CDE2310 Final Report G2

Group 8

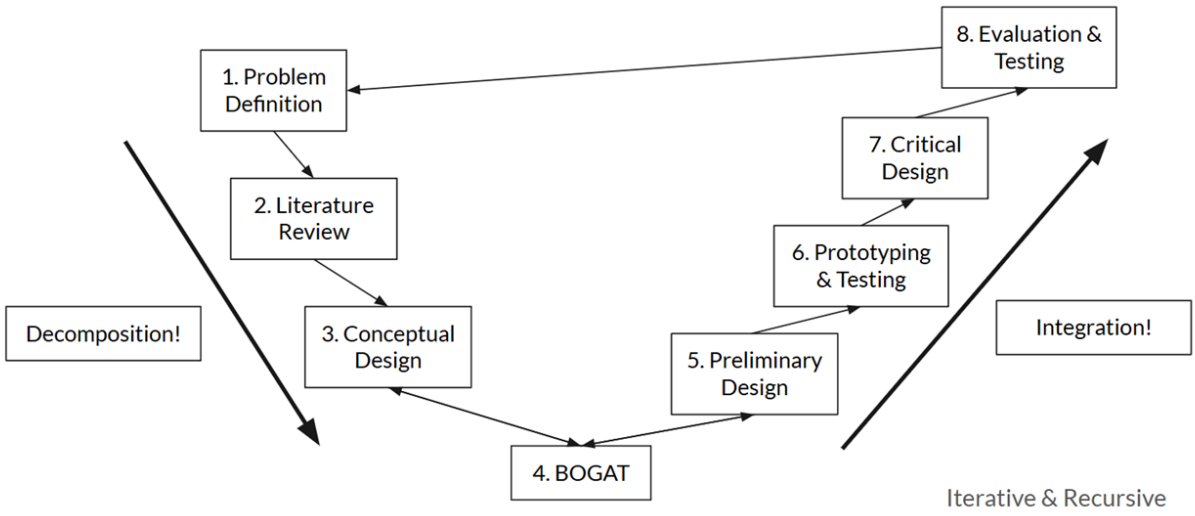
Student Name	Matriculation Number
Thia Yang Han	A0308175A
Justin Matthew Tunaldi	A0302732R
Mayukh Ghosh	A0312084N
Kim Hyunjin	A0286710A

Chapter 1 - Introduction

1.1 Course Content Description

This project report aims to document the learning journey of Group 8 throughout the course CDE 2310, following the design process of Zelephant bot from mission definition to its final evaluation. We adopted the CDE 2310 V-Model framework shown below.

1.2 NUS CDE 2310 V-Model



1.3 Organisation of Report

For the purpose of clarity and ease of access, we have systematically organized our documentation into five distinct documents. Each document focuses on a specific aspect of the project, ensuring that all relevant information is easily accessible and well-categorized.

The complete set of documents includes:

01 - Project Report (Contains Assembly Manual and Troubleshooting guide)
02 - Software Setup Guide
03 - CG Derivation
04 - End User Documentation
05 - Other Relevant Mechanical Files

All of the above documents are available in our GitHub repository under the folder titled **"Documentations".** The software programs used can also be found in our GitHub repository. You may access them at your convenience for a detailed understanding of each component of the project. This is the link to our GitHub repository:
https://github.com/hyunjinkim1112/r2auto_nav_CDE2310

Chapter 2 - Project Description

2.1 Background Information and Context

Natural disasters such as earthquakes can lead to disastrous consequences, such as survivors being trapped in dangerous locations. These locations may be unsafe as there may be unstable structures, collapsed buildings, or hazardous environments, making it risky for search and rescue teams to enter these places. Hence, using autonomous robots minimises the risk of secondary collapses or hazards for human rescuers. It also has advantages of being able to navigate through rubble and tight spaces, allowing access to more areas for search and rescue.

2.2 Mission Requirements

Our group has been tasked to design and build attachments to the turtlebot burger to be able to accomplish the following tasks:

1. The turtlebot is to autonomously navigate the disaster zone (maze), with walls height under 0.535m
2. The turtle bot is to be able to accurately detect heat signals, mimicking human body temperature of around 30 degree celsius.
3. Upon heat source detection, the turtlebot is to launch flares (ping pong balls) in close proximity with the following time delay, 2 seconds - 4 seconds - 2 seconds
4. Bonus mission criteria: Able to scale and carry out the mission on a 20 degree ramp.

2.3 Mission Analysis

2.3.1 Autonomous navigation within the maze.

1. The turtlebot must be able to navigate autonomously without colliding into obstacles

2.3.2 Identification of heat source

1. The turtlebot should be able to detect the heat signature and navigate towards it
2. The turtlebot must not collide into it

2.3.3 Firing of flares

1. The turtlebot should have a firing mechanism
2. Turtlebot should be able to hold 9 flares to complete the mission
3. The flares must be able to reach the minimum height required
4. The flares should observe 2s - 4s - 2s firing sequence

2.3.4 Power budgeting

1. Power source should be able to provide sufficient power for multiple runs of the maze for contingency purposes.
2. Turtlebot's weight and speed should be optimised to conserve power
3. Systems should be powered down if unused to reduce power draw

2.3.5 Mechanical stability

1. Turtlebot should be able to move around at any speed
2. Turtlebot should ensure flares do not drop or get dislodges prematurely
3. Turtlebot should not have any loose parts which could fall off affecting the run

2.3.6 Procurement plan

1. Project budget of S$80
2. Delivery cost and time of delivery should be accounted for to ensure time for testing and integration

2.4 Technical Performance Measures

1. 80% success rate in locating both survivors in 10 runs
2. 90% success rate in launching flares above height of 1.5m out of 10 attempts
3. Mission completion time less than 10 mins 80% of the time (without bonus mission) in 10 runs
4. 90% success rate in scaling the ramp when attempting bonus mission

2.5 Mission Safety Considerations

1. Avoid Subjecting Zelephant bot to harsh conditions such as dropping, burning, puncturing , crushing or exposure to water as such actions can compromise its integrity and functionality.
2. Any observed damage should prompt immediate disconnection from power sources for inspection and repair.
3. Adhere strictly to the prescribed battery charging procedures to safeguard against potential hazards. Use only the designated charger provided with the system, as employing incompatible chargers or prolonged charging periods may compromise battery performance and pose risks of fire or explosion.
4. Avoid running Zelephant bot for a prolonged period of time to prevent system overheating leading to irreversible damage of system components.

2.6 Workload Distribution For Final Evaluation

2.6.1 Pre-Mission Checks

1. End user documentation checking - Yang Han
2. Software Setup - Mayukh
3. Calibration of heat sensor - Hyunjin
4. Verification of appropriately-sized inflation radius - Matthew

2.6.2 Contingency response

1. Zelephant Bot Retrieval - Hyunjin
2. Flare Reset - Matthew
3. Physical System Diagnostics - Yang Han
4. Software System Diagnostics - Mayukh

Chapter 3 - Literature Review

3.1 Review of Current Literature on SLAM

The core technology enabling most autonomous robots to navigate is the **Simultaneous Localization and Mapping (SLAM) algorithm**. SLAM allows robots to build a map of their environment while simultaneously determining their position within that map, using a loop closure technique to eliminate errors accumulated from dead-reckoning data. This loop closure technique works by having the robot recognize previously visited locations, helping it to reorient itself and explore new areas.

SLAM can be implemented using various methods, such as **Visual SLAM (vSLAM)**, which utilizes cameras, and **LiDAR-based SLAM**. vSLAM itself includes several different approaches, such as **feature-based, direct, RGB-D**, and **event-based** methods [1]. The most prominent method is feature-based vSLAM, where features are tracked using **feature-descriptor algorithms** (e.g., SIFT, SURF, ORB) [2]. This allows the robot to detect previously visited features (such as a statue) and estimate its pose/location on the map, hence performing global map optimization. The map in vSLAM is typically constructed using a technique called **bundle adjustment**, which combines data from multiple camera angles to generate 3D information about the relative positions of objects in the environment. **Keyframes** are scenes where substantial data on the environment can be captured, and the most common keyframe selection algorithm is **heuristic-based uniform sampling by appearance**, which selects keyframes based on the number of detected features [3].

While feature-based vSLAM is low-cost and power-efficient, it is sensitive to texture and lighting conditions [1], making it less effective in environments with poor visibility, such as disaster zones where smoke or fire could hinder the algorithm's performance.

In contrast, LiDAR-based SLAM can be implemented using either **occupancy maps** or **graph-based approaches.** Occupancy maps are well-known and relatively easy to implement, but they are typically limited to small 2D environments. This is because generating an occupancy map for large-scale 3D environments requires significant memory. Additionally, occupancy maps have a more challenging loop closure process. Unlike vSLAM, which stores feature information, LiDAR-based systems primarily capture distance data between the robot and objects in the environment, along with geometric descriptors for object shapes. Since occupancy maps rely solely on distance data, the robot lacks features to track, making it difficult to accurately relocalize itself.

On the other hand, graph-based LiDAR SLAM uses both distance data and geometric descriptors to build a map, and relies on nodes (to represent the robot's pose) and edges (to represent the relative transformations between poses). This approach works well for large 3D environments and facilitates easier loop closure, making it a more suitable option for rescue robots. However, a significant challenge with graph-based SLAM is that it requires highly accurate estimates of the edges to construct an accurate map [1].

3.2 Review of Existing Methods and Designs on Autonomous Navigation, Ball Launching System, and Heat Detection System

In addition to our comprehensive review of Simultaneous Localization and Mapping (SLAM) techniques, we have conducted an in-depth analysis of various algorithms, engineering methods, and conventional design approaches relevant to the key subsystems of our project. These subsystems include the autonomous navigation system, the ball launching mechanism, and the heat detection module.

For each subsystem, we evaluated multiple design alternatives and engineering strategies currently used in research and industry. A summarized description of each method, along with its respective advantages and disadvantages, is presented in the following sections. This comparative review not only guided our own system design choices but also provides a valuable reference for future improvements or related projects.

	Description	Advantages	Disadvantages
Autonomous Navigation			
Exploration			
Frontier-based exploration	guides a robot to the boundary between known and unknown space to efficiently map an environment	Efficient for mapping unknown environments.	Can struggle in cluttered or dynamic environments due to reliance on clear frontier detection.
Sampling-based exploration	Randomly samples candidate points in the environment and evaluates them based on utility or information gain.	More flexible in complex or irregular environments, especially when frontiers are hard to define.	May require more computation and can be less efficient due to randomness in sampling.

Global Path Planning			
Wavefront algorithm	Spreads cost values from the goal across a grid to guide path planning.	Simple and guarantees a path in grid maps.	Not ideal for dynamic environments; lacks real-time obstacle avoidance.
A*	A heuristic-based graph search algorithm that finds the shortest path by combining actual cost and estimated cost to the goal.	Efficient and optimal with a good heuristic, making it faster than Dijkstra's.	Performance heavily depends on the quality of the heuristic function.
Dijkstra's Algorithm	A graph search algorithm that finds the shortest path from the start to all other nodes using uniform cost.	Guarantees the shortest path and works well without needing a heuristic.	Slower than A* because it explores all possible paths equally, even those far from the goal.
RRT (Rapidly-exploring Random Tree)	A sampling-based algorithm that builds a tree of possible paths by randomly exploring the search space.	Handles high-dimensional or complex environments well, especially in continuous spaces.	Paths are often suboptimal and require post-processing or smoothing.
Local Path Planning / Obstacle Avoidance			
Dynamic Window Algorithm	Selects real-time motion commands by evaluating velocity windows within robot dynamics.	Effective for local obstacle avoidance and smooth motion.	Can get stuck in local minima; needs a global planner for full navigation.
Mapping			
Cartographer	Provides real-time simultaneous localization and mapping using pose-graph optimization	Provides simultaneous localization and mapping in both 2D and 3D for	Requires extensive professional tuning using hardware platforms with

		multiple platforms	high-quality odometry
SLAM Toolbox	A set of tools and capabilities for 2D SLAM built upon the OpenKarto SLAM library; also a pose-graph optimization method	Reliably map large spaces in real-time, enables serialization of maps for continued mapping, allows for manual manipulation of the pose-graph, and comes with several operating modes.	It's primarily designed for 2D SLAM; not suitable for 3D mapping applications like drones or robots with depth sensors.
<u>Localization</u>			
AMCL	Uses particle filters to localize the robot	enable robust state estimation in dynamic environments and in the presence of noisy sensor data	performs poorly in dynamic environments with moving obstacles or changes in layout
GPS Localization	Uses GPS sensor to localize the robot	Popular approach for outdoor environment setting	Subject to dynamic change in the environment
Ball launching system			
Flywheel	Propels a projectile (ping pong ball) by using a pair of rapidly spinning wheels that grip and accelerate the ball as it passes between them.	Can shoot the ball straight upward	High power consumption to run the motors
Solenoid Plunger	Launches the ball with the linear motion created by the activation of the solenoid coils	Simple to implement	Difficult to shoot the ball high

Spring mechanism	launches a ball by storing potential energy in a compressed or stretched spring and then releasing it to convert the stored energy into kinetic energy	Does not require extra motors; relative easier to implement	Difficult to shoot the ball high
Lever	Uses the lever arm to amplify force with the power system to launch the ball	Relatively easier to implement	Difficult to shoot the ball high and straight
Heat detection system			
IR thermal camera	Captures a thermal image by detecting infrared radiation across a surface or area	Allows non-contact, wide-area temperature visualization	More expensive and complex than point sensors.
Temperature sensor	Measures temperature at a single point via physical contact.	Inexpensive and precise for localized measurements.	Only provides temperature at a specific point, no spatial info.

Chapter 4 - Conceptual Design

4.1 Design Objective

Our robot is designed to operate fully autonomously within a predefined maze environment. It is capable of performing a sequence of complex tasks without human intervention. The primary functions of the robot include:

1. **Autonomous Navigation:**
   The robot navigates through the maze using onboard sensors and algorithms to map its

surroundings, avoid obstacles, and determine an optimal path toward its target destination.

2. **Heat Source Detection:**
   Once within proximity of the target area, the robot utilizes a heat detection system to accurately identify and locate a heat source. This enables precise targeting in the subsequent task.
3. **Ping Pong Ball Launching:**
   After detecting the heat source, the robot activates its launching mechanism to deploy a ping pong ball aimed at the region around the heat source. This action simulates a task-specific response, such as firefighting or object delivery.
4. **Ramp Climbing (Optional):**
   In scenarios where the environment includes elevation changes, the robot is optionally capable of scaling a ramp. This function demonstrates its ability to handle multi-terrain navigation, enhancing its versatility.

Overall, the robot showcases a combination of intelligent perception, decision-making, and mechanical actuation, highlighting its effectiveness in performing complex tasks autonomously.

4.2 Preliminary ConOps

Below is the flowchart that shows the overall ConOps of the robot to complete the mission, which was presented during the Preliminary Design Review.

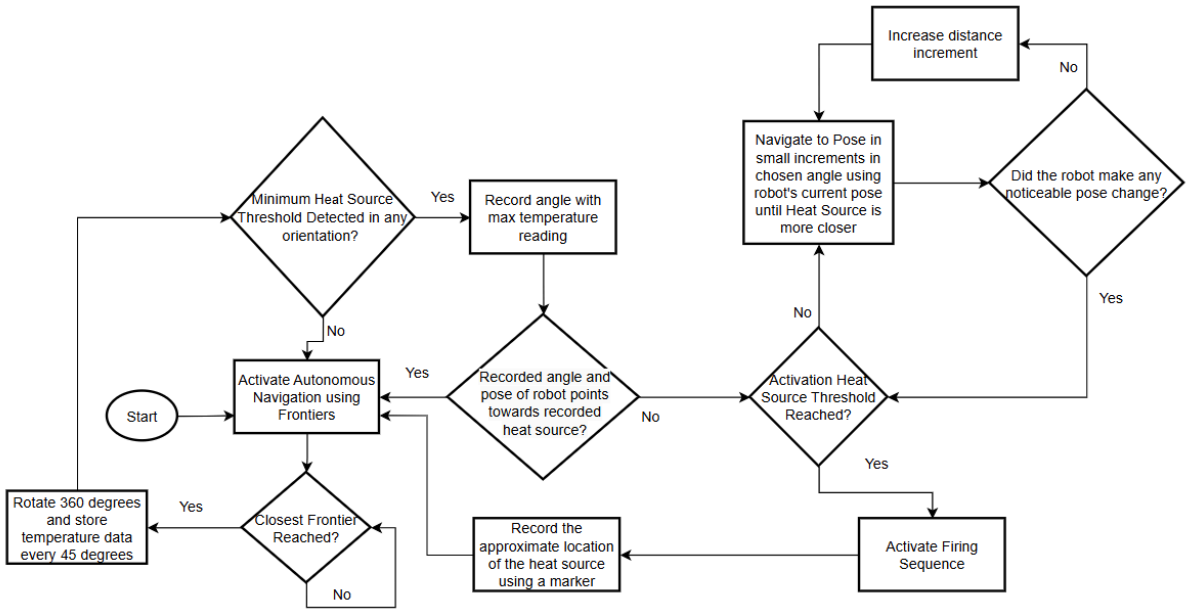


Fig 1: Preliminary ConOps Flowchart

Chapter 5 - Viability Analysis and Design considerations

After researching various components and algorithms, our team collaboratively evaluated the pros and cons of each option, carefully considering their feasibility from mechanical, electrical, and software integration perspectives. Based on this analysis, we determined which systems were best suited for inclusion in our preliminary design. The scope and outcomes of our discussion following the development of the conceptual design include, but are not limited to, the following points.

The software should be designed with a focus on both efficiency and reliability. The primary objective is to locate the three 'Heat Sources' and launch ping-pong balls, rather than to map the maze in the shortest possible time. With that in mind, the final robot should be programmed to stop at every heat source it detects and follow a 'shoot first, map later' strategy. This means it will initiate the firing sequence as soon as it gets close enough to a detected heat source, even if the maze is not fully mapped. This approach eliminates the need for a second traversal to reach each heat source again, and would hence significantly improve the mission success rate while reducing overall time and power consumption.

5.1 Exploration Algorithms

1. Mechanical: We require the robot to not be too wide to ensure compatibility with the maze environment and manoeuvrability through narrow passages.
2. Electrical: An inefficient algorithm would reduce the battery life and could lead to power outage before the mission is completed. Hence, the algorithm should be efficient and should not require too much power to run.

5.2 Heat Detection System

1. Mechanical: Heat detection range is limiting possible mounting points for the heat sensor. We require the sensor to be mounted between the second and third waffle plate of the turtlebot. As the turtlebot may be required to observe the heat source from a distance, the heat sensor is required to face slightly downwards
2. Electrical: Wires should be easily routable and not be too long as I2C signal integrity will decrease if the wire is too long.

5.3 Storage and Feeder System

1. Mechanical: Not blocking the LIDAR , we decided that the system should be less than 50cm tall.

2. Mechanical: Enough mounting points to reduce stress on the system, ensuring even weight distribution on each fastener.

5.4 Firing System

1. Mechanical: We wanted our system to use a flywheel launcher as it would optimise space utilisation. As opposed to a spring based system, we would not require it to be recocked which could be a key source of error. We could also ensure consistency as deformation of the flywheel is unlikely to pose a major concern.
2. Electrical: Motor choice is able to operate with 11.1v DC. Our battery supply is a 11.1v DC source. Hence we require the motors to be able to take at least 11.1v DC.

Chapter 6 - Preliminary Design

6.1 Navigation System

```
Require: queue_m // queue, used for detecting frontier points from a given map
Require: queue_f // queue, used for extracting a frontier from a given frontier cell
Require: pose // current global position of the robot

queue_m ← ∅
ENQUEUE(queue_m, pose)
mark pose as "Map-Open-List"

while queue_m is not empty do
    p ← DEQUEUE(queue_m)

    if p is marked as "Map-Close-List" then
        continue
    if p is a frontier point then
        queue_f ← ∅
        NewFrontier ← ∅
        ENQUEUE(queue_f, p)
        mark p as "Frontier-Open-List"

        while queue_f is not empty do
            q ← DEQUEUE(queue_f)
            if q is marked as {"Map-Close-List","Frontier-Close-List"} then
                continue
            if q is a frontier point then
                add q to NewFrontier
                for all w ∈ adj(q) do
                    if w not marked as {"Frontier-Open-List","Frontier-Close-List", "Map-Close-List"} then
                        ENQUEUE(queue_f,w)
                        mark w as "Frontier-Open-List"
            mark q as "Frontier-Close-List"
        save data of NewFrontier
        mark all points of NewFrontier as "Map-Close-List"
    for all v ∈ adj(p) do
        if v not marked as {"Map-Open-List","Map-Close-List"} and v has at least one "Map-Open-Space" neighbor then
            ENQUEUE(queue_m,v)
            mark v as "Map-Open-List"
    mark p as "Map-Close-List"
```

Fig 2: Wavefront Frontier BFS Pseudocode [4]

Our wavefront frontier detection algorithm is built upon a global, naive breadth-first search (BFS) approach, as outlined in the pseudocode provided in [4]. This algorithm systematically explores the occupancy grid map starting from the robot's current position and expands outward in a wavefront pattern, identifying frontiers, boundaries between known free space and unexplored regions.

The key idea behind using BFS is its ability to traverse the map layer by layer, ensuring that all reachable areas are considered in a structured manner. During traversal, each cell is checked for adjacency to unknown cells while being part of a known free space. If such conditions are met, the cell is classified as part of a frontier.

This method provides a simple yet effective way to detect potential exploration targets. However, since it's a naive BFS, it doesn't incorporate any heuristics or optimization strategies to prioritize frontiers based on utility, proximity, or safety. As a result, while the algorithm ensures comprehensive detection, it may include frontiers that are suboptimal, such as those too close to obstacles, too small to be useful, or already partially explored.

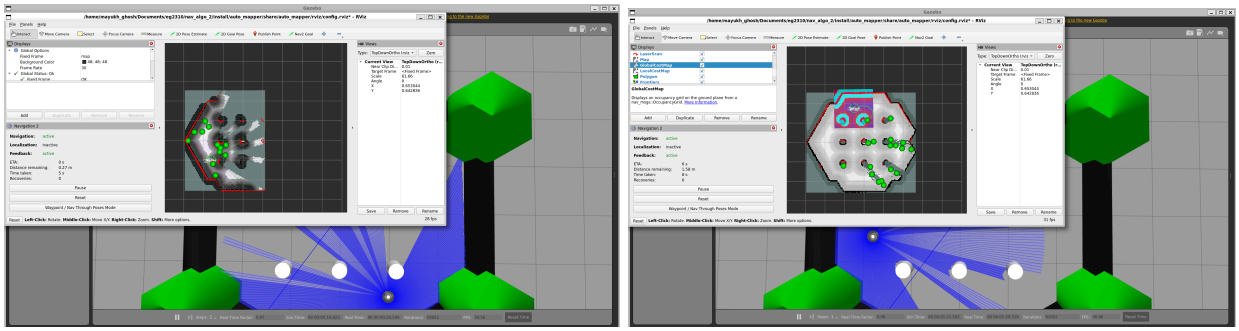


Fig 3: Screenshots of Frontier-based exploration in Gazebo Classic Simulation Environment

The Navigation 2 Stack was evaluated within the Gazebo simulation environment using an initial implementation of our frontier-based wavefront autonomous exploration algorithm. Upon launching the program, the navigation process would commence automatically and continue uninterrupted until the entire environment had been explored and no unexplored frontiers remained.

While the system successfully initiated and executed autonomous exploration, several issues were observed during testing. One major limitation was the algorithm's method of selecting frontiers. It frequently chose frontiers located directly adjacent to obstacles, which posed significant risks for collision or failed path planning due to narrow clearance margins.

Additionally, some selected frontiers were situated too close to the robot's current position, resulting in inefficient navigation behavior with minimal exploratory gain.

Another observed issue was the selection of frontiers that were already within previously explored regions. This redundancy led to unnecessary movements and time-consuming re-evaluations of areas that no longer contributed to mapping progress. These behaviors suggested the need for frontier selection logic, one that incorporates obstacle proximity filtering, minimum distance thresholds, and better distinction between truly unexplored and already visited areas.

Moreover, the Navigation 2 controller plugin chosen in this preliminary design was the Rotation Shim Controller with the primary controller being the DWB local planner controller. While this setup worked fine under basic simulation conditions, several issues were observed during real-world testing and in more complex simulation environments like the turtlebot3 house environment.

Firstly, the DWB controller often struggled with tight spaces and dynamic obstacle avoidance since the paths it chose were always dangerously close to walls and obstacles, which we believe were mostly due to the inherent biases present in the path align and goal align critic. It also showed difficulty in generating smooth paths when obstacles were close to the goal, which often caused the robot to get stuck or oscillate in place.

Secondly, the Rotation Shim Controller, although somewhat useful for improving stability by preventing the wide, sweeping turns often made by the DWB controller, sometimes introduced delays in path execution, particularly when the robot needed to make frequent orientation adjustments. This issue became especially apparent when the robot attempted to rotate near obstacles, where it struggled to turn confidently due to its proximity to surrounding objects. As a result, the robot often hesitated or failed to complete the rotation, leading to increased mission completion times and occasional unpredictable behavior when transitioning between rotational and translational motion.

Lastly, we had decided to use SLAM Toolbox instead of Cartographer for several key reasons. Firstly, SLAM Toolbox offers better integration with the Navigation 2 stack. This led to smoother operation, fewer compatibility issues, and easier debugging during development. In our simulation testing, SLAM Toolbox also demonstrated more stable mapping, with fewer distortions and more reliable loop closures, especially in environments with complex geometry. Another significant factor was the flexibility of SLAM Toolbox, which supports both synchronous and asynchronous modes, which allowed us to optimize SLAM performance according to the computational load and mission needs. Furthermore, the documentation and community support for SLAM Toolbox is more extensive and up-to-date compared to Cartographer. This

made development much more straightforward since the information on how to tune parameters was more understandable, and also ensured that the system is more scalable and maintainable in the long term.

6.2 Firing System

Aim: To fire ping pong balls in a 2s-4s-2s sequence 1 metre high reliably after the robot reaches close enough to the heat source.

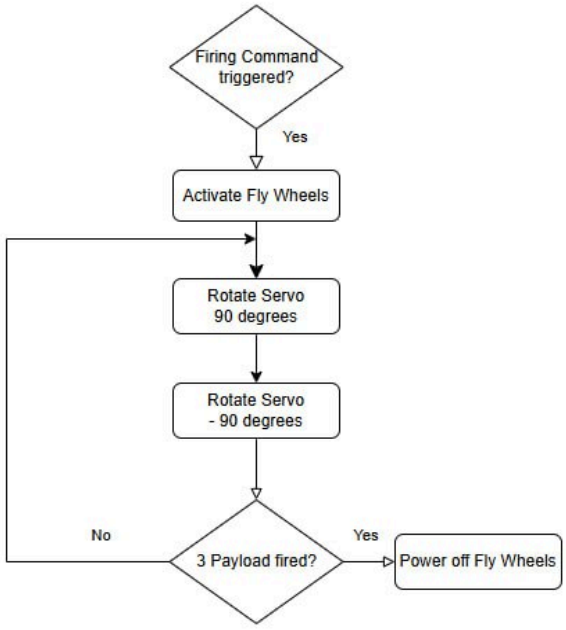


Fig 4: Flowchart of Firing Sequence

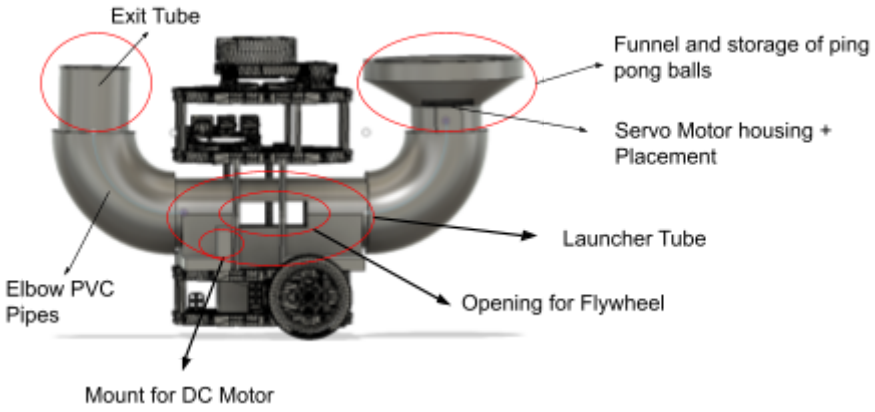


Fig 5: Preliminary Mechanical Design of the Robot

The launcher tube will be mounted on the 2nd layer of the turtlebot to ensure that the ball feeder does not block the LIDAR. We have chosen to use a funnel as the feeder system as it will allow us to carry as many balls without stacking them on top of another. There were a few considerations made as result of this preliminary design:

1. Using a funnel means that more than 1 ball could fall through at a time. Hence we chose to use a servo motor to control the timing of the feeder system

2. Since the 2nd layer of the turtlebot is occupied, the OpenCR which was originally placed there is now shifted to the 3rd layer, and the Raspberry Pi which was on the 3rd layer is now shifted to the underside of the 4th layer of the turtlebot.

3. An extra standoff on the 2nd layer of the turtlebot is required as the height of 1 standoff is only 4.5 cm and considering a ping pong ball has to flow through it, which is 4 cm in diameter, that leaves 0.5 cm to consider the thickness of the tube, thickness of the base of the launcher, etc, which is just not enough

The figure below shows how a ping pong ball would travel through the system

1. All of the balls would be stored inside the funnel and the initial position of the servo arm will ensure the balls stay in place

2. Once the command is triggered, the servo arm would quickly rotate out and back in to allow only 1 ball to flow through to the launch tube

3. The flywheel will then launch the ball through the elbow pipe and out of the exit tube

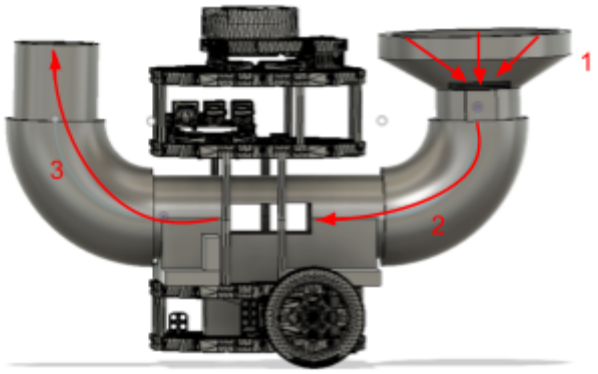


Fig 6: Direction of travel of a ping pong ball through the payload system

6.3 Heat Detection System

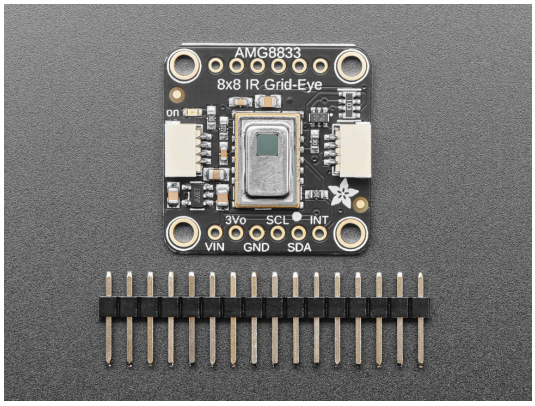


Fig 7: AMG8833 Infrared Sensor

The IR detection system features a thermal imaging camera that transmits data to a Raspberry Pi. We've selected the Panasonic Infrared Array Sensor Grid-EYE, model AMG8833, for this purpose. This sensor is well-suited for integration with the Raspberry Pi, as it operates on a 3.3V power supply and communicates via the I2C interface. One of the key advantages of the AMG8833 is its ability to detect temperatures ranging from -20°C to 80°C. It provides an 8x8 pixel thermal image, allowing us to calculate the average temperature in the camera's field of view by focusing on the central grid pixels, which is what we needed to know for our thermal detection algorithm.

6.4 Preliminary Electrical System Architecture

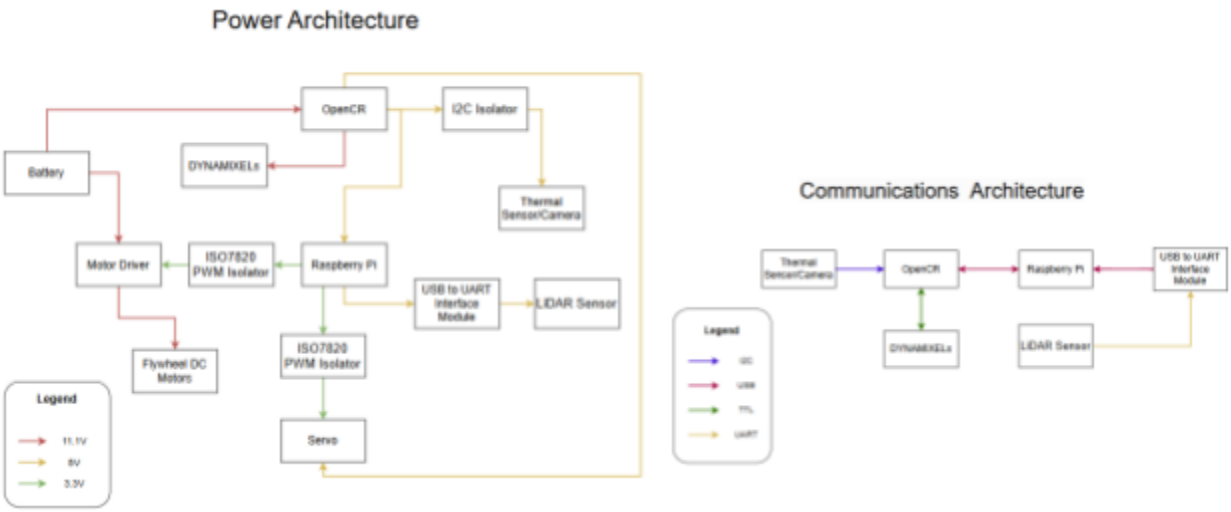


Fig 8: Preliminary Power and Communications Architecture

Initially for the power architecture, we thought that PWM isolators could be used in order to provide electrical isolation, which would prevent noise or damage from power fluctuations in high-power components like the Flywheel DC motors. Likewise, we believed that I2C isolators would be needed to ensure signal integrity, especially if the thermal camera is placed further away from the OpenCR. For the communications architecture, protocols such as CAN bus were not chosen as they introduce unnecessary complexity for this system and were hence. As the system does not require a distributed network with multiple devices, there was no justification for the use of CAN.

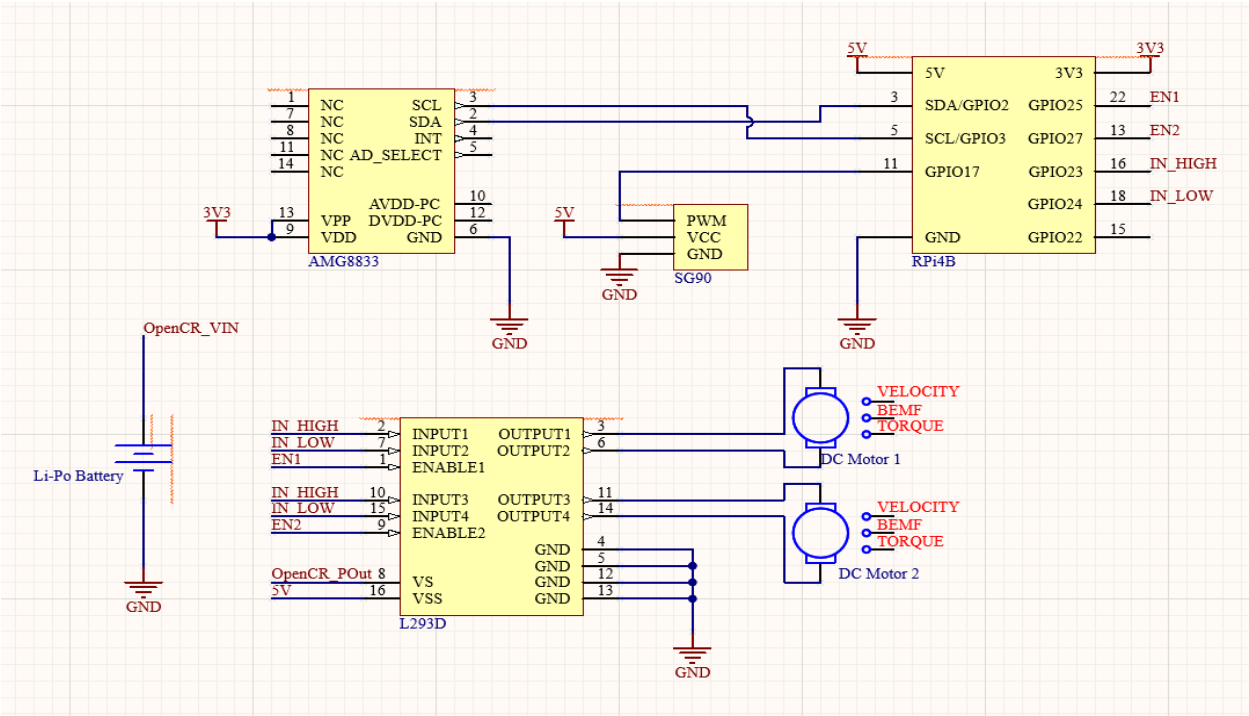


Fig 9: Preliminary Electrical Wiring Diagram

We initially planned on using the L293D motor driver to control the flywheel DC motors since it had a max continuous current of 600mA per channel which we believed to be more than enough to handle both motors. However, since the L293D was not available, we then decided to use the L298N in the final design which has the same configuration, and can keep enabled pins permanently turned on through the 5V regulator it has on board.

Chapter 7 - Prototyping and Testing

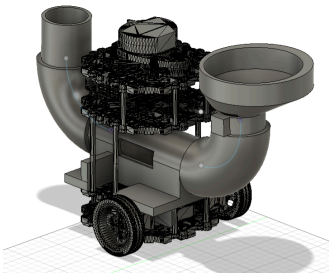
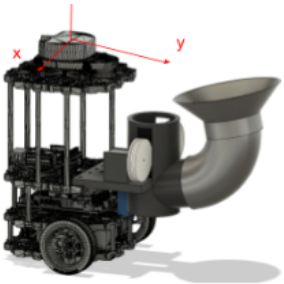
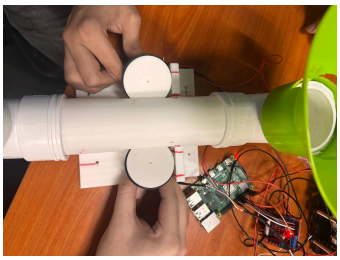
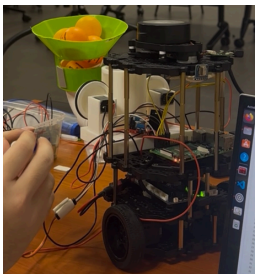
7.1 Ball Feeding Mechanism & Launching Mechanism

For the launching mechanism, a 12V JF-0520B solenoid plunger was tested with the low-fidelity prototype of the paper tube. From the testing, we figured out that a solenoid plunger can push up the ping pong ball by a small height, not enough to launch the ball high enough. Thus, we decided to select flywheel motors as our launching mechanism.

To find the best motor specification for our flywheel, 3 different motors have been tested:

- 9V DC motor
- 12V DC motor
- 24V DC motor

Different versions of design and proofs of concept are shown in the table below.

	Prototype 1	Final Design
CAD		
Proof of Concept		 
Note	When we tested the design, we realized that a significant amount of kinetic energy is lost when it hits the angled part of our connector, not being able to launch the ball 1 metre high. Thus, we decided to move on to the next design where the flywheels are placed vertically.	The biggest issue with the design is that placing the launcher significantly distorts the center of gravity. It could move normally in a straight line or a turn but will fall as it goes through the ramp. To minimize this effect, we decided to position the launching mechanism at the front, where it can be better supported by the wheel placement. We considered adding a ball caster either at the back of the robot or beneath the launcher to improve stability, but these options resulted in undesirable movement or more vulnerable design. Ultimately, we addressed the center of gravity issue by adding approximately 400g of weight to the rear of the robot as a counterbalance.

7.2 Testing and Improvement Log

Throughout our designing process we have had 2 main designs, a conceptual design and a low fidelity prototype, we will be documenting the process of selecting our final design

7.2.1 Testing and Improvement Log - Low Fidelity Cardboard Prototype

Test description	**Issue(s) Detected**	**Status**	**Renders Solution Unfeasible**	**Improvements Made/Remarks**
Firing system				
Able to launch ping pong ball above 1m	Solenoid unable to impart enough energy in the ball to travel 1 m vertically	Failed	Yes	Nil, we decided to use a flywheel based system instead.

7.2.2 Testing and Improvement Log - Prototype 1

General				
Able to mount onto turtlebot	The space between waffle plates was not enough for the tube in our prototype.	Failed	No	We decided to increase the space between the plates using standoffs. Trade Offs: the turtlebot was taller and less stable.
Turtlebot able to move around	Nil	Pass	No	Nil
Does not block the LIDAR	Nil	Pass	No	Nil
Firing system is able to operate with power supplied from OpenCR.	Nil	Pass	No	Nil
Storage				
Able to hold 10 ping pong balls	Nil	Pass	No	Nil
Balls able to slide down along the tube	Nil	Pass	No	Nil

Ball does not get stuck along the path	Nil	Pass	No	Nil
Motor				
Motors are spinning in the correct direction to propel the ping-pong balls	Nil	Pass	No	Nil
Motors able to start consistently	Nil	Pass	No	Nil
Motors able to spin flywheels at sufficient velocity to ensure ping-pong balls hit the intended height	The balls were able to be launched above the walls of the maze, which was the initial requirement. Due to the requirements change, we were unable to hit the 1.5 target set.	Failed	Yes	Despite preliminary calculations indicating feasibility we were unable to get the ball to the height of 1.5 as per the revised conditions due to the energy lost from direction change.
Feeder				
Turtlebot is able to control time interval between firing	Due to the bend design, we were unable to accurately ensure only one ball is fire, a common occurrence was firing 2 or 0 balls	Failed	Yes	We tried using a servo motor to allow only 1 ball to enter, However this was largely unsuccessful.

7.2.3 Testing and Improvement Log - Final Design

General				
Able to mount onto turtlebot	The space between waffle plates was not enough for the tube in our design	Failed	No	We decided to increase the space between the plates using standoffs. Trade Offs: the turtlebot was taller and less stable.

Turtlebot able to move around	Centre of gravity of robot misaligned causing instability	Fail	No	We decided to add 400g of counterweights.
Does not block the LIDAR	Funnel was too tall, blocking the lidar	Fail	No	We removed the top 40% of the funnel only retaining its ability to convert ping pong balls into a steady stream
Firing system is able to operate with power supplied from OpenCR.	Nil	Pass	No	Nil
Storage				
Able to hold 10 ping pong balls	Nil	Pass	No	Nil
Balls able to slide down along the tube	Balls tend to get stuck between the PVC connector and the 3D printed part	Fail	No	We created a platform between the connectors to ensure it is leveled to reduce the ball getting stuck
Ball does not get stuck along the path	Due to the weight of a single ball being miniscule, it might not reach the launching mechanism	Fail	No	We used 10 balls instead of 9 so that the 9th ball will be pushed into the launcher.
Motor				
Motors are spinning in the correct direction to propel the ping-pong balls	Nil	Pass	No	Nil
Motors able to spin flywheels at sufficient velocity to ensure ping-pong balls hit the vertical 1.5m mark	Nil	Pass	No	Nil

Motors able to fire ping-pong balls consistently.	Nil	Pass	No	Nil
Servo				
Turtlebot is able to control time interval between firing	Nil	Pass	No	Nil
SG90 is able to feed the ping-pong balls on command.	Nil	Pass	No	Nil
SG90 servo motor is able to feed the ball into the flywheel	Nil	Pass	No	Nil
Feeder				
Feeder tube is wide enough for ping-pong balls to roll through.	Nil	Pass	No	Nil

7.3 Temperature Sensing Algorithm

AMG8833 thermal camera has been tested separately with the minimal code. After testing the thermal camera itself, it was attached to the robot to develop the navigation logic during heat detection.

At the beginning, once the thermal camera detects the temperature higher than the detection threshold, the robot was programmed to move forward for a short period of time. However, we found out that if there is an obstacle in front of the heat source, the robot will hit the obstacle if it blindly moves forward. Thus, we decided to use a Navigation 2 goal to move toward the heat source in order to avoid any obstacle.

Chapter 8 - Critical Design

After many iterations of testing and refinement, we can now proudly present our final system for the ZelephantBot. Although there have been multiple changes with our final design that

came with our newfound knowledge on the subsystem's capabilities and shortfalls. This part of the documentation is meant to document the improvements made from the preliminary design and the considerations we made.

8.1 Key Specifications

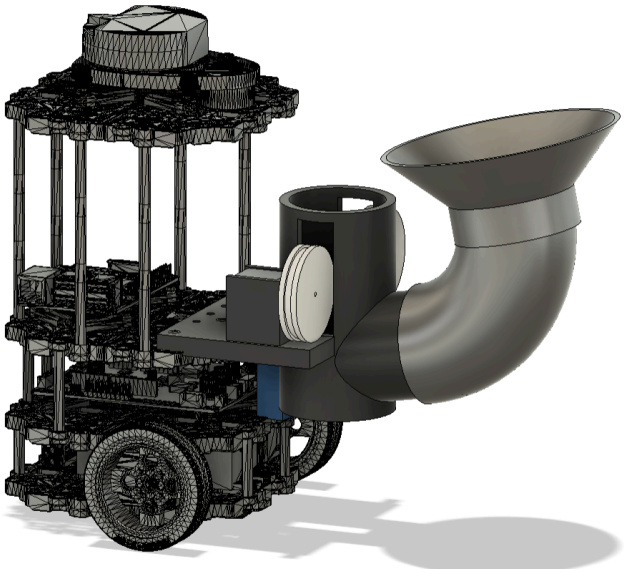


Fig 10: Turtlebot Assembly Visualisation

Max linear Speed (m/s) 0.22	**Max Rotational speed (deg/s)** 62	**Battery Capacity (mAh)** 1800
Total Mass 1.445 kg (without counterweights)	**Dimension W, L, H (cm)** 18, 35, 26	**Centre of Gravity X,Y,Z (cm)** 0.05, 3.84, -13.2
Navigation Algorithm Frontier-based exploration	**Temperature detection system** AMG8833	**Firing System** Dual Flywheel
Software Ros2 Humble	**Chassis** Robotis Co Turtlebot 3	

8.2 Robot Configuration

Our final integrated system, the ZelephantBot, is configured as shown below.

S/N	Description	Qty
01.	Lidar	01
02.	Raspberry Pi	01
03.	Launcher Tube	01
04.	Ball Feeder	01
05.	PVC Elbow Connector	01
06.	Flywheel	02
07.	Flywheel Motor	02
08.	SG90 Servo Motor	01
09.	OpenCR	01
10.	Dynamixel Motor	02
11.	LiPo Battery	01

8.3 Mechanical Design

8.3.1 Firing Design

Our final design has completely changed from the preliminary design after the testing due to 1 main issue. The ball does not travel 1 metre high (as we were told to at the time) since it loses too much momentum and kinetic energy going through the elbow pipe as it is launched from the flywheel.

After clarification on week 11, we were told that the ping pong ball only needs to be launched above the wall, which was only 53.5 cm in height. This meant that our preliminary design could have worked with some minor changes, since it could consistently reach that height.

However, since it was informed just 1 week before the test run, the design was already scraped to change 1 main thing, converting a horizontal launch tube to a vertical one.

8.3.2 Firing Sequence

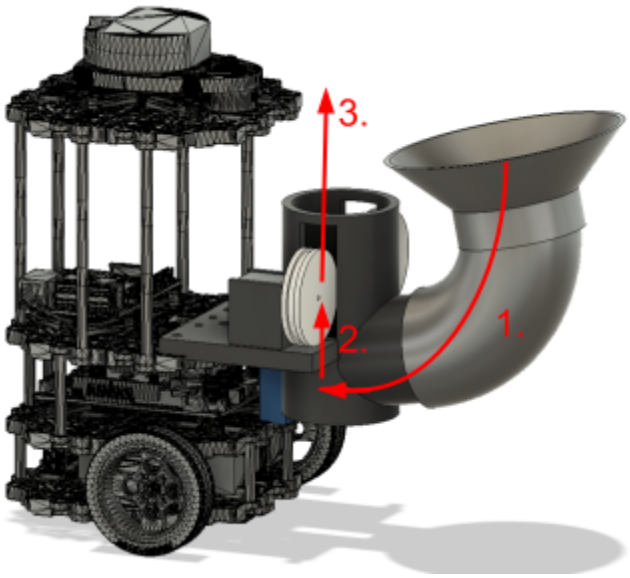


Fig 11: Direction of travel of ping pong ball through payload system

1. A funnel is still used as the feeder system to store all the ping pong balls. However, there is no longer a servo motor to control the number of balls that passes through to the launcher tube since the ball will not be straight in contact with the flywheels as it reaches the launcher tube. Additionally, the elbow connector and the opening of the launcher tube is only wide enough for 1 ball.

2. Instead, the servo motor would be used to flick the ball towards the flywheel, which is only slightly above the ball's initial position inside the launcher tube

3. As the ball reaches the flywheel, the flywheel would launch the ball straight upwards

Although the design has changed, we still took some learning points following the initial prototype testing. Additionally, some considerations following our design were made to fit this design onto the turtlebot.

1. Raspberry Pi can now be shifted back onto the 3rd layer of the turtlebot and the OpenCR is shifted back onto the 2nd layer.

2. However, the OpenCR has to be slightly shifted to the back to ensure that the servo motor has enough space to rest on the waffle plate.

3. Extra standoffs on the 3rd layer is required as the funnel would have blocked the LIDAR, considering that we are using the same elbow connector and funnel from the preliminary design
4. Counterweights were placed on the back of the turtlebot if we chose to attempt the ramp as the CG of the payload imbalances the overall CG of the robot, not enough to make it unstable if the robot is going straight or turning but is enough when travelling through the ramp

8.4 Electrical Subsystem

8.4.1 Power Budget

The power budgeting has been done for the scenario where our robot attempts one full mission without the bonus mission (scaling the ramp).

Component	Voltage (V)	Current (A)	Qty	Power (W)	Time (min or sec)	Energy (J)	Remark
Servo Motor (SG90)	3.37	0.15	1	0.5	20 sec	10	
DC Motor (Flywheel)	11.1	0.5	2	11.1	30 sec	333	
AMG8833	3.3	0.0045	1	0.015	10 min	9	
Turtlebot During initial bootup	11.1	1.35	1	15.0	30 sec	450	
Turtlebot During standby	11.1	1.02	1	11.4	4 min	2736	
Turtlebot During operation	11.1	1.54	1	17.1	6 min	6156	The speed was set as 0.12.

Total:	10663	Safety factor of 0.1 of the power was added. (There is power loss caused by the motor driver as well.)

In the power budget above, one entire run is assumed to take 10 min in the power budgeting above. Energy of the 11.1V battery is 14Wh. (Energy = (voltage * capacity) * 0.7 = 11.1V * 1.8Ah * 0.7 = 14 Wh; 0.7 is an inefficiency factor since our robot cannot move when the battery is lower than 30%.) From the power budget, energy required for one trial run is 2.86 Wh (energy = 10663 J = 2.96 Wh). Thus, approximately 4 trial runs can be done with the fully charged battery.

8.4.2 Final Electrical Subsystem

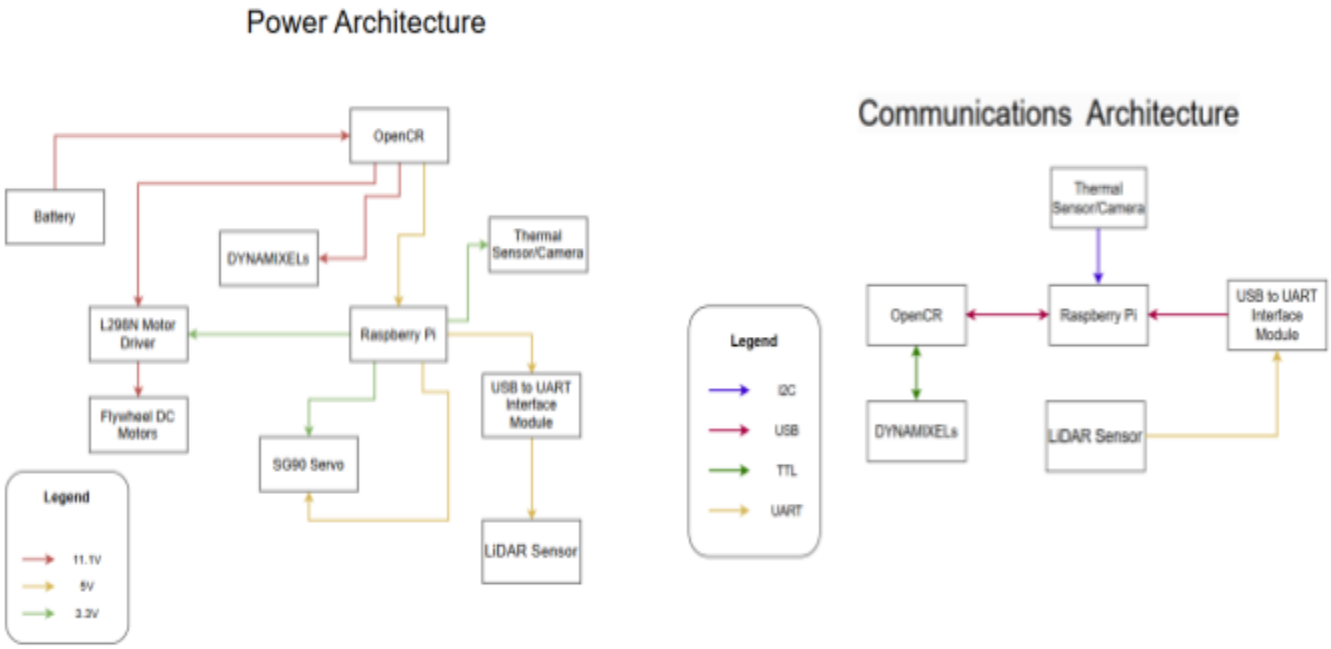


Fig 12: Power and Communications Architecture

For our final electrical architecture design, we decided to remove the isolators from the power architecture, as they were deemed unnecessary due to the absence of noise or grounding issues in our system during testing. Additionally, we opted to interface the thermal sensor directly with the Raspberry Pi, as this proved to be significantly easier both in terms of software integration and mechanical layout.

From a mechanical standpoint, the OpenCR board is located on the second-lowest layer of the chassis in the final design, making it physically challenging to run a wire from the thermal sensor to the OpenCR, even with the use of cable holes in the chassis plates. From a software perspective, interfacing the sensor with the OpenCR would require creating a dedicated ROS node on the OpenCR to publish thermal data, or alternatively transmitting the data to the Raspberry Pi for further processing, both of which would introduce unnecessary complexity. This change has also been reflected in the updated communication architecture.

In addition to these modifications, we also revised our choice of motor driver. As mentioned in the preliminary design, we switched from the L293D to the L298N. While availability was one factor, the L298N also offered practical advantages such as easier mounting due to its PCB screw holes, which made integration into the chassis more secure and convenient.

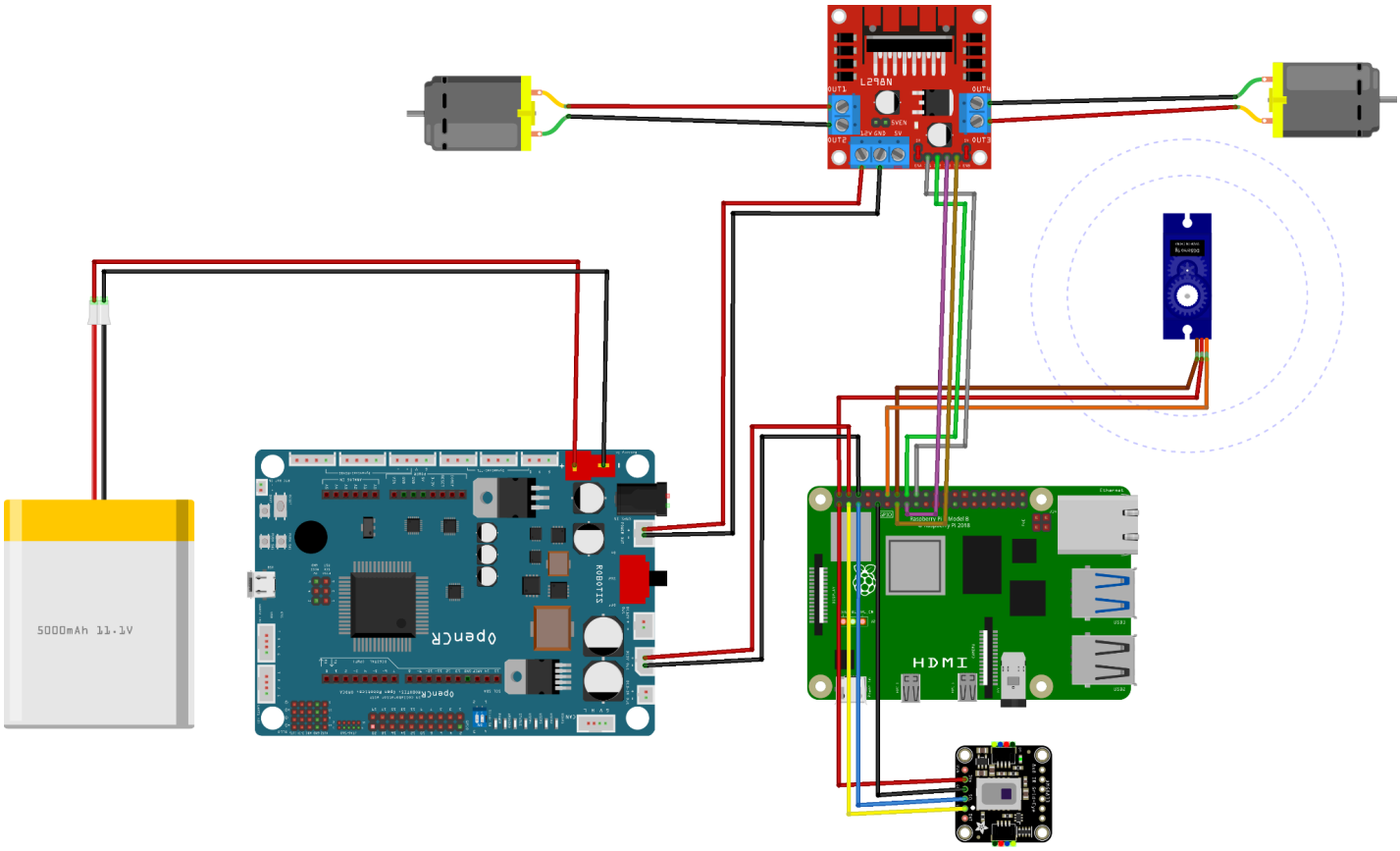


Fig 13: Physical Wiring Layout

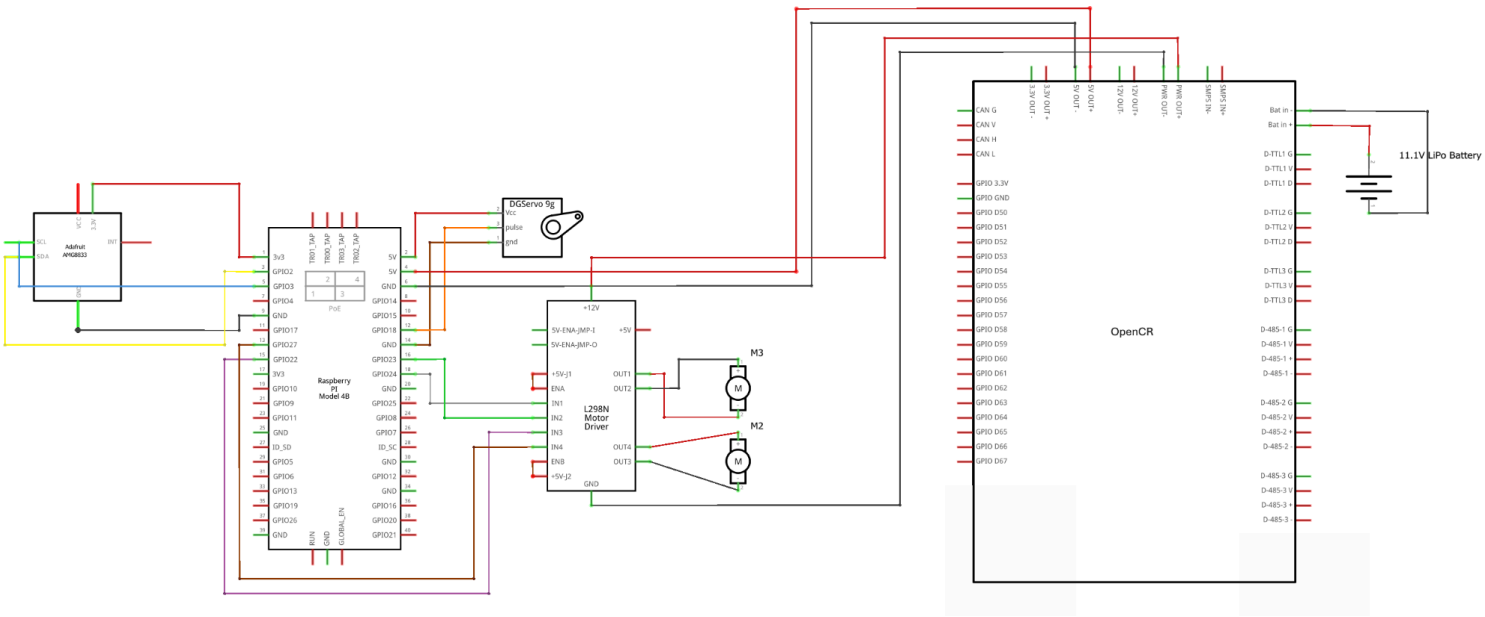


Fig 14: Schematic Wiring Layout

The diagrams above show the final wiring of the system circuit. Components such as the Dynamixel and LiDAR are not included, as they come with the TurtleBot package.

8.5 Software Subsystem

8.5.1 Final ConOps

The Concept of Operations for the ZelephantBot is illustrated in the flowchart below. Compared to the preliminary ConOps, the updated version reflects key improvements aimed at enhancing system robustness and reliability.

One major revision is the removal of the previously required heat activation threshold that determined when the robot should stop seeking a heat source. Instead, this mechanism has been entirely replaced with Navigation 2 goals, which guide the robot to move closer to the detected heat source in a more controlled and precise manner. This not only simplifies the logic but also allows for better integration with the robot's autonomous navigation stack, enabling it to reach optimal positions near the heat source without relying on arbitrary temperature thresholds which can change in different environments. We have also increased the number of temperature readings taken during each 360-degree rotation. By collecting more data points per spin, the system can project more reliable and accurate navigation goals toward the detected heat source.

Additionally, we have integrated a backup recovery navigation algorithm named **R2AutoNav**, which is a heavily customized version of the original script `r2autonav.py` provided by our instructors in the Engineering Design and Innovation Centre (EDIC). This algorithm acts as a fallback mechanism to ensure continued navigation capability in cases where the primary frontier-based exploration system fails.

Moreover, this algorithm effectively manages ramp scaling. The Navigation 2 inflation layer surrounding the walls adjacent to the ramp ensures that the ZelephantBot can safely traverse the ramp without falling, provided a frontier is detected at the end of the ramp. This behavior is a result of the inflation layer acting as a potential field. Since the ZelephantBot's LiDAR is mounted higher than the peak of the ramp, the ramp itself is not detected and is instead interpreted as free space in the occupancy grid. Consequently, the area near the end of the ramp is often identified as a frontier when the ZelephantBot is positioned at the base of the ramp. However, it is important to note that the detection of a frontier at the ramp's end is not guaranteed, due to potential noise in the LiDAR scan data.

Further enhancements include comprehensive failure handling logic throughout the system, covering scenarios such as unreachable goals, or goal rejections. These improvements collectively make the system more robust and fault-tolerant, which is critical for mission success in real-world environments. Overall, this refined and streamlined ConOps supports easier debugging, facilitates modular upgrades, and lays the groundwork for future feature expansion without the need for significant structural overhauls. This is because this ConOps also outlines the functionality of our central program, `coordinator_node.py`, which is responsible for managing and orchestrating all the other individual modules. It does so by making decisions on when to initiate specific processes, ensuring the robot operates in a synchronized and orderly manner throughout the mission.

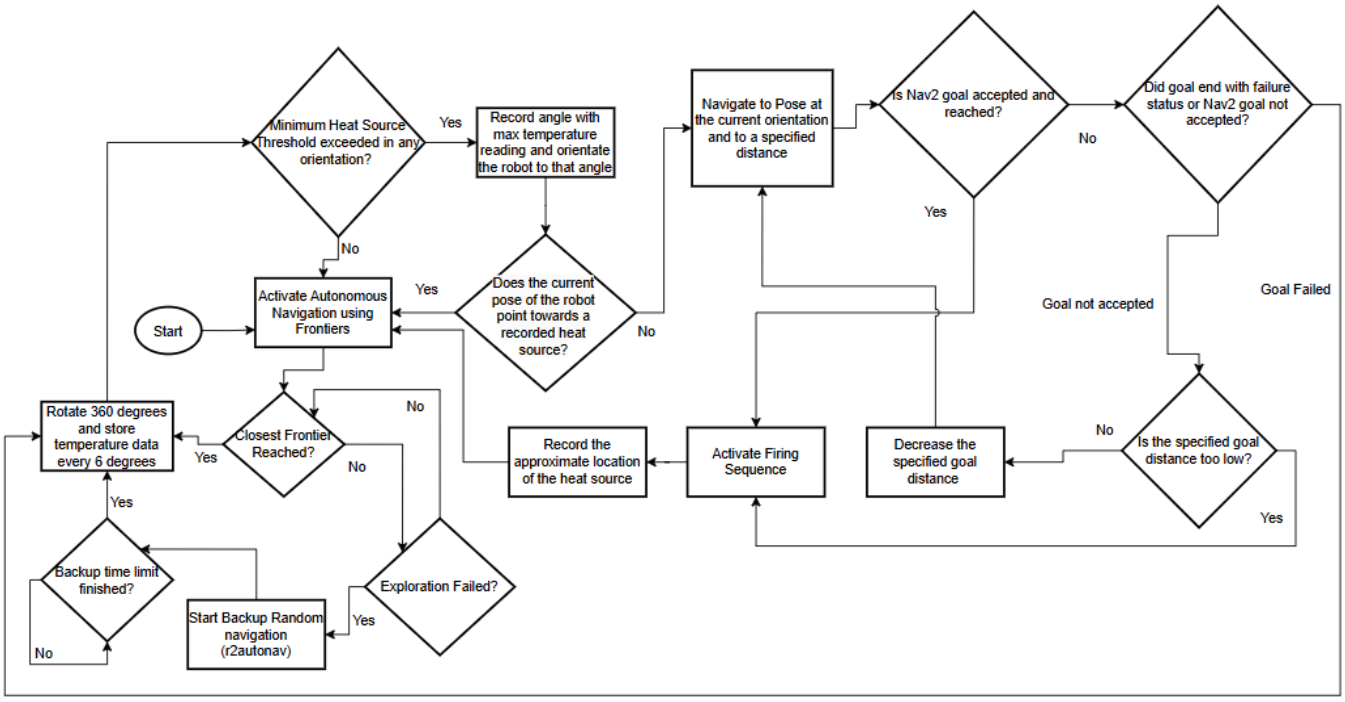


Fig 15: Final ConOps

8.5.2 Frontier Based Exploration Algorithm

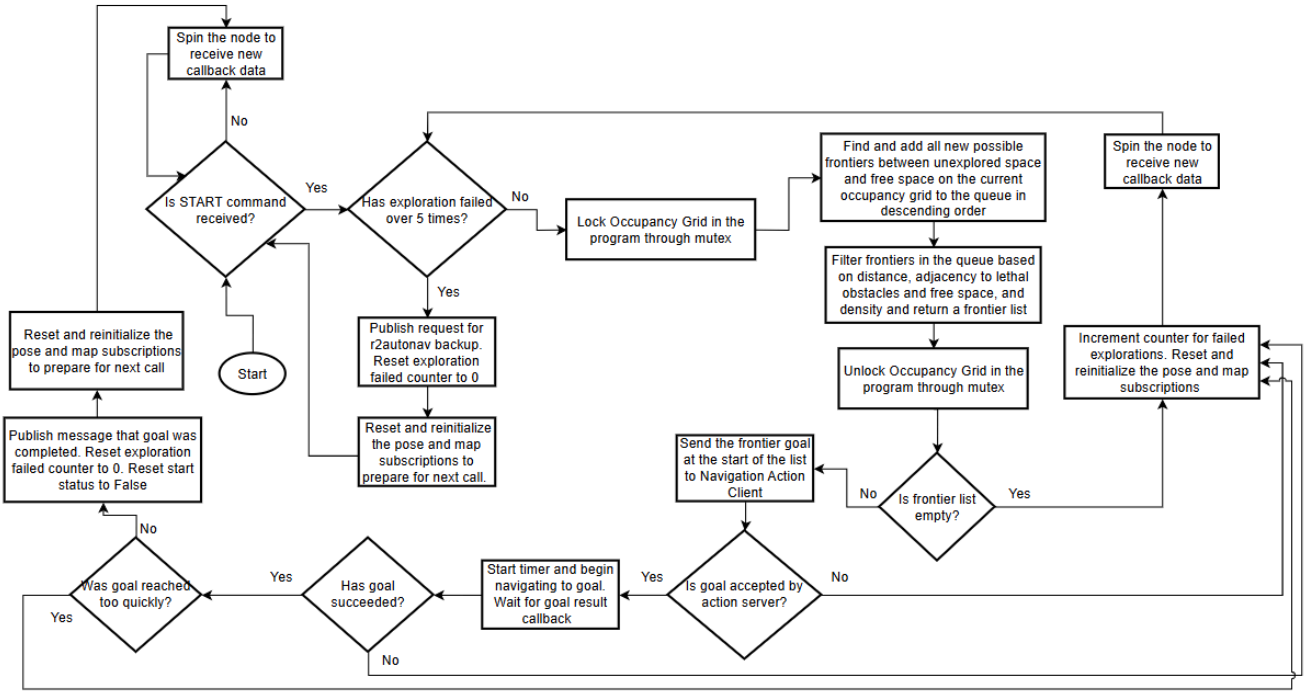


Fig 16: Flowchart of Frontier-Based Exploration Program (Automapper)

Our final frontier-based exploration algorithm includes several key improvements over the initial version to make the exploration process more reliable and efficient. One major enhancement is the implementation of frontier filtering. Rather than attempting to navigate to every detected frontier, which often includes redundant or unreachable points, the algorithm now applies filters based on criteria mentioned below. This helps in selecting only meaningful and reachable frontiers, thereby reducing unnecessary navigation attempts and improving the overall efficiency of the exploration.

In addition to filtering, we introduced a goal timer and a failure counter into the navigation process. These mechanisms monitor the robot's progress toward each selected frontier goal. If the robot finishes a goal too quickly, or if it encounters repeated failures when attempting to reach a goal or detect frontiers, the counter is incremented. Once it exceeds a certain threshold, the program publishes the r2autonav backup request message to the central coordinator program. By adding these features, the robot is now capable of detecting when it's stuck or when frontier-based exploration is no longer effective. This results in a more robust and fault-tolerant exploration system, as the robot can gracefully handle navigation failures and recover autonomously without requiring manual intervention.

.

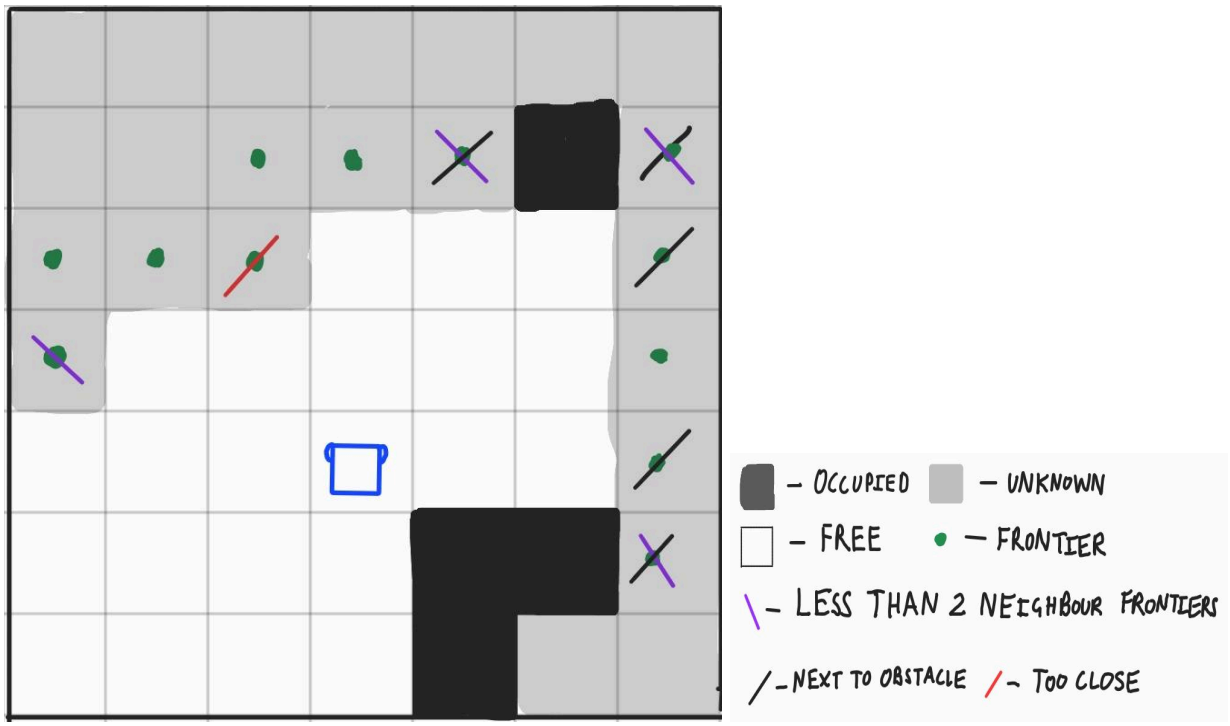


Fig 17: Example of Frontier Filtering Algorithm

The illustration above shows how frontiers are detected and filtered during autonomous exploration using the occupancy grid. In this occupancy grid, a frontier refers to the boundary between free space cells (represented in white) and unknown space cells (shown in grey). Initially, frontiers are identified by examining each cell in the map and checking whether it lies adjacent to both free space and unexplored space. If so, it's marked as a candidate frontier. However, not all detected frontiers are useful or safe for navigation. Therefore, a filtering process is applied to remove undesirable candidates:

1. **Valid frontiers** are shown as green dots without any crosses. These are well-positioned and meet all criteria for safe exploration targets.

2. A **Red cross on a frontier** indicates that the frontier is too close to the robot. These are filtered out to prevent the robot from wasting time exploring areas it's already near or has covered. Having too close frontiers can also the Navigation 2 Stack to get stuck as it may either spend too long trying to make the robot reach the exact position of the frontier as the frontier is too close to follow an approximate path, or immediately complete the frontier which may lead in increased mission completion time as thermal

detection would start immediately afterwards.

3. A **Black cross on a frontier** indicates that the frontier is located next to lethal obstacles which could be walls or other hazards. Including these frontiers could lead the robot into dangerous or barely-navigable regions, so they are excluded. It also acts as a safety barrier by not forcing the Navigation 2 Stack from trying to get too close to obstacles, which minimizes the risk of collision.

4. A **Purple cross on a frontier** indicates that the frontier is sparse, which means that the frontier does not have enough neighboring frontiers around them. A dense cluster of frontiers usually signifies a meaningful boundary or an unexplored region, while isolated ones may just be noise on the occupancy grid and would increase mission completion time for no justifiable reason.

8.5.3 Modified R2AutoNav Backup Algorithm

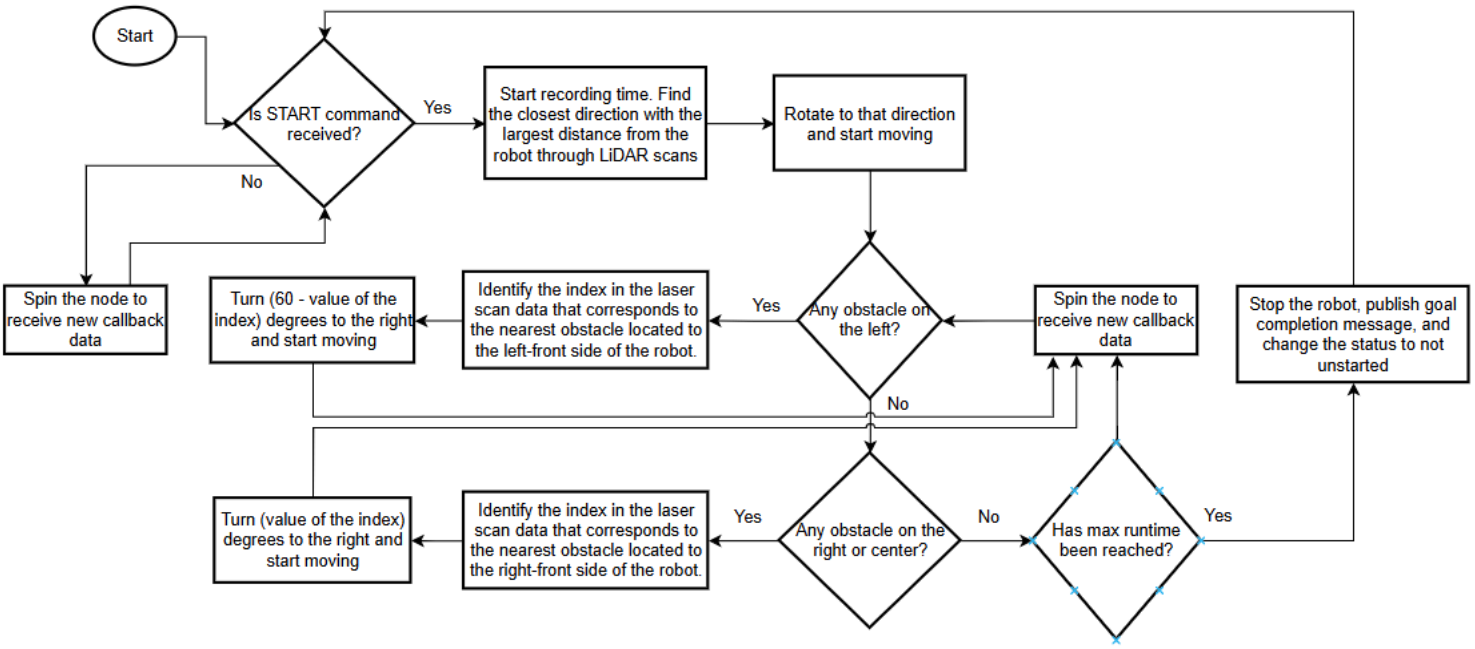


Fig 18: Flowchart of R2AutoNav algorithm

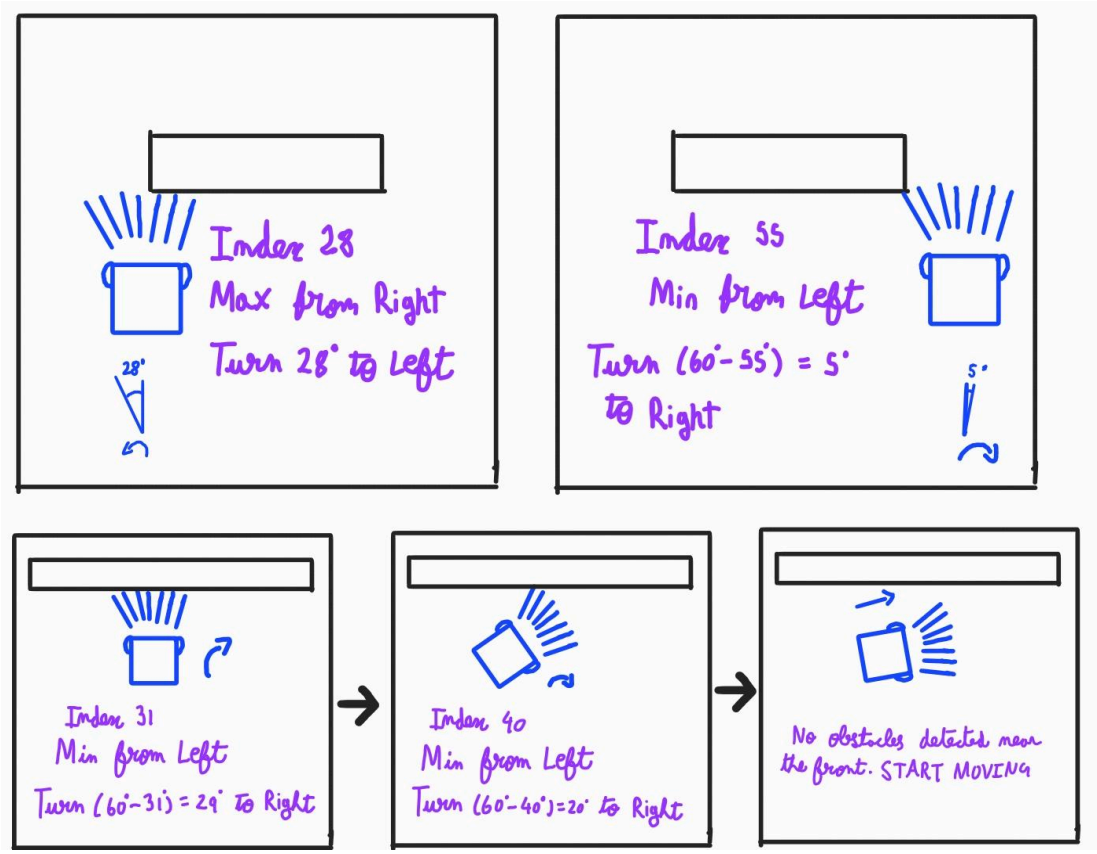


Fig 19: Example Illustrations on R2AutoNav obstacle avoidance

Initially, the ZelephantBot checks its LiDAR scan data and moves in the direction of the first maximum distance detected. However, if it detects any obstacles within a predefined stop distance, it then decides how to navigate around that obstacle. This decision is made by analyzing the LiDAR scan data, specifically the indices corresponding to the angles in the front-facing region of the robot. The LiDAR scan provides an array of distances, each associated with a specific angle relative to the robot's orientation. In the implementation, the region directly in front of the robot is defined by a symmetric angle window from -30 to +30 degrees. If any values in this region fall below the stop distance threshold, the robot recognizes that there is an obstacle directly ahead and must take evasive action.

The logic distinguishes between obstacles on the left and right sides of the front region using the indices of the distance array. Indices greater than 30 are considered to be on the left side, while those less than or equal to 30 are considered on the right or center. If an obstacle is detected on the left, the robot turns right, and if on the right or center, it turns left. To determine how much to turn, the program calculates a spin duration based on how far the closest obstacle is from the center of the robot's view. For left-side obstacles, it looks for the

closest obstacle index greater than 30 (i.e., the obstacle closest to the center of the view but still on the left). It then calculates the deviation from the center (index 30), which represents how much to turn. A similar approach is used for right-side obstacles by finding the closest index less than 30. The **illustration above** shows multiple examples on how this is achieved. Overall, this method provides a lightweight but responsive mechanism for obstacle avoidance that does not require full path planning or map awareness, making it well-suited as a backup navigation strategy to get the ZelephantBot unstuck when needed.

8.5.4 Heat Detection Algorithm

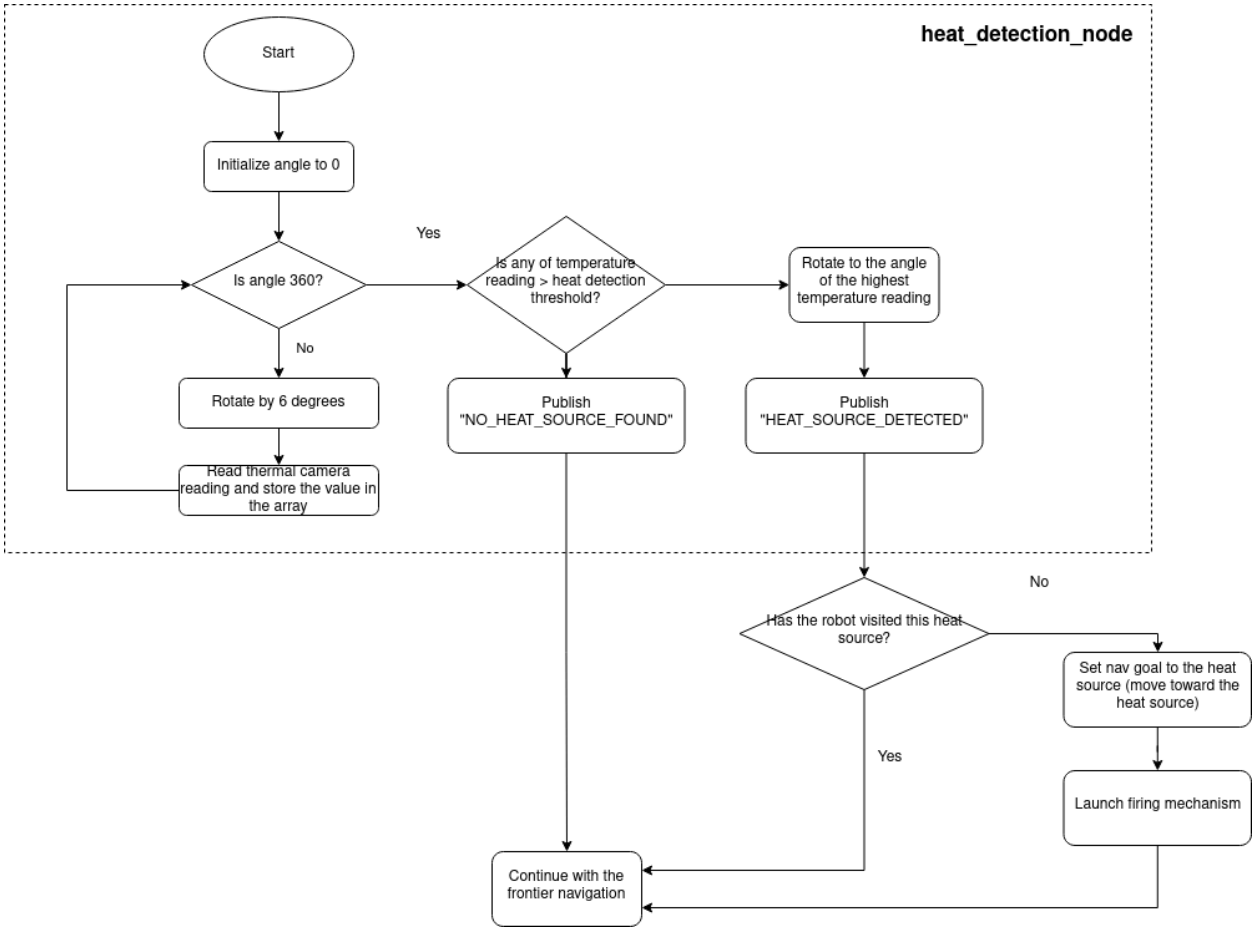


Fig 20: Flowchart of heat detection algorithm

As shown in the flowchart, when the robot reaches a frontier, it begins rotating. Every 6 degrees, the thermal sensor captures data, from which the average temperature of the central 3×3 grid is computed. After a full 360° scan, the system checks for any temperature above a defined threshold. If detected, the robot rotates toward that direction, moves forward (by

setting the nav goal), and activates the firing mechanism. Otherwise, it resumes frontier navigation. We transitioned from directly moving toward the heat source to using a navigation goal to avoid potential obstacles in its path. The sequence in the dotted box is handled by the heat_detection_node.

8.5.5 Selection of Key Parameters

8.5.5.1 Selection of Key Parameters for Navigation 2 Stack

Inflation Radius (Global and Local Costmap):
The inflation radius should be set large enough to generate a smooth potential field across the map. This ensures that the ZelephantBot avoids getting too close to walls and obstacles, promoting safer navigation. A wider inflation radius also encourages the planner to generate lower-cost paths by limiting the available free space, effectively guiding the robot through more optimal, buffered routes. Our chosen value after testing extensively was **0.24**.

Cost Scaling Factor (Global and Local Costmap):
The cost scaling factor should be tuned to create a more gradual potential field. A less steep gradient allows the ZelephantBot to navigate more fluidly through the costmap, rather than aggressively avoiding slightly inflated areas, which could otherwise result in jerky or overly conservative path planning. Our chosen value was **2.58**.

Robot Radius (Global and Local Costmap) :
Although the ZelephantBot is not a perfectly circular robot, we approximate it as circular to be compatible with the NavFn planner, which does not account for full kinematic constraints. A robot radius of **0.23m** was selected, which was 0.01m smaller than the actual radius. This reduction was intentional, as the actual size of the ZelephantBot already limits available navigable paths. A larger configured radius would further constrain pathfinding, potentially making it impossible for the planner to find feasible routes through narrow spaces.

8.5.5.2 Selection of Key Parameters for Frontier Exploration

Minimum Frontier Density:
A value of **0.1** was selected to ensure that isolated frontier cells are not considered for exploration. This is important because the SLAM-generated map often marks individual pixels (each 0.05m in size) within already explored regions as unexplored. By enforcing a minimum density, only frontiers with at least one neighboring frontier cell are considered valid. This helps avoid inefficient mapping behavior and unnecessary movement.

Minimum Distance to Frontier:

A minimum frontier distance of **0.65m** was chosen to prevent the robot from targeting frontiers that are too close to its current position. This value is significantly higher than our navigation goal tolerance of **0.15m**, ensuring that each exploration step leads the robot to a genuinely new area, rather than triggering redundant movement within tolerated bounds.

Minimum Free Threshold:

This parameter ensures that the robot only attempts to explore unknown space that has sufficient nearby free space for safe navigation. A threshold value of **2** was used, meaning the frontier must be surrounded by at least two free neighboring cells to be considered viable. This helps prevent the robot from entering narrow, cluttered, or risky areas.

Maximum Obstacle Threshold:

To avoid navigating toward unsafe areas, this parameter limits the number of obstacle cells allowed near a frontier. A value of **1** was selected, meaning any frontier cell directly adjacent to a lethal obstacle is excluded from exploration. This increases safety and reduces the risk of collision during mapping.

8.5.5.3 Selection of Key Parameters for Coordinator Program

Heat Goal Distance Threshold:

Through testing, it was observed that the most significant change in detected heat occurs at a distance of approximately **0.4 m** from the heat source. Therefore, this value was selected as the heat goal distance threshold. It serves as an optimal distance for publishing a Navigation 2 goal toward the projected location of the heat source. This choice also aligns well with the Navigation 2 Stack's default goal tolerance of 0.15 m, ensuring that the ZelephantBot can reliably reach and recognize the target area without requiring extremely precise localization.

Radius of Search:

This parameter defines the radius around an already flared heat source within which any newly detected heat source is considered to be the same, preventing duplicate detections. This ensures that the same heat source is not repeatedly detected and logged. Based on testing and environmental constraints, a value of **0.4 m** was selected as an effective threshold.

8.5.5.4 Selection of Key Parameters for Backup R2AutoNav Algorithm

Stop Distance:

Given the ZelephantBot's radius of 0.24 m, a buffer of at least 0.1 m is required to ensure safe stopping before reaching any walls. Therefore, a stop distance of **0.4 m** was selected to provide sufficient clearance and avoid collisions.

Maximum Runtime:
A maximum runtime of **25 seconds** was chosen before the algorithm halts. While the robot operates at relatively low speeds, this longer runtime increases the likelihood of the ZelephantBot exploring unexplored areas of the map, which is particularly useful in complex environments.

8.5.5.5 Selection of Key Parameters for Heat Detection

Heat Detection Threshold:
The threshold should be slightly above average room temperature but below the temperature of a heat source at a distance of approximately 0.4 m. This approach reduces the need for frequent calibration while enabling longer-range heat detection. A threshold value of **25.5°C** was chosen as a suitable compromise, though performance may vary if the ambient temperature is significantly higher.

Temperature Capture Angle:
For accurate angle estimation of heat sources, a finer angular resolution (e.g., 1–2°) would be ideal. However, this is limited by the frame rate of the AMG8833 thermal sensor. A capture interval of **6 degrees** was selected to maintain a good balance between detection accuracy and acceptable robot spin time during scanning

8.5.5.6 Selection of Key Parameters for SLAM Toolbox

Max Laser Range:
Reducing the maximum laser range limits the area scanned at once, which encourages the ZelephantBot to detect more frontiers and make more frequent heat detection stops. This ensures a more thorough exploration of the maze. A value of **1.1 m** was found to be an effective balance through testing, enhancing both frontier detection and heat source localization.

8.6 ROS2 Topic-Node Block Diagram

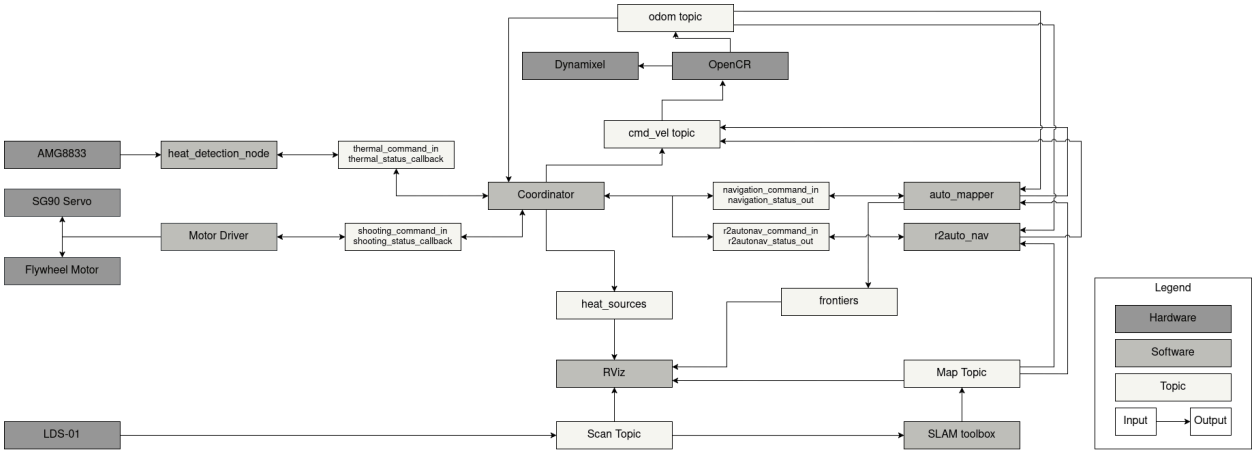


Fig 21: ROS2 Topic-Node Block Diagram

The Topic-Node Block Diagram shows how hardware, software, and topics interact in the system. Sensor inputs (e.g., lidar, thermal camera) are processed by dedicated software and sent via ROS topics to the decision-making logic. Commands are then published to trigger actions like activating flywheel and servo motors or adjusting the robot's speed and direction. Do note that the MarkerArray Rviz Visualization Topics for both frontiers and heat sources are not shown in the above diagram as they are primarily used for testing and development, rather than the operation of the system.

8.7 System Finances

A Turtlebot set was provided to create the system. A budget of up to 80 dollars was provided for supplementary parts. The breakdown of parts and their cost are indicated below in the bill of materials

No.	Part	QTY	Unit cost (SGD)	Total Cost (SGD)
Mechanical				
1	Turtlebot 3	1	Provided	Provided
2	M2x8mm cross-head screws	6	Provided	Provided
3	M3x20mm SS screws	4	Provided	Provided
4	Standoffs	12	Provided	Provided

5	Launcher	1	27.25	27.25
6	Flywheels	2	5.00	10.00
7	O rings	10	0.05	0.50
8	PVC pipe connector 55mm	1	2.40	2.40
9	Funnel	1	1.40	1.40
Electrical				
1	SG90 servo motor	1	provided	provided
2	Motor Driver Dual H Bridge L298N	1	Provided	Provided
3	Jumper wires	12	provided	provided
4	24v DC motors	2	6.00	12.00
5	AMG8833	1	39.90	39.90
Software				
1	AWS server	1	provided	provided

Total Cost: 93.45 SGD

Chapter 9 - Assembly Instructions

9.1 Layer 1 Assembly

The 1st layer of the turtlebot has remained exactly the same as the original turtlebot, hence, refer to the assembly manual for TurtleBot3 Burger for assembly of the waffle-plate, dynamixel, and battery

9.2 Layer 2 Assembly

The 2nd layer assembly is similar to the assembly manual for the 2nd layer of the TurtleBot3 Burger. The only difference is that the OpenCR is slightly shifted to the back to allow space for the servo motor that is connected to the launch tube.

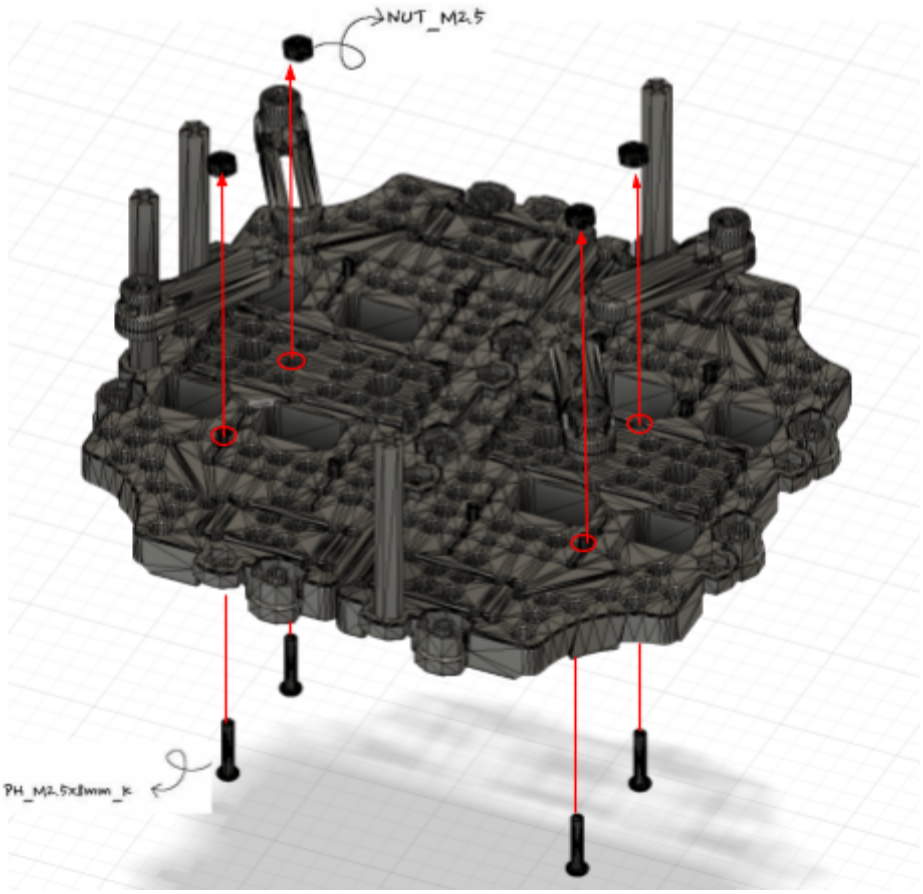


Fig 22: Mounting Supports for RPi

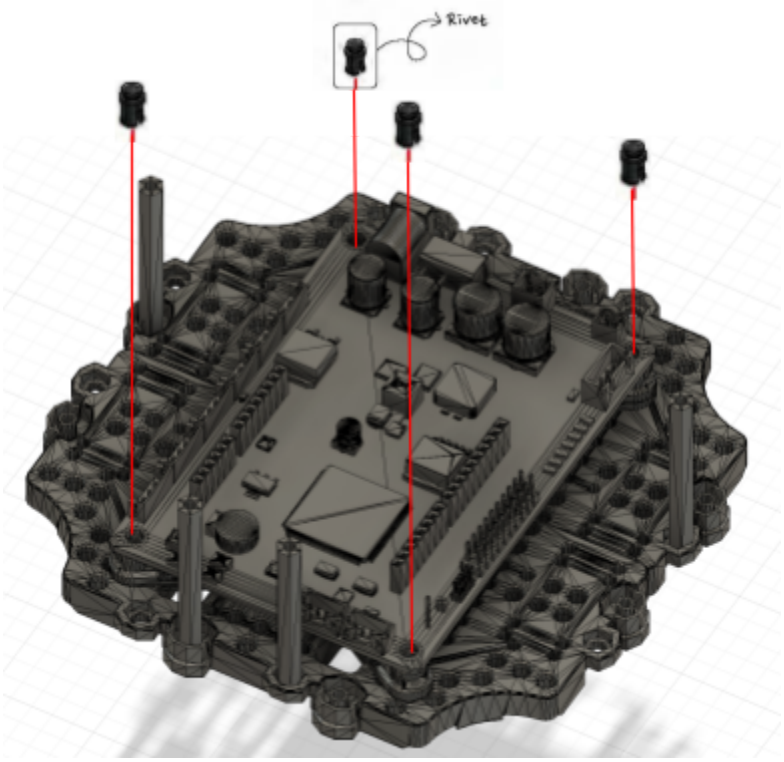


Fig 23: Mounting RPi onto Waffle Plate using Supports

Layer 2 Component List

Component Name	Quantity
OpenCR	01
PH_2.5x8mm_K	04
NUT_M2.5	04
Rivet	04
Plate_Support_M3x45mm	04

9.3 Layer 3 Assembly

Mount of the Raspberry Pi and the USB2LDS on the third layer is exactly the same as the one shown on the assembly manual for TurtleBot3 Burger.

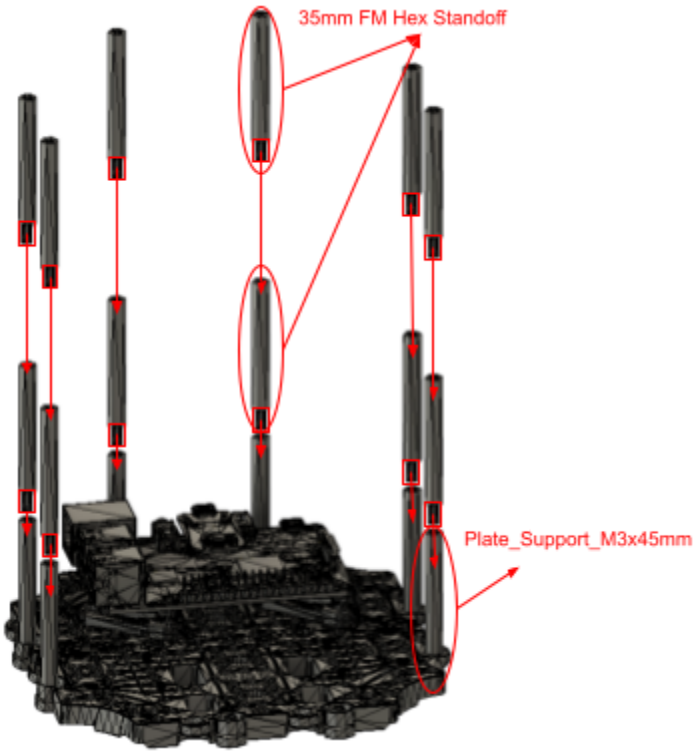


Fig 24: Pictorial Representation of additional Standoffs used

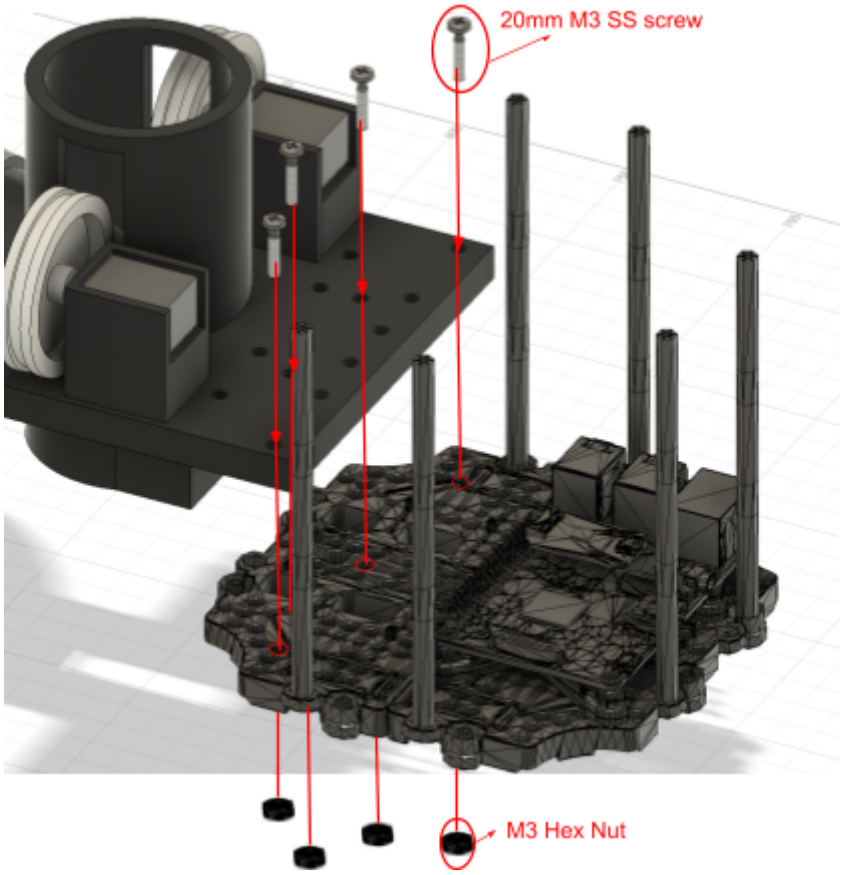


Fig 25: Fastening Launcher onto Waffle Plate

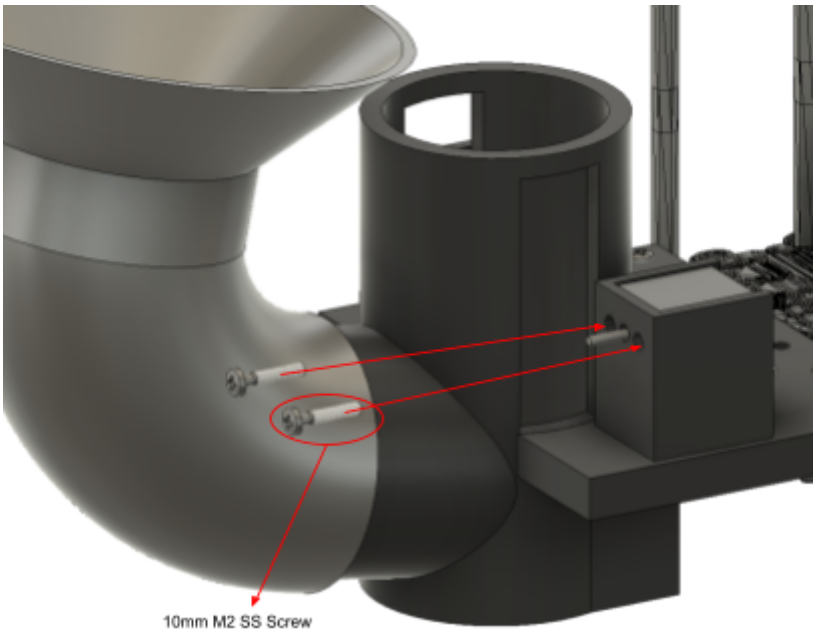


Fig 26: Securing DC Motor into its housing

Repeat this for the DC Motor on the other side.

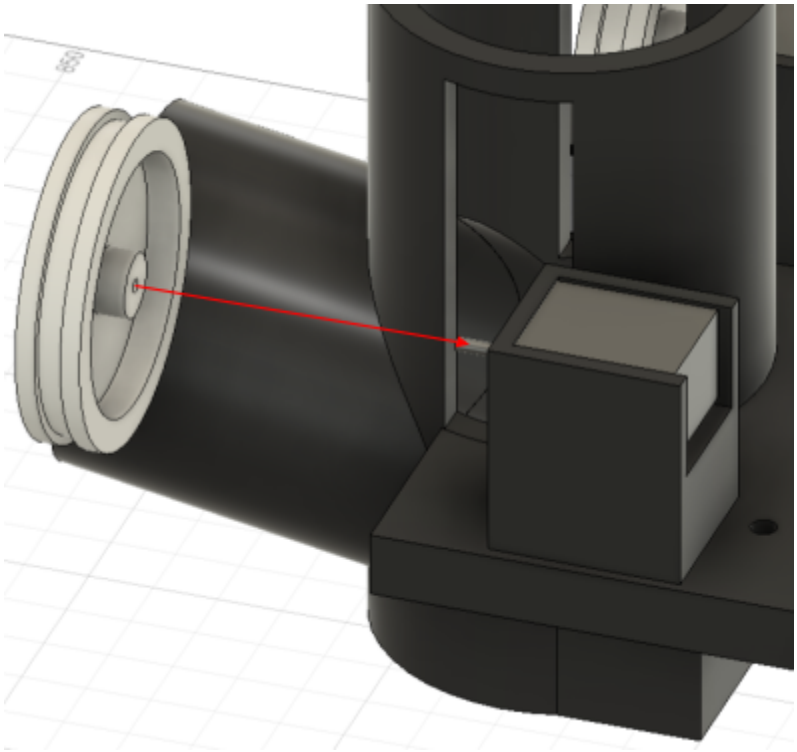


Fig 27: Tight Fit Mount between Flywheel Shaft and Flywheel

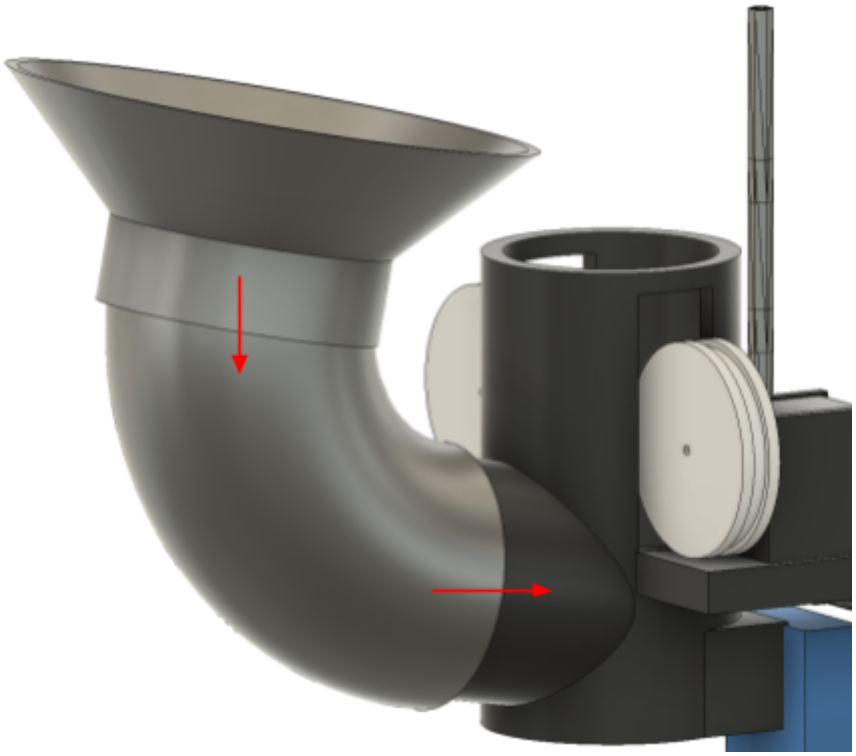


Fig 28: Tight Fit Connection between Elbow PVC Connector and Ball Feeder

Layer 3 Component List

Component Name	Quantity
Raspberry Pi	01
USB2LDS	01
NUT_M3	04
20mm M3 SS Screw	04
Plate_Support_M3x45mm	06
35mm FM Hex Standoff	06
10mm M2 SS Screw	04
24V DC Motor	02
Flywheel	02
Launch Tube	01
PVC Elbow Connector 55mm	01
Ball Feeder (Funnel)	01
SG90 Servo Motor	01

9.4 Layer 4 Assembly

The 4th layer of the turtlebot has remained exactly the same as the original turtlebot, hence, refer to the assembly manual for TurtleBot3 Burger for assembly of the LIDAR onto the waffleplate.

9.5 Final Assembly

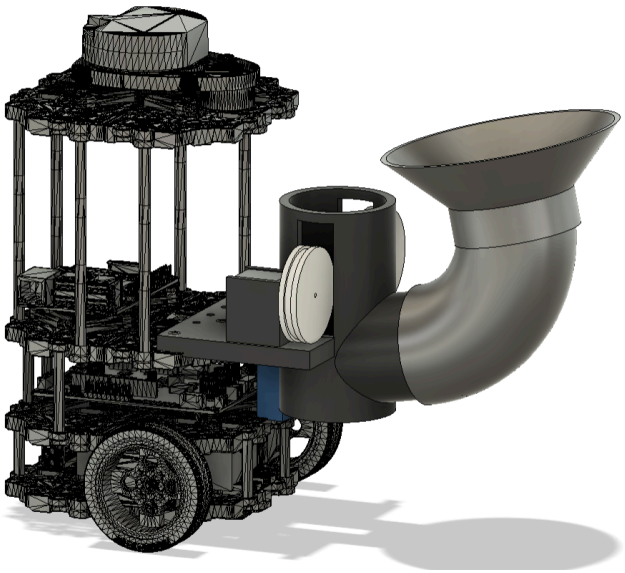


Fig 29: Final Assembled ZelephantBot

Chapter 10 - Final Evaluation and Testing

10.1 Pre-Mission Evaluation Testing and Documentation

10.1.1 Factory Acceptance Test (FAT)

The FAT is meant to ensure that each individual system of the ZelephantBot is operational and functioning as intended before the mission. It also serves as a way to isolate faults or for modular testing if the need arises. It is recommended that the FAT be conducted before every mission to ensure a successful mission. Refer to our End User Documentation in our GitHub Repository for the FA Tests.

10.1.2 Acceptable Deferred Defects Log

The Acceptable Deferred Defects Log aims to log key defects with ZelephantBot and its impact on its ability to complete the mission. The log is updated and presented prior to every mission run. Refer to our End User Documentation in our GitHub Repository for the Acceptable Deferred Defects Log.

10.1.3 Maintenance and Part Replacement Log

The Maintenance and Part Replacement Log aims to keep track of the list of replaced components damaged during testing. This enables us to diagnose and pinpoint where faults lie and can occur. Refer to our End User Documentation in our GitHub Repository for the Maintenance and Part Replacement Log.

10.2 Evaluation Setting

The evaluation stage of the ZelephantBot was set in a 5x5m closed maze with three heat sources. The maze consisted of multiple walls and obstacles (in the form of buckets and tin cans) that obscured the heat sources from direct view as shown below. However, on our best-scoring run, we had not attempted the bonus ramp mission, which is why it is blocked by a wall in the illustration. The approximate path and decisions our ZelephantBot took is shown in the illustration as well.

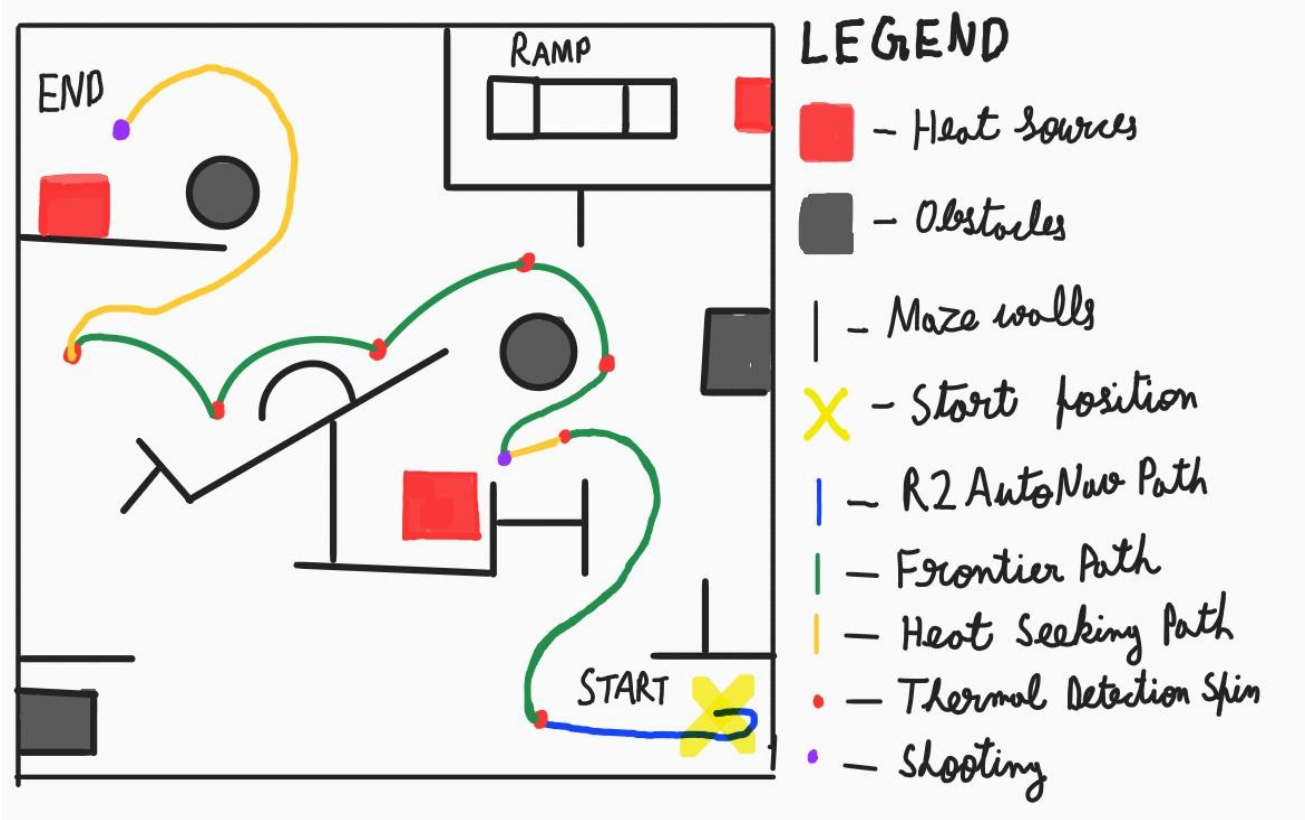


Fig 30: Pictorial Representation of the successful first mission run by ZelephantBot

10.3 Evaluation Results

Success:

Zelephant bot was the only robot that was able to complete the entire mission. ZelephantBot was able to reach the first survivor autonomously at 1.30 (as shown in the above image). It then proceeded to fire **two** flares. Subsequently, the ZelephantBot navigated to the second survivor at 6:30 (as shown in the above image). Similarly, It then proceeded to fire **two** flares. Finally, we concluded the successful mission run at 7:19. The run was scored 89.5 out of 100, narrowly losing out on partial credit for flare firing and poor structural integrity. What surprised us was that our ZelephantBot could detect the heat source from behind the wall, and send a well placed Navigation 2 Goal such that it could easily avoid any walls and obstacles and make its way to the heat source in an unexplored region.

Failures:

Unfortunately, the ZelephantBot was unable to fire the second ball on cue. There was some obstruction in the feeder mechanism that prevented the second ball from reaching the flywheel. Another key issue that we had noticed was that the ZelephantBot's program handle was locked in Frontier-Based Exploration for far too long, as Navigation 2 could not determine whether the goal was reached, even when it was. This was the key reason why our mission time completion increased to 6:30 when trying to explore the map to find the second heat source.

Fig 31: Screenshot of Rviz showing ZelephantBot moving towards a frontier (in green) and a heat source already flared (in red) in final mission run

Chapter 11 - Troubleshooting

11.1 Troubleshooting - Mechanical

1. Servo motor not feeding balls into flywheel
    a. Ensure wires are connected
    b. Comment out flywheel motor command and run motor_driver.py
    c. Insert one ball and observe it come into contact with stationary flywheels
    d. Rectify servo placement
    e. Replace servo if unsuccessful
2. Balls not being shot
    a. Ensure flywheel is spinning
    b. Check motor's wire connection
    c. Test flywheel rotation direction
    d. Flip wire connection if required
3. Turtlebot not moving
    a. Ensure no loose wire connection
    b. Check battery level
        i. Below 30% will cause ros bring up to not run (Stack smashing detected error)
    c. Replace DYNAMIXEL wires

11.2 Troubleshooting - Firmware

1. Heat sensor malfunctioning
    a. Ensure wires are connected
    b. Run heat_detection_node.py on the RPi to check for temperature. Ensure that the initial temperature reading is not 0.0
    c. If it is 0.0, try connecting the sensor to a spare RPi. If is still 0.0 there, replace the AMG8833 as it is faulty
2. LiDAR map not publishing
    a. Restart our frontier-based exploration program "automapper". Ensure that is_sim is False if you are not testing in a simulation environment. It may take multiple tries for the Map to be received by Rviz. If it still fails, do try running the command `ros2 daemon start`. If it still fails to work, we suggest cleaning up your directory and reinstalling the necessary software from our setup manual again.

11.3 Troubleshooting - Software

1. Laptop unable to connect to the Turtlebot
    a. Ensure that the laptop and the RPi are connected to the same network.
    b. Ensure that the ROS_DOMAIN_ID in the ~/.bashrc file of the laptop and the RPi is the same.
    c. Change the ROS_DOMAIN_ID on both the laptop and the RPi to prevent conflict of communication with those on the same network.
    d. Instead of using `sshrp`, try to log in using `ssh ubuntu@{IP-Address of RPi on your network}`.

Chapter 12 - Future Improvements

Our observations from the final mission run of the ZelephantBot suggest that there is still considerable potential to be unlocked in both its firing and software system. We have identified several improvements that could enhance its efficiency and adaptability in even more complex environments.

.

12.1 Centre of Gravity

This current version of Zelephant bot is back-heavy where the center of gravity lies more towards the back half of the robot. This results in the robot being unstable as it navigates the maze. We realise that with the current design, it would not be able to travel at speeds higher than 0.11 m/s. Hence a solution would be to shrink the print down to its basic components, removing much of the support and thickness of the 3D print. This will significantly reduce the uneven weight distribution.

12.2 Feeder, Servo Arm

In the interest of time and cost, we decided to make the feeder system using corrugated cardboard which gives rise to inconsistencies, resulting in the first ball not being able to reach the flywheel as illustrated in the evaluations of our mission run. Hence given more time and budget, we would instead 3D print the servo arm which would ensure a more consistent feeding of the flares.

12.3 Mounting of AMG8833

As part of the firmware, the AMG8833 was required to be mounted at the front and center of the robot. However we also had to mount the flywheel launcher there hence we had to attach the thermal camera on the funnel system which was not ideal. An improvement we could make was to hollow out or drill out part of the PVC connector to mount the thermal camera. This would ensure that the thermal camera is securely mounted and consistently pointing at a specific direction.

12.4 Wiring and Cable Management for the Electronics

Despite successful closed circuit connections of all the components to the RPi, we could have improved on the wire management. We could have labelled the different jumper wires leading to each component, aiding troubleshooting of the system, especially during the testing phase, where our ZelephantBot hit obstacles and fell down the ramp, which messed up the wires sometimes.

12.5 Navigation Planner

One of the key limitations of traditional navigation planners, such as the standard NavFn, is that they treat the robot as a point mass or a circular approximation, without considering its actual kinematic constraints or physical footprint in detail. This can lead to unrealistic paths that the robot cannot follow in real-world conditions, especially in tight or cluttered environments. Likewise, if the robot's radius is too large even when it has a narrow but long footprint like our ZelephantBot, many narrow paths would be judged to be impossible to navigate through due to the large radius we have to set in the Navigation 2 parameters.

As a future improvement, we suggest implementing a kinematically feasible planner such as Smac Lattice, which is part of the Navigation 2 stack. Smac Lattice integrates both search-based planning and vehicle-specific motion constraints. It considers the robot's exact dimensions, turning radius, and motion limitations during path generation. This leads to more paths being found that are not only theoretically optimal but also physically executable by the robot. Perhaps, it might also help avoid collisions, reduce unnecessary re-planning or recovery behaviors, and ensure smoother and more efficient navigation. Moreover, integrating a kinematically aware planner would align well with future scalability, for example, if we modify the robot's chassis in any way, then the footprint can also easily be adjusted to allow navigation in more complex and tight environments.

12.6 R2AutoNav Algorithm

Currently, the r2autonav recovery navigation algorithm relies on simple obstacle-avoidance based movements when the main frontier-based exploration fails. While this helps prevent the robot from stalling, it can be inefficient and may result in the robot wasting time navigating within already known areas of the map. As a future improvement, we suggest enhancing the r2autonav logic by projecting a goal toward the most open and unexplored direction, based on real-time laser scan data. This can be achieved by identifying which maximum laser scan index points towards an unexplored region as shown in the occupancy grid, and then moving towards that direction using the current obstacle-avoidance algorithm. This approach increases the chances of rediscovering frontiers by updating the map with new data. As a result, the robot can escape deadlocks more efficiently and continue exploring without manual intervention. This enhancement would make r2autonav a more reliable fallback strategy, especially in complex environments where frontier-based exploration struggles due to incomplete or noisy maps.

12.7 Frontier-Based Exploration Algorithm

Sometimes, during the process of navigating to a frontier, the ZelephantBot remains at exactly the same distance from its assigned goal for an extended period without making any progress, as observed in our final mission run. Although the Navigation 2 Stack has a progress checker, we cannot always rely on it, as there is a chance that the entire controller server might have failed and recovering from it would take too long. Therefore, we propose implementing a timeout or stagnation detection mechanism during goal execution without the use of the Navigation 2 Stack. Specifically, if the robot remains within a fixed distance range from the goal for longer than a predetermined threshold (e.g., 20–25 seconds), the program should automatically cancel the current goal and trigger the r2autonav backup algorithm. This would decrease mission completion time and also reduce the likelihood of complete navigation failure.

Another potential improvement would be to completely switch from using the occupancy grid to relying solely on the costmap for frontier detection. In this approach, a range of cost values can be defined to represent obstacles, rather than relying on fixed occupancy values. This would allow for greater flexibility and control when detecting and filtering frontiers. Specifically, it would enable the implementation of a customizable safety buffer around obstacles by excluding frontiers that fall within a specified cost range. As a result, this would lead to more robust and safer frontier selection, reducing the chances of the robot attempting to explore areas that are too close to walls or other hazards, and improving overall navigation efficiency.

12.8 Heat Detection

Instead of the current approach, where the robot stops and performs a full 360-degree spin to collect temperature data at every goal point before checking for nearby heat sources, we propose a more efficient alternative using K-Means clustering. This would involve continuously accumulating temperature data across the entire map during exploration and then applying K-Means clustering to identify potential heat source regions once mapping is complete. The core advantage of this method lies in its ability to generate a temperature heatmap, from which cluster centroids can be selected as candidate goal points for closer inspection. This would significantly reduce mission completion time, especially in larger and more complex mazes, where stopping repeatedly to scan can be time-consuming and inefficient.

However, this method comes with trade-offs. Firstly, it would require substantial restructuring of the existing software system, as the logic would need to shift from local detection to global data analysis. Secondly, the reliability of this method in pinpointing precise heat source locations is still uncertain. It may not be as accurate as the current approach, which is designed for high-confidence detection in smaller or more precise missions, like our current one where heat localization accuracy is crucial for mission goal completion. That said, in scenarios where the quick identification of general regions containing heat sources is more valuable than the pinpoint accuracy of their exact locations, this K-Means-based approach holds great potential.

12.9 Runtime-Tuning of Navigation 2 Parameters

Another potential improvement is enabling runtime tuning of key parameters within the Navigation 2 stack. Currently, most parameters such as acceleration limits, planner tolerances, and obstacle inflation radii are defined in static YAML files and require restarting the Navigation 2 nodes to take effect. This limits adaptability during a mission, especially in dynamic environments or when unexpected behaviors are observed.

By implementing runtime parameter tuning, either through dynamic reconfiguration tools like modifying behaviour trees, or through ROS2 service calls, the robot could automatically adjust its navigation behavior based on the current region of the maze, whether it is a wide or narrow. This would significantly enhance robustness and adaptability to different types of environments, while also facilitating easier experimentation and debugging during development and testing. However, we are unsure whether these parameters could indeed be dynamically tuned through the previous methods, so this suggestion would most likely be extremely difficult to implement.

References

[1] C. Debeunne and D. Vivet, “A review of visual-LiDAR fusion based simultaneous localization and mapping,” Sensors, vol. 20, p. 2068, Apr. 2020. doi: 10.3390/s20072068.

[2] D. Bojanic, K. Bartol, T. Pribanic, T. Petkovic, Y. D. Donoso, and J. S.Mas, “On the comparison of classic and deep keypoint detector and descriptor methods,” in 2019 11th International Symposium on Image and Signal Processing and Analysis (ISPA), IEEE, Sep. 2019, pp. 64–69. doi: 10.1109/ispa.2019.8868792.

[3] N. Dias, G. Laureano, and R. Costa, “Keyframe selection for visual localization and mapping tasks: A systematic literature review,” Robotics, vol. 12, p. 88, Jun. 2023. doi: 10.3390/robotics12030088.

[4] A. Topiwala, P. Inani, and A. Kathpal, Frontier based exploration for autonomous robot, 2018. arXiv: 1806 . 03581 [cs.RO]. [Online]. Available: https://arxiv.org/abs/1806.03581.