SYSC4907-A
Engineering Capstone Project

Autonomous Mapping and Navigation
Final Report

Group 75

Himanshu Singh 101152189
Ray Prina 101172837
Sangat Buttar 101147990
Sagar Syal 101150341
Sundar Vengadeswaran 101143449

Table of Contents

1.0 Introduction

The purpose of this document is to present all the finalized achievements that were made on the Autonomous Mapping and Navigation project. Furthermore, this document will update all changes made to any requirements and plans from the Progress Report.

1.1 Motivation

In today's ever growing high-tech industry, the emergence of autonomous rovers has proven to be extremely beneficial in many situations. These can range from dangerous military operations to scientific exploration in locations that humans cannot easily travel to (e.g., space exploration, deep sea exploration, nuclear radiation zones, etc.). As such, this project aimed to develop techniques to efficiently analyze performance differences in autonomous mapping and navigation techniques, from both a hardware and software perspective.

1.2 Project Scope

The Autonomous Mapping and Navigation system utilized the Hiwonder JetAuto robot as its base, which included a RPLIDAR A1 for obstacle detection. The system employed Robot Operating System (ROS1) to manage and allow communication between Simultaneous Localization and Mapping (SLAM) algorithm nodes. Our initial evaluation of the Autonomous Mapping and Navigation system was centred around the replication of a miniature self-driving car. Following extensive testing and thoughtful consideration, we have refined our project scope to implement an autonomous exploration and mapping rover, rather than a self-driving car. This adjustment is due to shortcomings in sensor capabilities, mapping/exploration methodologies, and overall practicality.

Through ROS and SLAM, the project purpose entailed of autonomously exploring and mapping a track environment akin to a parking lot, after which the robot localized itself and navigated to a specific position within the generated map. Furthermore, the project attempted to discern the difference in mapping and trajectory performance between three separate hardware configurations, to be described in Section 3. However, due to an unexpected hardware safety issue, described in Section 2, the project scope needed to be shifted to a simulation focus. This shift included analyzing mapping performance based on differences in trajectory parameters such as start point, end point, turning radius, etc. Additionally, the simulation focus included a workspace migration from ROS1 to ROS2.

2.0 Project Overview

2.1 Health and Safety

Throughout the year there were multiple safety issues concerning the robot base used in the project. These entailed of power supply issues and overheating issues, ultimately culminating in an electrical fire and loss of the robotic system, as seen in Figure 1 below.

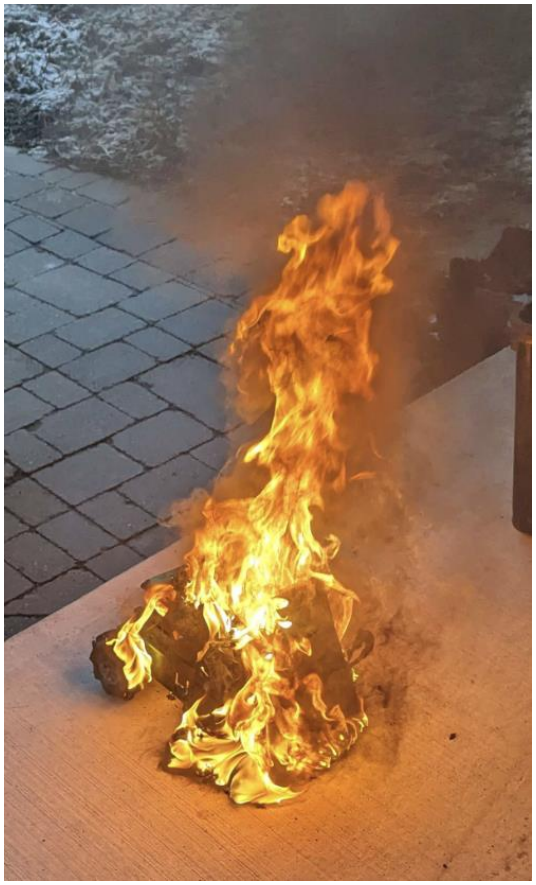

Figure 1: HiWonder JetAuto Self-Combustion

While working on the project, there were power issues that occurred with the robot, which resulted in hardware components being shut down. Through extensive usage, the LiDAR module unexpectedly shut down several times, requiring a power cycle to enable further usage. Furthermore, multiple nodes that were run in conjunction also unexpectedly shut off, causing problems during testing. For instance, the unexpected shut down of the Servo Manager node resulted in a subsequent chain reaction, causing the Odometry Publisher, State Publisher, and Base Length nodes to also cease operation. This was believed to be a complete power supply issue, as the battery installed on the robot was unable to sustain the robot for longer periods of time. For further details, refer to section 4.4.

After extensive work periods, the robot's chassis would emanate heat, alongside the release of a potent chemical odor, presumably associated with Lithium. It was believed that the primary cause of the robot overheating stemmed from a short circuit, wherein an unintended surge of current occurred. This resulted in damage to the robot's electrical system, causing the wire to combust.

The aforementioned issues then converged with one another, resulting in the combustion of the entire robot. Having been low on battery, the robot was placed on charge for an hour before unexpectedly undergoing self-combustion, thereby initiating an electrical fire, and emitting a significant amount of lithium gas. Various limiters had been implemented on the charger to prevent such incidents; the battery has an overload protection of 110%-160% [1]. These circumstances suggest a potential fault within the robot's battery system, such as a short circuit.

Lithium Polymer (LiPo) Batteries were subject to specific storage conditions, deviation from which could result in significant damage. Storing LiPo batteries in excessively high temperatures could result in capacity degradation and trigger thermal runaway, potentially leading to fire or an explosion. Similarly, subjecting batteries to extremely cold temperatures could impair their performance and inflict damage on the battery [3].

Additional precautions could have been taken to mitigate the occurrence of such events. For enhanced safety measures, LiPo batteries are to be stored in fireproof LiPo charging bags to decrease the risk of fires or explosions. It was imperative to keep the robot away from direct sunlight or any heat-emitting sources. Furthermore, the LiPo batteries were recommended to be stored at 50% of their max capacity to prevent over-discharge, alongside regular inspections to ensure there is no damage, leakage or swelling on the battery [3].

2.2 Engineering Professionalism

Sangat Buttar:

Sangat's professional responsibilities were comprised of completing delegated tasks by the leadership roles. His responsibilities comprised of completing the tasks they were delegated in section 2.5.1, along with updating the leading members of the team with updates regarding the progress of tasks assigned to him. Sangat also occasionally assisted other team members with the tasks they were assigned, depending on the difficulty of tasks that required more members to complete by the deadline set in the Gantt chart.

Ray Prina:

Ray's professional responsibilities were composed of completing the tasks that were delegated to him by the leaders. His role and tasks were delegated in section 2.5.1. Additionally, he was also assigned to update the leaders about the progress of the other

tasks that were assigned to him. Ray was also in charge of the creation and constant update of the Gantt chart, receiving schedule changes and schedule additions from the leaders to input onto the chart.

Himanshu Singh:

Himanshu's professional responsibilities were comprised of finishing tasks assigned by team leaders, detailed in section 2.5.1. He ensures regular updates to the leadership about his progress. Additionally, Himanshu supports other team members with their assigned tasks, especially when they involve complex challenges that necessitate more hands to meet the deadlines set in the Gantt chart.

Sagar Syal:

Sagar’s professional responsibilities were comprised of taking one of the two leadership roles. His responsibilities comprised delegating tasks for his team members based on their strengths and ensuring that they were completed effectively and efficiently. Sagar was also responsible for adjusting the schedule when necessary.

Sundar Vengadeswaran:

Sundar’s professional responsibilities were predominantly composed of taking one of two leadership positions within the group. This included delegating roles to the other team members and ensuring that tasks were consistently being completed. As a result, due to the individual tasks he completed, paired with the coordination and hard work of his team members, the final goal of achieving autonomous mapping and navigation was successfully realized.

2.3 Project Management

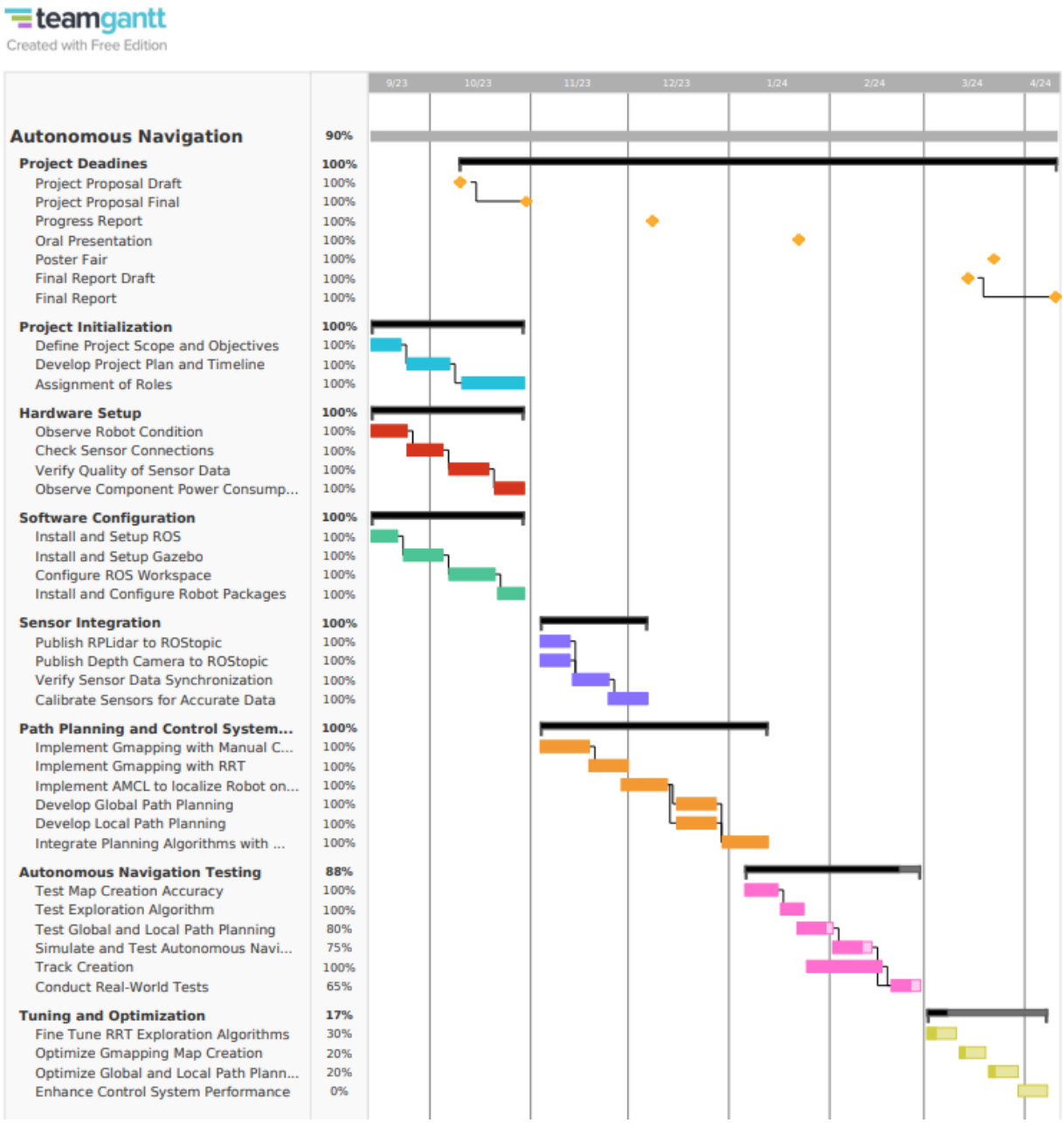

Figure 2: Updated Gantt Chart

2.4 Justification and Suitability for Degree Program

This project allowed us to demonstrate many concepts we have learned throughout our engineering program. The revised iterations of the robot system allowed us to demonstrate concepts of distributed systems, computer communication (TCP, IP, Wireshark, Firewalls), and advanced programming, as the system was originally

designed to function solely on the JetAuto locally. In order to incorporate remote computation, the concepts of computer communication and computer security allowed us to setup a system with increased performance, while avoiding issues with time delay and computer security. In the process of comparing the different mapping, exploration, and navigation algorithms, the engineering concept of providing analytical data to solidify choices in algorithms were applied. This project could also be adapted for other engineering program applications, such as the inclusion of a complex simulation system for software engineering application or adaptation of other autonomous projects together (road sign detection, collision avoidance, lane detection, etc.) to demonstrate the wider application of the concepts we have applied to the autonomous navigation projects.

2.5 Individual Contributions

2.5.1 Project Contributions

- **R** denotes the individual who is responsible for executing the task
- **A** denotes the individual who is ultimately accountable for the task
- **I** denotes the individual who is consistently informed on the progress of the task
- **S** denotes the individual who can provide support for the task

Task	Sundar Vengadeswaran	Sagar Syal	Sangat Buttar	Himanshu Singh	Ray Prina
Mapping	R, A	S	A	I	S
Exploration	R, A	R, A	I	S	S
Navigation	S	R, A	S	I	I
Simulation	S	I	R, A	S	A
Sensors	I	I	S	R, A	I
Testing	I	S	I	A	R, A

Table 1: Updated Responsibility Matrix

- **Mapping**
  - This Task involves the integration of the lidar and the mapping algorithms.
- **Exploration**
  - This task involves the set-up and integration of the exploration algorithms.
- **Navigation**
  - This task involves the setup and the integration of the navigation stack.
- **Simulation**
  - This task involves the setup and creation of the simulated environment for virtual deployment.
- **Sensors**
  - This task involves the setup of the sensors installed on the robot.
- **Testing**
  - This task involves the creation of the test environment and the evaluation of metrics.

2.5.2 Report Contributions

Name	Sections
Sundar Vengadeswaran	1, 3, 5
Sagar Syal	3, 4, 5
Sangat Buttar	2, 3, 4
Ray Prina	2, 3, 6
Himanshu Singh	4, 5

Table 2: Report Contributions

3.0 System Design

The project scope mainly centred on the creation of an autonomous mapping and navigation rover using the HiWonder JetAuto robot. The system came equipped with a LiDAR sensor and an RGB-D camera system [1]. The primary objective of the autonomous mapping and navigation system was to emulate the functionality of modern autonomous rover mechanisms while tailoring it toward a parking lot environment. However, due to the nature of the software requirements, described in Section 3.1.2, it was determined that the RGB-D camera was not needed.

The project split this high-level task into two main operations: autonomous mapping, and autonomous navigation. The end-goal of the rover was to reliably explore an unknown parking lot environment while simultaneously generating a map (autonomous mapping), and then navigating to a specified point on the map all the while avoiding dynamic obstacles (autonomous navigation). To accomplish this, the project employed the use of the JetAuto to compare multiple SLAM mapping, exploration, and navigation planing algorithms, paired with multiple hardware configurations, detailed below.

3.1 Component Design

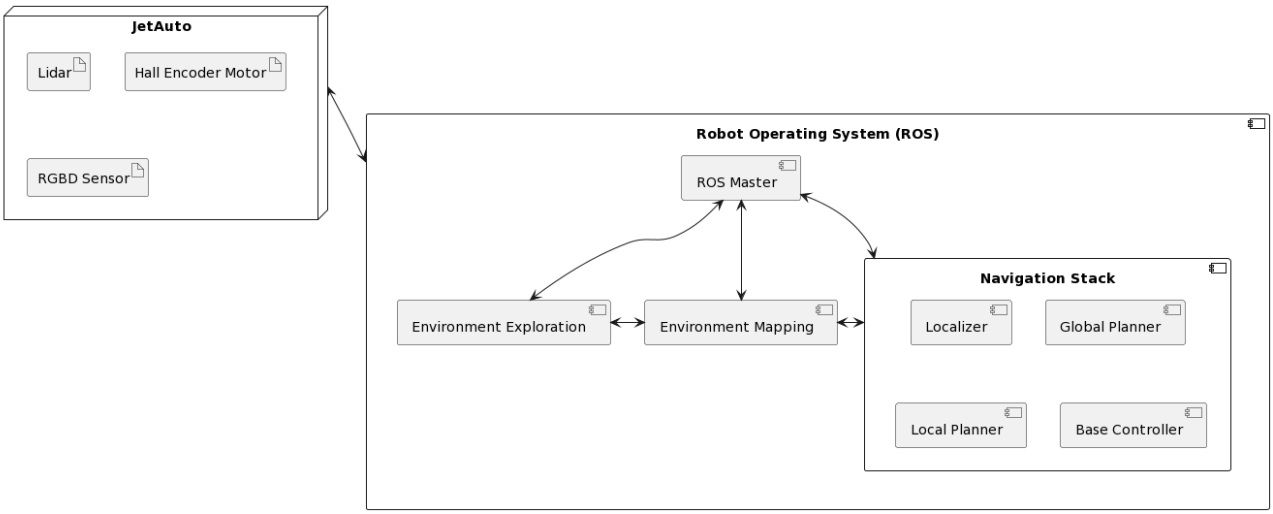

Figure 3: Use Case Diagram

3.1.1 Hardware Design

The hardware components used in this project are:

1. LiDAR Sensor
2. Hall Encoder Geared Motors
3. 4-Channel Encoder Motor Driver
4. ROS Control System and Expansion Board
5. 11.1V 6000mAh Lithium Phosphate Battery

The components used in this project each had their own function that they were required to fulfill. The **LiDAR Sensor** which was located on the front of the robot was used to scan the surrounding area. This data was a major component utilized to generate a map of the surrounding room [2]. There are four **Hall Encoder Motors** on the bottom of the robot, each motor was used to enable the robot's movement [1]. The four motors were connected to a **4-Channel Encoder Motor Driver Board** [1]. The motors were controlled by a **Servo Controller**, which provided precise control over the robot's movement. In conjunction with the 4-channel encoder, the encoder was used to obtain accurate odometry measurements [2]. The **ROS Control System and Expansion board** was the motherboard of the JetAuto, connecting all the hardware components together. A 11.1V 6000mAh **Lithium Phosphate Battery** powered the entire JetAuto [1], which proved to be faulty during hardware and software testing of the system.

Together, each component was used to complement one another to ensure the project was successful. Firstly, the user inputted the desired destination on one of the external devices. After the robot determined which direction it should take, the Encoder Motor Driver Board controlled all 4 Hall Encoder Motors and moved the robot towards the

desired destination. The LiDAR sensor was used to scan and map the robot's surroundings. The obtained LiDAR data was used in conjunction with multiple nodes to generate a map of the robot’s surroundings and outputs the map onto the external device. The robot then determined the optimal path it took to ensure it arrived at the desired destination while avoiding collisions.

There were 3 iterations of the hardware design, which adjusted the method of running the ROS nodes and components of the system on multiple computers. The first iteration involved running all components on the JetAuto itself. The 2nd iteration involved publishing the odometry and LiDAR data from the robot on to a ROS topic. The remote computer then subscribed to the ROS topics and completed computation for mapping, navigation, and exploration. The 3rd iteration included the addition of another remote computer to split the task of the mapping and exploration on one of the computers, while navigation was processed on another computer. The sensor data publishing task was still handled by the JetAuto through publisher topics, however there were now 2 subscribers: the computer handling mapping/exploration and the computer handling navigation.

3.1.2 Software Design

The project objective was developed and accomplished using ROS1 (ROS Melodic) system architecture. ROS is a combination of software frameworks and libraries, written in C++ and Python, allowing for the communication and transfer of data between robotic hardware components, for both cross-device and cross-network purposes [2]. In order to enable this communication, ROS employs the use of three major components: nodes, topics, and messages [2].

A ROS node is a process or an executable which runs a certain operation. Nodes use ROS topics, which are akin to communication channels, to transfer data with other nodes [2]. ROS nodes can either subscribe or publish to a topic [2]. ROS messages are made up of the actual data that is sent via the topics [2]. Lastly, we have the ROS Master, which is another node which allows other nodes to find each other when establishing communication [2]

3.1.2.1 ROS Nodes

Below is a list of the major ROS nodes that were used:

Sensors:

Lidar: rplidarNode

Subscribed Topics: None	
Published Topics: 1	
sensor_msgs/LaserScan	ROS Laser Scan message

Table 3: LiDAR Sensor Node [2]

The LiDAR node above publishes a LaserScan format, based on the LiDAR sensor's output [2]. The LaserScan message is then used by the SLAM nodes that are listed below. Additionally, the LaserScan messages can be visualized using the RViz tool, the red markers indicate the objects the LiDAR sensor scans [2].

Integration:

hiwonder_servo_manager

Subscribed Topics: 5	
servo_controllers/port_id_1/servo_states	Servo controller status
joint1_controller/state	Status update from wheel joint 1
joint2_controller/state	Status update from wheel joint 2
joint3_controller/state	Status update from wheel joint 3
joint4_controller/state	Status update from wheel joint 4
Published Topics: 5	
servo_controllers/port_id_1/servo_states	Updated servo controller status
joint1_controller/command_duration	Execution time for wheel joint 1
joint2_controller/command_duration	Execution time for wheel joint 2

joint3_controller/command_duration	Execution time for wheel joint 3
joint4_controller/command_duration	Execution time for wheel joint 4

Table 4: Hiwonder Servo Manager Node

The Hiwonder Servo Manager node described above in Table 4 publishes the time taken for each wheel joint to execute a specific command. This time is crucial as it is used to calculate odometry, which is then subsequently published to the mapping algorithms.

joint_states_publisher

Subscribed Topics: 4	
joint1_controller/state	Status update from wheel joint 1
joint2_controller/state	Status update from wheel joint 2
joint3_controller/state	Status update from wheel joint 3
joint4_controller/state	Status update from wheel joint 4
Published Topics: 1	
joint_states	Cumulative wheel joint status

Table 5: Joint States Publisher Node

The Joint States Publisher node outlined above in Table 5 publishes the overall status of wheel joints together. The status relates to whether data such as odometry is readable to the system.

robot_state_publisher

Subscribed Topics: 1	
joint_states	Cumulative wheel joint status
Published Topics: 1	
Tf/tfMessage	Published wheel data to be transformed

Table 6: Robot States Publisher Node

The Robot State Publisher node seen above in Table 6 publishes the joint wheel status to the TF topic, which can be further used by any mapping and exploration algorithms.

jetauto_odom_publisher

Subscribed Topics: 2	
jetauto_controller/cmd_vel	Currently executing velocity command
cmd_vel	Updated velocity command
Published Topics: 2	
odom_raw	Raw odometry data
set_pose	Updated robot pose

Table 7: JetAuto Odometry Publisher Node

The Jetauto Odometry Publisher node described above in Table 7 publishes raw odometry data, paired with an updates position point, which can then be used to localize the robot within the map.

ekf_localization

Subscribed Topics: 5	
odom_raw	Raw odometry data
set_pose	Updated robot pose
cmd_vel	Updated velocity command
imu	Inertial Measurement Unit status
Tf/tfMessages	Published wheel data to be transformed
Published Topics: 1	
Tf/tfMessages	Published odometry data to be transformed

Table 8: EKF Localization Node

The EKF (Extended Kalman Filter) Localization node outlined above in Table 8 subscribs to raw odometry, updated position, current velocity commands, inetial data, and exsiting Transform data to publish updated odometry data. This data is then subsequently transformed and used by the mapping and exploration algorithms.

Mapping (SLAM):

The following SLAM algorithms were tested and compared with each other in terms of performance and accuracy.

slam_gmapping

Subscribed Topics: 2	
sensor_msgs/LaserScan	ROS Laser Scan message to a create map
tf/tfMessage	Transforms message into laser, base and odometry data
Published Topics: 3	
nav_msgs/MapMetaData	Map data, periodically updated
nav_msgs/OccupancyGrid	Map file, periodically updated
sd_msgs/Float64	Estimated distribution entropy over rover position

Table 9: GMapping SLAM Node [2]

The slam_gmapping node subscribes to the LaserScan messages that were published by the LiDAR node [2]. Slam_gmapping then uses its own tf topic to transform the obtained sensor data into the format the mapping algorithm requires (e.g. Odometry) [2]. The slam_gmapping node then publishes the map data (OccupancyGrid), and the maps metadata (MapMetaData), both can be updated periodically [2].

hector_mapping

Subscribed Topics: 2	
sensor_msgs/LaserScan	ROS Laser Scan message to create a map
std_msgs/String	System Commands, e.g., “Reset”

Published Topics: 3	
nav_msgs/MapMetaData	Map data, periodically updated
nav_msgs/OccupancyGrid	Map file, periodically updated
geometry_msgs/PoseStamped	Estimated robot position
geometry_msgs/PoseWithCovarianceStamped	Estimate robot position including calculated covariance (uncertainty)

Table 10: Hector SLAM Node [2]

The hector_slam node subscribes to the LaserScan messages that were published by the LiDAR node [2]. However, unlike the slam_gmapping node, hector_slam does not require any transformation of its obtained data [2]. The hector_slam node then publishes the map data (OccupancyGrid), and the maps metadata (MapMetaData), both can be updated periodically.

slam_karto

Subscribed Topics: 2	
sensor_msgs/LaserScan	ROS Laser Scan message to create map
tf/tfMessage	Transforms message into laser, base and odometry data
Published Topics: 3	
nav_msgs/MapMetaData	Map data, periodically updated
nav_msgs/OccupancyGrid	Map file, periodically updated
Visualization_msgs/MarkerArray	Graph of robot position, periodically updated

Table 11: Karto SLAM Node [2]

The slam_karto node subscribes to the LaserScan messages that were published by the LiDAR node [2]. The slam_karto node, similar with slam_gmapping, also requires the use of a tf topic to transform its data into odometry data [2]. The slam_karto node then publishes the map data (OccupancyGrid), and the maps metadata (MapMetaData), both can be updated periodically [2].

Exploration:

RRT Exploration Algorithm:

global_ rrt_frontier_detection

Subscribed Topics: 2	
nav_msgs/OccupancyGrid	Map file, continuously updated as the rover moves
geometry_msgs/PointStamped	5 points that are used to define the global region to explore
Published Topics: 2	
geometry_msgs/PointStamped	Detected frontier points
visuaization_msgs/Marker	Line shapes using Rviz

Table 12: RRT Global Node [2]

local_rrt_frontier_detection

Subscribed Topics: 2	
nav_msgs/OccupancyGrid	Map file, continuously updated as the rover moves
geometry_msgs/PointStamped	5 points that are used to define the local region to explore
Published Topics: 2	
geometry_msgs/PointStamped	Detected frontier points
visuaization_msgs/Marker	Line shapes using Rviz

Table 13: RRT Local Node [2]

The global_rrt_frontier_detection and local_rrt_frontier_detection nodes are similar, as they are both used to detect frontiers (e.g., obstacles, borders, etc.) in the robot's range [2]. These frontiers are used as targets for the robot to explore in real-time.
Both nodes subscribe to the map data (OccupancyGrid) and five points in the robot's vicinity (PointStamped) [2]. They then generate the frontier points and provide markers on the relative locations of the frontier points, to be used by the filter and assigner nodes [2].

The difference between the two nodes is that the global_rrt_frontier_detection node defines the total exploration area that the robot will explore at a given time, whereas the local_rrt_frontier_detection node defines the immediate exploration targets in the robot's vicinity [2].

frontier_opencv_detector

Subscribed Topics: 1	
nav_msgs/OccupancyGrid	Map file, continuously updated as the rover moves
Published Topics: 2	
geometry_msgs/PointStamped	Detected frontier points
visuaization_msgs/Marker	Line shapes using Rviz

Table 14: RRT Frontier Node [2]

The frontier_opency_detector is another node used to explore frontier targets. However, this node does not use the traditional algorithm that the previously mentioned frontier detection nodes use but rather uses the OpenCV library to detect frontiers in the robot's vicinity [2]. This node is used as a comparison and backup to enhance the frontier detection capabilities of the robot [2]. The OpenCV detection node, once again, generates the frontier points and provides markers on the relative locations of the frontier points, to be used by the filter and assigner nodes [2], as shown below.

filter

Subscribed Topics: 2	
nav_msgs/OccupancyGrid	Map file, continuously updated
geometry_msgs/PointStamped (goals topics)	Detected frontier points
Published Topics: 3	
visuaization_msgs/Marker (frontiers)	Received frontier points
visuaization_msgs/Marker (centroid)	Filtered frontier points
filtered_points	An array of filtered frontier points

Table 15: RRT Filter Node [2]

The filter subscribes to the PointStamped topic generated by the frontier detection nodes that were previously mentioned. The node then uses a filtration process to remove invalid and redundant frontier points generated by the three detection nodes and publish the newly filtered points to the assigner node [2], as explained below.

assigner

Subscribed Topics: 3	
nav_msgs/OccupancyGrid	Map file, continuously updated as the rover moves
nav_msgs/OccupancyGrid (Filtered)	Filtered frontier points on the map
goals_topic	Detected frontier points
Published Topics: None	

Table 16: RRT Assigner Node [4]

The assigner node does not publish any nodes. However, it subscribes to the filtered frontier points that were generated by the filter node and uses ROS services to send commands to the navigation stack [2].
For example, the move_base node which will be explained below.

explore_client

Subscribed Topics: 1	
geometry_msgs/PointStamped	Clicked points from rviz too
Published Topics: 1	
visualization_msgs/Marker	Boundary visualization via clicked points.

Table 17: Frontier Exploration Client Node [2]

The explore_client node subscribes to the “geometry_msgs/PointStamped” topic, receiving the points the user clicked from RVIZ – the locations are then designated for exploration by the user [2].
It publishes markers using the “visualization_msgs/Marker” topic, visually representing the exploration area boundaries derived from the selected points [2].

explore_lite

Subscribed Topics: 2	
nav_msgs/OccupancyGrid	The map which will be used for exploration planning. Can be either costmap from move_base or map created by SLAM
map_msgs/OccupancyGridUpdate	Incremental updates on costmap
Published Topics: 1	
visualization_msgs/MarkerArray	Visualization of frontiers considered by exploring algorithm

Table 18: Explore Lite Node [2]

The explore_lite node in ROS enables greedy frontier-based exploration by subscribing to the “nav_msgs/OccupancyGrid” topic and “map_msgs/OccupancyGridUpdate” topic, receiving the points that were clicked by the user on RVIZ, and constructing the map [2]. It publishes the “visualization_msgs/MarkerArray” topic for boundary representation [2]. The greedy algorithm focuses on quickly exploring areas without considering the entire environment [2].

Navigation Stack:

amcl

Subscribed Topics: 4	
sensor_msgs/LaserScan	ROS Laser Scan message to create map
tf/tfMessage	Transforms message
geometry_msgs/PoseWithCovarianceStamped	Estimated rover position with covariance, initializes filter
nav_msgs/OccupancyGrid	Map file, periodically updated
Published Topics: 3	
geometry_msgs/PoseWithCovarianceStamped	Updated estimated rover position with covariance

geometry_msgs/PoseArray	An array of estimated positions maintained by filter
tf/tfMessage	Transformed odometry message

Table 19: AMCL Node [2]

The AMCL node is a localizer node which subscribes to the map file generated by the mapping node (OccupancyGrid). It then uses the original LaserScan messages from the sensor and the tf topic, to publish the robot's estimated position [2].

global_planner

Subscribed Topics: None	
Published Topics: 1	
nav_msgs/Path	Global path

Table 20: Global Planner Node [2]

The global_planner node publishes an estimated global path based on certain robot parameters such as localization, orientation, etc. [2].
This path is continuously updated as more reliable mapping data is available [2].

dwa_local_planner::DWAPlannerROS

Subscribed Topics: 1	
nav_msgs_Odometry	Odometry information
Published Topics: 2	
nav_msgs/Path (global_plan)	Global plan of path to follow
nav_msgs/Path (local_plan)	Local plan (immediate trajectory)

Table 21: Local Planner Node [2]

The dwa_local_planner node uses the Dynamic Window Approach (DWA) algorithm to generate a local path, encompassing the immediate trajectory for the robot to follow [2]. The local planner subscribes to the odometry data that was generated by the mapping nodes [2], mentioned above. The local planner also updates the global planner based on any changes that occur in real-time [2].

move_base

Subscribed Tooccur1	
geometry_msgs/PoseStamped	Robot position with timestamp
Published Topics: 1	
geometry_msgs/Twist	Stream of velocity commands (cmd_vel)

Table 22: Movement Node [2]

The move_base node subscribes to the robot's position, generated by the SLAM mapping node [2]. The node then publishes a stream of velocity commands known as "cmd_vel" [2].
The "cmd_vel" topic uses the JetAuto_controller (The robot's local base controller) to move the robot to the intended position [2].

teleop_twist_keyboard

Subscribed Topics: None	
Published Topics: 1	
geometry_msgs/Twist	Stream of velocity commands (cmd_vel)
geometry_msgs/TwistStamped	Stream of velocity commands (cmd_vel) with timestamp

Table 23: Teleoperated Movement Node [2]

The teleop_twist_keyboard node uses ROS parameters and services to read keyboard strokes and publishes a "cmd_vel" topic to JetAuto_controller [2].
This node is used to implement controlled movement during the testing phases of the robot [2].

3.1.2.2 ROS Node Architecture

The ROS nodes described in the section above operate together to form a fully functional mapping and navigation system. In addition to this, specific nodes execute on specific machines depending on the hardware implementation outlined in Section 3.1.1.

The ROS architecture of implementation 1 can be seen in Figure 4 below:

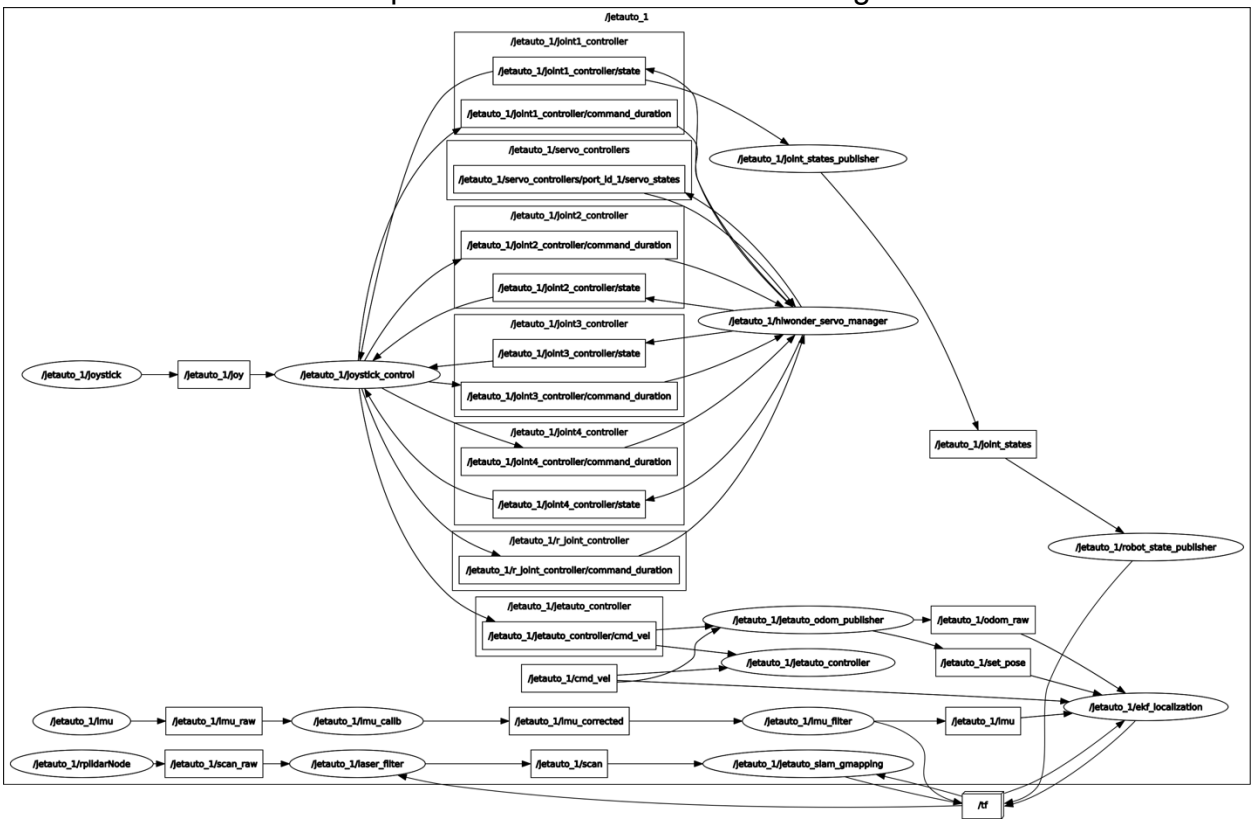

Figure 4: ROS Node architecture diagram of Implementation 1

The ROS architecture of implementation 2 can be seen in Figure 5 below:

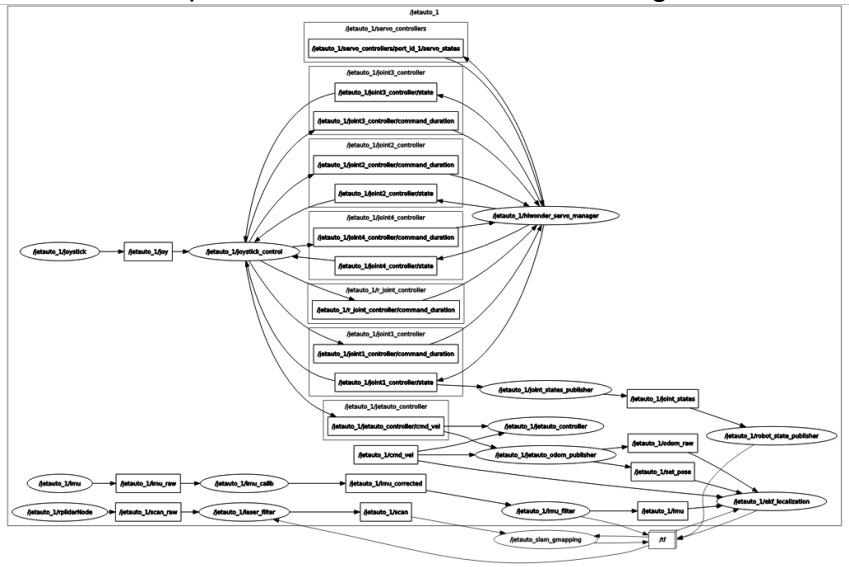

Figure 5: ROS Node architecture diagram of Implementation 2

Unfortunately, due to the major safety incident described in Section 2.1, the data required for implementation's ROS node architecture was destroyed.

3.1.2.3 ROS2 Simulation Architecture

Owing to the fact of the safety incident, the final system needed to be quickly and efficiently transitioned to a simulation focused environment. Hence, the decision to migrate to a ROS2 workspace stemmed from this specific matter of concern. Unlike, ROS1, ROS2 offered a more streamlined and singular package approach [4]. Rather than employing the use of several modules on separate devices, ROS2 allows for the launching of all mapping related nodes from one singular module [4].

Slam_Toolbox:

The module in question is the slam_toolbox package, ported from the ROS1 slam_karto module, for further details refer to section 3.1.2.1 [4]. The slam_toolbox package was required to be launched in conjunction with either teleoperation or navigation2, ROS2's navigation stack [4]. The results of the slam_toolbox module can be seen in Section 5.2.

3.1.3 Gazebo Simulation

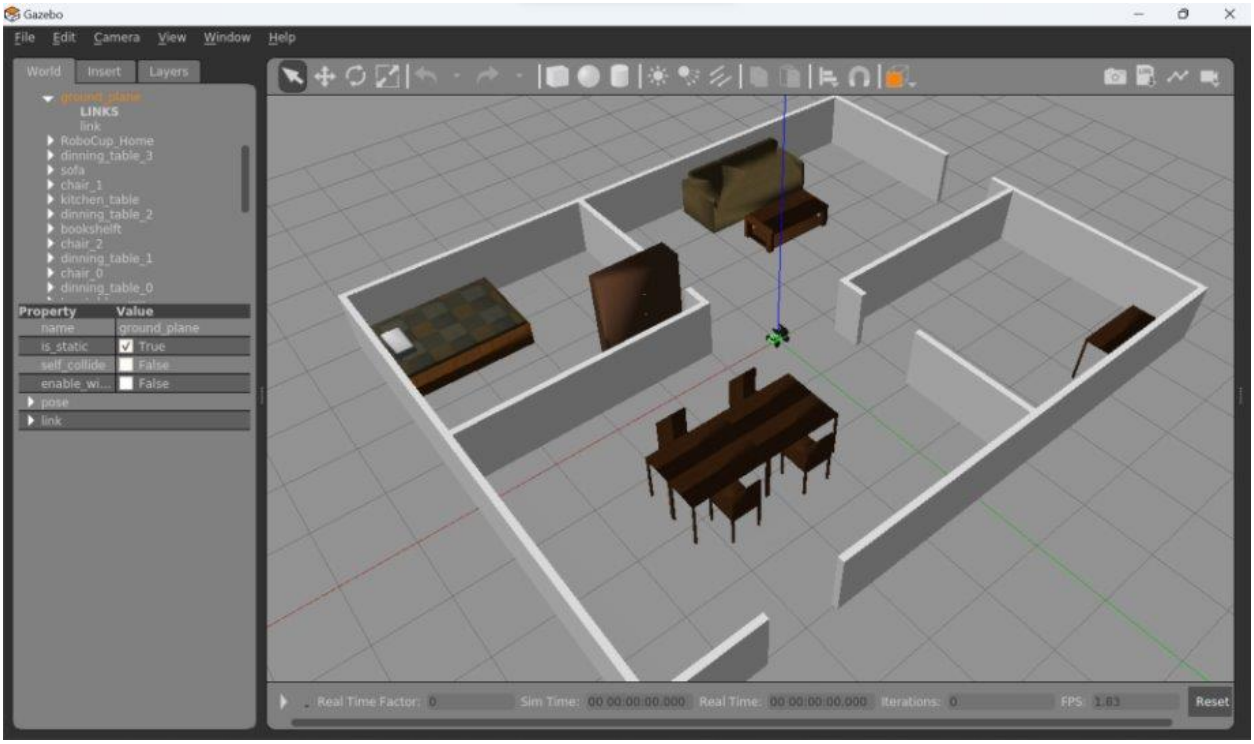

Figure 6: Gazebo Simulator with Example Environment

Gazebo is a simulation software which was used to test the exploration and mapping algorithms in complex environments [2]. It allowed us to import a robot model, along with its components, such as the rpLiDAR and astra camera, to test it with the ability to integrate real life components. Gazebo supports physics-based models, which allowed us to emulate the behaviour of the robot as accurate to real life as possible [2]. It also had collision modelling, so that the robot stopped once it had collided with objects at specified speeds [2]. It also allowed us to create geometric relations between its components, such as the movement of the wheels of the robot [2]. All of this allowed us

to simulate the robot with a high degree of accuracy, while being less time consuming and providing similar test data to a real demo.

3.1.4 RVIZ

RViz is a 3D visualization tool for ROS that allows the user to visualize various sensor data, robot states, generated maps, etc. [2].
Implemented RVIZ Functions:

1. **Sensor Data Visualization:**
   - RVIZ enables the visualization of sensor data from sources such as cameras, laser scanners, and 3D point clouds [2]. This is useful in understanding how a robot perceives its environment.
2. **Robot State and Map Display:**
   - The position and orientation of the robot can be visualized relative to the generated map [2]. This was important for monitoring the robot's behavior and verifying that it meets expectations.
3. **Node Map Visualization:**
   - RVIZ can display a theoretical map of all active nodes that were running on the ROS system [2].

The RViz tool was used to visualize the map file and the Robot’s relative position and orientation. Furthermore, it was used to test the functionality of the exploration and mapping algorithms.

3.1.5 System Calibration

3.1.5.1 Timing

Due to a timing issue, outlined in Section 4.3, leading to the implementation of a Network Time Protocol (NTP) Server for synchronization across the system's devices. The NTP server was critical in addressing problems where ROS messages were dropped, deemed out of date due to time discrepancies among the distributed system. Without synchronized timing, messages arrived with timestamps that did not match the receiver's clock, resulting in the system discarding them as outdated. By integrating an NTP server, time alignment across all devices was achieved, eliminating the issue of messages being dropped due to being perceived as out of date [6].

3.2 Testing and Simulation

A real test track environment was created to test the JetAuto’s navigation, exploration, and navigation capabilities. As in Figure 6, this environment contained multiple paths and tight corners, which would allow testing of the accuracy of the different algorithms, along with the limitations each method encounters. This environment was first modeled in AutoCAD and followed by a model in Gazebo Simulator.

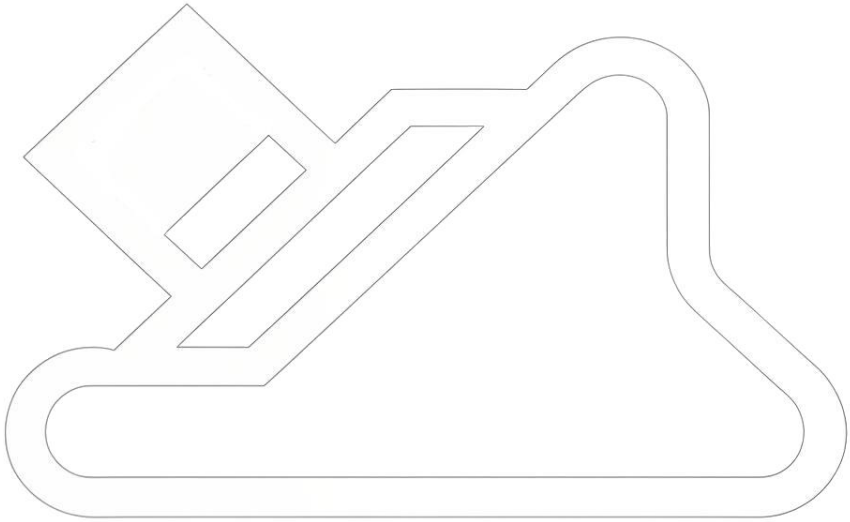

Figure 7: Previous Track

The previous track layout as in Figure 7 was remodeled to allow for in depth analysis of several exploration algorithms, shown in Figure 8. The increased number of obstacles and blind corners allow us to test the limitations of different exploration techniques.

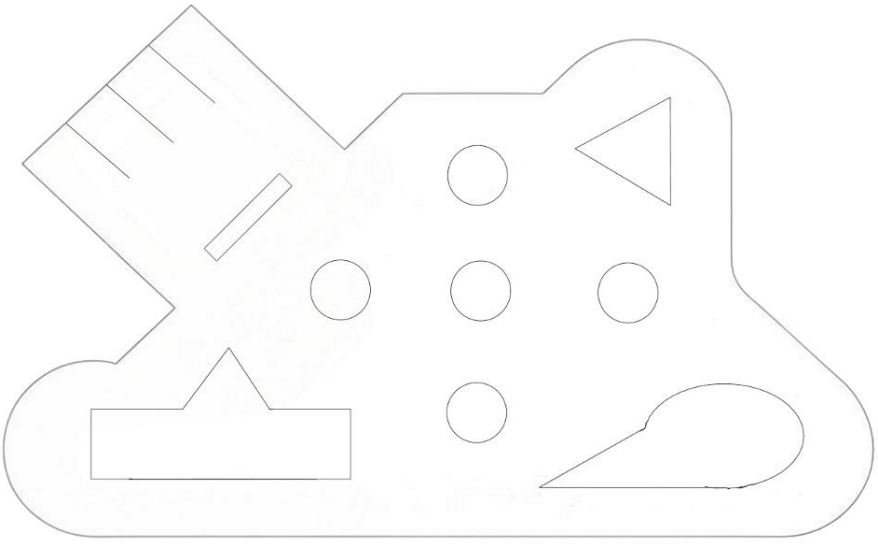

Figure 8: Remodeled Track

Following real-time testing of the JetAuto on the test track in Figure 8, a remodel was created as in Figure 9 as the corners were too tight for the JetAuto to make successfully without colliding with the exterior walls of the track. This led to a remodel of the track with squared off edges rather than the semi-circle turns, which allowed the JetAuto to successfully take the turns, while maintaining the mapping accuracy of the sharper turns in the previous model Figure 8.

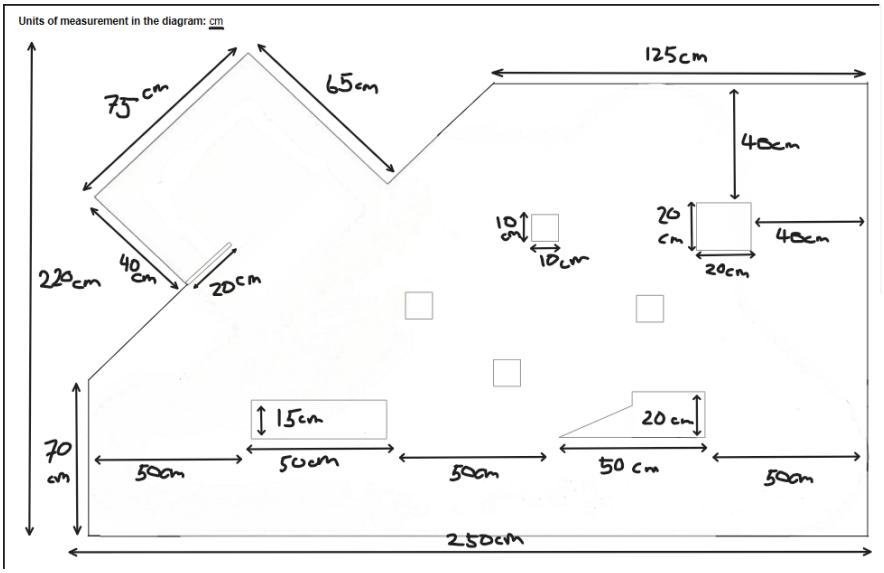

Figure 9: 3rd Remodeled Track with Squared Off Corners

The final track as in Figure 9, was made with cardboard to prevent damage to the sensitive components of the JetAuto and allowed testing of the turning radius of the robot without causing damage to the JetAuto.

Figure 10: Alternative Track create for Robot Testing

Figure 10 is the track that was created to test out the robots mapping capabilities using slam_gmapping, the drawn map resulted with mentioned in Figure 11.

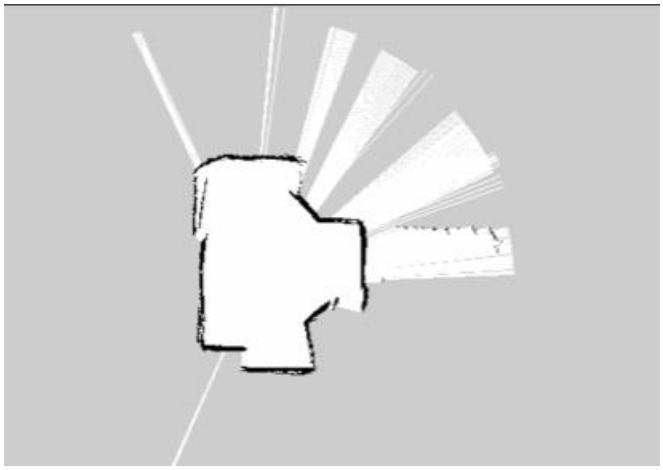

Figure 11: Gmapping Physical Map

Figure 11 is the drawn map obtained from slam_gmapping, the real-life version of the map is shown in Figure 10.

4.0 Challenges

4.1 Communication

During the course of the project, both Windows Subsystem Linux (WSL) and virtual machines (VM) were employed by individuals on the team. Initially, for individuals using WSL, the latest version (WSL 2) was employed. This later caused issues with communication due to how WSL 2 employed IPs on a network and how ROS required IPs. WSL 2 utilized a subnet approach where the system created a virtual IP under the Windows system. This clashed with the needs of ROS where machines in the system needed to be on the same network as the master. To remedy this, the WSL 2 system was ported over to WSL 1 which acted like a conventional machine on a network and had a separate IP from its Windows host. A similar network connectivity issue was encountered with the virtual machines. To address this, the network settings for the VMs were adjusted from the default Network Address Translation (NAT) to a bridged configuration. This allowed the VMs to appear as individual hosts on the network, thereby enabling the necessary ROS communication across the same network as the master.

4.2 Firewall

ROS1 utilized a network port which was blocked by the firewall of the remote computer. This prevented communication of the ROS nodes with the other remote computers and the JetAuto itself. This challenge was solved by adding an exception to the firewall for the port utilized for ROS communication.

4.3 Time Delay

Due to synchronization challenges, the robot was not maintaining accurate time alignment with the multiple devices that it communicated with. The initial design omitted a CMOS battery, as such, the robot's internal clock could only be updated once powered on. Consequently, the robot's time needed to be manually adjusted to ensure synchronization with other devices. This process was primarily conducted through linux's date command; a task that was both time-consuming and inefficient.

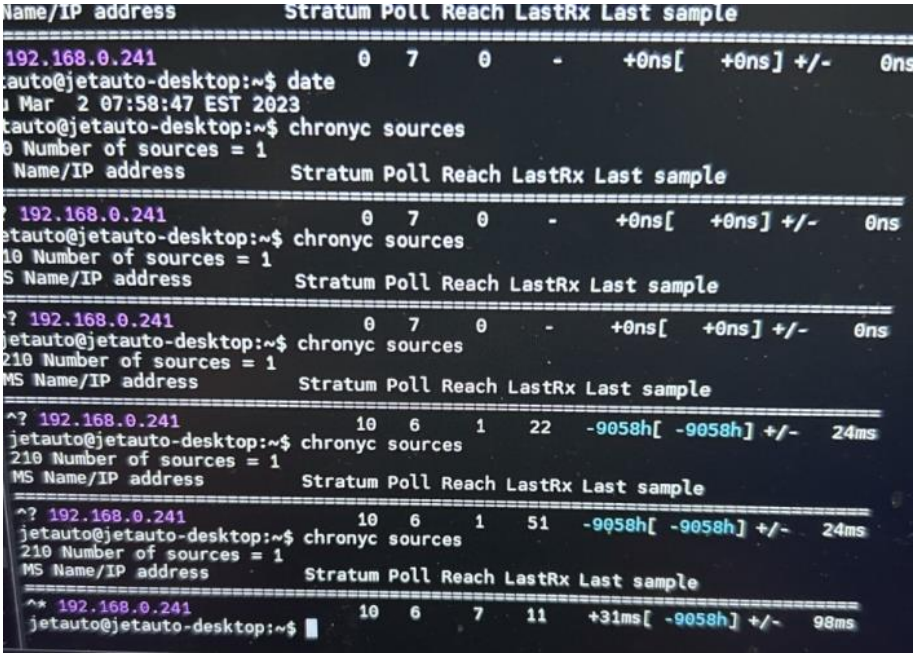

Figure 12: Chrony sources list

To address this, an NTP (Network Time Protocol) server was implemented via Chrony, designating one device as the host for other devices, including the robot, to subscribe to. This automation significantly reduced the need for manual time updates, enhancing efficiency in device synchronization. Figure 12 displays a sequence of IP addresses, as provided by Chrony, indicating the robot's connections. From top to bottom, the transition from question marks to an asterisk symbolizes the robot's identification of a device as the NTP server, effectively recognizing it as the host.

4.4 Power Issues

Throughout the process of testing the Hiwonder JetAuto, a multitude of issues were uncovered surrounding power delivery to the components of the robot. This was first highlighted during testing of the RGB-D and LiDAR sensor and required occasional power cycling of the robot. This issue also caused issues with the motor controller, as it would cause the node to crash when sufficient power was not available, leading to a cascading list of crashing nodes.

4.5 Deprecated software support

The Hiwonder Jetauto operating system was Linux 18.04, with robot operating system 1. Linux 18.04 had many issues with support for the libraries needs to run the Gazebo Simulation and ROS nodes, as there were many bug fixes and updates created just for Linux 20.04. This issue was also encountered during the utilization of ROS1, as many mapping, navigation, and exploration algorithms were updated for ROS2. This created many issues which required our own solution implementation, as the solution given in the documentation revolved around updating the system to ROS2 or a new fork of Linux. The solution to this challenge was to spend more time evaluating issues in the system, as many solutions involved breaking down the problem to

5.0 Final Achievements

The following section summarizes all the achievements that were made throughout the course of the project.

5.1 Physical Achievements

5.1.1 Multiple Implementations

During the project, several iterations were executed to enhance performance and achieve optimal outcomes.

5.1.1.1 Local Implementation

In the initial phases of the project a local implementation was employed for all nodes. As such, mapping and exploration, navigation, and sensor nodes were run on the HiWonder JetAuto. The resulting maps can be seen in Figure 13 below.

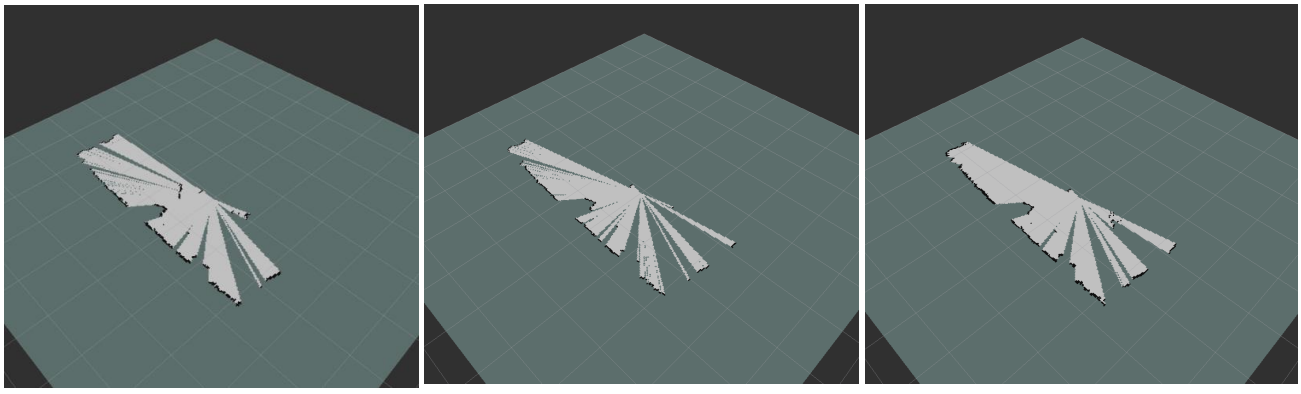

Figure 13: Local Implementation of all Nodes

During this initial implementation, we established the baseline performance. However, the implementation faced significant challenges, notably in the form of extensive time delays. These delays adversely affected map generation, which in turn impacted maneuverability, as the exploration process heavily relies on the availability of accurate maps.

5.1.1.2 Offloaded to One Machine

In the second phase of the project, all sensor data collection nodes remained active on the JetAuto, while other computational nodes were offloaded to a separate machine to enhance efficiency and resource management as seen in figure 14.

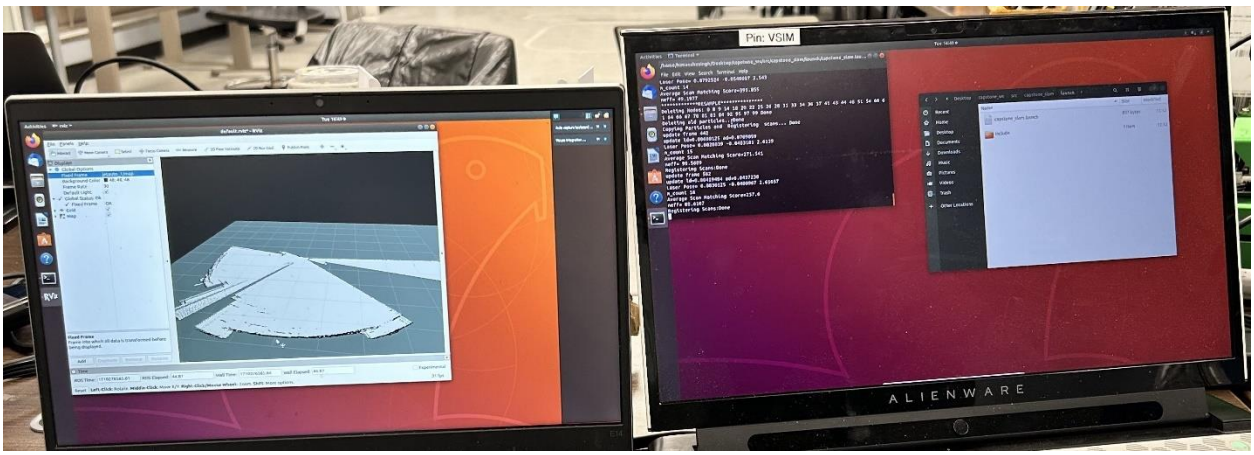

Figure 14: Offloaded to a Singular Machine

The second implementation significantly enhanced performance in map generation and navigation by utilizing distributed computing, by offloading the computational tasks to another machine, as seen in Figure 14 above. This approach allowed the JetAuto to respond to map updates more effectively, especially when detecting obstacles. Distributed computing refers to the strategy of dividing computational tasks between multiple machines, which can lead to more efficient processing and quicker response times. By distributing the workload, the JetAuto was able to handle map updates more efficiently, reducing the delay during the exploration phase of the model. However, even with this improvement, there was still a delay in map generation, suggesting potential areas for further optimization. After investigation this delay was then discovered to be primarily due to a memory bottleneck, as the machine to which tasks were offloaded had only 8GB of memory. This limited memory capacity constrained the amount of data that could be processed simultaneously, leading to delays in generating and updating the map.

5.1.1.3 Offloaded to Multiple Machines

In the final implementation of the project all sensor data was left on the JetAuto, mapping and exploration nodes were put on a separate machine and the navigation stack was put on a secondary machine. The third iteration of this project achieved optimal performance. By reapplying the principles of distributed computing, we effectively resolved the memory bottleneck. This enhancement enabled JetAuto to perform at its best, generating maps with negligible delay. As a result, it allowed for rapid responses to obstacles and facilitated efficient exploration.

5.1.2 Mapping Algorithms

5.1.2.1 Hector

During the testing phase, Hector SLAM did not meet performance expectations. Initially chosen for its ability to perform real-time mapping without the need for high-quality odometry data, relying on LIDAR inputs, its strengths were apparent in environments that are flat and less structurally complex[2]. However, in more demanding settings, particularly those lacking distinct features, Hector SLAM's performance was suboptimal. This led to challenges in generating accurate maps, as the algorithm struggled with maintaining consistency and detail in areas of complexity as you can see from figure 15 it doesn’t have the same level of figure 17.

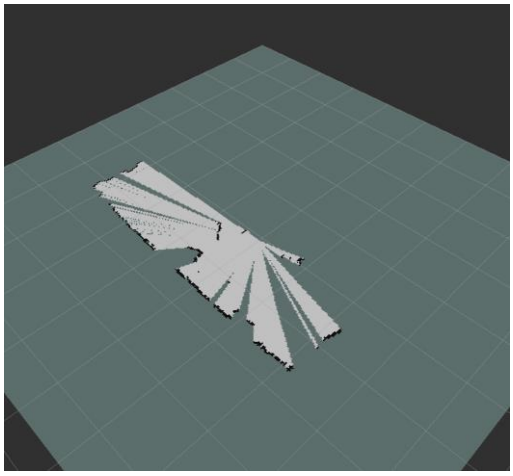

Figure 15: hector_slam Map

5.1.2.2 Karto

In the evaluation phase, Karto demonstrated superior performance when compared to Hector. Karto uses both LIDAR and odometry data, contributing to its better performance [2]. This integration allows for more accurate and detailed map construction, especially in environments with complex structures and varied landscapes as seen as figure 16.

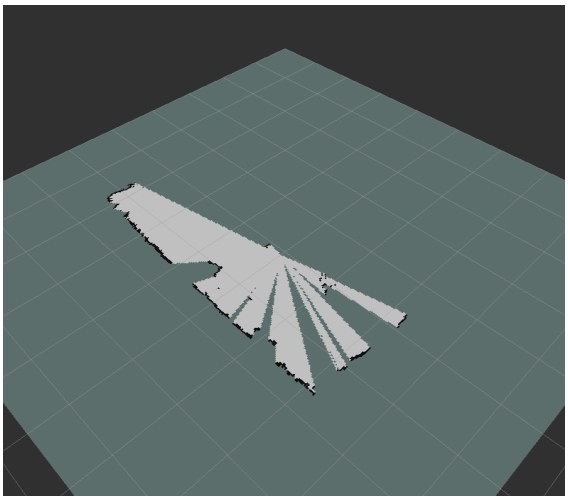

Figure 16: karto_slam Map

5.1.2.3 Gmapping

Gmapping had the best performance when compared to the other mapping algorithms due to its effective use of the Rao-Blackwellized Particle Filter, which integrates odometry and LIDAR data to create accurate maps [2]. This method allows Gmapping to efficiently handle the uncertainties associated with robot localization and map construction, leading to highly detailed and reliable maps. Figure 17 displays this by showing the most area as well as showing the most obstacles.

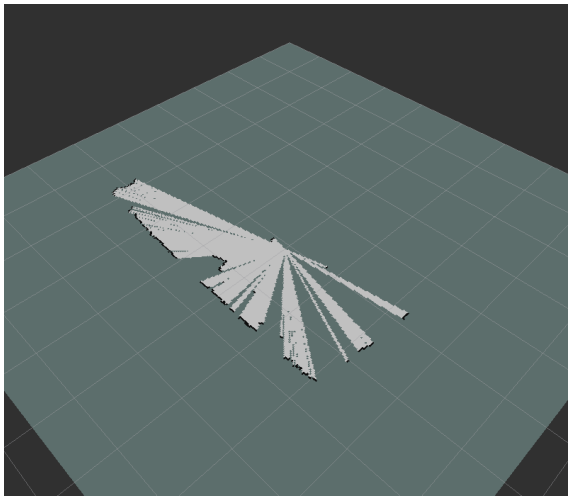

Figure 17: slam_gmapping Map

5.1.3 Exploration Algorithms

5.1.3.1 Frontiers

Frontiers was an effective algorithm as it explored the environment systematically, focusing on the boundaries between known and unknown areas. This approach allowed for mapping of large, open spaces by sequentially moving towards these 'frontier' regions, which are the edges of unexplored territory. However, it encountered difficulties in more complex environments where numerous obstacles were present. The presence

of obstacles expanded the quantity of frontiers, thereby expanding the expanse of areas unknown, necessitating more comprehensive exploration [2].

5.1.3.2 ExploreLite

Although Explorelite streamlines the exploration process by reducing the redundant travel it has a harder time dealing with dynamic environments. Due to how Its method of prioritizing nearby unexplored areas can sometimes lead to a lack of thoroughness, potentially overlooking important distant regions [2]. This proximity-based decision-making may result in missed opportunities for significant discoveries. Additionally, the algorithm's focus on efficiency compromises the depth of exploration in complex settings, where a more systematic approach could uncover valuable insights.

5.1.3.3 RRT Exploration

RRT lead to the best results due to the balance it achieves in exploration, efficiently navigating through complex environments with its rapid tree expansion [2]. This approach allows a thorough of the environment adapting to diverse terrains. The versatility of RRT in striking a balance between exploring vast, open areas and probing through cluttered spaces contributed significantly to its superior performance.

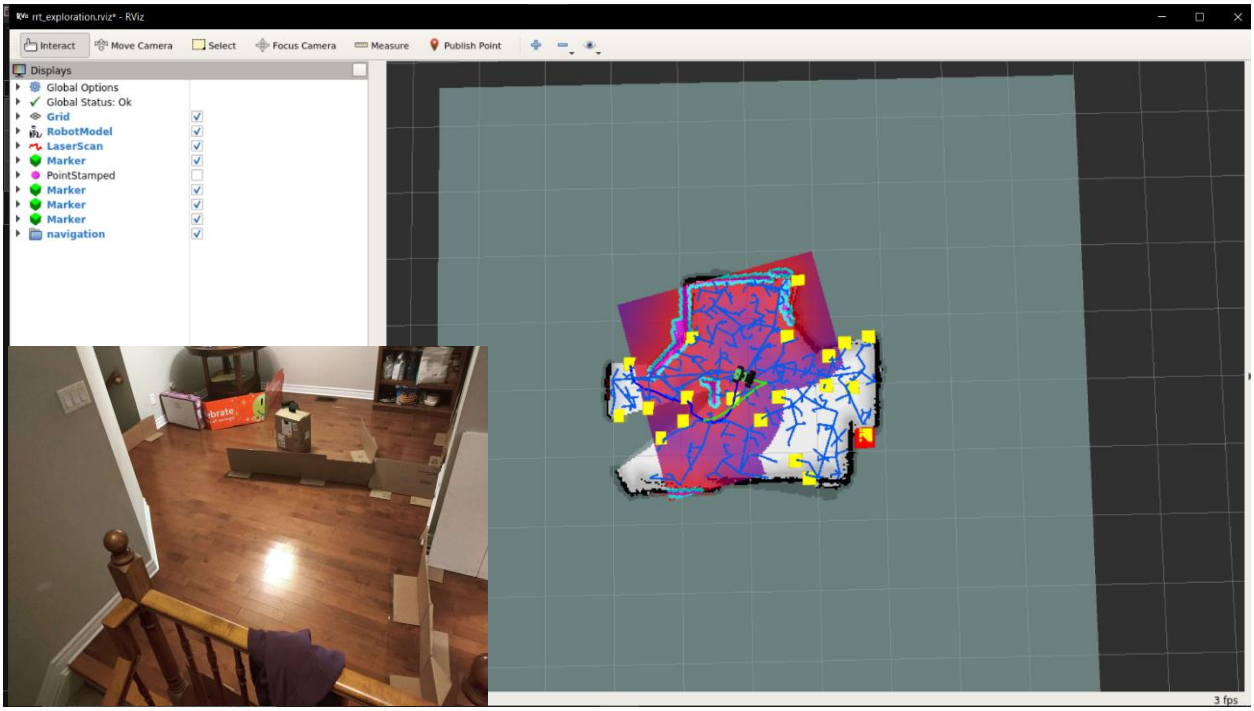

Figure 18: Result of RRT exploration

Figure 18 shows the result of an RRT exploration algorithm visualized in Rviz. The yellow square are the global frontiers that are points that the JetAuto tries to navigate to. The RRT package has branches exploring a large amount of the map comparing it to the real track.

5.2.3 EVO Python Package

Recent progress has been made by researching and planning the usage of the EVO Python package. This package provides a small library for dealing with trajectory, odometry and SLAM algorithms [5].

Next Steps:

1. **Evaluation Metrics:**
    - Uses the function evo_ape to measure how accurately JetAuto estimates its position compared to the robot's actual position.
    - Uses the function evo_rpe to understand how well JetAuto maintains position estimates over time.
    - Uses the function evo_traj for a detailed analysis of JetAuto's trajectory.
2. **Result Comparison:**
    - Uses the function to compare results from different setups and configurations.
3. **Incorporating Ground Truth:**
    - Establish a 'ground truth' trajectory by tracking JetAuto's actual movements.
4. **Integration with evo:**
    - Integrate trajectory files with the evo package for evaluation.

5.3 Simulation Performance Analysis

As mentioned in sections above, due to the safety incident, numerical performance analysis was completed on simulation rather than the physical implementation, paired with a migration to ROS2. To accurately measure and analyse mapping performance, the EVO software tool was used, specifically measuring odometry data. The slam_toolbox module, described in section 3.1.2.3 is heavily reliant on odometry data in order to publish its map.

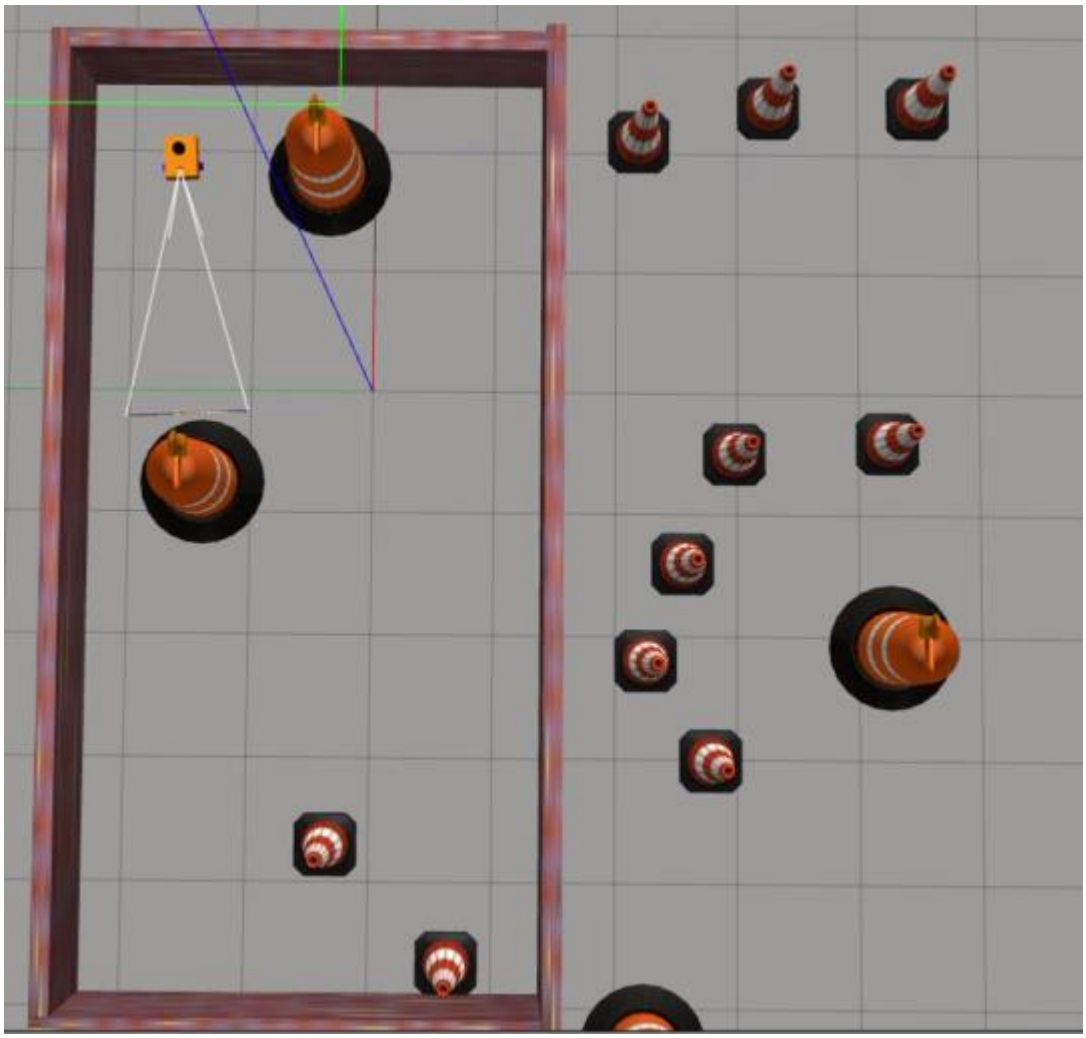

Figure 19: Environment In Gazebo

Figure 19 above presents a top-down perspective of the environment developed in Gazebo, designed for our robot to navigate and map using SLAM_ToolBox. The robot in the simulation is controlled using teleoperation. This setup enables the robot to gather odometry data across several trials, which are saved in rosbags. Rosbags are a file format used to record and playback messages being passed over ROS. These rosbags serve as input for the EVO Python package, which can then produce a range of statistics displayed graphically. The robot has predetermined paths it would go alone to maintain the consistency of the data.

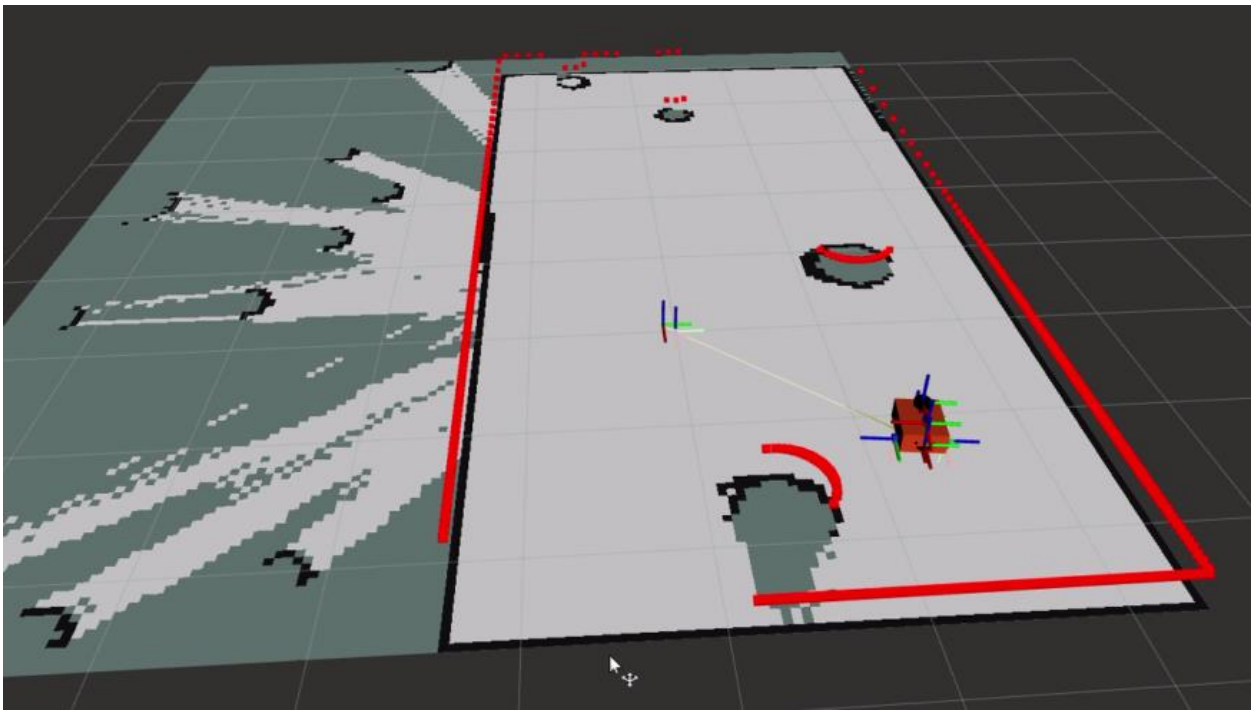

Figure 20: Mapped Environment in RVIZ

Figure 20 above shows an RVIZ representation of a map generated from the environment within Gazebo. The streaks visible on the left represent areas of the environment that have been mapped by SLAM_ToolBox, attributed to the relatively small size of the enclosing walls. The circular patches (in dark gray) that remain unmapped within the environment correspond to pylons.

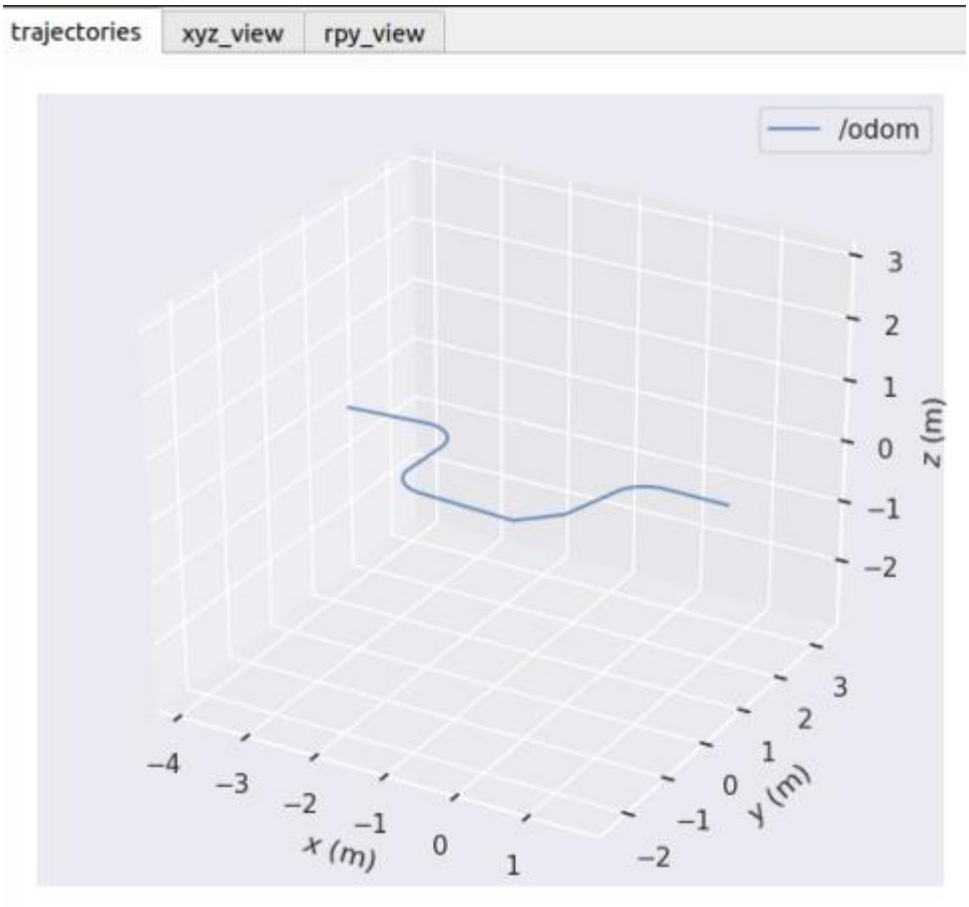

Figure 21: Robot Trajectory

Figure 21 above shows 3D trajectory view. It visualizes the path that the robot has taken in three-dimensional space. It remains consistent along the z axis which indicates that the environment is flat, and the turns being made are the robot maneuvering around the pylons.

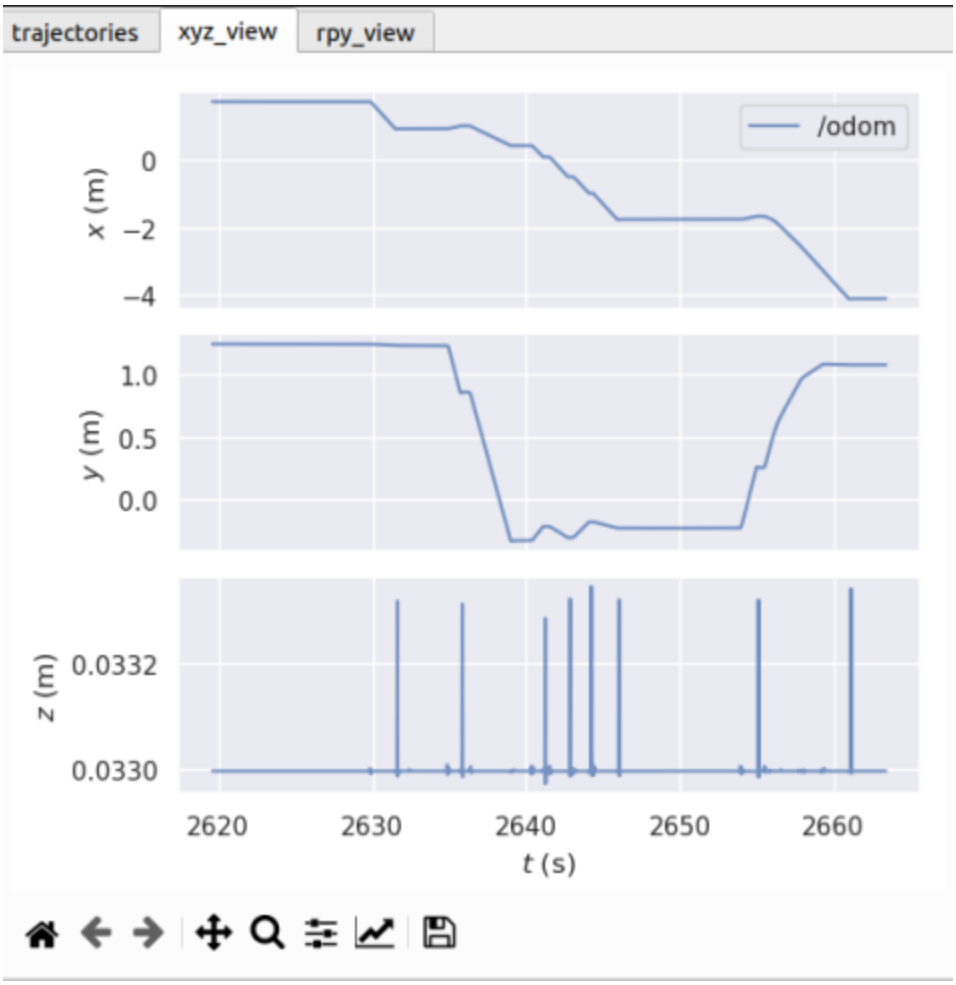

Figure 22: Robot Movement In X, Y, Z Axis over time

Figure 22 above is a plotted representation on the xy-plane of the robot's movement over time (t). It shows how the x, y, and z positions change with time. The x and y plots show a series of stepped changes, indicating controlled, incremental movements. The z plot shows small spikes, which could represent moments when the robot navigated over small bumps or irregularities in the environment.

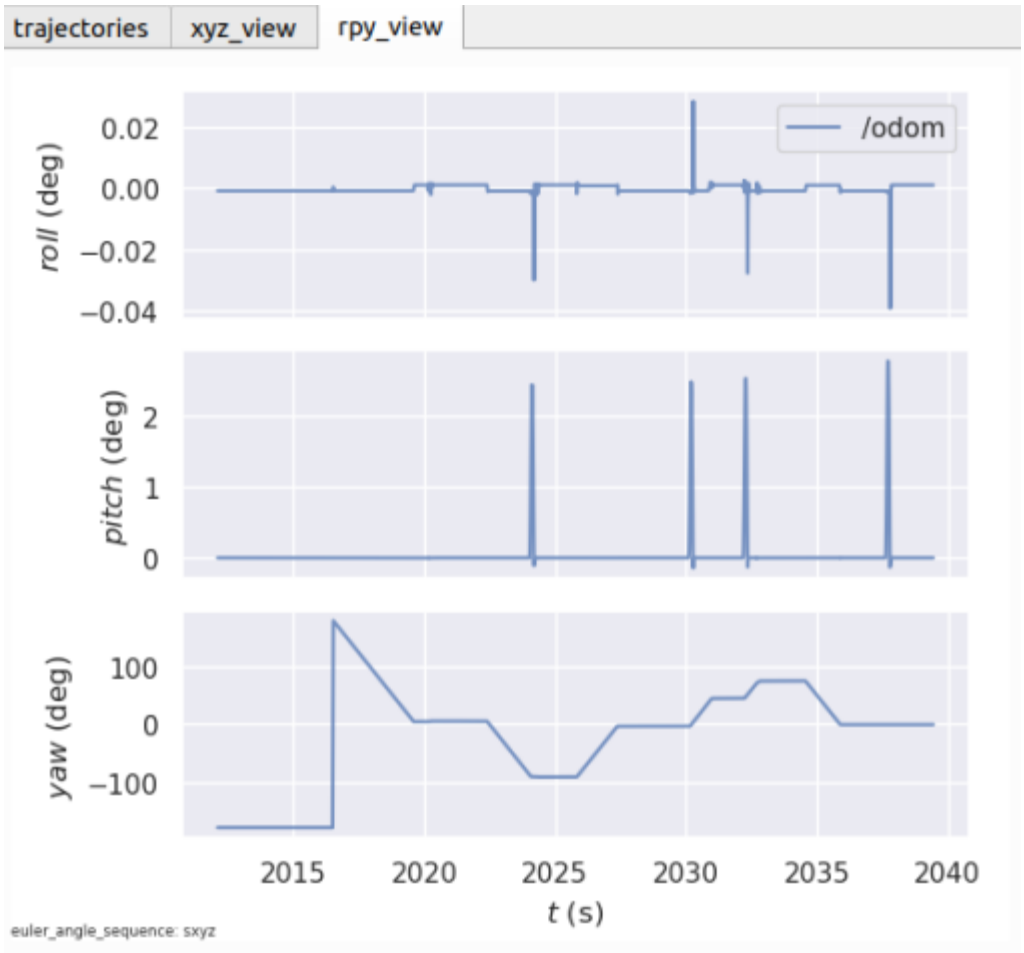

Figure 23: Robot Roll, Pitch, and Yaw

Figure 23 above shows the roll, pitch, and yaw (RPY) of the robot in degrees. Roll and pitch seem to have small fluctuations around zero, indicating that the robot is mostly stable but may be encountering some minor unevenness or obstacles. Yaw has larger variations, which suggests changes in the robot's orientation - it's turning left or right as it moves.

6.0 Conclusion

In conclusion, a lot of extensive changes have been made to the mapping algorithm, exploration algorithm, and the navigation stack to ensure the capstone project was a success. New nodes were added to the software design to ensure the project functioned as intended and some nodes were removed as they were deemed obsolete. The physical test track was remodeled for a third time as the robot was incapable of making sharp turns around the curved track, the remodeled track was squared off and some obstacles were removed. During the test phase the robot functioned as intended, mapping, exploration and navigation were working, and the robot was set for the metric tests. This was when the robot caught fire and exploded, all the files saved on the robot were lost.

Due to the loss of the robot the team was forced to adapt and switch into a more digital approach, shifting priority to the simulation. ROS2 was learned to work with the more modern version of the simulator.

7.0 References

[1] “Hiwonder Jetauto Ros Robot car powered by Jetson Nano with lidar depth,” Hiwonder, https://www.hiwonder.com/products/jetauto?variant=39962923860055 (accessed Apr. 9, 2024).

[2] “Wiki,” ros.org, https://wiki.ros.org/ (accessed Apr. 9, 2024).

[3] “Optimal storage conditions: Temperature and humidity guidelines for lipo batteries,” LiPol Battery Co., https://www.lipobatteries.net/lipo-battery-news/optimal-storage-conditions-temperature-and-humidity-guidelines-for-lipo-batteries/#:~:text=Storage%20Temperature%20Range%3A%20LiPo%20batteries, battery’s%20capacity%20and%20prevents%20degradation. (accessed Apr. 10, 2024).

[4] “Ros 2 Documentation,” ROS 2 Documentation - ROS 2 Documentation: Foxy documentation, https://docs.ros.org/en/foxy/index.html (accessed Apr. 10, 2024).

[5] M. Grupp, “Python Package for the evaluation of Odometry and SLAM,” Evo, https://michaelgrupp.github.io/evo/ (accessed Dec. 8, 2023).

[6] “How to Install and Configure NTP on Linux”, https://timetoolsltd.com/ntp/how-to-install-and-configure-ntp-on-linux/