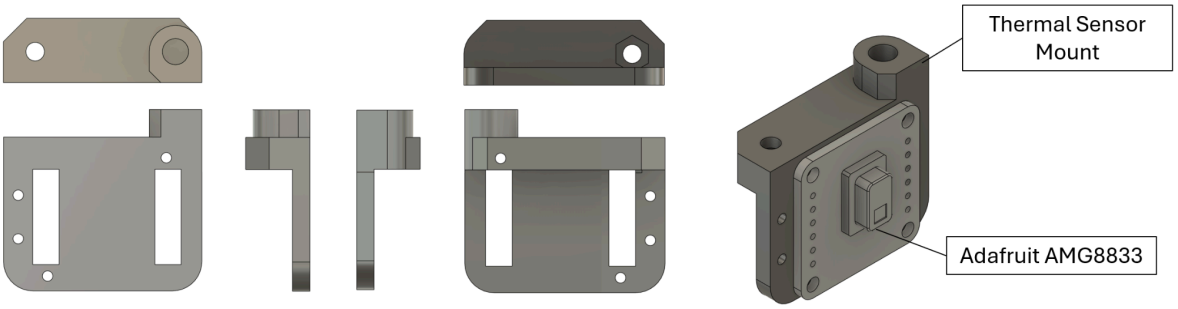
Launcher:



Magazine:



Thermal Sensor:



Appendix B

Parts to be 3D printed with support

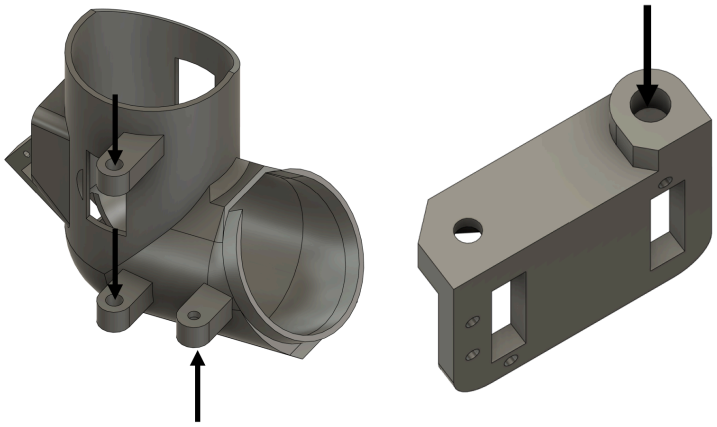
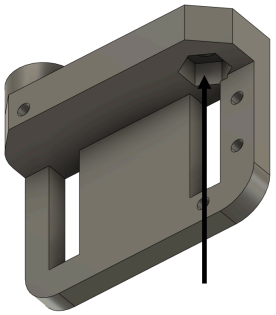
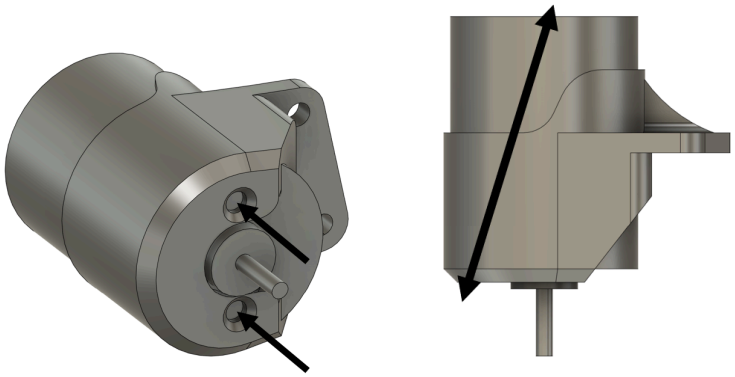
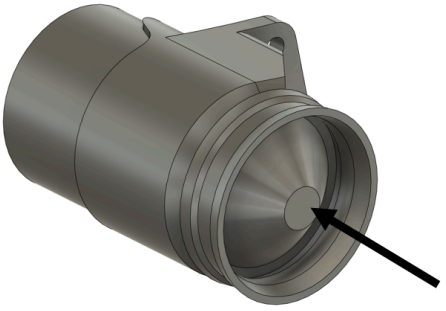
Part Name	Quantity
Barrel	1
Motor Mount	2
Hopper	1

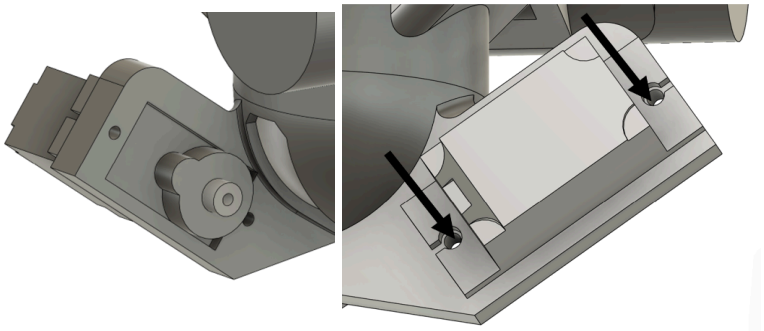
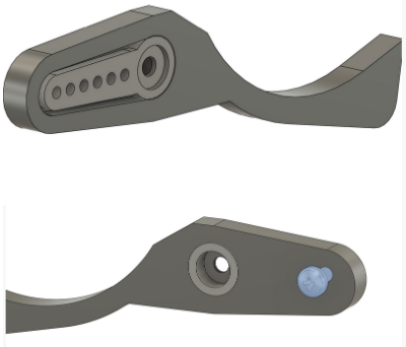
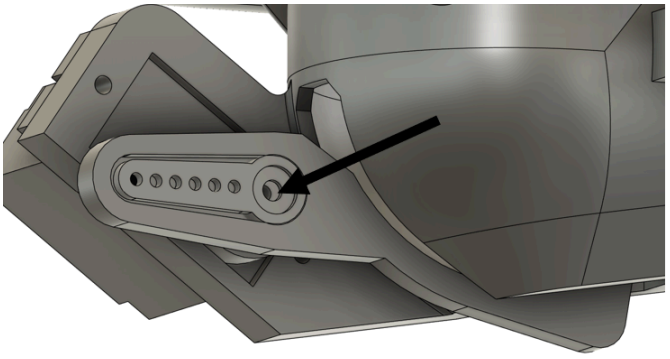
Parts to be 3D printed without support

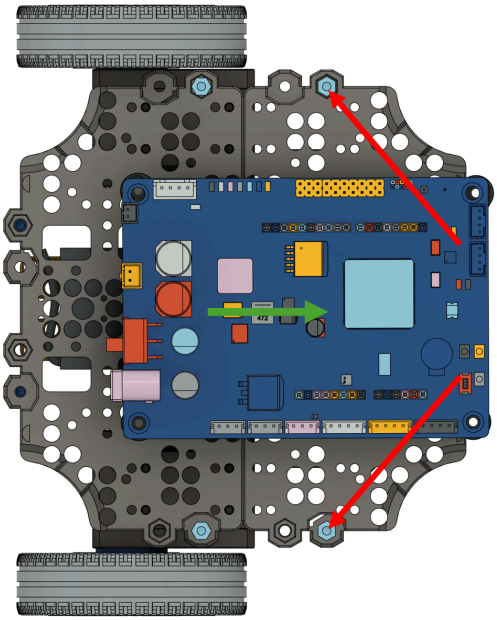
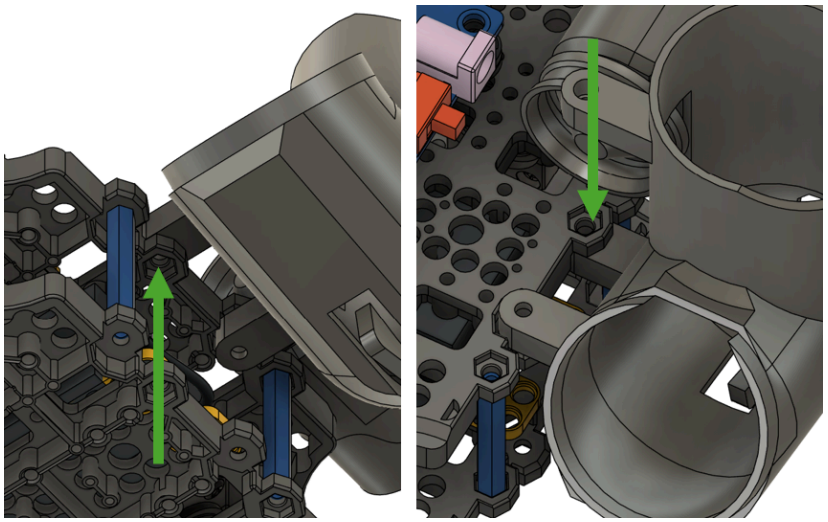
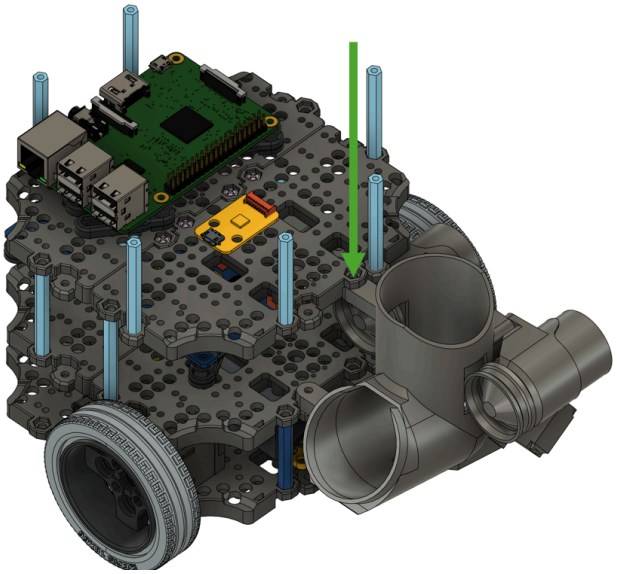
Part Name	Quantity
Flywheel	2
Arm	1
Adapter	1
Thermal Sensor Mount	1

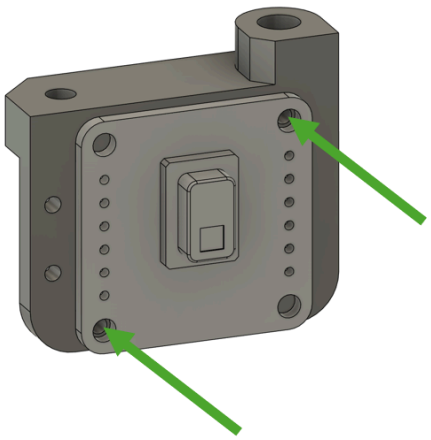
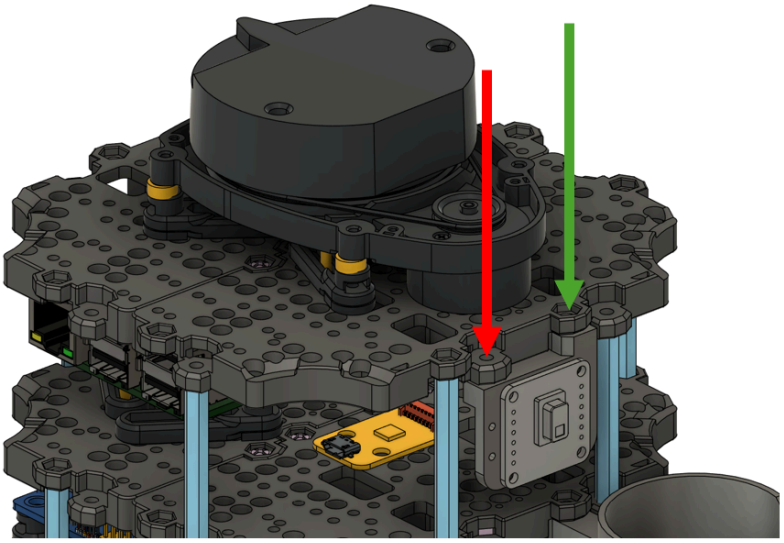
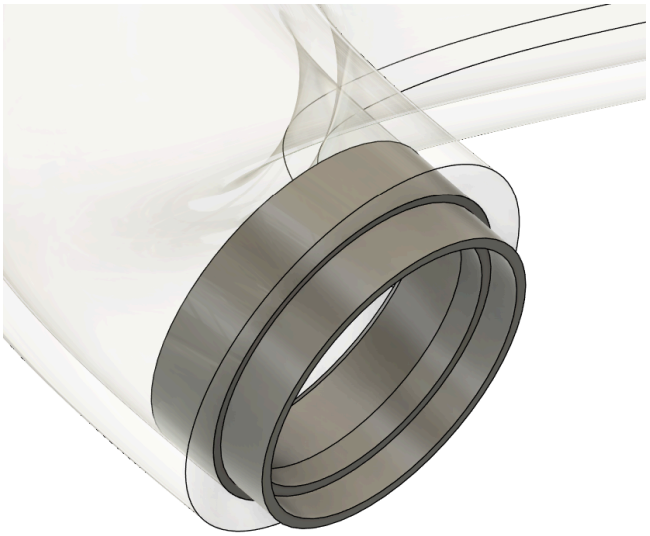
All parts designed with and to be printed with ± 0.4 mm tolerance.

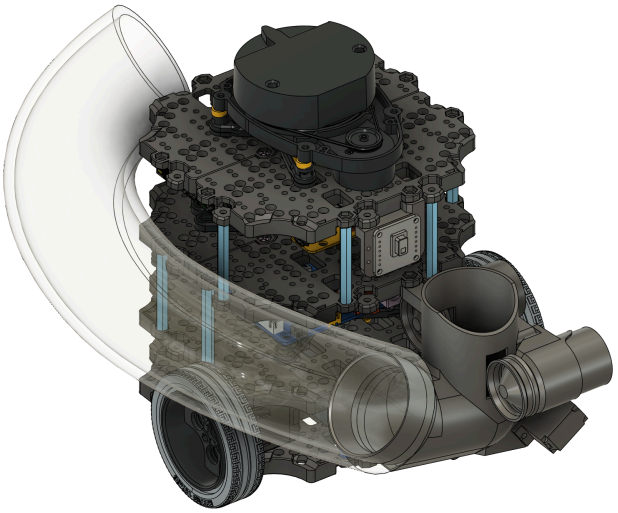
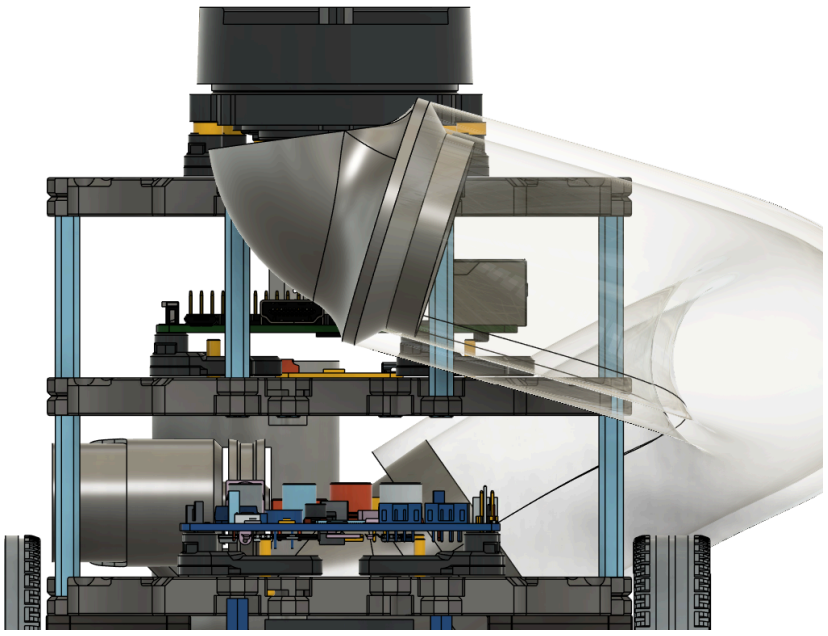
Hardware Assembly:

Step	Images	Description
1		Place and set 3 heat set inserts into the **Barrel** and **Thermal Sensor Mount** in the locations indicated by the arrows.
2		Trap an M3 nut in the slot of the **Thermal Sensor Mount** shown by the arrow
3		Attach the motor to **Motor Mount** using screws (if possible; they are not metric). Wrapping tape tightly around the mount and motor in the direction shown in the right image is adequate. Do for both sides.
4		Press fit **Flywheel** onto motor shaft. Do this for both motors and flywheels. Wrap the o-rings around the grooves of the flywheel.

5		Screw **4x M3x10** screws into the holes in the **Barrel** to attach the **Motor Mounts.** Slots go on the bottom.
6	—	Test the motor directions then wire them together according to the wiring diagrams.
7		Attach the Servo to the **Barrel** using the 2x self-tapping screws
8		Use 1x self-tapping screw to attach the **Arm** to the Servo Horn.
9		Use 1x servo spline screw to attach the **Arm** to the Servo Spline.

10		Partially disassemble the turtlebot to show the 2nd layer. Relocate the standoffs (red arrows) and shift the OpenCR back by at least 18 mm. Re-adjust the mounts appropriately.
11		Attach the **Launcher** sub assembly onto the turtlebot's 2nd layer using 2x M3x6 screws.
12		Reattach the 3rd floor onto the 2nd. Use 1x M3x6 screw to attach the **Launcher** to the 3rd layer.

13		Use 2x M2x6 screws to attach the AMG8833 to the **Thermal Sensor Mount.** Wire the AMG8833 to the RPi.
14		Use 1x M3x10 screw (red) and 1x M3x6 screw (green) to attach the **Thermal Sensor t**o the 4th layer. Reattach the 4th layer to the 3rd.
15	—	Cut ducting pipe to 300 mm length
16		Superglue the **Adapter** to the pipe. Take note of the orientation of the **Adapter**.

17		Superglue the Magazine **Adapter** to the Launcher **Barrel.** Wrap the Magazine **Pipe** around the bot and secure it with wire ties.
18		Ensure proper orientation of **Hopper** before supergluing it to the Magazine **Pipe**