Hochschule Bremerhaven

Navigation Team Report

Submitted in accomplishment

of

Lab Report

elemental in the degree of

M.Sc. in Embedded Systems Design

by

Krish Shah - 41564

Nirzer Gajera - 41638

Yogachand pasupuleti- 41588

Shamanth Adiga - 41527

Niranjan Channayya - 41559

Person in Charge

Prof. Dr.-Ing. Oliver Prenzel

Subject

ESD PROJECT

Preface

Authors

- Krish Shah – 41564
- Nirzer Gajera – 41638
- Yogachand pasupuleti– 41588
- Shamanth Bandimata Adiga – 41527
- Niranjan Channayya – 41599

Change History

1st Draft Version – V1.0 (02.01.2025)

2nd Draft Version – V1.1 (27.02.2025)

Final Draft Version – V2.0 (7.03.2025)

Table of Contents

Table of Figures

List of Tables

List of Abbreviations

Abbreviations	Meaning
ROS2	Robot Operating System 2
GNSS	Global Navigation Satellite System
RTK	Real-Time Kinematic
GPS	Global Positioning System
SLAM	Simultaneous Localization and Mapping
LIDAR	Light Detection and Ranging
HDDM	High-Definition Distance Measurement
AMCL	Adaptive Monte Carlo Localization
Rviz	ROS Visualization
BT	Behaviour Trees
UDP	User Datagram Protocol
PLL	Phase Locked Loop
DGPS	Differential Global Positioning System
INS	Inertial Navigation System
GUI	Graphical User Interface
QoS	Quality of Service
DDS	Data Distribution Service
Nav2	Navigation2
IEEE	Institute for Electrical and Electronics Engineers
GPS	Global Positioning System
URDF	Unified Robotics Description Format
PID	Proportional Integral Derivative
Wi-Fi	Wireless Fidelity
TF	Transform
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
ICP	Iterative Closest Point
BT	Behaviour Tree
AMCL	Adaptive Monte Carlo Localisation

1 Introduction- *{Nirzer Gajera (41638)}*

1.1 Abstract- *{Nirzer Gajera (41638)}*

Autonomous navigation is a vital capability for mobile robots, allowing them to function effectively in dynamic and unknown environments. This project develops an autonomous navigation system utilizing a mobile robotic platform that integrates advanced sensors, motor controllers, and computing technologies. The NVIDIA Jetson AGX Orin powers the system and employs ROS2 Humble as its middleware for real-time control. Key components include a SICK TIM781 LiDAR for obstacle detection, a Swift DGPS/INS module for accurate localization, and a RoboClaw Motor Controller for motor actuation.

The ROS2 Nav2 Stack is utilized for path planning and obstacle avoidance, while ROS2 SLAM is employed for mapping unknown environments. A tailored PyQt6-based GUI facilitates real-time monitoring and waypoint setting, with DDS middleware ensuring efficient communication. This report outlines the system architecture, hardware-software integration, navigation implementation, and testing methodologies. Performance evaluations underscore the system's effectiveness and discuss challenges, limitations, and potential improvements.

1.2 Short Introduction- *{Nirzer Gajera (41638)}*

Autonomous navigation is a key field in robotics, allowing mobile robots to explore, map, and operate in environments without human intervention. Applications include industrial automation, warehouse logistics, autonomous vehicles, and planetary exploration [1]. Despite significant advancements, achieving reliable autonomy in unstructured and dynamic environments remains challenging due to factors like real-time decision-making, sensor accuracy, localization drift, and computational constraints [2].

This project addresses these challenges by developing a ROS2-based autonomous navigation system, leveraging **NVIDIA Jetson AGX Orin** for high-performance computing [3]. The system integrates sensor fusion techniques with a **SICK TIM781 LiDAR** for precise obstacle detection and mapping [4], and a **Swift DGPS/INS module** for accurate localization [5]. The **ROS2 Nav2 stack** handles global and local path planning [2], while **ROS2 SLAM** enhances mapping in unknown environments [6]. A **PyQt6-based GUI** provides an intuitive interface for user interaction, allowing waypoint setting and monitoring.

In contrast to existing solutions that rely on **single-sensor navigation**, this project emphasizes **multi-sensor integration** for improved robustness [7]. Additionally, real-time communication via **DDS middleware** ensures efficient and low-latency data exchange between system components [8]. The development aligns with the increasing demand for autonomous solutions in industries such as **logistics, agriculture, and security robotics** [9].

This report presents the design, implementation, and performance evaluation of the navigation system. It discusses the challenges encountered, potential improvements, and future extensions to enhance the system's capabilities.

1.3 Motivation- {*Nirzer Gajera (41638)*}

The motivation behind this project arises from its role as a mandatory subject within the Master's program in Embedded Systems Design, making it a critical component of the curriculum. This initiative provides students with hands-on experience that bridges theoretical concepts with real-world applications, particularly in autonomous robotic systems and embedded technologies. By participating, students deepen their understanding of sensor integration, real-time data processing, and autonomous navigation—fundamental principles in modern embedded systems engineering.

The project offers students exposure to cutting-edge hardware and software technologies, including the NVIDIA Jetson AGX Orin, ROS2 Humble, and advanced sensor fusion tools such as the SICK TIM781 LIDAR, SWIFT PIKSI DGPS, and INS modules. These components enable real-time obstacle detection and precise positioning, emphasizing the practical challenges of autonomous systems design. This experience enhances competencies in hardware-software interfacing, data acquisition, and real-time processing, which are essential for professional growth.

A key motivation is the emphasis on collaborative teamwork and multidisciplinary cooperation. Coordination with parallel teams working on robotic arm integration and depth-sensing camera applications fosters cross-functional collaboration, enhancing problem-solving, critical thinking, and communication skills. Students learn to manage complex system architectures, delegate tasks effectively, and integrate diverse subsystems into a cohesive solution.

Ultimately, this project supports the learning objectives of the Master's program while preparing students for careers in industries where embedded systems and autonomous technologies are rapidly evolving. By working on a comprehensive autonomous rover platform, students gain the confidence and expertise needed to contribute to advancements in robotic automation, scientific exploration, and industrial innovation.

1.4 Project Vision- {*Nirzer Gajera (41638)*}

The vision of this project is to develop an autonomous mobile platform capable of seamlessly navigating both indoor and outdoor environments using advanced perception, localization, and path planning technologies. The system will integrate ROS2 middleware, Nav2 navigation stack, SICK TiM781 LiDAR, and SWIFT PIKSI DGPS/INS to achieve precise real-time obstacle detection and adaptive waypoint-based navigation. Indoors, the platform will utilize SLAM-based localization, while outdoors, it will rely on RTK-GNSS and IMU-based positioning for high-accuracy navigation. A wireless GUI will allow users to monitor real-time sensor data, switch between manual and autonomous control, and adjust waypoints dynamically, ensuring flexibility in operation.

This project aims to create a versatile, intelligent, and scalable robotic system that can operate autonomously with real-time obstacle avoidance while also supporting open-loop waypoint execution with obstacle detection as a fallback mechanism. The modular architecture ensures that the system can adapt to future improvements, such as multi-robot coordination, 3D mapping, and cloud-based navigation services. By leveraging ROS2's communication framework and DDS middleware, the platform will remain robust, efficient, and expandable, making it a reliable solution for autonomous industrial, research, and field applications.

1.5 Structure of report- {*Nirzer Gajera (41638)*}

This project report is a comprehensive document detailing the **design, development, and testing** of an **autonomous mobile platform** utilizing **ROS2, Nav2, SICK TiM781 LiDAR, and SWIFT PIKSI DGPS/INS** for navigation. The report is structured to provide a **systematic flow of information**, beginning with the **motivation and objectives**, followed by the **technical design, implementation, and evaluation** of the system.

The introduction lays the foundation for the project by presenting an abstract summarizing the system's capabilities, goals, and implementation strategy. The short introduction provides context on autonomous navigation and why it is crucial for mobile robotics. The motivation section discusses the challenges in autonomous navigation, emphasizing the need for robust real-time obstacle avoidance, waypoint-based navigation, and human intervention capabilities. The project vision defines the objectives and desired outcomes, outlining the planned functionalities of the system. Finally, the structure of the report describes how the document is organized, ensuring a logical flow for the reader.

This section covers the planning, execution, and management of the project. The milestones subsection provides an overview of the hardware and simulation development progress. The time plan presents a structured timeline of development, testing, and validation phases. The team roles subsection defines the responsibilities of each member, ensuring a collaborative and efficient workflow. The software management tool segment explains the use of GitHub for version control and collaborative development.

The **system specification** section outlines the **core functionalities** of the mobile platform. It describes the **user story**, detailing **indoor and outdoor navigation capabilities**. This section also specifies the **requirements and constraints**, ensuring a realistic scope for the implementation.

The **background and technology overview** section provides an in-depth discussion of the **state-of-the-art technologies** used in the project. It covers **ROS2 as a middleware solution**, **PyQt6 for GUI development**, and **Data Distribution Service (DDS) for efficient real-time communication**. Additionally, it explains the **Nav2 framework** for navigation and **SWIFT PIKSI's GNSS-RTK capabilities** for achieving centimetre-level localization accuracy. It also highlights all the software tools used for this project. It also contains a description of all the components used for the project. It provides a brief breakdown of the **core computing unit (NVIDIA Jetson AGX Orin), LiDAR sensor (SICK TiM781), GNSS-INS module (SWIFT PIKSI), stereo camera (ZED-X), and motor controllers**.

The **architecture and design** section presents the **functional architecture** of the system, both hardware and software illustrating how data flows between sensors, control algorithms, and actuation modules. It also describes the component level architecture for SICK TIM781 LiDAR and Swift PIKSI Navigation module. The list of Components is also included in this chapter.

This section focuses on software development, including ROS2 workspace setup, custom node implementation, and URDF-based rover simulation. The Python-based GUI development is explained in detail, highlighting the key libraries, functions, and classes. The section also discusses SLAM, NAV2, and obstacle avoidance strategies, with an emphasis on static map generation and real-time navigation. Additionally, the ROS2 driver implementation for SWIFT PIKSI is covered.

The test plan outlines different testing stages, including navigation and path planning validation. The test results are analysed for both indoor SLAM-based navigation and outdoor GPS-based navigation. The section also includes a summary of findings, system limitations, and recommendations for future improvements.

This section summarizes the key achievements of the project. It identifies the limitations and the goals which were not achieved. The report concludes with suggestions.

The final section contains IEEE-formatted references used in the report. Additionally, code listings for major system components are included for reproducibility and future development.

2 Project Management- *{Shamanth B Adiga (41527)}*

Agile project management provided a structured yet adaptive framework for our project, enabling us to effectively navigate evolving requirements and stakeholder expectations. By incorporating agile sprints, we established a dynamic and collaborative environment that emphasized flexibility, shared responsibility, and continuous iteration. Our project was systematically divided into smaller, more manageable tasks, ensuring that each team member had a clear focus and well-defined responsibilities. As a team of five, we strategically allocated specific aspects of the project to individual members, optimizing efficiency and leveraging our collective expertise.

To maintain alignment and ensure steady progress, we held weekly meetings with our professor every Wednesday, during which we provided comprehensive project updates, discussed emerging challenges, and incorporated feedback to refine our approach. These regular check-ins reinforced accountability and allowed us to swiftly adapt to any new insights or necessary modifications. Through sprint planning, stand-up meetings, and iterative refinements, we fostered a culture of transparency and collaboration, ensuring that our efforts remained targeted and our deliverables met the highest standards. This agile methodology not only streamlined our workflow but also enhanced our ability to respond proactively to challenges, ultimately leading to the successful execution of our project.

2.1 Milestones – *{Shamanth B Adiga (41527)}*

2.1.1 Hardware Status

Tasks	**Hardware Status**	**Timeline**
1. Setting up tools and software	Completed	10/2024
2. Getting Data from IMU and GPS	Completed	02/2025
3. Getting Data from Sick Lidar	Completed	01/2025
4. Setting up the GUI and communication with the rover	Completed	12/2024
5. Manual Control of Rover from Gui	Completed	01/2025
6. Basic Implementation of path planning from the GUI	Completed	02/2025

Table 1. Hardware Status

2.1.2 Simulation Status

Tasks	Simulation Status	Timeline
1. Setting up Ubuntu, Ros2 Humble, Gazebo, Rviz	Completed	10/2024
2. Building our rover in the simulated environment	Completed	11/2024
3. Adding a Lidar plugin, Camera plugin	Completed	12/2024
4. Setting up SLAM TOOLBOX for map generation	Completed	01/2025
5. Autonomous navigation using Nav2	Completed	02/2025

Table 2. Simulation Status

2.2 Time Plan- *{Shamanth B Adiga (41527)}*

Month	Krish	Nirzer	Yoga	Niranjan	Shamanth
October	Getting familiar with ROS2, Online course, Documentation	Getting Familiar with ROS2 from ROS2 Documentation, GitHub, Python	Getting familiar with ROS2, Online course	Installing ROS2 and dependencies and getting familiar.	Getting started with ROS2 with the help of turtlebot3.
November	Configuring Roboclaw controller, setting up Architecture and working on URDF for the rover.	SICK LIDAR hardware study and thorough implementation of Sick_scan_d GitHub Repository for ROS2	Getting started with lidar for mapping	Roboclaw usage. Getting started with Swift piksi multi-unit for navigation.	Built a robot using URDF files and spawned it in Gazebo.
December	Started development on Target PC, Debugging solutions for Odometry data.	Modifying launch files and Rviz files specific to TIM781 took Nav2 a short time to understand and decide on the project's	Worked on safety stop with lidar data and started NAV2 & SLAM	Processing IMU data, calculating distance and publishing it on a topic in ROS and Analysing the issues with it	Worked on Slam Toolbox for mapping.

		architectur e.			
January	Testing for odometry data using encoders and ROS2 driver based data.	Design File for GUI, PyQT6 based GUI and also PySide6 based GUI, DDS communic ation for Motor Control	Implemented safety stop, mapping with SLAM in simulation. Worked on NAV2	Building and installing ROS2 driver for Swift Piksi.	Worked on backend of the GUI.
February	Worked towards simple solution, alongside trying to implement NAV2 using Visual odometry.	Creation of a basic map with some obstacles on the GUI. Multiple points selection on the GUI and sending it to Rover for basic navigation.	Worked and implemented NAV2 for autonomous navigation	Applicability of Swift Piksi for outdoor applications- GPS standalone data	Worked with the NAV2 simulation.

Table 3. Time Plan

2.3 Team Roles- *{Shamanth B Adiga (41527)}*

Team Members	Working Areas and Components
Krish	Motor Control, URDF, Ros2 Architecture, NAV2 Implementation
Nirzer	LIDAR, GUI, Basic Implementation
Yoga	ROS2, LIDAR, SLAM, NAV2, simulation of robot
Niranjan	Handling Swift navigation piksi unit for navigation
Shamanth	SLAM, NAV2, Simulation of the robot, GUI

Table 4. Team Roles

2.4 Software Management tool – GitHub- *{Shamanth B Adiga (41527)}*

- **Centralized Repository Management:** A common GitHub repository was established to serve as the single source of truth for all project-related code. This ensured seamless access to the latest versions of the codebase for all team members.
- **Collaborative Development and Code Integration:** The team, consisting of five members, actively contributed to the repository by uploading and maintaining their respective code components. This enabled efficient collaboration, minimized integration conflicts, and ensured a smooth development process.

ESDBOT/ navigation: - It has our whole repository which includes the motor control functionality, URDF files related to the simulated rover, files related to IMU and GPS and GUI related files.

3 System (Performance) Specification - *{Nirzer Gajera (41638)}*

3.1 User Story

User Role:

As an engineer, we want to develop a mobile platform that can drive autonomously between waypoints, using Nav2 for navigation, SWIFT PIKSI for DGPS/INS-based localization, and a wireless GUI for manual control, so that it can operate both indoors and outdoors with real-time obstacle avoidance and fallback open-loop waypoint navigation.

3.1.1 Phase 1: Indoor Autonomous Navigation & Wireless Control

Scenario 1: System Initialization and Wireless GUI Setup

User Goals:

- The **Nav2 stack** is configured with **LiDAR-based SLAM for localization and path planning**.
- The **wireless GUI (built with PyQt6)** is connected over **Wi-Fi** to allow **robot control and monitoring**.
- The **SICK TiM781 LiDAR** is used for **real-time obstacle detection and mapping**.

Acceptance Criteria:

- The **robot initializes Nav2 and SLAM**, successfully generating an indoor map.
- The GUI displays the **real-time LiDAR map, robot status, and navigation controls**.
- The user can define **a start and end point** for autonomous movement.

Scenario 2: Indoor Autonomous Navigation without Nav2 & Realtime Obstacle Avoidance

User Goals:

- The user **selects a start and goal location** within the mapped indoor space in GUI.
- Path is followed by rover with open loop movement **without replanning**.
- The **TiM781 LiDAR actively detects obstacles** and stops the rover.

Acceptance Criteria:

- The robot follows the **path without collisions**.
- It **stops** when encountering an obstacle.
- The GUI **draws the robot's path.**

Scenario 3: Manual Control via Wireless GUI

User Goals:

- The user can **switch from autonomous to manual mode** using the GUI.
- The **directional buttons** on the GUI allow manual navigation.
- The user can override with an emergency stop at any time for safety reasons.

Acceptance Criteria:

- The robot responds **immediately** to manual control inputs.
- The user can **seamlessly switch between manual and autonomous modes**.

3.1.2 Phase 2: Indoor Autonomous Navigation & Wireless Control (with Nav2 and SLAM)

Scenario 4: Indoor Autonomous Navigation with Nav2 & Realtime Obstacle Avoidance

User Goals:

- The user selects a **start and goal location** within the indoor map using the **GUI**.
- The **robot autonomously calculates a path** using **SLAM (Simultaneous Localization and Mapping)** and **real-time LiDAR data** while utilizing Nav2.
- The system **avoids obstacles dynamically** using sensor feedback.

Acceptance Criteria:

- The robot follows the generated path without collisions.
- It **adjusts its route dynamically** when an obstacle is detected.
- The GUI **shows the robot's position, and path, and detected obstacles in real-time**.

3.1.3 Phase 2: Outdoor Navigation with Wireless Control

Scenario 5: Transitioning to Outdoor Environments

User Goals:

- The user **switches the navigation mode** to operate outdoors, where DGPS is available.
- The robot integrates DGPS **data with LiDAR-based localization**.
- The system adapts to **terrain variations and larger obstacle sizes**.

Acceptance Criteria:

- The robot correctly integrates D**GPS and LiDAR SLAM** for positioning.
- The GUI updates **position and mapping data dynamically** for outdoor conditions.
- The robot **maintains stable navigation even with DGPS fluctuations**.

3.2 Requirements Specification

The system requirements for the autonomous rover were defined to cover hardware, software, communication, and environmental factors.

Functional Requirements

- The rover shall navigate autonomously using sensor data for localization and obstacle avoidance.
- The rover shall allow manual override through a wireless controller.
- The rover shall generate and update a map using onboard sensors.
- The rover shall broadcast real-time sensor data (LiDAR scans, camera images, odometry) over ROS2 topics.
- The rover shall maintain a **TF tree** to correctly relate sensor and robot frames.
- The rover shall log all sensor data and control commands for offline analysis.

Performance Requirements

- Maximum speed: **1 m/s** on flat terrain.
- Mapping accuracy: **±10 cm** positional error after 10 m of travel.
- Obstacle detection range: **10 meters** using LiDAR.
- Minimum battery life: **60 minutes** under average load.

Hardware Requirements

- Drive system: Dual tracked chassis.
- Sensor suite: Front-facing stereo camera, front and rear LiDAR, DGPS/INS.
- Processing: Jetson AGX Orin with Ubuntu 20.04 and ROS2 Humble.
- Power: Rechargeable battery.

Software Requirements

- Operating system: Ubuntu 20.04.
- Middleware: ROS2 Humble.
- Sensor drivers: Official or custom-developed for full compatibility.
- Navigation: Nav2 stack (where feasible) with fallback to custom motion control.

Safety Requirements

- Emergency stop functionality through safety node monitoring proximity sensors.
- Automatic motor shutdown if a critical fault (overcurrent, encoder failure) is detected.

Communication Requirements

- Wireless telemetry for manual control and debugging.
- Real-time communication over DDS (ROS2 standard).

3.3 Constraints & Limitations

The project faced several constraints and inherent limitations due to hardware capabilities, software support, and environmental factors.

Hardware Constraints

- Uneven weight distribution of the rover poses big challenges for testing phases.
- Encoder resolution and slippage on tracks resulted in inaccurate odometry.
- Limited payload capacity on short platform-constrained sensor and battery options.
- Limited access to Network access for testing and remote development capabilities.

Software Constraints

- Lack of native ROS2 driver for RoboClaw required custom driver development.
- ROS2 packages for certain sensors (e.g., DGPS/INS) required adaptation from ROS1.
- Simulation challenges due to lack of direct support for tracked locomotion in URDF and Gazebo.

Operational Constraints

- Manual tuning was required for many system parameters (e.g., PID gains, TF tree offsets), requiring time-consuming field trials.
- Sensor synchronization (time stamping) across LiDAR, camera, and IMU was challenging due to non-uniform update rates.

Time and Team Limitations

- The project was conducted within the constraints of an **academic timeline**, requiring compromises between ideal design and practical implementation.
- Limited team experience with ROS2 and advanced sensor fusion techniques restricted the sophistication of algorithms used.

Conclusion

These constraints directly influenced design decisions, including the shift from full autonomy using **Nav2** to the **simplified motion control approach**, as well as the reliance on open-loop control due to unreliable odometry feedback. Understanding these limitations helps to **contextualize performance results** and **identify areas for future improvement**.

4 Background

4.1 State of the Art & Technology

This part of the project description talks about the advanced technologies used to build the robot's navigation system. ROS2 serves as the main software framework. A real-time LiDAR sensor gathers information about the environment, and a graphical interface built with PyQt6 shows that information to the user. The systems that create the map (SLAM) and plan the robot's path (Nav2) were tested and developed inside a simulation. Understanding how the real-time parts and the simulated parts work together is important to see how the navigation system works.

4.1.1 ROS2: A Next-Generation Middleware for Advanced Robotics *{yogachand pasupuleti (41588)}*

ROS2 represents a significant advancement in open-source robotics middleware, designed to address the limitations of its predecessor, ROS1, and meet the evolving demands of modern robotic applications. Built on a foundation of distributed computing, ROS2 facilitates the efficient development, deployment, and maintenance of complex robotic systems.

Key Innovations and Enhancements:

- **Real-Time Performance:** ROS2 introduces robust real-time capabilities, enabling its use in safety-critical applications like autonomous vehicles and industrial automation, where deterministic timing is paramount.
- **Enhanced Lifecycle Management:** A formalized lifecycle management system provides predictable node behaviour and simplifies debugging, significantly improving system reliability.

- **Middleware Abstraction:** By abstracting the middleware layer, ROS2 offers greater flexibility and adaptability, allowing developers to select the most suitable middleware for their specific deployment needs. This allows for increased portability across different hardware and software platforms.
- **Strengthened Security:** ROS2 incorporates comprehensive security features, including authentication and access control, to protect against unauthorized access and data manipulation, addressing critical security concerns in modern robotic systems.

ROS2: Core Architecture and Communication Model

ROS2 provides a powerful framework for developing complex robotic systems, built upon a node-based architecture. This architecture decomposes robotic systems into individual, specialized nodes that communicate and coordinate to achieve overall system functionality.

Key Features:

- **Node-Based Architecture:** ROS2 systems are composed of interconnected nodes, each responsible for a specific task. This modularity promotes code reusability and simplifies system development.
- **Publish-Subscribe Communication:** A core feature of ROS2 is it publish-subscribe communication model.
  Publishers: Nodes that broadcast messages containing data.
  Subscribers: Nodes that receive and process messages from publishers.

This event-driven paradigm facilitates efficient and flexible data exchange between nodes, forming the backbone of ROS2's communication architecture.

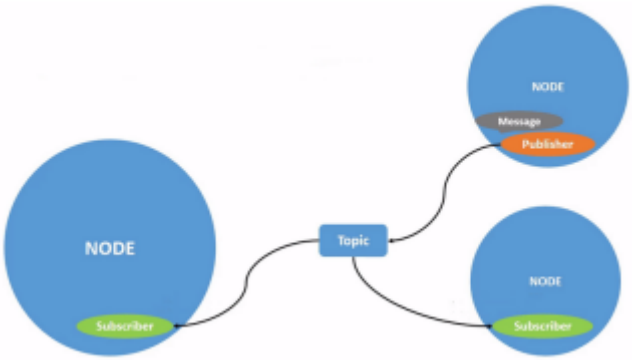

Figure 1. Topic-Based Data Exchange in a Node Network [10]

Service-Client Communication.

Service-Client communication facilitates request response interactions between nodes. This communication model allows nodes to provide and consume services, where the service server offers a particular functionality, and another node (the service client) can request and receive that functionality. Service-Client communication is essential for arranging actions, making real-time requests, and executing tasks that involve multiple steps.

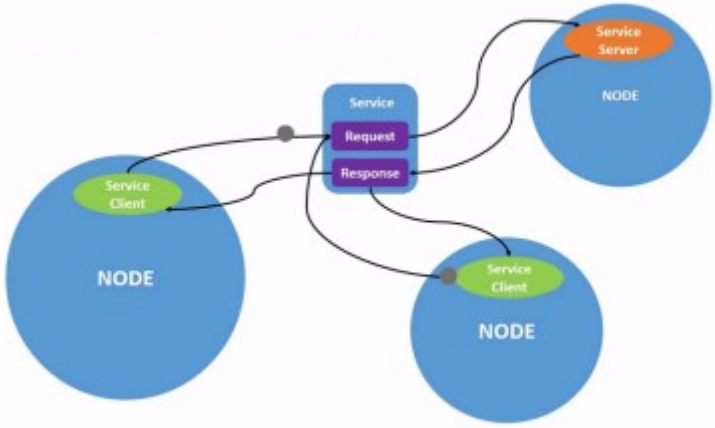

Figure 2. Client Request and Server Response [10]

4.1.2 PyQt6– A Powerful GUI Toolkit for Python- {*Nirzer Gajera (41638)*}

PyQt6 is a set of Python bindings for the **Qt6 application framework**, which allows developers to create cross-platform Graphic User Interfaces (GUIs). The Qt Company developed **Qt**6, a **powerful C++ framework** widely used for building professional desktops and embedded applications. PyQt6 makes these features accessible to Python developers, combining **Python's simplicity** with **Qt's advanced GUI capabilities**. [11]

Why Use PyQt6 for GUI Development?

PyQt6 provides an extensive **widget toolkit**, making it highly suitable for building modern, responsive, and feature-rich applications. Some key advantages include:

- **Cross-Platform Support** – Works on Windows, Linux, and macOS.
- **Rich Set of Widgets** – Includes buttons, labels, text boxes, tables, and custom graphics.
- **Event-driven programming** – Uses **signals and slots** to handle user interactions.
- **Customizable Styling** – Supports CSS-like **QSS styling** for UI customization.
- **Integration with ROS2** – Easily communicates with **DDS middleware** for robotics applications.

4.1.3 DDS- {*Nirzer Gajera (41638)*}

Data Distribution Service (DDS) is a **real-time, decentralized, and scalable** middleware protocol designed for **high-performance communication** in distributed systems. **ROS2 (Robot Operating System 2)** uses DDS as its underlying communication mechanism, allowing robotic applications to exchange messages efficiently. [10]

Why DDS in ROS2?

- **Decentralized Communication** – No single point of failure.
- **Low-Latency & Real-Time Support** – Critical for autonomous robots.
- **Flexible QoS Policies** – Customizable for different robotic applications.
- **Interoperability** – Supports multi-robot and cloud communication.

DDS follows a **publish-subscribe model**, meaning nodes do not directly communicate but instead use a shared data bus.

Core Concepts of DDS in ROS2

DDS Concept	ROS2 Equivalent	Functionality
Data Writers & Readers	Publishers & Subscribers	Nodes that send and receive messages
Topics	Topics	Channels for communication (e.g., /cmd_vel, /lidar_scan)
Domain Participants	ROS2 Nodes	Entities that participate in communication
QoS Policies	QoS in ROS2	Configurable settings that determine message delivery behaviour
Discovery	Automatic Discovery	Nodes automatically detect each other without a master

Table 5. DDS and ROS2 Equivalence

Unlike ROS1, which relied on a **centralized master**, ROS2's DDS-based architecture enables **peer-to-peer discovery**.

DDS Implementations Available in ROS2

ROS2 allows **multiple DDS vendors** through the **RMW (ROS Middleware) abstraction layer**.

DDS Implementation	Developed By	Performance	Best Use Case
Fast DDS (Default)	eProsima	◆ High	General ROS2 applications
Cyclone DDS	Eclipse Foundation	◆ Low latency	Embedded systems
Connext DDS	RTI	◆ Very high	Aerospace, safety-critical
OpenSplice DDS	ADLINK	◆ Industrial-grade	Medical, industrial robots
CoreDX DDS	Twin Oaks	◆ Lightweight	IoT, constrained environments

Table 6. ROS2 DDS Middleware Comparison [12]

DDS Configuration in ROS2

```
os.environ["ROS_DOMAIN_ID"] = "10"
os.environ["RMW_IMPLEMENTATION"] = "rmw_fastrtps_cpp"
```

4.1.4 QoS for ROS2- {*Nirzer Gajera (41638)*}

ROS 2 offers a wide variety of Quality of Service (QoS) policies that allow you to tune communication between nodes. With the right set of Quality-of-Service policies, ROS 2 can be as reliable as TCP or as best-effort as UDP, with many possible states in between. Unlike ROS 1, which primarily only supported TCP, ROS 2 benefits from the flexibility of the underlying DDS transport in environments with lossy wireless networks where a “best effort” policy would be more suitable, or in real-time computing systems where the right Quality of Service profile is needed to meet deadlines.

A set of QoS “policies” combine to form a QoS “profile”. Given the complexity of choosing the correct QoS policies for a given scenario, ROS 2 provides a set of predefined QoS profiles for common use cases (e.g. sensor data). At the same time, developers are given the flexibility to control specific policies of the QoS profiles.

QoS profiles can be specified for publishers, subscriptions, service servers, and clients. A QoS profile can be applied independently to each instance of the aforementioned entities, but if different profiles are used, they may be incompatible, preventing the delivery of messages. [13]

QoS Policy	Description	Possible Values
Reliability	Controls if messages are guaranteed to be delivered.	**Best Effort** – Fast, but messages may be lost. **Reliable** – Ensures message delivery, but may add latency.
Durability	Defines whether messages are stored for late subscribers.	**Volatile** – Messages are lost if no subscriber is active at publish time. **Transient Local** – Messages are stored for late subscribers.
History	Controls how many past messages are stored in memory.	**Keep Last (N)** – Stores only the most recent N messages. **Keep All** – Stores all messages until the memory limit.
Depth	Works with **History** to define the exact number of messages stored (only applicable when using KEEP_LAST).	**Integer Value (N) –** Number of messages to store.
Deadline	Defines the maximum allowed time for a message to arrive.	**Short Deadline** – Messages must arrive within a few ms. **Long Deadline** – Messages can arrive with longer delays.

Liveliness	Ensures a node is still active by requiring periodic updates.	**Automatic** – DDS manages liveliness (default). **Manual By Topic** – The publisher must manually assert liveliness.
Latency Budget	Defines acceptable delay before a message is considered late.	**Low Latency** – Messages must arrive quickly. **High Latency** – Messages can tolerate delays.
Lifespan	Controls how long a message remains valid before being discarded.	**Short Lifespan** – Messages expire quickly (e.g., 1s). **Long Lifespan** – Messages remain available for a long time (e.g., 1 hour).

Table 7. ROS2 QoS Policies

QoS Configuration for GUI

```
qos_profile = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
```

4.1.5 Lifecycle Nodes: Enhanced Robotic System Control *{yogachand pasupuleti (41588) & Shamanth B Adiga (41527)}*

ROS 2 introduced the concept of lifecycle nodes or managed nodes, a new type of node that can be managed through 4 states. Because lifecycle nodes that fail to launch can be reconfigured and reactivated together at runtime, they provide roboticists finer-grained control of their robotics systems. [14]

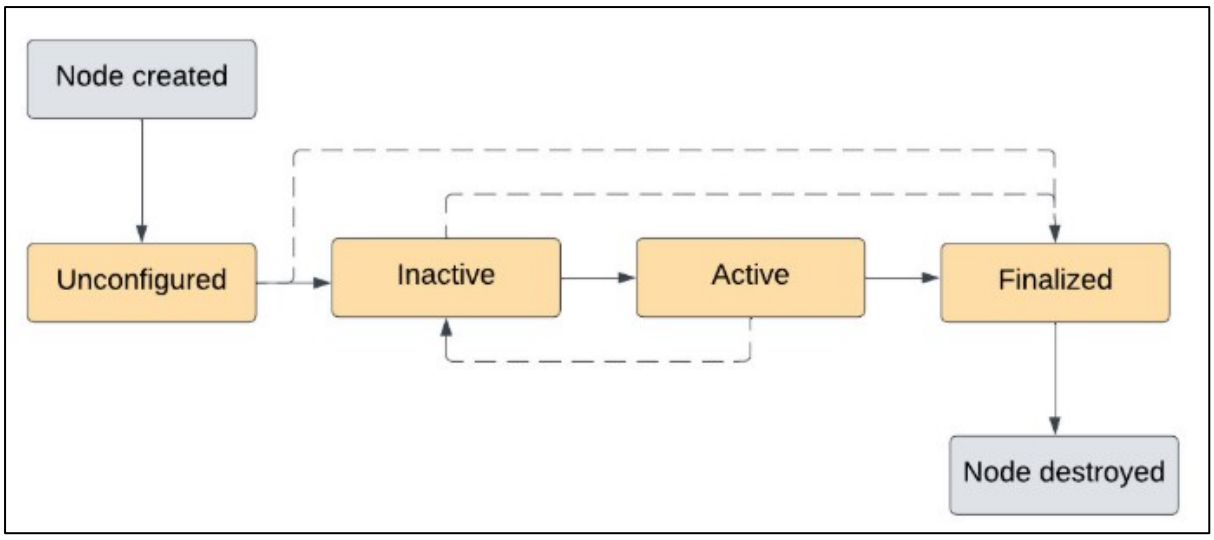

Figure 3. Lifecycle Node State Diagram [14]

NAV2 and SLAM, ROS 2's most famous navigation stack, use lifecycle nodes to manage transitions and ensure that the stack can successfully execute without missing any nodes. Nodes are configured and activated in a particular sequence to start up the stack, then brought down in reverse order to a finalized state. A failed state transition of one node blocks others from becoming active, and unresponsive servers trigger the shutdown of all nodes. This behavior has been achieved within a simulation environment, demonstrating the robust management of lifecycle nodes. The specific usage of lifecycle nodes within the Nav2 and SLAM implementations is shown in Sections 7.8 & 7.9.

4.1.6 Nav2_lifecycle_manager *{yogachand pasupuleti (41588) & Shamanth B Adiga (41527)}*

Nav2's lifecycle manager is used to change the states of the lifecycle nodes to achieve a controlled startup, shutdown, reset, pause, or resume of the navigation stack. The lifecycle manager presents a lifecycle_manager/manage_nodes service, from which clients can invoke the startup, shutdown, reset, pause, or resume functions. Based on this service request, the lifecycle manager calls the necessary lifecycle services in the lifecycle managed nodes. Currently, the RVIZ panel uses this lifecycle_manager/manage_nodes service when user presses the buttons on the RVIZ panel (e.g. Startup, reset, shutdown, etc.), but it is meant to be called on bringup through a production system application.

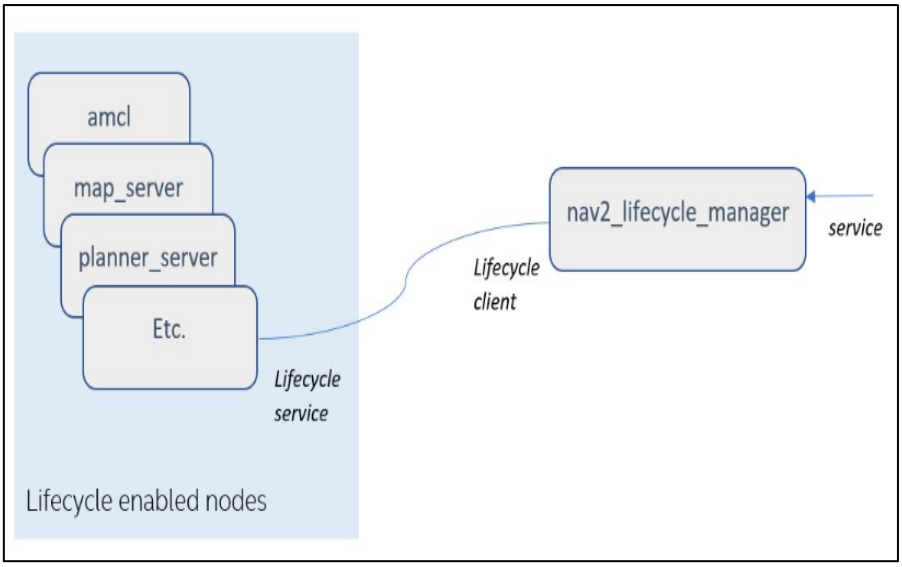

Figure 4. Lifecycle Manager Operations [15]

To start the navigation stack and be able to navigate, the necessary nodes must be configured and activated. Thus, for example when startup is requested from the lifecycle manager's manage_nodes service, the lifecycle managers calls configure() and activate() on the lifecycle enabled nodes in the node list. These are all transitioned in ordered groups for bringup transitions, and reverse ordered groups for shutdown transitions.

The lifecycle manager has a default nodes list for all the nodes that it manages. This list can be changed using the lifecycle manager's "node_names" parameter.

4.1.7 SLAM (Simultaneous Localization and mapping) *{yogachand pasupuleti (41588) & Shamanth B Adiga (41527)}* [16]

For fully autonomous deployed systems to operate in large and changing environments such as warehouses, factories etc they require tools that can be used to accurately map an area specified for their operation, update it over time, and scale to handle mapping of some of the largest indoor and outdoor spaces imaginable. The field of Simultaneous Localization and Mapping (SLAM) aims to solve this problem using a variety of sensor modalities, including laser scanners, radars, cameras, encoders, gps and IMUs. For localization and mapping in this project have used a Lidar to scan the surroundings and generate a map and the provider is SICK LIDAR. SLAM methods using laser scanners are generally considered the most robust in the SLAM field and can provide accurate positioning in the presence of dynamic obstacles and changing environments.

Previously existing open-source laser scanner SLAM algorithms available to users in the popular Robot Operating System (ROS) include GMapping, Karto, Cartographer, and Hector.

However, few of these can build accurate maps of large spaces on the scale of the average Walmart store. Even fewer can do so in real-time using the mobile processor typically found in mobile robot systems today. The only package that could accomplish the above was Cartographer. However, it was abandoned by Google and it is no longer maintained.

To rectify the above disadvantages a new fully open-source ROS package, SLAM Toolbox, to solve this problem. SLAM Toolbox builds on the legacy of Open Karto , the open-source library from SRI International, providing not only accurate mapping algorithms, but a variety of other tools and improvements. SLAM Toolbox provides multiple modes of mapping depending on need, synchronous and asynchronous, utilities such as kinematic map merging, a localization mode, multi-session mapping, improved graph optimization, substantially reduced compute time, and prototype lifelong and distributed mapping applications.

Slam_toolbox is a comprehensive and versatile open-source software package developed for use with the Robot Operating System (ROS), primarily designed for 2D Simultaneous Localization and Mapping (SLAM) applications.

SLAM is a fundamental concept in robotics that allows a robot to build a map of an unknown environment while simultaneously determining its position within that environment. The core challenge that SLAM solves is the "chicken-and-egg" problem—where a robot needs a map to localize itself, but it also needs to know its location to build the map accurately. Slam toolbox effectively addresses this issue through advanced algorithms and optimization techniques, making it ideal for real-time operations and offline map optimization.

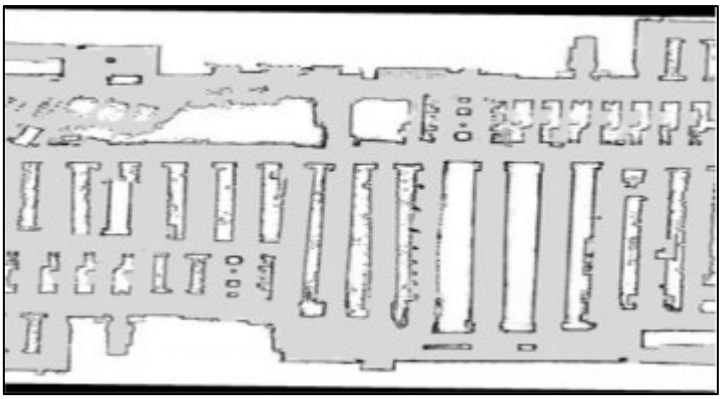

Figure 5. Retail store map created using SLAM Toolbox (Source: Macenski, 2019)

Working - SLAM_TOOLBOX [17]

The working usually happens in 3 steps:

1. Data Collection and Sensor Input
2. Mapping Process (Occupancy Grid Generation)
3. Localization Process

Data Collection and Sensor Input: The process begins with a robot equipped with sensors such as LiDAR (Light Detection and Ranging), depth cameras, or other range-finding sensors. These sensors collect data about the surrounding environment by measuring distances to obstacles and free space. This data typically comes in the form of laser scans or point clouds.

Mapping Process (Occupancy Grid Generation): During mapping mode, **Slam_toolbox** uses the sensor data to create an **occupancy grid map**—a 2D grid where each cell represents the likelihood of being occupied by an obstacle. The robot moves through the environment, continuously scanning and collecting data. As it gathers data, the toolbox performs:

- **Scan Matching:** Each incoming scan is compared to the previous scans to estimate the robot's movement between two points. Algorithms like Iterative Closest Point (ICP) or Correlative Scan Matching are used to align scans and refine position estimates.
- **Pose Graph Creation:** The robot's position at each time step (called a pose) is recorded as a node in a graph. Edges between these nodes represent the relative transformations (movements) between poses. This allows the system to track the robot's movement across time and space.
- **Loop Closure Detection:** When the robot revisits a previously mapped area, Slam_toolbox detects this and attempts to correct any accumulated drift by adjusting the pose graph. This ensures the map remains consistent and accurate.
- **Graph Optimization:** Once loop closures are detected, pose graph optimization techniques, such as Non-linear Least Squares Optimization (using solvers like Ceres), are applied to minimize errors across the graph, improving the accuracy of the overall map.

Localization Process: After generating the map, Slam_toolbox can switch to localization mode. In this phase, the system does not update or build the map but instead uses the existing map to determine the robot's current position. The localization process includes:

- Sensor Data Matching: Incoming laser scans from the robot's sensors are compared to the previously generated map to find the most likely position of the robot.
- Pose Estimation: The system continuously updates the robot's estimated pose (position and orientation) in real time, helping the robot understand its exact location within the environment.
- Dynamic Re-localization: If the robot loses track of its position (due to sensor noise or sudden displacement), Slam_toolbox offers global localization algorithms to help it recover by matching scans across the entire map.

4.1.8 NAV2 *{Yogachand pasupuleti (41588) & Shamanth B Adiga (41527)}*: [18]

GENRALISED NAV2 ARCHIETECTURE:

At the core of the architecture lies the Behavior Tree (BT)-based navigation structure, which governs the robot's behavior through a series of pre-defined and customizable behaviors that are executed based on environmental feedback and the robot's current state. This modular setup allows developers to define different navigation strategies depending on specific tasks or robot requirements.

Another vital part of the architecture is the Costmap System, which consists of two types of costmaps: the global costmap and the local costmap. The global costmap provides a comprehensive map of the environment for long-term planning, while the local costmap dynamically updates based on real-time sensor input to help the robot avoid newly detected obstacles. The Global Planner node takes responsibility for computing the shortest, safest path from the robot's starting position to the goal using algorithms such as A* or Dijkstra. On the other hand, the Local Planner (Controller Server) refines this global path into smaller, executable motion commands that take into account dynamic obstacles and real-time environmental changes.

The architecture also includes components like the Map Server, which provides static maps (usually generated through SLAM or other mapping algorithms), and the Localization Server that uses algorithms like Adaptive Monte Carlo Localization (AMCL) to continuously estimate the robot's position within the map. Additionally, the Recovery Server implements behaviors to recover from navigation failures, like moving backward or clearing the costmap when the robot is stuck.

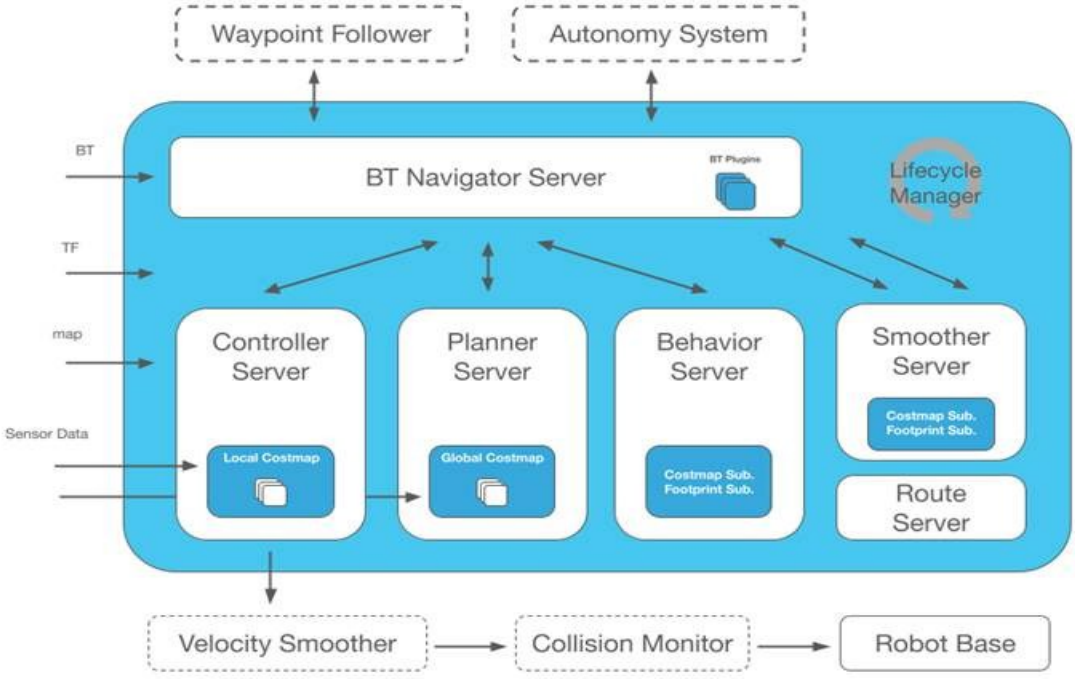

Figure 6. Generic NAV2 Architecture [19]

FEATURES:

The NAV2 stack comes with several advanced features that make it a robust and reliable solution for enabling autonomous navigation in mobile robots. One of its core features is its path planning capability, which allows the robot to navigate from its current location to a defined goal point in the environment. The global planner calculates an efficient route that avoids obstacles and optimizes factors like travel time and distance, while the local planner dynamically adjusts the route based on real-time feedback from onboard sensors such as LiDAR and cameras.

Another standout feature is obstacle avoidance. The NAV2 stack uses real-time costmaps to map out obstacles dynamically. These costmaps are continuously updated using sensor data, allowing the robot to adapt its movement and re-plan paths when new obstacles are detected. Additionally, NAV2 supports dynamic reconfiguration of parameters during runtime, enabling developers to tweak behavior parameters such as speed limits, obstacle inflation radius, and safety boundaries without restarting the system.

A notable architectural enhancement in ROS2 NAV2 is the incorporation of Behavior Trees (BTs) for decision-making. This approach allows for a more flexible and modular representation of robot behaviors, making it easier to implement complex navigation strategies and recover from failures. Behavior trees can be customized to handle specific tasks like docking, following, or stopping under particular conditions.

Localization and mapping are also integral features of the NAV2 stack. Using packages like AMCL or integrating with SLAM systems allows the robot to maintain an accurate position estimate within a given map. This ensures that the robot understands where it is always within its environment, which is critical for effective navigation.

Another essential feature is the lifecycle management system, which ensures systematic initialization, activation, and shutdown of all the navigation components, making the system more robust and fault tolerant. In addition to this, the stack supports multi-robot navigation—multiple robots can navigate collaboratively within a shared environment using coordinated path-planning and collision avoidance algorithms.

Working Of Different Nodes in NAV2:

AMCL:

The AMCL (Adaptive Monte Carlo Localization) node in the Nav2 stack is responsible for estimating a robot's position and orientation within a known map. It uses particle filters to probabilistically track the robot's pose by incorporating sensor data, typically from a LiDAR or camera, and comparing it with the pre-built map. Initially, the AMCL node generates a set of particles distributed across the map, each representing a potential robot pose.

As new sensor data is received, the particles are weighted based on how well they match the map's features, with particles that align better with the sensor data being given higher weight. Over time, the AMCL node resamples the particles, ensuring that the most likely poses are given priority, and continuously refines the robot's estimated position. This process helps the robot adapt to changes in the environment, providing an accurate and reliable localization solution in dynamic settings.

BT NAVIGATOR:

The BT Navigator in the Nav2 stack is responsible for executing high-level behaviours and missions in a robotic system using Behaviour Trees (BT). It provides a flexible and structured way to define the robot's decision-making process, allowing for complex tasks to be broken down into simpler, hierarchical behaviours. The BT Navigator interprets and executes a Behaviour Tree, where each node represents a specific action or condition, such as moving to a goal, avoiding obstacles, or performing a task. The node's state is evaluated based on the robot's current status, sensor data, and environmental conditions. The BT Navigator continuously monitors the progress of the active behaviour and transitions between tasks accordingly, ensuring that the robot behaves intelligently and efficiently. By using Behavior Trees, the BT Navigator provides better modularity and scalability in designing autonomous navigation tasks, while also enabling easy adjustments and enhancements to the robot's behavior.

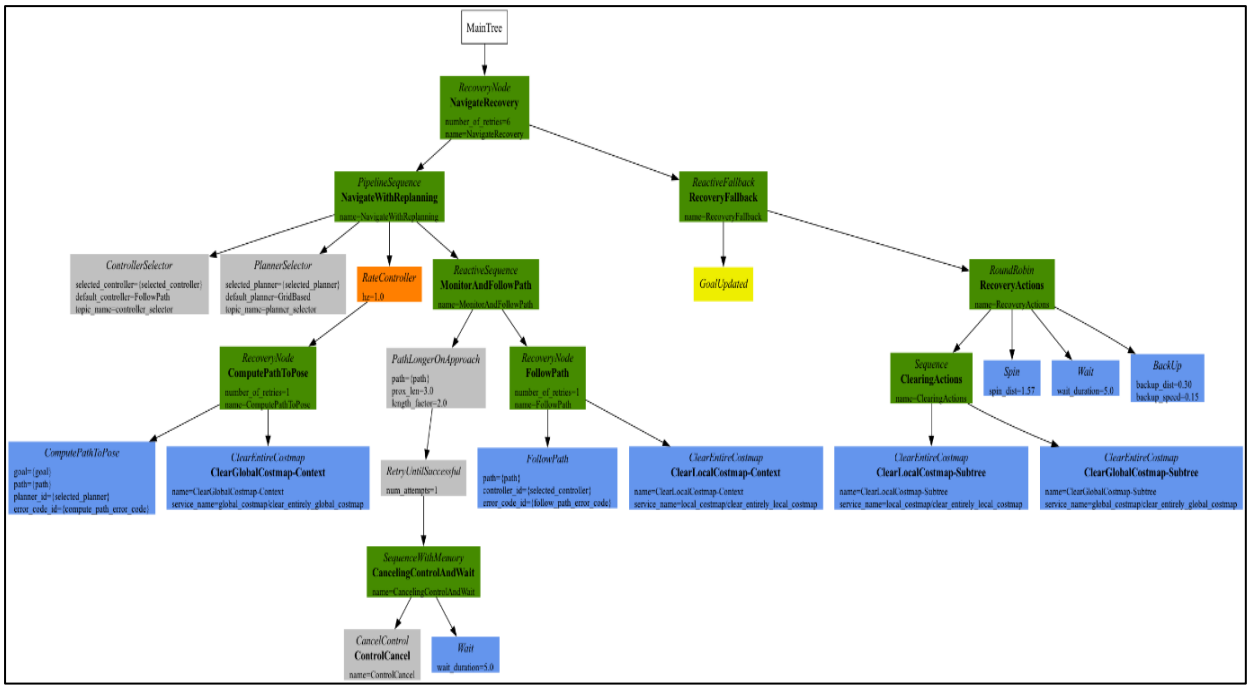

Figure 7. A flow chart demonstrating the working of BT Navigator [19]

PLANNER SERVER, CONTROLLER SERVER, BEHAVIOUR SERVER:

- **Planner Server**: The Planner Server is responsible for generating a global path for the robot to follow. It takes the robot's current position, the goal pose, and the map of the environment as input, and produces a path that avoids obstacles while reaching the target. The Planner Server uses algorithms like A* or Dijkstra to compute this global path. This server typically runs in the background and provides a safe, optimal path for the robot based on the static map and dynamic obstacles.
- **Controller Server**: The Controller Server is responsible for executing local control actions to follow the path generated by the Planner Server. While the Planner Server provides the global trajectory, the Controller Server computes the appropriate velocity commands to move the robot smoothly along this path. It takes into account the robot's current position, velocity, and dynamic obstacles, adjusting the robot's speed and direction in real-time to ensure safe and accurate path following. This is usually handled by algorithms like Dynamic Window Approach (DWA) or Elastic Band.
- **Behaviour Server**: The Behavior Server is responsible for orchestrating high-level decision-making and coordinating the overall task execution in the navigation system. It uses a Behavior Tree (BT) to structure the robot's actions based on the current state of the system, environmental conditions, and sensor data. The Behavior Server decides when to call upon the Planner Server or the Controller Server, ensuring that the robot reacts to dynamic changes in its environment, such as unexpected obstacles or changes in the goal. The Behavior Server enables modular, adaptable, and scalable decision-making for more complex navigation tasks.

4.1.9 Transform Frames *{yogachand pasupuleti(41588)}*

TF2, the Transform Library in ROS, provides a centralized and efficient method for managing coordinate frame transformations crucial for robotic systems. It organizes coordinate frames, such as a robot's base or sensor locations, into a hierarchical tree structure, where nodes represent frames and edges define the transformations between them. By storing these transformations and automatically calculating the relationships between any two frames within the tree, TF2 eliminates the need for manual and complex calculations. This system allows developers to define static or dynamic transformations, even those changing over time, and simply query TF2 for the required coordinate frame conversions. Consequently, TF2 simplifies code, enhances robustness, and facilitates seamless interoperability between different components of a robotic system, enabling developers to focus on higher-level tasks rather than the intricacies of coordinate frame management.

Static Transforms:

Static transforms in ROS define fixed spatial relationships between coordinate frames, representing the unchanging physical configuration of a robot. These transforms are fundamental for establishing the robot's geometric layout and are crucial for tasks like SLAM (Simultaneous Localization and Mapping), as further applied and discussed in section 7.5.1

Dynamic Transforms:

Dynamic transforms, on the other hand, represent relationships between coordinate frames that change over time, reflecting the robot's motion and position within its environment. These transforms are vital for tasks like navigation and real-time localization. Examples include the map to odom transform, which provides the robot's position in a global map, and the odom to base_link transform, which tracks the robot's position relative to its starting point using odometry data. Dynamic transforms are typically published by active sensor processing systems and localization algorithms, such as AMCL or the robot_localization package. These transforms provide real-time updates on the robot's pose, enabling it to navigate and interact with its surroundings effectively after the initial static map is created. as further applied and discussed in section 7.8

4.1.10 Functioning of LiDAR Technology *{yogachand pasupuleti (41588)}*

A 2D LiDAR (Light Detection and Ranging) sensor operates by emitting laser pulses and measuring the time it takes for these pulses to travel to an object and reflect back to the sensor, a process known as Time-of-Flight (ToF). The sensor typically uses a rotating mirror or prism to scan the laser beam in a horizontal plane, capturing distance measurements at multiple angles. By collecting these data points, a 2D representation of the surroundings is generated, often in the form of a point cloud. These sensors are widely used in robotics, automation, and security for obstacle detection, mapping, and navigation. However, 2D LiDAR lacks depth perception as it only scans in a single plane, making it more suitable for applications requiring surface monitoring and 2D mapping, rather than full 3D object recognition. [4]

Overview and Technical Description of SICK TIM781 LiDAR is discussed in **Section (4.3.2)**.

4.1.11 Principle of Operation (LiDAR): - *{Yogachand Pasupuleti (41588)}*, - *{Nirzer Gajera (41638)}*

Measurement principle:

1. **Distance Measurement: -**

The TiM 2D LiDAR sensors scan their environment using the state-of-the-art optical time-of-flight measurement method, the time-of-flight (TOF) technology. This involves sending out laser pulses via a rotating mirror and detecting their reflection. The longer it takes for a laser pulse to return to the sensor, the further away the detected object is. This combined with the strength of the incoming signal allows the position of objects in space to be determined with millimetre accuracy. The TiM updates this representation of its surroundings up to 15 times per second, thereby allowing real-time orientation, navigation, and control. The pulse data of the laser sensors are processed using HDDM+ technology.

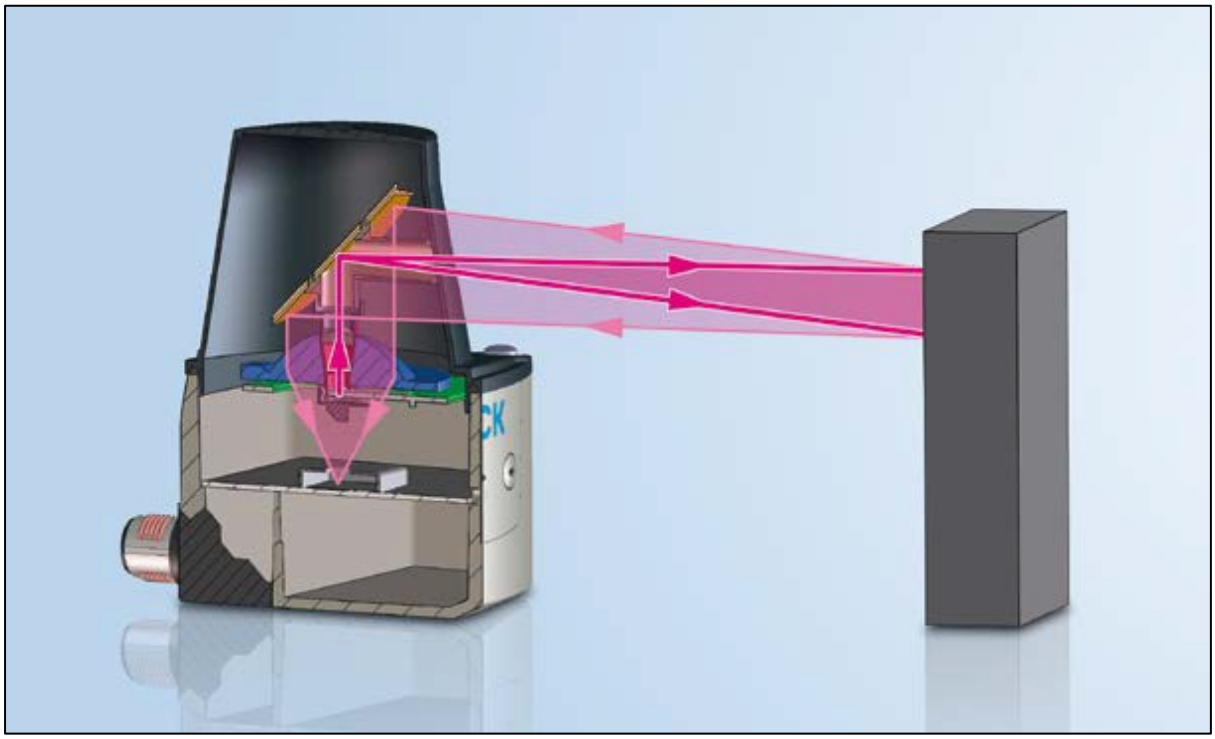

Figure 8. TIM781 Laser Functioning Diagram [4]

The High-Definition Distance Measurement Plus (HDDM+) is an advanced signal processing technology developed by SICK. It enhances measurement accuracy by reducing noise and improving object detection reliability. The HDDM+ method processes edge hits particularly well, which is advantageous for localization and anti-collision applications in a changing environment. Unlike traditional ToF measurements that rely on a single pulse reflection, HDDM+ captures multiple laser pulses for each measurement point and applies statistical averaging to reduce errors.

The device emits beams pulsed by a laser diode. If a laser pulse hits an object or person, it is reflected on the surface of the object or person in question. The reflection is registered by a photosensitive element in the device receiver. With this measurement process, a measured value is formed by adding together multiple single pulses. The device calculates the distance from the object based on the elapsed time that the light requires between emitting the beam and receiving the

reflection. Radar systems apply this “pulse time-of-flight measurement” principle similarly. [4]

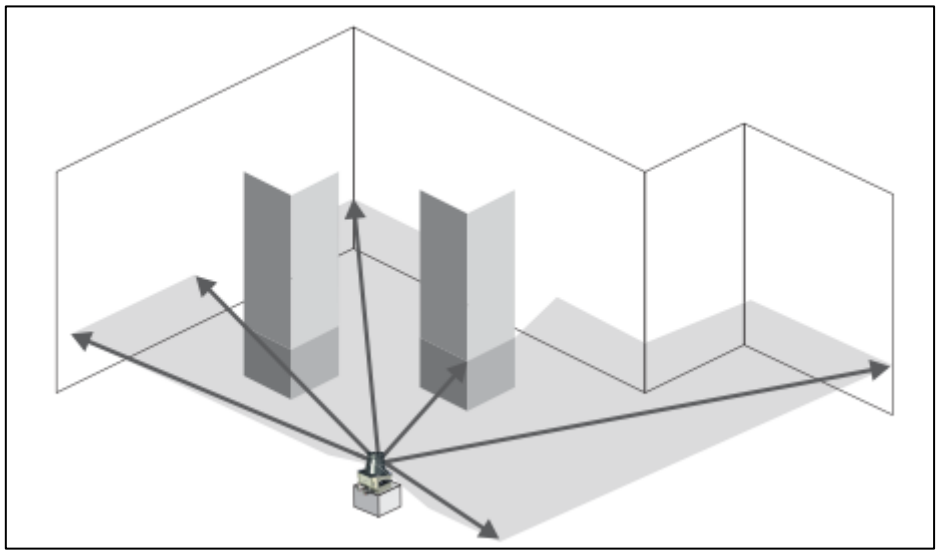

Figure 9. TIM781 scanning environment [4]

Direction Measurement: -

The device uses a rotating mirror to deflect the emitted laser beams, thereby scanning its surroundings in a circular pattern. The measurements are triggered internally by an encoder in regular angle increments. The measuring procedure uses the averaging from multiple pulses to determine individual measured values. A measuring point is the average value of several measurements combined.

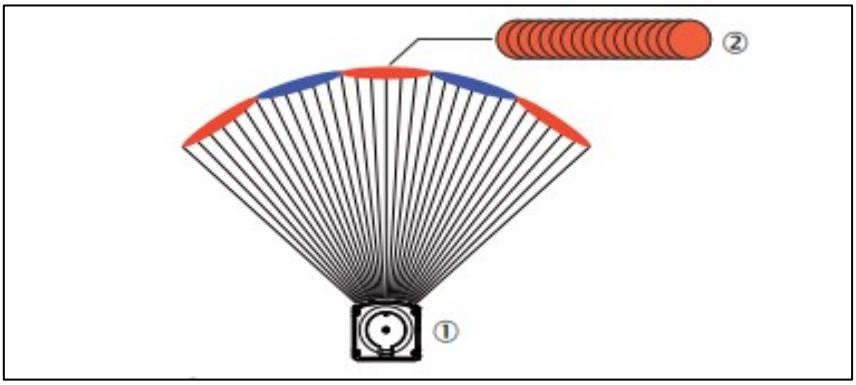

Figure 10. TIM781 angle measurement [4]

The angular resolution of 0.33° ensures that the TiM781 can detect fine variations in object position, making it highly reliable for applications requiring precise directional measurements. The encoder-based system continuously updates these angular values, ensuring consistent tracking of moving and stationary objects. Additionally, this precise angular measurement allows the sensor to differentiate between multiple objects within its scanning range. [4]

4.1.12 Swift Navigation: GNSS and RTK Technology – *{Niranjan Channayya (41559)}*

Swift Navigation has emerged as a key player in the field of high-precision satellite-based positioning systems, offering cutting-edge solutions that enhance localization accuracy for autonomous systems. The Piksi Multi GNSS receiver, one of Swift Navigation's flagship products, incorporates Real-Time Kinematic (RTK) technology, enabling centimetre-level positioning accuracy, a significant improvement over traditional Global Navigation Satellite System (GNSS) solutions that typically provide accuracy within a few meters. This advancement is critical for robotics, autonomous vehicles, precision agriculture, and industrial automation, where precise positioning plays a vital role in navigation and operational efficiency.

RTK-based navigation relies on a differential correction technique that uses a stationary base station to provide real-time error corrections to a moving rover unit, such as the one employed in this project. By leveraging corrections from the base station, the rover unit can mitigate common sources of GNSS error, including ionospheric delays, satellite clock drifts, and multipath effects. Unlike conventional GPS receivers, which estimate position solely based on satellite signals, RTK employs carrier phase measurements, significantly refining position estimates and reducing drift over time.

A distinguishing feature of the Swift Piksi Multi GNSS receiver is its multi-frequency and multi-constellation support. Traditional GNSS receivers are often limited to a single frequency and a specific satellite system, leading to increased susceptibility to atmospheric disturbances and satellite outages. The Piksi Multi receiver, however, supports multiple satellite constellations, including GPS, GLONASS, BeiDou, and Galileo, thereby improving availability, robustness, and accuracy even in challenging environments where satellite visibility is obstructed. This redundancy ensures continuous operation in urban areas, dense forests, or partially obstructed landscapes, where conventional GPS signals may degrade.

Another critical component of the Swift Navigation system is its integration with Inertial Navigation Systems (INS). The Bosch BMI160 IMU (Inertial Measurement Unit), embedded in the Piksi Multi, enables dead reckoning capabilities by providing real-time acceleration and gyroscope data. This sensor fusion approach is particularly useful in scenarios where GNSS signals are weak or temporarily lost, such as inside tunnels or areas with significant electromagnetic interference.

By combining IMU data with GNSS corrections, the system maintains accurate localization, reducing reliance on satellite signals alone.

From a technological perspective, Swift Navigation has optimized the RTK-based GNSS system for real-time applications, ensuring low-latency communication through protocols such as the Real-Time Streaming Protocol (RTSP) and the

Networked Transport of RTCM via Internet Protocol (NTRIP). The integration of these protocols allows the GNSS receiver to receive correction data from remote base stations over the internet, eliminating the need for a dedicated physical base station in some deployments. Additionally, the Swift Navigation ROS2 driver, implemented in this project, facilitates seamless communication between the GNSS receiver and other robotic components, enabling real-time localization updates within the ROS2 framework.

Despite its advantages, RTK-based GNSS navigation presents certain challenges. The primary limitation is its dependence on a base station or NTRIP service to provide real-time corrections. Without these corrections, the GNSS receiver operates in a standalone mode with reduced accuracy, typically within a meter-level range. Furthermore, multipath errors—caused by satellite signals reflecting off nearby buildings or objects—can introduce inaccuracies, particularly in urban environments. To mitigate such issues, the system employs error filtering and sensor fusion techniques, where GNSS data is combined with odometry and LiDAR-based SLAM algorithms to enhance localization robustness.

As GNSS technology continues to evolve, new developments such as Precise Point Positioning (PPP) and 5G-assisted localization hold the potential to further improve accuracy and reliability. Unlike RTK, which requires a nearby base station, PPP relies on precise satellite orbit and clock corrections, allowing high accuracy positioning even in remote areas. Additionally, advancements in multi-sensor fusion, leveraging GNSS, IMU, LiDAR, and vision-based localization, are shaping the future of autonomous navigation, reducing the reliance on any single sensor modality.

In this project, the Swift Piksi Multi GNSS receiver, integrated within the ROS2 environment, provides a robust and high-precision localization solution, facilitating accurate waypoint tracking and autonomous path planning. Its compatibility with Nav2 and SLAM can enhance the overall system reliability, ensuring the rover's ability to navigate structured and unstructured terrains with high positional confidence. The incorporation of RTK technology, multi-constellation GNSS, and IMU-based dead reckoning represents a significant step forward in achieving precise and robust autonomous navigation capabilities.

4.1.13 Principle of Distance calculation using Bosch BMI160 IMU unit – *{Niranjan Channayya (41559)}*

The IMU unit provides us with raw acceleration values in x, y and z-axis. Raw form here means in Least Significant Bit (LSB) form, which is the smallest unit of digital information the sensor can output. This needs to be converted to meters per second square (m/s^2) for further calculation of distance in meters.

Converting raw accelerations to m/s^2

Converting the raw accelerations to m/s^2 involves understanding the sensor sensitivity and scale factor. In our case, the Bosch BMI160, the acceleration mode varies based on selected range of gravitation(g) –

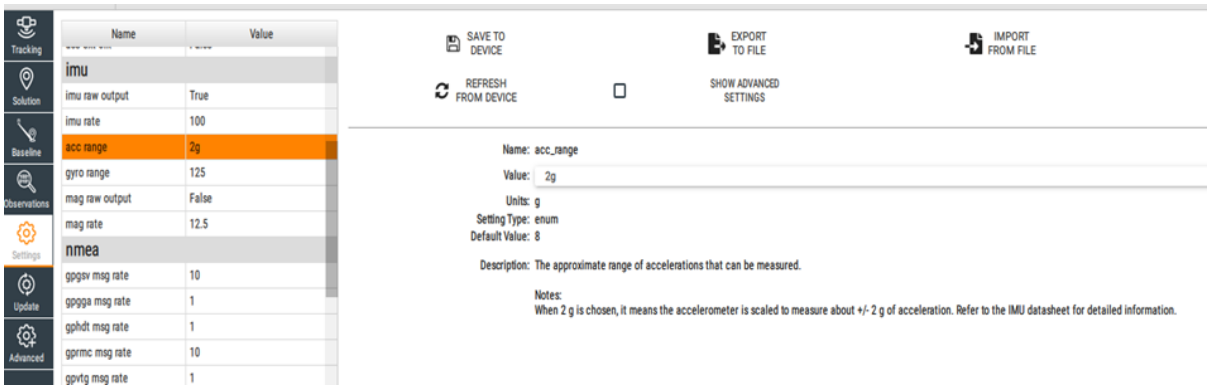

Figure 11. Selection of range of acceleration measurement in Piksi Console

- ±2g: Sensitivity is approximately 16384 LSB/g
- ±4g: Sensitivity is approximately 8192 LSB/g
- ±8g: Sensitivity is approximately 4096 LSB/g
- ±16g: Sensitivity is approximately 2048 LSB/g

In our case, we have ±2g sensitivity, which is set using Piksi console. This effectively means that we can measure acceleration in the range of -2g to +2g. Here, g indicates gravitational constant which is 9.81 m/s^2. Therefore, we can measure acceleration in range of -19.62 m/s^2 to +19.62 m/s^2.

Converting the sensor readings from LSB to m/s^2 involves 2 steps:

1. Converting the raw integer value to acceleration in g using the sensitivity-

$$Acceleration\ (g) = \frac{Raw\ Reading}{Sensitivity}$$

2. Since 1g= 9.81 m/s2, multiply the previous result by it-

$$Acceleration\ = \left(\frac{Raw\ Reading}{Sensitivity}\right) * 9.81$$

Calculation of Distance from Acceleration

Now, we get the acceleration value from the sensor as per the rate (in Hz) we set. For instance, if we set the rate as 100 Hz, we get acceleration values 100 times per

second. Next, we will have to process this data to get distance travelled / displacement. For this we will have to integrate acceleration twice over time.

1. **<u>Acceleration → Velocity (First Integration)</u>**
   Here we integrate acceleration to get velocity by using the formula –

$$v(t) = v(t-1) + a(t) \times \Delta t$$

   Where:
   - *v(t)*: velocity at time t (in m/s)
   - *a(t):* acceleration at time t (in m/s2)
   - Δt: time between 2 samples (in sec)

2. **<u>Velocity → Distance (Second Integration)</u>**
   Now, we integrate the obtained velocity from previous step into distance by using the formula –

$$d(t) = d(t-1) + v(t) \times \Delta t$$

   Where:
   d(t): distance at time t
   v(t): velocity at time t

   we can use these two formulas in a python code to calculate the distance travelled by the rover in all the three axis – x, y & z.

   But using only IMU data to calculate distance travelled leads to inaccuracies as a consequence of Noise and Drift. Because of these small errors accumulate rapidly.

4.2 Software Tools

Software tools are essential programs and applications that streamline the development, testing, and deployment of robotic systems. They provide specific functionalities, from code compilation and debugging to simulation and data visualization. In robotics, these tools enable developers to manage complex systems, analyse sensor data, and ensure reliable robot behavior. They range from operating system frameworks like ROS 2, to build systems like CMake, and visualization platforms like RViz2, each playing a vital role in the robotics development lifecycle.

4.2.1 ROS2

Overview of ROS2 is discussed in **Section (4.1.1)**.

4.2.2 Visual Studio {*Yogachand pasupuleti (41588) & Shamanth B Adiga (41527)}*

Visual Studio Code (VS Code) is a free, open-source, and highly versatile code editor developed by Microsoft. It provides a streamlined environment for coding, debugging, and managing software projects across various programming languages and platforms. Extensibility, through a rich ecosystem of extensions, makes VS Code highly adaptable to diverse development workflows, including robotics development with ROS 2.

VS Code Usage in ROS 2 Project:

In ROS 2 project, VS Code served as the central editor for all coding and configuration tasks:

Python Package Development:

- VS Code provided a clean and efficient environment for writing and editing Python code used in building ROS 2 packages. Features like syntax highlighting, code completion, and debugging tools enhanced the development process.

CMake Integration:

- VS Code facilitated editing and managing CMakeLists.txt files, essential for building the ROS 2 workspace. This allowed for efficient configuration of build processes and dependency management.

Configuration File Editing:

- VS Code was used to edit XML and YAML configuration files, crucial for configuring robot parameters, launch files, and other ROS 2 settings. This ensured accurate and consistent configuration throughout the project.

ROS 2 Workspace Management:

- Beyond coding, VS Code simplified ROS 2 workspace management, allowing for easy navigation of project files, editing launch files, and integration with Git for version control.

4.2.3 RViz2 *{yogachand pasupuleti (41588) & Shamanth B Adiga (41527)}*:

RViz (ROS Visualization) is a powerful 3D visualization tool essential to the ROS framework. It visualizes robot sensor data, state, and environment in real-time for debugging, development, and testing. RViz visualized SLAM and NAV2

functionalities, displaying robot position, navigation paths, and maps. RViz displayed costmaps, localization data (AMCL), and enabled interactive goal setting. RViz visualized LiDAR data for SLAM validation and sensor integration. RViz provided real-time diagnostics, including trajectories, obstacles, and navigation feedback, crucial for autonomous robot development.

4.2.4 Gazebo *{yogachand pasupuleti (41588) & Shamanth B Adiga (41527)}*:

Gazebo Classic is a powerful open-source robotics simulator used to test and validate our robot's navigation and SLAM capabilities within a virtual replica of our laboratory room. Its high-fidelity physics and support for diverse sensors, including LiDAR through specialized plugins, provided crucial, real-time data for mapping and localization within this simulated environment. These LiDAR plugins enabled accurate visualization of sensor data, simulating real-world LiDAR behavior. Gazebo's visualization tools enabled real-time monitoring, facilitating debugging and refinement of SLAM algorithms and Nav2 configurations. By simulating the specific conditions of our laboratory, Gazebo helped validate our robot's path planning and obstacle avoidance. Its seamless ROS integration allowed for efficient testing of Nav2 functionalities, proving essential for verifying our robot's autonomous behavior before real-world deployment.

4.2.5 SOPAS ET Software (for LiDAR Configuration)- {*Nirzer Gajera (41638)*}

SOPAS ET (Engineering Tool) is SICK's proprietary software used for configuring, visualizing, and monitoring SICK sensors, including LiDAR. It provides an intuitive graphical interface that allows users to set up, adjust parameters, and diagnose sensor performance.

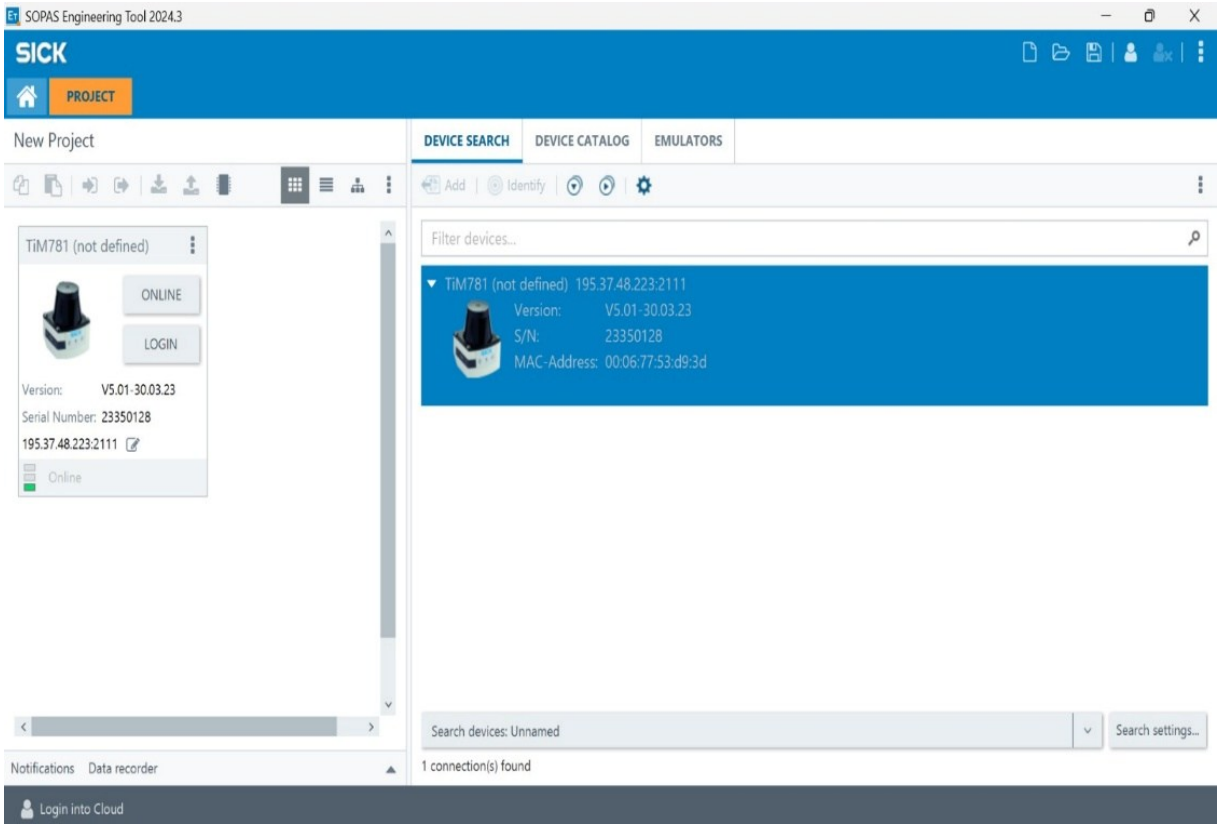

Figure 12. Device Management in SOPAS ET

Device Search and Connection Management

The above image shows the device search interface of SOPAS ET:

- Device Found: The TiM781 sensor is identified, showing:
  - Version: V5.01-30.03.23.
  - Serial Number (S/N): 23350128.
  - MAC Address: 00:06:77:53:D9:3D.
  - IP Address & Port: 195.37.48.223:2111.
- Connection Status: The device is online and synchronized.
- Options to Login & Configure: Users can log in to access full configuration settings. [20]

This screen is essential for device discovery, troubleshooting, and ensuring a stable connection.

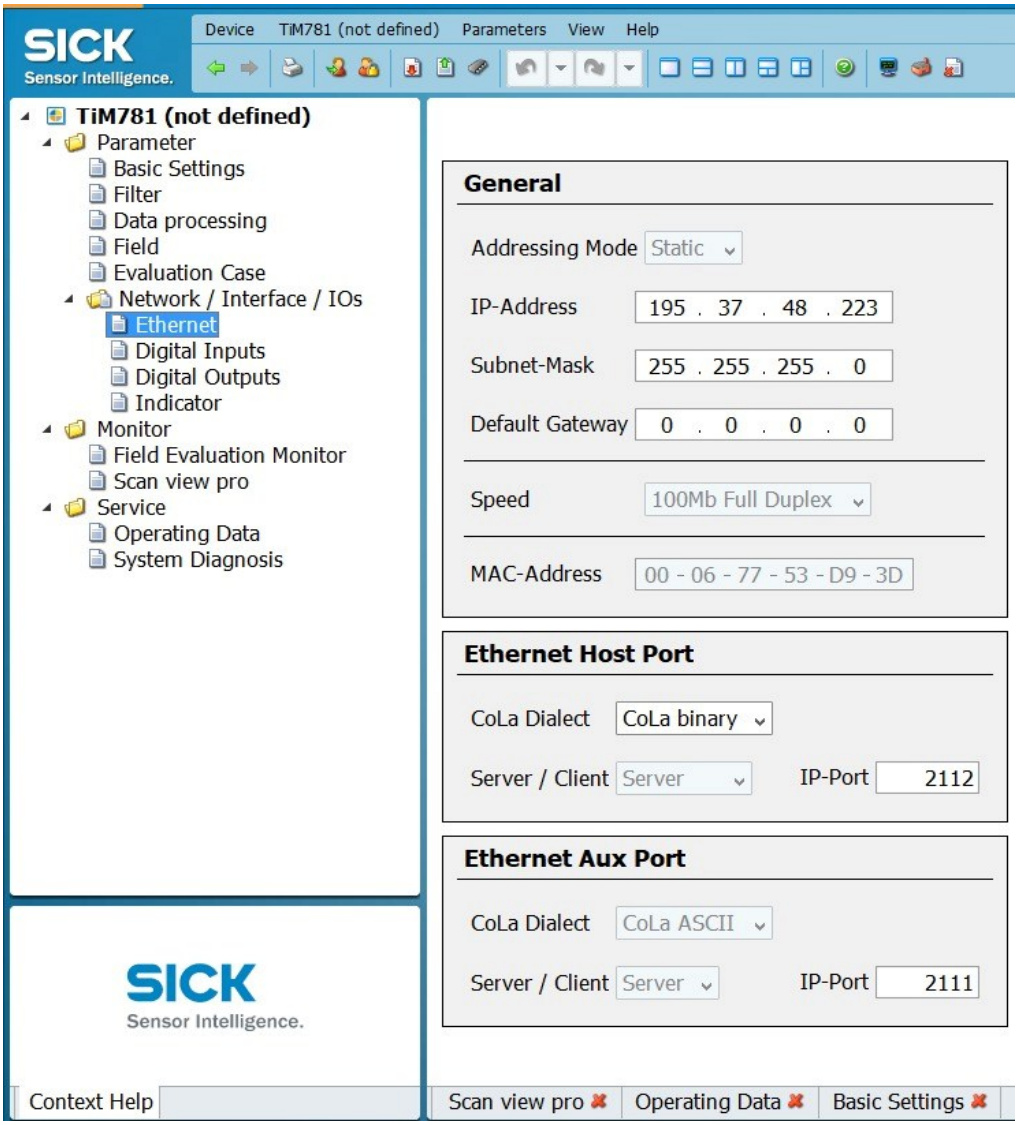

Figure 13. Device connection and network configuration

In the picture above, one can configure the Ethernet settings of the TiM781:

- Addressing Mode: Set to Static, meaning the sensor has a fixed IP address.
- IP Address: 195.37.48.223 – The assigned network address for communication.
- Subnet Mask & Default Gateway: These are used to ensure proper network communication.
- Speed: Set to 100Mb Full Duplex, which optimizes data transmission.
- Ethernet Host & Aux Ports:
    - CoLa Dialect: Defines the communication protocol (CoLa binary or ASCII).
    - IP Ports: 2112 (Host) and 2111 (Aux) for data exchange. [20]

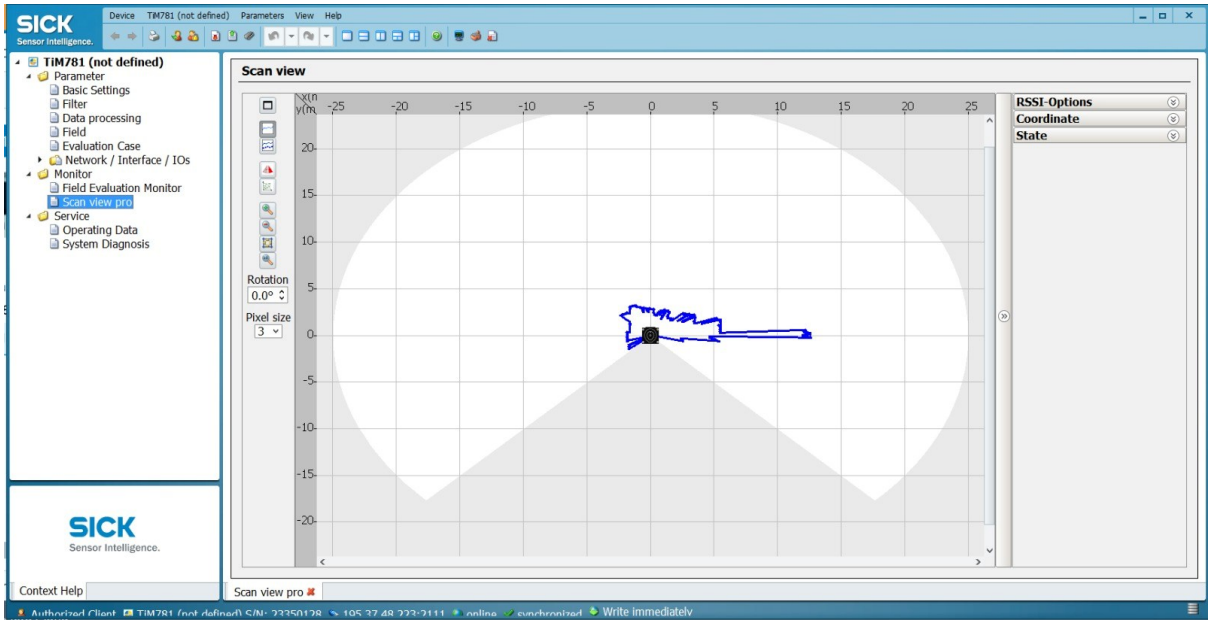

Figure 14. Real-Time Scan Visualisation

In the picture above, you are using the Scan View Pro tool in SOPAS ET:

- The white semi-circular region represents the 270° scanning field of the TiM781.
- The blue line represents real-time detected objects or surfaces.
- Pixel Size & Rotation Controls allow users to adjust the visual representation of the scan data. [20]

4.3 Description of System and Components

4.3.1 Nvidia Jetson AGX ORIN Dev. Kit- *{Nirzer Gajera (41638)}*

Figure 15. Nvidia Jetson ORIN Development Kit [3]

The NVIDIA Jetson Orin Development Kit is a powerful AI and robotics platform designed for edge computing, making it the best choice for our autonomous navigation project. It delivers up to 275 TOPS of AI performance with an NVIDIA Ampere GPU featuring 2048 CUDA cores and 64 Tensor cores, paired with a 12-core Arm Cortex-A78AE CPU for efficient processing. The kit includes 32GB or 64GB LPDDR5 RAM, expandable storage via an M.2 NVMe SSD slot and a microSD card slot, and offers extensive connectivity options like 10Gb Ethernet, PCIe Gen 4, USB 3.2, HDMI, and MIPI CSI for camera integration. Running on the JetPack SDK with support for CUDA, TensorRT, and DeepStream, it provides the ideal platform for AI-driven applications such as robotics, autonomous systems, smart cities, and medical imaging, ensuring robust performance for our navigation system. [3]

4.3.2 SICK TiM-781 2D LIDAR- *{Nirzer Gajera (41638)}*

The **SICK TiM781 LiDAR** is a high-performance 2D LiDAR sensor designed for precise environment perception in industrial and autonomous applications. It features **270° scanning coverage** with a range of up to **90 meters**, making it ideal for mapping, navigation, and collision avoidance. With its **high-resolution scanning technology and multi-echo capability**, the TiM781 ensures accurate detection even in challenging environments with dust, rain, or varying light conditions. Its compact and lightweight design allows for easy integration into mobile robots, AGVs, and industrial automation systems. [4]

Figure 16. SICK TIM781 LIDAR [4]

Equipped with **Ethernet connectivity** and support for various communication protocols, the TiM781 enables seamless data transmission and remote configuration. The sensor's **low power consumption and IP67-rated enclosure** ensure reliable operation in demanding environments. Its advanced filtering algorithms and adjustable field settings allow for **customized detection zones**, enhancing flexibility for different applications. Whether for autonomous navigation, security monitoring, or industrial safety, the **SICK TiM781 LiDAR** delivers precise, real-time 2D perception for robust and efficient system performance. [4]

Specification	Details
Scanning Angle	270°
Detection Range	Up to 90 meters
Measurement Accuracy	±3 cm (depending on conditions)
Angular Resolution	0.33° to 1° (configurable)
Scan Frequency	15 Hz
Connectivity	Ethernet (TCP/IP, UDP)
Output Data Format	Ethernet-based protocols, including SOPAS and ROS support
Power Supply	9V - 28V DC
Power Consumption	Approx. 4.6 W
Protection Rating	IP67 (dust and water-resistant)
Operating Temperature	-25°C to +50°C
Dimensions	60 mm x 60 mm x 86 mm
Weight	Approx. 250 g

Table 8. SICK TIM781 LIDAR Technical Description

4.3.3 Swift Navigation Piksi Multi – *{Niranjan Channayya (41559)}*

The Swift Piksi Multi is a high-precision **GNSS (Global Navigation Satellite System)** receiver designed for applications requiring centimetre-level accuracy. Piksi Multi supports multiple satellite constellations like GPS, GLONASS, BeiDou, Galileo, and others, enhancing positioning accuracy and reliability. Swift Piksi has two main types of positioning systems-

1. Piksi Multi – Standalone Position
2. Piksi Multi - GNSS RTK Position with Stationary Base

Standalone Positioning uses a single GNSS receiver without RTK corrections from a base station. In this mode, the Swift Piksi GNSS receiver relies solely on satellite signals to estimate the rover's position, typically achieving accuracy in the 1–3 meter range.

Real-Time Kinematic positioning technology provides location solutions that are up to 100 times more accurate than traditional GPS, with centimeter-level precision. This is critical for applications like autonomous vehicles, precision agriculture, drones, and robotics where exact positioning is key. [5]

Piksi Multi OEM board also includes a Bosch BMI160 IMU and Bosch BMM150 Magnetometer sensors. As of firmware release v1.0.11, it is possible to stream raw IMU values from the BMI160. The BMI160 is an ultra-small, low-power, 16-bit inertial measurement unit designed by Bosch Sensortec. It integrates both a triaxial accelerometer and a triaxial gyroscope into one package.

4.3.4 Router- {*Nirzer Gajera (41638)*}

The Teltonika RUT956 is an industrial-grade cellular router designed for robust and reliable connectivity in demanding environments. It features dual SIM slots with automatic failover, 4G LTE connectivity, and multiple WAN/LAN ports, making it ideal for remote monitoring, IoT applications, and mobile networking solutions. With GPS tracking capabilities, an RS485 interface for industrial communication, and multiple I/O ports, the RUT956 ensures seamless integration with SCADA, telemetry, and automation systems. Its rugged design and wide power input range (9-30V) make it highly suitable for vehicle, industrial, and outdoor deployments, while its web-based configuration interface simplifies network setup and management.

Figure 17. Router

4.3.5 Stereolabs ZED-X Camera– {*Krish. Shah* (41564)}

The **ZED X** camera, developed by **Stereolabs**, is a high-performance **stereo vision camera** designed specifically for **robotics, autonomous vehicles, and industrial automation**. It builds upon the legacy of the earlier ZED camera series but is enhanced for **robotic perception** under challenging real-world conditions. Unlike traditional monocular cameras, the **ZED X** leverages **stereo vision** technology to estimate depth by analysing the disparity between the images captured by its two horizontally offset cameras. [21]

- **Principle of Operation**

The core working principle of the **ZED X** camera is **stereo triangulation**, where two images taken from slightly different perspectives (left and right cameras) are compared to calculate **depth information**. Each pixel in the left image is matched to the corresponding pixel in the right image, and the **shift (disparity)** between them determines how far away the object is. This process, known as **stereo matching**, enables the camera to generate a **dense depth map.** The calculated depth map is then used to reconstruct a **3D model of the environment** in real-time.

In addition to stereo imaging, the **ZED X** includes an integrated **IMU (Inertial Measurement Unit)**, which provides real-time acceleration and angular velocity data. By fusing IMU data with visual information, the camera can improve its **visual odometry** performance, making it more robust to temporary feature loss. [21] This combination of **visual features** and **inertial sensing** is often referred to as **visual inertial odometry (VIO)**, a technique widely used in modern mobile robotics.

Figure 18. Zed-X Camera

- **Technical Features**

The **ZED X** camera has several key features designed for **robotics applications**:

- **Resolution:** Supports up to 2208 x 1242 pixels per camera.
- **Depth Range:** Effective depth sensing from **0.3 meters to 20 meters**.
- **Field of View:** Wide **120° horizontal field of view** for broad environmental perception.
- **IMU Integration:** Built-in 9-axis IMU for improved motion tracking.

- **IP66 Rating:** Dustproof and waterproof, ideal for outdoor autonomous systems.
- **Synchronization:** Multi-camera synchronization over hardware triggers, allowing multiple ZED X cameras to work together.

Interface: Native **ROS2 support** via the **official ZED ROS2 wrapper**, simplifying integration into robotic frameworks.

➢ **Applications in the Autonomous Rover Project**

In the autonomous rover project, the **ZED X** camera was intended to serve as a **multi-purpose perception sensor**, fulfilling three critical roles:

1. **Depth Perception:** By continuously generating real-time depth maps, the camera provides a comprehensive **3D understanding of the environment**, helping the rover detect obstacles and understand terrain geometry.
2. **Visual Odometry:** The **ZED X** was also responsible for estimating the rover's **relative movement** over time by tracking visual features between consecutive frames. This feature tracking allowed the rover to compute **its position and orientation** (pose) without relying solely on wheel encoders or GPS.
3. **Localization Support:** The visual odometry data from the ZED X was intended to complement data from other sensors (such as LiDAR, IMU, and GPS) within a **sensor fusion framework**, improving the accuracy and robustness of the rover's localization. [21, 22]

➢ **Advantages of ZED X in Robotics**

The **ZED X** offered several advantages that made it well-suited to outdoor mobile robotics applications:

- **Passive sensing:** Unlike LiDAR, which emits laser pulses, the ZED X passively observes the environment using natural light. This makes it energy-efficient.
- **Rich Perception:** It provides both **RGB imagery** (for visual scene understanding) and **depth data** (for spatial reasoning), which allows multi-modal sensor fusion.
- **Robust to Environmental Conditions:** Its IP66-rated enclosure makes it suitable for **dusty, wet, and outdoor environments**.
- **ROS2 Integration:** The availability of the official **ZED ROS2 Wrapper** ensured rapid integration into the rover's ROS2 ecosystem.

➢ ***Limitations and Challenges***

Despite its capabilities, the **ZED X** also introduced certain limitations and challenges during the project:

- **Featureless Environments:** Visual odometry depends on detecting **distinct features** in the environment. In feature-poor areas (e.g., plain walls, flat fields), tracking becomes unreliable [22].
- **Computational Load:** Processing stereo images and generating dense depth maps is **computationally expensive**, placing a significant burden on the **Jetson Nano** computing platform.
- **Low-Light Sensitivity:** Although stereo matching works in low light to some extent, the overall accuracy reduces as lighting deteriorates.

➢ **Relevance to Modern Robotics**

Stereo cameras like the ZED X are becoming increasingly popular in **autonomous robots** due to their ability to provide **visual, depth, and motion data** from a single device. When fused with other sensors like LiDAR, IMU, and GPS, they contribute to building a **robust multi-sensor localization and mapping system** [23].

Conclusion

The **ZED X camera** represented a modern, multi-functional perception sensor for the autonomous rover, allowing it to perceive its surroundings, estimate its own motion, and aid in localization. Its capabilities made it a valuable tool for sensor fusion, despite the **computational challenges and limitations in low-texture environments**.

4.3.6 RoboClaw Motor Controller

➢ **Introduction**

The **RoboClaw 2x60A** is an advanced **dual-channel motor controller** developed by **BasicMicro**. It is designed for **precise control of brushed DC motors**, offering support for **PWM speed control, quadrature encoders, closed-loop PID control, and multiple communication interfaces**. This makes it a widely used motor controller in robotics, automation, and autonomous vehicle projects (BasicMicro, n.d.).

The **2x60A** model specifically supports **two independent motors**, each capable of handling up to **60A of continuous current**, making it suitable for **high-power mobile robots** like the autonomous rover.

- **Working Principle**

The **RoboClaw 2x60A** operates as an **H-Bridge motor driver** with additional features for **encoder-based closed-loop control**. The core working principles involve:

- **PWM (Pulse Width Modulation) Motor Control**: Varying the duty cycle of the PWM signal controls the motor speed.
- **Quadrature Encoder Feedback**: The motor encoders provide real-time position and velocity feedback, allowing the controller to **self-correct errors**.
- **PID Speed and Position Control**: The onboard PID controller adjusts motor speed based on encoder feedback, maintaining precise movement [24].
- **Current Sensing and Protection**: Built-in overcurrent, thermal, and voltage protection safeguards the motors and controller from damage.

- **Communication Interfaces**

One of the key advantages of the RoboClaw is its **multiple communication interfaces**, making it highly flexible:

- **USB Communication**: Direct PC communication via **Motion Studio** for configuration and tuning.
- **Serial UART**: Standard serial communication for microcontroller or embedded system control.
- **RC (Radio Control) Mode**: Allows direct control using an RC receiver, useful for manual testing.
- **PWM Mode**: Direct pulse width signal control.
- **I²C Mode**: Supports communication with I²C-enabled devices (though less commonly used).
- **Encoder Feedback via ROS2 Integration**: The motor controller natively supports **encoder feedback**, which can be processed in **ROS2-based robotic systems** [25] [24]

- **Applications in the Autonomous Rover Project**

In this project, the **RoboClaw 2x60A** was used for:

1. **Motor Speed Control**: Receiving **velocity commands** from ROS2 (cmd_vel) and executing motor movements accordingly.
2. **Encoder-Based Odometry**: Reading quadrature encoder data to estimate the rover's movement.

3. **Current Monitoring**: Ensuring that the motors operated within safe limits.

The motor controller was **configured via Motion Studio** before integration into ROS2.

[25]

➢ **Advantages**

- **High Current Capacity:** Supports **60A per motor**, suitable for heavy-duty applications.
- **PID Control for Precision:** Enables **closed-loop speed control**, reducing drift and improving accuracy.
- **Multiple Interfaces:** Allows connection via USB, UART, and PWM, ensuring compatibility with various robotic platforms.
- **Built-in Safety Mechanisms:** Features **overcurrent, voltage, and temperature protection**, ensuring motor longevity.
- **Quadrature Encoder Support:** Provides **real-time feedback**, useful for precise movement tracking in robotics.

➢ **Limitations and Challenges**

Despite its benefits, the **RoboClaw 2x60A** presented certain challenges:

- **Not natively supported in ROS2:** The lack of an official ROS2 driver required the development of **a custom ROS2 node** to communicate with the controller.
- **Serial Communication Instability:** ROS2 serial communication occasionally encountered **packet loss** and **timing mismatches**, requiring debugging.
- **Encoder Drift in Low-Speed Movement:** The encoder readings were less reliable at lower speeds, leading to **odometry inaccuracies**.
- **Complex Configuration in Motion Studio:** Setting **PID values and encoder parameters** required extensive tuning and testing.

Conclusion

The **RoboClaw 2x60A** was an essential component of the autonomous rover, allowing for **precise motor control and encoder-based odometry feedback**. Despite challenges with ROS2 integration and encoder drift, it provided a **high-power, reliable motor-driving solution** with essential **safety features** and **real-time monitoring**.

4.3.7 Rover Chassis by Ulrich Robotics

- **Introduction**

The **tracked wheel chassis** used in the autonomous rover was developed by **Ulrich Robotics**. This type of chassis is designed for all-terrain mobility, providing superior **traction and stability** compared to traditional wheeled platforms. Tracks distribute weight over a larger surface area, making them particularly well-suited for **uneven, soft, or rugged surfaces**, which was a crucial consideration for the rover's intended outdoor operations [26].

- **Design and Structure**

The tracked chassis consists of:

- **Two independent motor-driven tracks** — one on each side of the rover.
- **Drive sprockets and idler wheels** supporting the continuous tracks.
- **Suspension system** allowing the tracks to conform to irregular terrain.
- **Mounting platforms** for attaching sensors, electronics, and the payload.

The entire platform was constructed from **high-strength aluminium** [26], balancing **durability and weight** to support heavy payloads like the battery, sensors, and onboard PC.

- **Working Principle**

The chassis operates on the **differential drive principle**, where the speed and direction of the left and right tracks are independently controlled. By varying the relative speeds of the two tracks, the rover can:

- **Move forward/backward** (both tracks move at the same speed).
- **Turn left/right** (tracks move at different speeds or in opposite directions).

This type of locomotion provides a **tight turning radius**, even enabling the rover to turn in place (tank steering), which is essential for navigating confined or obstacle-rich environments [27].

- **Applications in the Autonomous Rover Project**

The Ulrich Robotics tracked chassis was chosen specifically for:

- **All-terrain mobility** required for outdoor field testing.
- **High payload capacity** to support the extensive sensor suite.
- **High manoeuvrability** for navigating confined environments.

- **Compatibility with the RoboClaw motor controller**, which directly drove the left and right tracks.

➢ **Advantages**

- **High Traction:** Tracks offer better grip than wheels, particularly on soft, loose, or uneven terrain.
- **Load Distribution:** Tracks spread the vehicle's weight over a larger surface area, reducing ground pressure.
- **Obstacle Handling:** Tracks can climb over small obstacles and adapt to varying terrain heights.
- **High Payload Capability:** The robust aluminium structure supports a fully loaded sensor and power system.

➢ **Limitations and Challenges**

- **High Friction and Power Consumption:** Continuous contact with the ground increases rolling resistance, requiring more power.
- **Track Wear and Maintenance:** Tracks experience more wear than wheels and need regular inspection.
- **URDF Simulation Challenges:** Gazebo/URDF does not natively support continuous tracks, requiring creative workarounds (replacing tracks with equivalent large wheels).
- **Slippage in Odometry:** Since tracks do not roll like wheels, encoder-based odometry is **less reliable**, especially on loose surfaces.

Conclusion

The **Ulrich Robotics tracked chassis** provided the autonomous rover with exceptional **terrain adaptability, payload capacity, and manoeuvrability**, making it ideal for outdoor testing and real-world navigation tasks. However, these benefits came at the cost of **higher power consumption, increased maintenance, and complex simulation modelling**, requiring customized approaches to both control and simulation.

5 Architecture & Design – {*Krish. Shah* (41564)}

5.1 System / Hardware Architecture – {*Krish. Shah* (41564)}

The autonomous rover is designed with a modular hardware architecture to ensure flexibility, scalability, and robustness in operation. It is structured to integrate multiple sensing, processing, and control units that work in tandem to achieve autonomous navigation and obstacle avoidance.

5.1.1 Core Components

Computing Unit

- NVIDIA Jetson running ROS2 Humble, serving as the central processing hub for sensor fusion, navigation algorithms, and motor control [28]
- Utilizes Python and C++ for implementing ROS2 nodes, ensuring compatibility with the rover's sensors and actuators [10]

Motor Controller: [29]

- RoboClaw 2x60A motor controller, responsible for managing the differential drive system.
- Provides precise control over motor speed and direction based on sensor feedback.
- Implements PID-based velocity control for smoother movement and responsiveness.

Locomotion System: [30]

- Dual caterpillar tracks powered by independent high-torque motors, ensuring maximum traction and manoeuvrability.
- Supports movement in rough terrain and uneven surfaces for robust outdoor navigation.
- Track width: 0.5 meters, designed for stability in different environmental conditions.

5.1.2 Sensor Suite

LiDAR System (Sick TIM781) [4]

- Front LiDAR sensors enable accurate mapping and obstacle detection.
- Provides a 2D point cloud representation of the environment, essential for SLAM (Simultaneous Localization and Mapping).
- Works in real-time to update the environment map dynamically.

Camera System (ZED X Stereo Camera): [21] [21]

- Offers high-resolution depth perception for visual odometry and obstacle recognition.
- Additional Support for navigation using the Visual Odometry (VIO)
- Captures RGB and depth data to support visual SLAM and terrain classification.

GPS & INS (Swift DGPS/INS): [5]

- Provides high-accuracy global positioning for outdoor navigation.
- Integrates inertial measurements for drift correction and precise localization.
- Enables real-time georeferencing of the rover's position, improving path planning accuracy.

5.1.3 Communication Interfaces

Motor Communication: [31]

- Serial-based control interface for seamless interaction with the RoboClaw motor controller.
- Allows real-time feedback for encoder data, motor currents, and system diagnostics.

Sensor Integration:

- USB and CSI interfaces for camera modules, ensuring high-speed data transfer.
- Ethernet/Wi-Fi for communication with remote monitoring systems.
- ROS2 topics and messages used for data transmission between different hardware components.

System Scalability and Upgradability:

- Modular architecture allows easy integration of additional sensors and computing resources.
- Future enhancements include terrain adaptation, and improved autonomy.
- Designed to support multi-sensor fusion for advanced navigation capabilities.

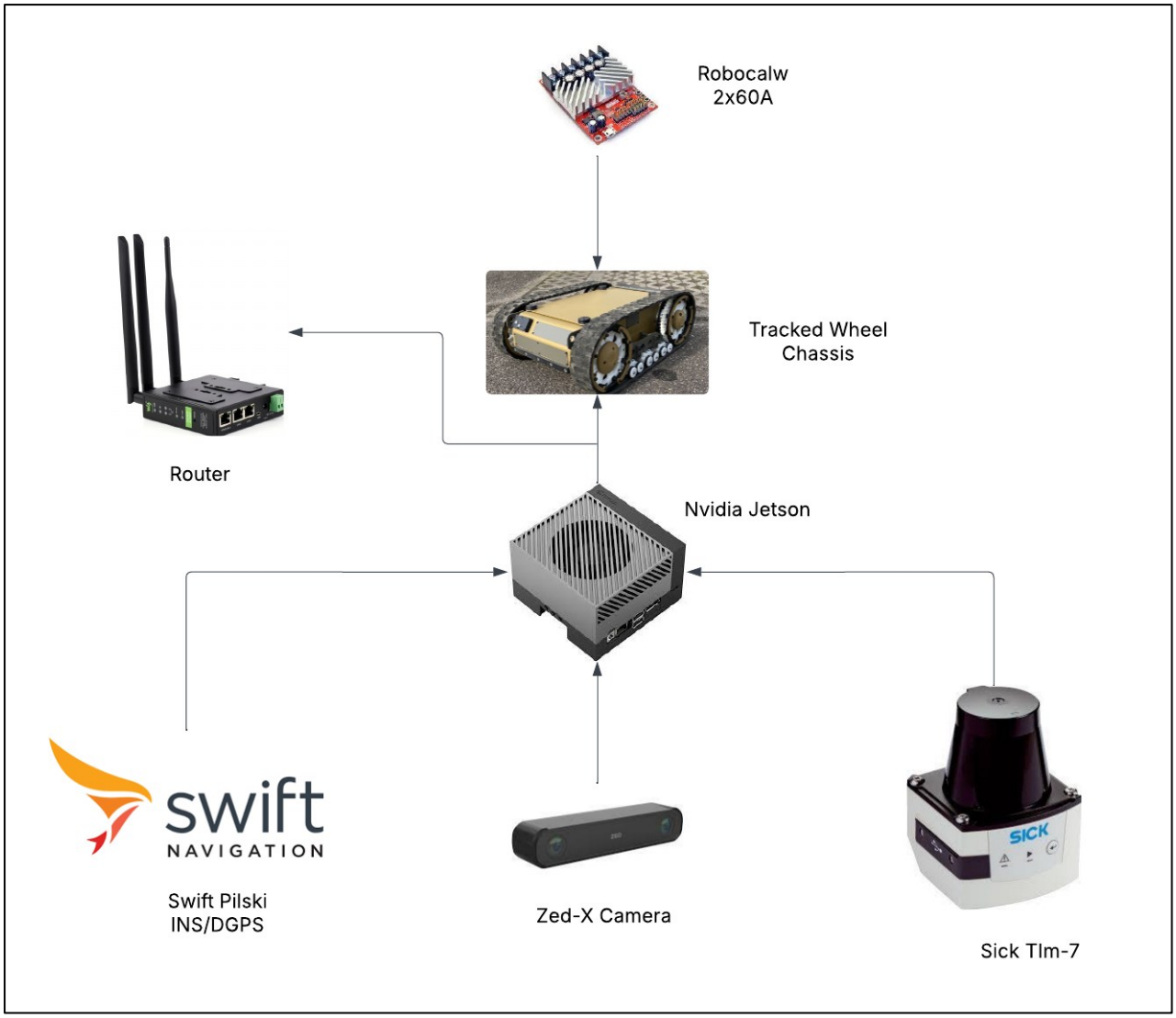

Figure 19. Hardware Overview [29]

5.2 List of Components – {*Krish. Shah* (41564)}

Below is the comprehensive list of components used in the autonomous rover:

No.	Component Category	Description
1	**Computing Unit**	NVIDIA Jetson Orin AGX
		MicroSD Card (128GB) for OS and storage
		Power Adapter (5V/4A)
2	**Motor System**	RoboClaw 2x60A Motor Controller
		High-Torque DC Motors (x2)
		Caterpillar Tracks (x2)
		Power Cables and Connectors
3	**Sensor Suite**	Sick TIM781 LiDAR (x2 - front and rear)
		ZED X Stereo Camera System
		Swift DGPS/INS
		IMU Sensor (Integrated with ZED X or Swift INS)
4	**Power System**	24V Lithium-ion Battery Pack
		Power Distribution Cabels
		Emergency Stop & Warning Light
5	**Communication & Connectivity**	Ethernet and Wi-Fi Module
		Serial and I2C cables
		DDS Service Communication
6	**Chassis & Mounting**	Aluminium Frame
		Custom Mounts for Sensors

Table 9. List of Components

Each component has been selected to optimize performance, reliability, and scalability for autonomous navigation and robotic applications.

5.3 Software Architecture – {*Krish. Shah* (41564)}

5.3.1 ROS2 Package Overview

The software is divided into multiple **ROS2 packages**, each responsible for a specific subsystem:

Package Name	Description
motor_control	Implements the **simple motion control solution** using a **custom driver** for the RoboClaw motor controller. Contains ROS2 nodes for sending commands and reading encoder data.
my_robot_description	Contains the **URDF and Xacro files** for defining the robot model, including sensors, base links, and joint configurations.
my_slam_config	Holds configuration files for **SLAM Toolbox**, enabling 2D mapping of the environment using LiDAR data.
my_tf_broadcaster	Implements **TF (Transform) broadcasting**, defining relationships between coordinate frames such as base_link, odom, and sensor frames.
nav2_config	Stores configuration files for the **Nav2 stack**, including cost maps, planner parameters, and controller configurations (though Nav2 implementation was unsuccessful).
roboclaw_hardware_interface	Handles low-level **hardware communication** with the RoboClaw motor controller, providing an interface for motor commands.
roboclaw_serial	Implements **serial communication** between Jetson and the RoboClaw motor controller using /dev/ttyACM0.
ros2_roboclaw_driver	A ROS2 driver for RoboClaw, used to interface with the motor controller, read encoders, and send movement commands.
rover_bringup	Contains **launch files** for bringing up the entire rover system, including sensors, motor nodes, etc..
rover_motor_bringup	A separate package dedicated to **bringing up only the motor control system**, ensuring motor drivers

	are initialized properly before other nodes.
sick_scan_xd	Handles the **Sick TIM781 LiDAR sensor**, providing ROS2 nodes for publishing LiDAR point cloud data.
zed-ros2-interfaces	Interfaces for the **ZED X Stereo Camera**, defining custom messages for depth perception and vision-based localization.
zed-ros2-wrapper-master	Wrapper package for **ZED ROS2 integration**, allowing the rover to use **stereo vision and depth mapping** within ROS2.

Table 10. ROS2 Packages

5.3.2 Architectural Design in ROS2 for Rover

The software architecture is structured in layers:

- **Hardware Abstraction Layer**: Contains low-level communication interfaces for the motor controller and sensors.
- **Perception Layer**: Handles LiDAR and camera processing, providing environment perception data.
- **Navigation and Motion Control Layer**: Implements either **full navigation (Nav2) or direct motion control (motor_control package)**.
- **System Bringup and Configuration Layer**: Defines launch files, TF broadcasting, and startup procedures.

This modular approach ensures that individual subsystems can be tested and debugged independently, facilitating easier improvements and integration. The software architecture also provides the ability to switch between different levels of autonomy, from **basic motor control** to **high-level navigation (when functional)**.

This structured software architecture supports **both control strategies**, with a clear separation of **autonomous navigation (Nav2) and direct motor control (custom driver)**. The flexibility of the system allows for adaptability in different operational conditions and debugging scenarios.

This structured software architecture supports **both control strategies**, with a clear separation of **autonomous navigation (Nav2) and direct motor control (custom driver)**. The flexibility of the system allows for adaptability in different operational conditions and debugging scenarios.

The software architecture of the autonomous rover is divided into two approaches: **Autonomous Navigation using Nav2** and **Basic Time-based Motion Control using and a Custom Motor Driver**. The selection between these depends on the use case and level of autonomy required. [32]

1. Autonomous Navigation using Nav2 [19]

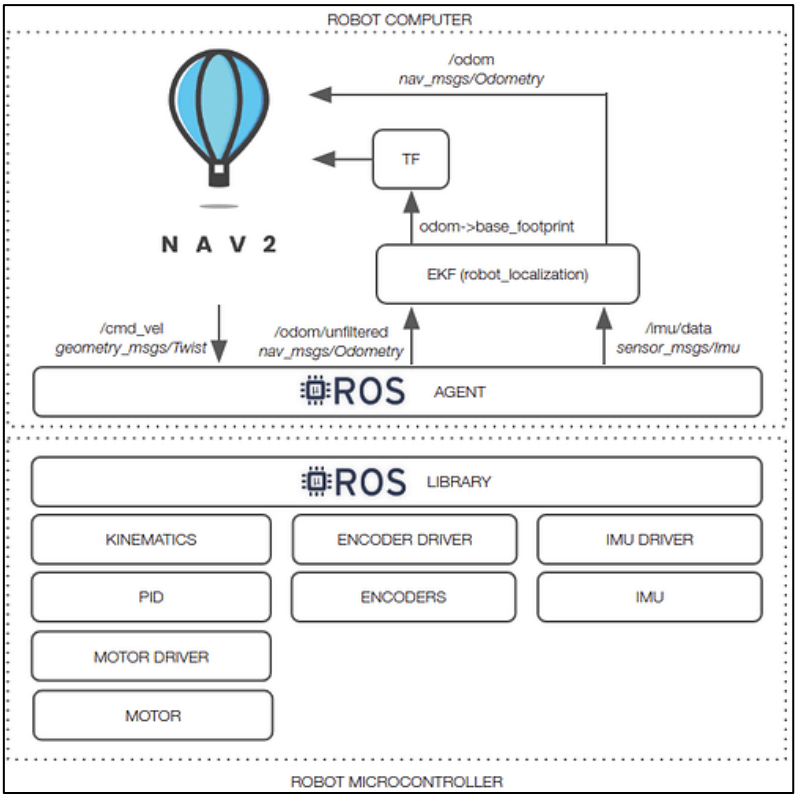

Figure 20. General NAV2 Architecture used with Mobile Robotics

- **Overview**: Uses the ROS2 Navigation Stack (Nav2) to enable autonomous path planning and obstacle avoidance.
- **Components**:
    1. **Localization**: Utilizes LiDAR, GPS, and IMU to determine the rover's position.
    2. **Mapping**: Generates a 2D cost map using LiDAR and occupancy grid algorithms.
    3. **Path Planning**: Implements global (A*) and local (DWA, TEB) planners to find and execute paths.
    4. **Velocity Control**: Sends velocity commands (/cmd_vel-) based on planned paths.
    5. **Obstacle Avoidance**: Dynamically adjusts navigation to avoid collisions using sensor feedback.

Detailed NAV2 architecture for ROVER (in relation to project):

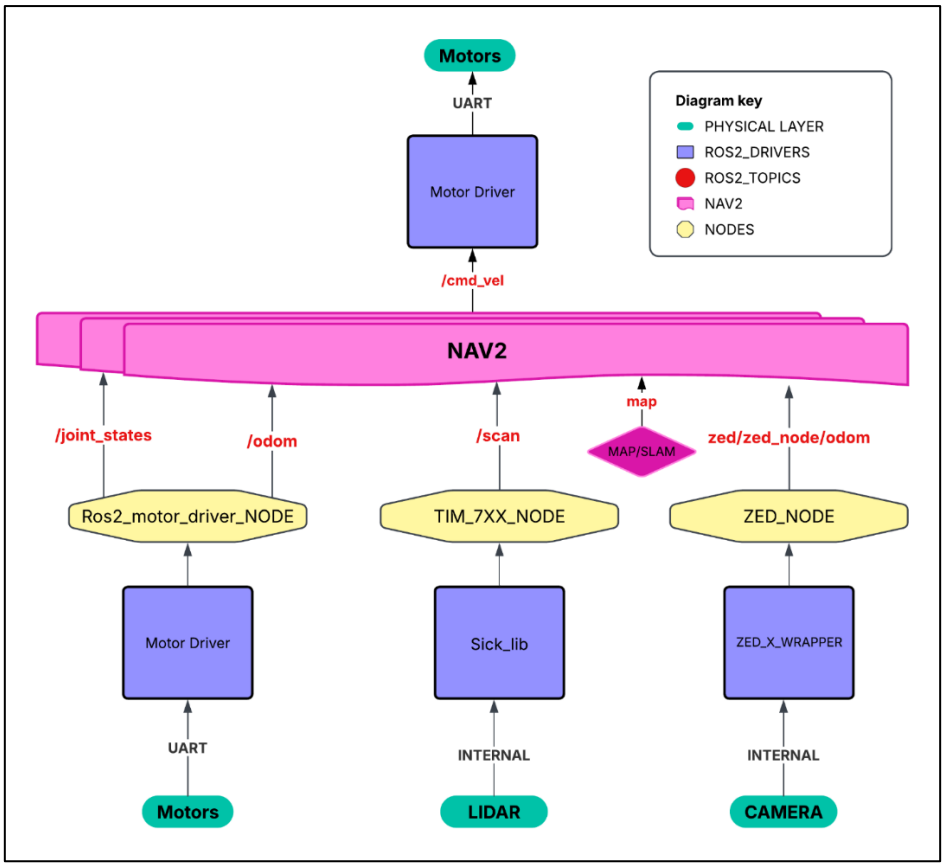

Figure 21. Detailed NAV2 architecture for ROVER

The software architecture of the autonomous rover was originally designed to follow a **ROS2-based structure** that integrates **sensors, motor control, and navigation**, with all components interacting via standard ROS2 communication channels (topics, services, and actions). The uploaded architecture diagram represents this intended system.

❖ **Explanation**

The diagram defines how the core components interact within the ROS2 system to enable autonomous navigation. Each element serves a specific role:

➢ **Physical Layer (Hardware)**

- **Motors**: These are the physical actuators responsible for the rover's movement.
- **LiDAR (TIM781)**: Used for obstacle detection and mapping.
- **ZED Camera**: Provides stereo vision and depth perception.

Each of these hardware components is directly controlled or accessed by a corresponding driver.

- **Driver Layer (ROS2 Drivers)** [25] [3] [32] [33] [21]

This layer handles **direct communication with hardware**. Drivers manage lower-level hardware interaction, such as sending serial commands to motors or configuring LiDAR scan parameters.

- **Motor Driver** interfaces with the motors via UART.
- **Sick_lib** manages data acquisition from the TIM781 LiDAR.
- **ZED_X_WRAPPER** handles image capture and depth processi ng from the ZED camera.

- **Node Layer (ROS2 Nodes)**

Each hardware component's driver is wrapped inside a dedicated ROS2 node that acts as the **interface between hardware and the rest of the system**.

- **Ros2_motor_driver_NODE**: Subscribes to /cmd_vel (movement commands) and sends these to the motor driver, while also publishing joint_states and odom (wheel encoder feedback and calculated odometry).
- **TIM_7XX_NODE**: Collects LiDAR data from Sick_lib and publishes it to the /scan topic.
- **ZED_NODE**: Collects stereo images, depth data, and odometry from the camera and publishes them on the topic zed/zed_node/odom.

- **Topics (Communication Channels)**

The **topics** (in red in the diagram) are the ROS2 communication links between different nodes.

- **/cmd_vel**: Command velocities produced by Nav2 and sent to the motor controller.
- **/joint_states**: Published by the motor node to report the current wheel positions and speeds.
- **/odom**: Provides real-time odometry data based on encoder feedback.
- **/scan**: The processed LiDAR data published by the LiDAR node.
- **zed/zed_node/odom**: The odometry derived from the ZED stereo camera.

- **Nav2 Stack (Navigation Layer)** [32]

At the centre of the architecture is **Nav2**, the core navigation module. Nav2 consumes all the sensor data and state feedback to perform autonomous navigation:

- It receives the **current position** from /odom and zed/zed_node/odom.
- It uses the **LiDAR scan** to detect obstacles in real time.
- It receives the **map** (created from SLAM or preloaded) to plan global paths.
- Based on this data, Nav2 generates new movement commands (/cmd_vel), continuously updating to account for changes in the environment.

- **Map/SLAM Layer**

The mapping system (e.g., SLAM Toolbox) consumes the /scan data from the LiDAR to build a **2D occupancy grid map**, which Nav2 uses for path planning.

Intended Data Flow (as shown in diagram)

1. **Sensors (LiDAR, Camera, Motors)** provide raw data.
2. **Drivers** (Sick_lib, ZED_X_WRAPPER, Motor Driver) read and manage this data.
3. **ROS2 Nodes** wrap the drivers and expose data via ROS2 topics.
4. **Nav2** consumes all data: map, odometry, scan.
5. **Nav2** generates movement commands (cmd_vel).
6. **Motor Node** converts cmd_vel into actual motor commands.
7. **Motor Node** continuously updates odometry (/odom), feeding back into Nav2.

Overall Objective

This design aimed to create a **closed-loop autonomous navigation system**, where real-time sensor feedback continuously refines localization and obstacle detection, allowing dynamic adjustment of the robot's path to safely reach its destination.

This explanation directly describes the **uploaded system architecture diagram** and reflects the intended design goal of the project.

2. Basic Motion Control using Time based Control and Custom Motor Driver

- **Overview**: Implements a straightforward motion control system without complex navigation algorithms.
- **Components**:
    1. **Predefined Movement Sequences**: Uses fixed motion patterns for movement (e.g., move forward for X seconds, turn for Y seconds).
    2. **Custom Motor Driver**: Sends direct speed and direction commands to the RoboClaw motor controller using serial communication.
    3. **Simple Obstacle Detection (Optional)**: Stops or redirects the rover when an obstacle is detected based on sensor input.
    4. **Manual Control Support**: Allows remote control via ROS2 topics or basic command-line inputs.

Both approaches serve different use cases, where **Nav2** is used for full autonomy and adaptability, while **Basic Motion Control** provides a simpler alternative for controlled environments or testing purposes.

Detailed Architecture for Simple Navigation Approach:

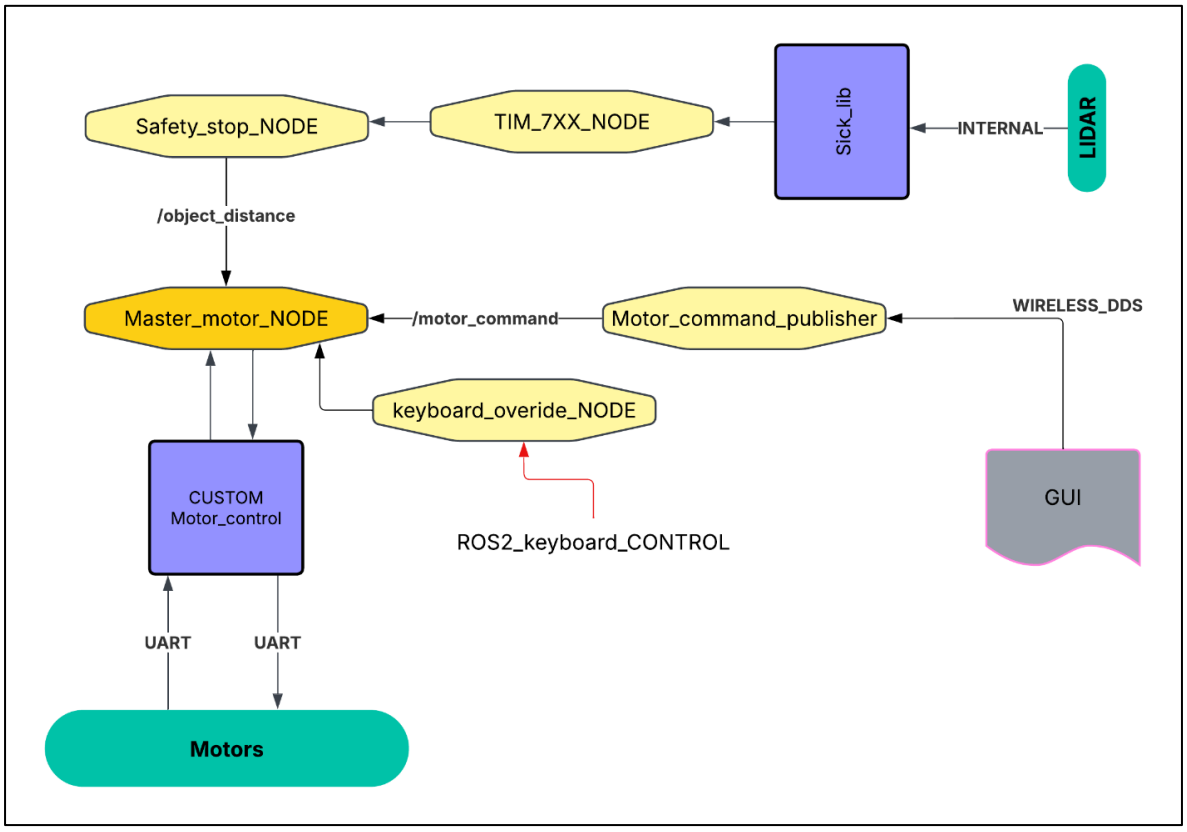

Figure 22. Detailed NAV2 architecture for ROVER

The second uploaded architecture diagram represents the **simplified custom motion control system**, which was actually implemented after the Nav2-based autonomous approach failed. This diagram reflects the fallback system focusing on direct control and obstacle safety handling.

❖ **Explanation**

This architecture demonstrates how the simpler **custom motion control solution** works using direct motor commands and basic obstacle detection logic.

➢ **Physical Layer (Hardware)**

- **Motors**: These are the physical actuators responsible for moving the rover.
- **LiDAR (TIM781)**: Used for basic obstacle detection, without any mapping or complex SLAM process.

This reduced hardware layer only focuses on what is needed for basic motion and safety.

➢ **Driver Layer (ROS2 Drivers)**

- **Custom Motor Control Driver**: Directly communicates with the motors using **serial UART**.
- **Sick_lib**: Interfaces with the TIM781 LiDAR, handling raw data acquisition.

This layer ensures hardware components can operate, but without higher-level planning involved.

➢ **Node Layer (ROS2 Nodes)**

The logic is distributed across a few **custom ROS2 nodes** that handle distinct responsibilities:

- **TIM_7XX_NODE**: Collects LiDAR data and publishes obstacle distances.
- **Safety_stop_NODE**: Monitors obstacle distances and decides if emergency stops are necessary.
- **Motor_command_publisher**: Publishes high-level movement commands.
- **keyboard_override_NODE**: Allows manual override of motion commands using wireless keyboard input.
- **Master_motor_NODE**: Acts as the central motion manager. It receives movement commands from either the motor_command_publisher (autonomous sequences) or keyboard_override_NODE (manual input), applies safety logic from Safety_stop_NODE, and sends final commands to the custom motor control node.

- **Custom Motor Control Node**: Converts these final motor commands into **low-level serial commands** sent directly to the RoboClaw motor controller.

This layered approach allows both **predefined motion commands** and **manual overrides**, with a safety layer checking obstacles before actual movement.

- **Topics (Communication Channels)**

The communication between nodes happens via ROS2 topics:

- **/object_distance**: Published by Safety_stop_NODE, it reports detected obstacles to the master motor node.
- **/motor_command**: The main channel for publishing intended movement commands, which can originate from either the autonomous publisher or manual override.

- **Wireless Interface (GUI)**

A **GUI (Graphical User Interface)** was connected via **Wireless DDS** to allow remote manual control. Commands sent from this GUI enter the system through the keyboard_override_NODE, which injects them into the command flow at the same level as autonomous commands.

- **Data Flow (As shown in diagram)**

1. **TIM781 LiDAR** constantly scans for obstacles, feeding data to TIM_7XX_NODE.
2. TIM_7XX_NODE processes the data and reports it to Safety_stop_NODE.
3. Safety_stop_NODE decides if motion should be allowed or blocked based on proximity to obstacles and sends /object_distance.
4. **Movement commands** are generated either from predefined sequences (motor_command_publisher) or manual input (keyboard_override_NODE).
5. Master_motor_NODE decides the final command to send to the motor, applying safety logic from Safety_stop_NODE.
6. This final command is sent to the **Custom Motor Control Node**, which directly communicates with the motor driver over UART.

Overall Objective

This system was designed to offer basic, reliable motion control with an **emergency stop mechanism** based on simple obstacle detection. It does not rely on complex localization, mapping, or planning, making it **simpler and more reliable for immediate deployment** after Nav2 failed.

This explanation directly describes the **second uploaded system architecture diagram** and reflects the **actual working system** in the project.

5.4 Component Level Architecture

5.4.1 LiDAR Software Architecture- {*Nirzer Gajera (41638)*}

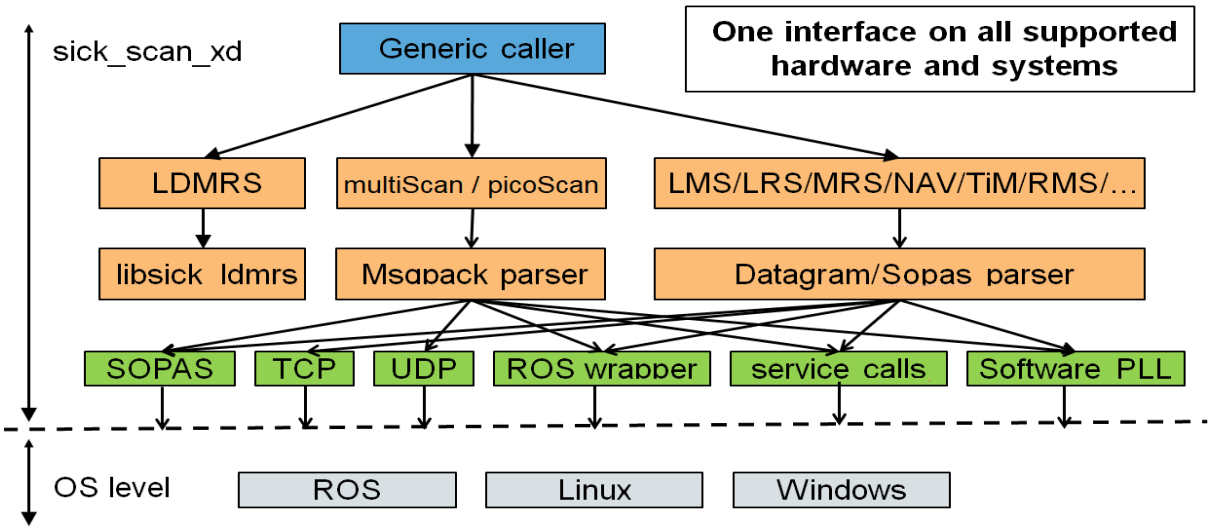

Figure 23.SICK Generic caller for SICK LiDAR Devices [20]

This chapter discusses the software architecture of SICK LiDAR for TIM781 with the reference from sick_scan_xd repository for SICK LiDAR's.

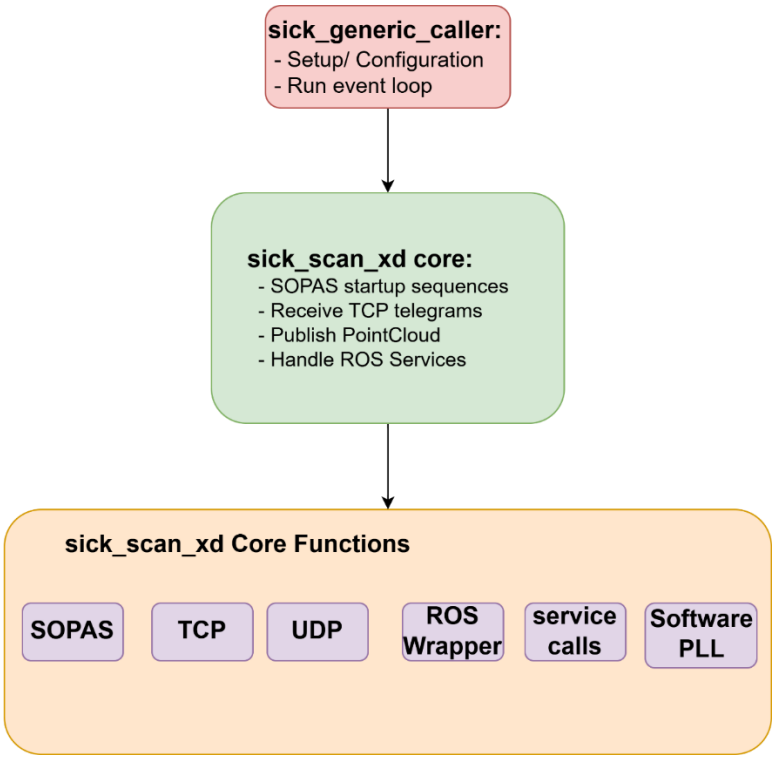

Figure 24. sick_scan_xd architecture for TIM781

This diagram above represents the **sick_scan_xd** framework, a unified interface for integrating **SICK LiDAR sensors** across multiple hardware and operating systems. At the core, the **Generic Caller** module provides a single interface to manage various **LiDAR models**, including **LDMRS, multiScan/picoScan, and LMS/LRS/MRS/NAV/TiM/RMS**. These models utilize different **data parsers** such as **libsick_ldmrs, MScanPack parser, and Datagram/SOPAS parser** to process sensor data. The framework supports multiple **communication protocols and integration methods**, including **SOPAS, TCP, UDP, ROS wrappers, service calls, and Software PLL**, ensuring flexible connectivity. This architecture is designed to work across **ROS, Linux, and Windows**, enabling cross-platform compatibility for robotic and automation applications. [20]

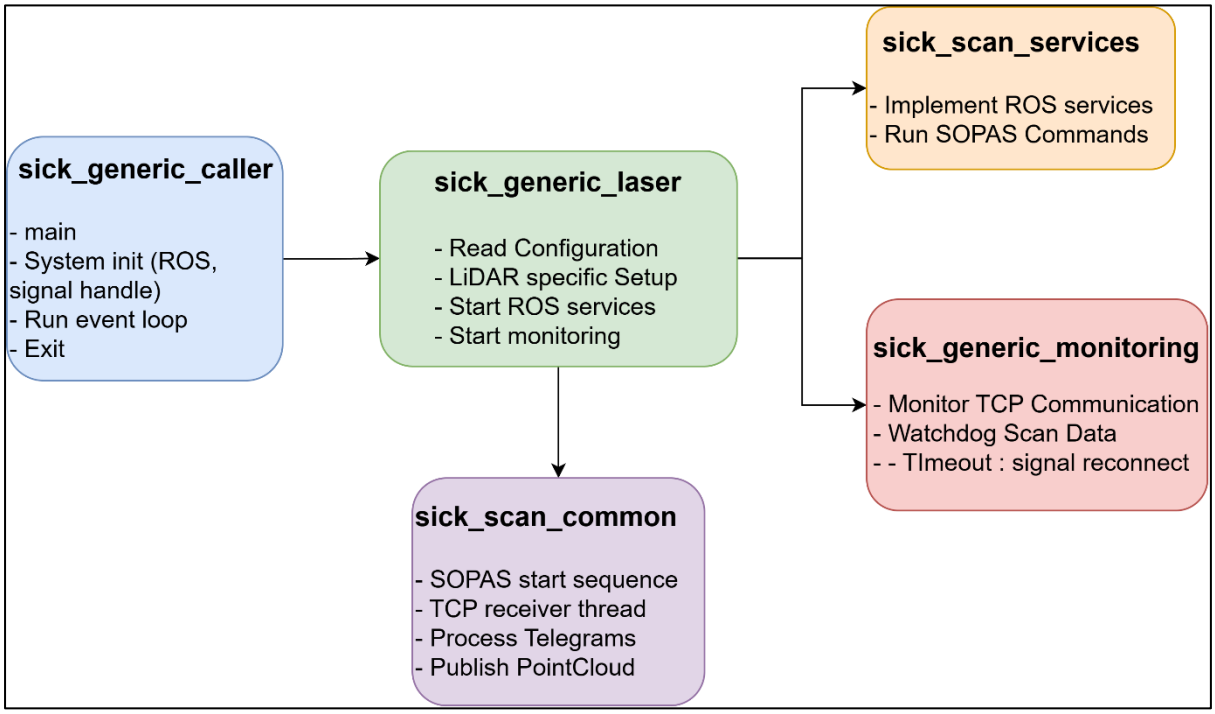

Figure 25. Software Functional blocks of sick_scan_xd

This diagram illustrates the **sick_scan_xd** architecture, highlighting its core components and functions. At the top, the **sick_generic_caller** module is responsible for **setup, configuration, and managing the event loop**, ensuring that the scanning process is properly initialized. The **sick_scan_xd core** is the central processing unit, handling **SOPAS startup sequences, receiving TCP telegrams, publishing PointCloud data, and managing ROS services.** At the bottom, the **sick_scan_xd core functions** include various communication protocols and interfaces, such as **SOPAS, TCP, UDP, ROS Wrapper, service calls, and Software PLL**, providing flexible connectivity options for real-time LiDAR data acquisition and processing.

Overview of sick_scan_xd Functional Blocks

The **sick_scan_xd** package consists of five primary functional blocks that facilitate the initialization, setup, communication, and monitoring of **SICK TIM781 LiDAR** within a **ROS-based system**. Each block plays a specific role in ensuring proper data acquisition, parsing, and system resilience, as depicted in the provided diagram.

1. Initialization and Setup Modules

- **sick_generic_caller** handles the **main system initialization**, including **ROS setup, signal handling, and execution of the event loop**.
- **sick_generic_laser** is responsible for **reading configuration parameters** from launch files, which are retrieved using:
    - **ROS 2 and generic setups:** LaunchParser from the ROS wrapper.
- This module also conducts **LiDAR-specific setup**, defining parameters such as the **number of layers and angular resolution** using sick_scan_xd::SickGenericParser, and converts scan data into **ROS sensor messages** (sensor_msgs::LaserScan).

2. Common LiDAR Handling Module

- **sick_scan_common** serves as a universal interface for various LiDAR models, including **LMS, LRS, MRS, NAV, TiM, and RMS sensors**.
- It is implemented through SickScanCommon and SickScanCommonTcp, utilizing SickGenericParser to process **LiDAR-specific properties and messages**.
- This module executes core functions such as:
    - **SOPAS startup sequence**
    - **Receiving and processing TCP messages**
    - **Parsing scan data and converting it into a PointCloud**
    - **Publishing PointCloud data to ROS**

3. Monitoring and Error Handling

- **sick_generic_monitoring** continuously monitors TCP communication and **scan data integrity**.
- If a **timeout or network error** occurs, the system triggers **reinitialization** using:

- SickScanMonitor for general scan data.
- PointCloudMonitor for PointCloud validation and recovery.

4. ROS Service Implementation

- **sick_scan_services** enables **ROS service integration**, executing commands via the **SOPAS protocol**.
- The class sick_scan_xd::SickScanServices facilitates **service registration and message conversion** between ROS and SOPAS.

5.4.2 Autonomous navigation using Piksi – {*Niranjan Channayya (41559)*}

The architecture below illustrates the high-level architecture of the autonomous navigation system utilizing **Swift Piksi (GPS/RTK)** for precise localization. The system integrates multiple sensor inputs, including the **BMI160 IMU** and **Wheel Encoders**, to enhance positioning accuracy through **sensor fusion**, implemented using a **Kalman filter**. This fusion process refines the robot's pose estimation by combining GNSS, inertial, and odometry data, compensating for potential inaccuracies or signal losses in individual sensors.

The sensor fusion output is then processed by the **Global Planner**, responsible for generating a high-level path to the desired goal while avoiding obstacles. The **Local Planner** refines this trajectory in real time, considering dynamic obstacles and environmental constraints. Finally, the computed motion commands are sent to the **cmd_vel Publisher**, which interfaces with the motion controller to drive the robot along the planned path.

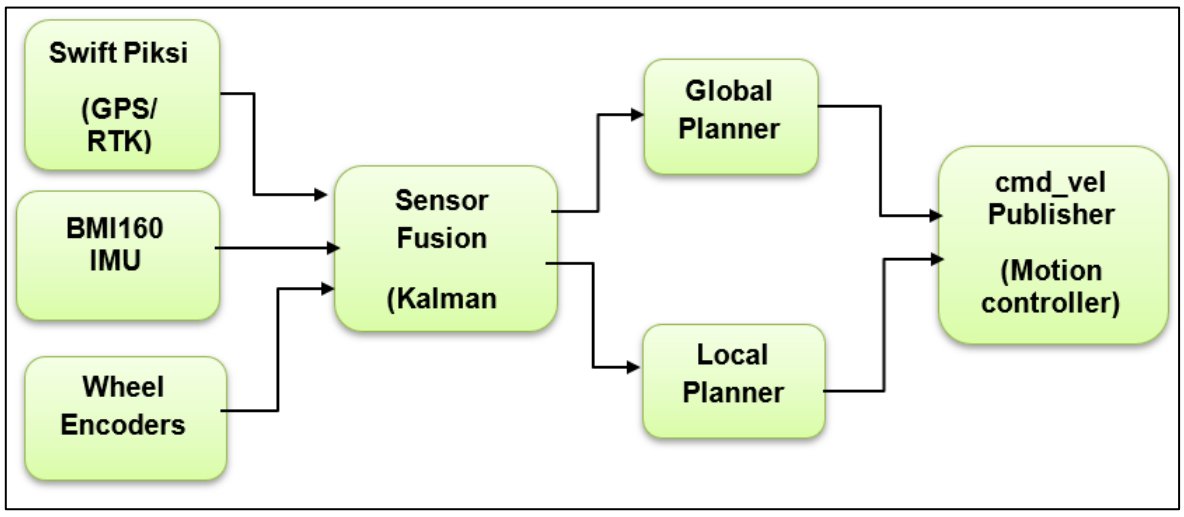

Figure 26.Basic architecture for autonomous Navigation using Swift Piksi

This modular approach ensures robust and precise autonomous navigation, leveraging RTK-corrected GNSS positioning, IMU-based dead reckoning, and odometry data for reliable movement in both structured and unstructured environments.

6 Hardware Setup and Testing

6.1 Motor Testing on Windows – {*Krish. Shah* (41564)}

The initial phase of hardware testing was conducted on a Windows PC to ensure that the sensors and motor controller functioned as expected before transitioning to the main computing platform.

Additionally, the RoboClaw motor controller was tested by sending serial commands to ensure proper response and control of motor speed and direction which was partly unsuccessful. This phase provided a controlled environment to debug and validate the components before their integration with Jetson. Multiple testing iterations were performed to ensure reliability, during which different configurations were assessed to optimize sensor responsiveness and motor control stability.

Various debugging techniques, including serial monitoring by sending python commands, and motor response analysis, were employed to ensure that data acquisition was both accurate and efficient.

6.2 Motor Controller and Encoder Testing – {*Krish. Shah* (41564)}

A dedicated ROS2 motor control node was implemented to interface with the RoboClaw motor controller using serial communication. The motor response to /cmd_vel messages was tested to ensure smooth acceleration and deceleration.

Encoder feedback was examined to accurately measure motor rotations and determine the actual speed of the rover. Issues were identified where the rover performed well at high speeds but displayed inconsistencies at lower speeds, leading to further debugging of the encoder readings and adjustments in motor tuning parameters which.

Extensive calibration was conducted to balance motor power distribution and try to improve the accuracy of distance estimation based on encoder data. Real-time speed feedback was also introduced to mitigate inconsistencies, allowing for more refined speed adjustments based on encoder output but the data was unsuccessful.

6.3 Camera Integration – {Krish. Shah (41564)}.

The ZED X stereo camera system was set up with basic communication established, though its full integration into the navigation pipeline was not proven beneficial for development. Custom ROS2 message formats were defined to ensure efficient handling of sensor data across different modules.

More detailed information can be found here 7.10.4 Using ZED-X Camera – {Krish. Shah (41564)}

6.4 Swift Navigation Piksi Multi -*Niranjan Channayya (41559)*

The Swift Piksi unit has a built-in IMU sensor – Bosch BMI160. This sensor gives us linear acceleration and angular velocity data.

6.4.1 Distance calculation using BMI 160 IMU unit

Below we see comparison of graphs when the IMU unit is displaced for 0.5 meters (50 cms) in z axis. Here, acceleration is sampled at a very high rate of 200 Hz and we can see the calculation of velocity and distance over time.

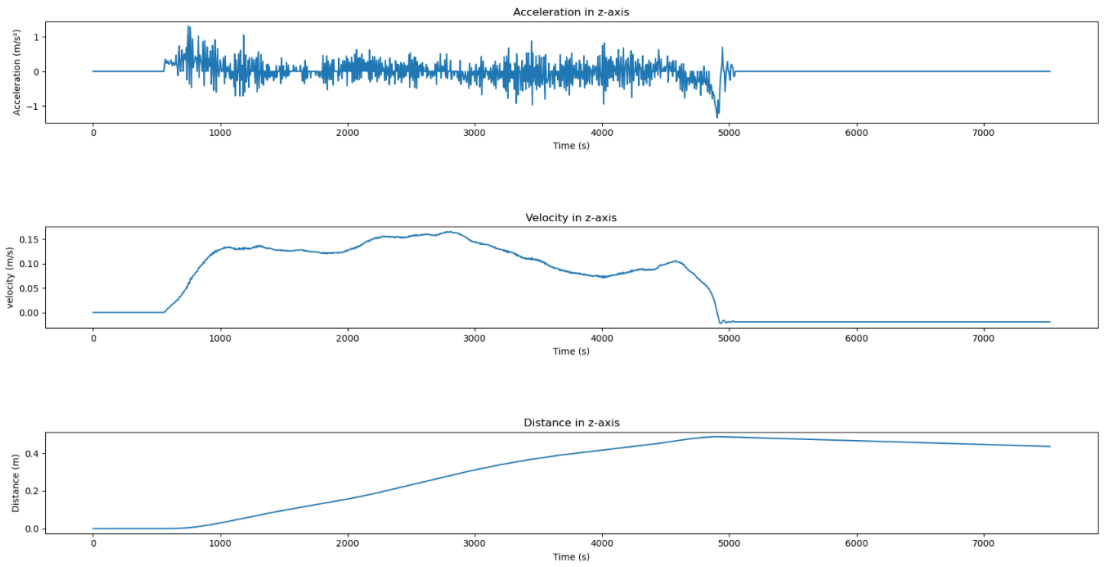

Figure 27. Moved the IMU unit for 50 cm - 1st sample

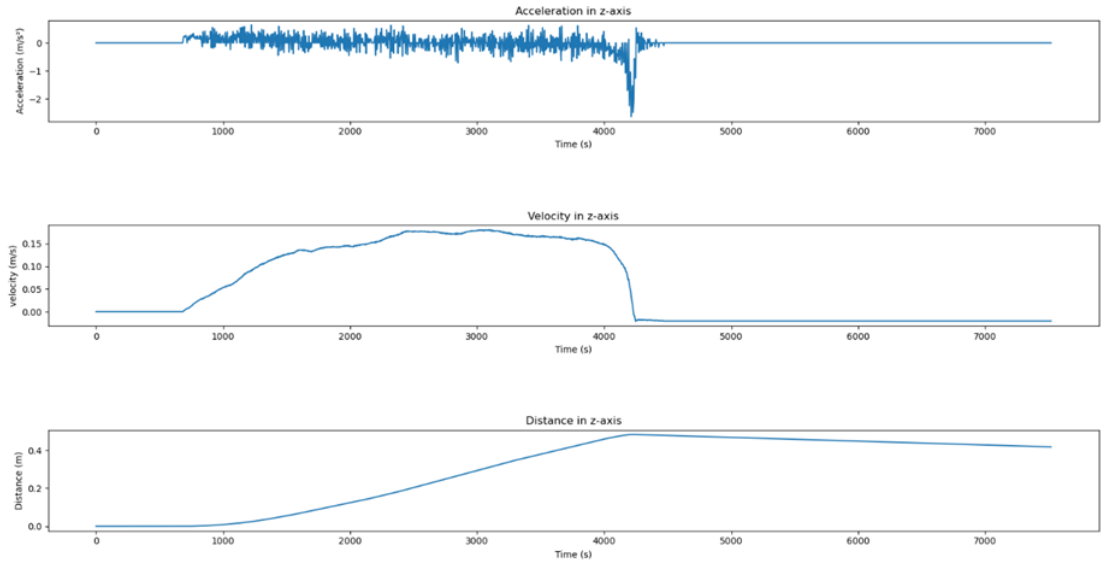

Figure 28. Moved the IMU unit for 50 cm - 2nd sample

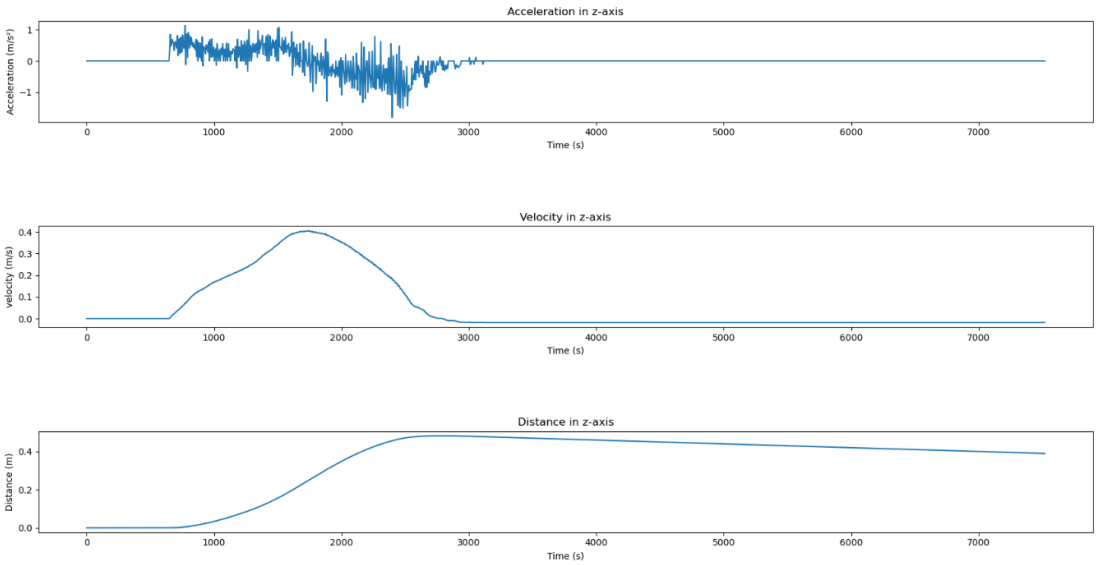

Figure 29. Moved the IMU unit for 50 cm – 3rd sample

- **Acceleration:** The acceleration plot shows a clear spike at the start of motion, fluctuating around zero during movement. There is noticeable noise in the acceleration data, causing irregular variations. The sudden negative spike is observed at the end of motion, due to the stopping force.

- **Velocity:** Velocity gradually increases as expected, reaching a peak when the sensor is moving at its highest speed. After stopping, the velocity does not return to exact zero immediately but instead drifts slightly, indicating integration drift. Small acceleration noise accumulates into velocity drift, which affects distance calculations.

- **Distance:** The estimated distance increases progressively and reaches approximately 0.5 meter as expected. The distance value does not remain completely stable after movement stops, showing slight variations due to double integration drift.

Basically, the distance calculated is almost correct until the sensor is in movement. But as soon as the unit comes to halt, the calculated distance starts to reduce gradually even though the unit is in halt. These errors are due to –

1. **Sensor Noise:** The Bosch BMI160 accelerometer introduces random noise in acceleration readings, which is amplified further when we integrate. These high frequency noise in acceleration leads to small velocity fluctuations which accumulate over time as errors in distance calculations.

2. **Bias Drift:** The IMU sensor often shows a bias drift, meaning there is some acceleration readings even when the unit is in rest. This bias introduces errors in distance calculation.

3.

Implemented Solutions to increase accuracy:

1. **Senor Offset Calculation:** There is a constant bias or error present in sensor readings when the sensor is stationary. Ideally, when the accelerometer is stationary on a flat surface, it should measure only the gravitational acceleration (approximately 9.81 m/s² on the axis aligned with gravity) and 0 m/s² on the other axes.

   However, due to manufacturing tolerances, temperature variations, and electrical noise, raw sensor readings often include a constant bias that must be removed for accurate measurements.

   In our case, these biases were removed by calculating the offset when the unit was still and subtracting that value in the further readings.

2. **Acceleration threshold**: An acceleration threshold is a simple yet effective method for reducing noise in IMU sensor readings. It works by ignoring small fluctuations in acceleration that are likely due to sensor noise rather than actual motion. This is particularly useful when integrating acceleration data to calculate velocity and distance, as small errors can accumulate and lead to significant drift.

   In our case, we added a threshold of around 0.05 m/s^2 to remove this noise.

$$af = \begin{cases} 0, & |a| < 0.05\ m/s2 \\ a, & |a| \geq 0.05\ m/s2 \end{cases}$$

   Here,

   af is acceleration after thresholding

   a is raw acceleration reading (in m/s^2)

Possible Solutions to increase accuracy:

1. **Low Pass Filtering**: By applying a low pass filter to the accelerometer data it is possible to remove high-frequency noise from acceleration data before integration. This will reduce the final drift that is observed in distance calculations.

   There are two possible filtering techniques-

   Simple Moving Average (SMA) Filter- Here we average a fixed number of recent samples. This is a simple filter, but introduces delay.

This can be applied by using the formula-

$$y[n] = \frac{1}{N}\sum_{k=0}^{N-1} x[n-k]$$

Where:

y[n] is the filtered signal.

N is the window size.

x[n] is the raw acceleration at time n.

Exponential Moving average (IIR) Filter-This Filtering technique gives more weight to recent data points. This is more efficient for real time filtering as it has less delays.

This can be applied using the formula-

$$y[n] = \alpha \cdot x[n] + (1-\alpha) \cdot y[n-1]$$

Where:

α is the smoothing factor (between 0 to 1)

x[n] is the current input

y[n-1] is the previous input

A smaller *α* means stronger smoothing (better noise reduction), but more lag.

2. **Zero Velocity Update:** Identify moments when the sensor is stationary and force velocity to zero to correct drift. This significantly reduces accumulated errors in distance estimation. By using this technique, we will be able to significantly reduce drift in distance calculation.
3. **Kalman filtering:** This is a more advanced technique to estimate acceleration while correcting for noise and drift. This approach helps to-
    - Smooth out noise from raw sensor data.
    - Correct sensor drift over time.
    - Estimate Velocity and distance more accurately from acceleration data.

    A **Kalman Filter** can combine data from two or more sensors, leveraging their strengths to produce a more accurate and stable estimate. This approach is called **sensor fusion**.

6.5 Lidar Integration - *{Yogachand pasupuleti (41588)}*

6.5.1 Detecting SICK devices in the network

To initiate the project, establishing a dependable communication pathway with the SICK scanning devices was essential. This began with network discovery, utilizing the sick_generic_device_finder.py script, which employs UDP broadcast to locate available scanners. Verification of the correct network broadcast address was conducted using ifconfig, with adjustments made to the UDP_IP variable as needed. As illustrated in the figures within section 4.2.6, following successful device discovery, the LiDAR's specific network parameters were configured through the SICK SOPAS ET software which is discussed extensivel in chapter **4.2.5**. This included assigning a unique IP address and ensuring the settings were persistently stored in the EEPROM. Finally, network connectivity was validated using ipconfig and ping commands, confirming distinct IP addresses for the host PC and LiDAR, thereby ensuring the foundational network setup was robust and reliable for subsequent development and testing phases.

6.5.2 Integration of TiM781 LiDAR Sensor

To integrate the TiM781 LiDAR sensor into the system, the sick_scan_xd driver was utilized. The installation and configuration process were carried out on both Windows (without ROS) and Linux (with ROS 2) platforms. On windows, the driver was built using CMake and VS studio, and the sick_generic_caller executable was used to launch the sensor with the appropriate launch file (sick_tim_7xx.launch).

On Linux, the driver was built using colcon in a ROS 2 workspace, and the sensor was launched using the ros2 launch command with the same launch file as shown in figure. The hostname parameter was specified to connect to the sensor's IP address, ensuring proper communication. This setup enabled seamless data acquisition and integration of the TiM781 LiDAR sensor into the project.

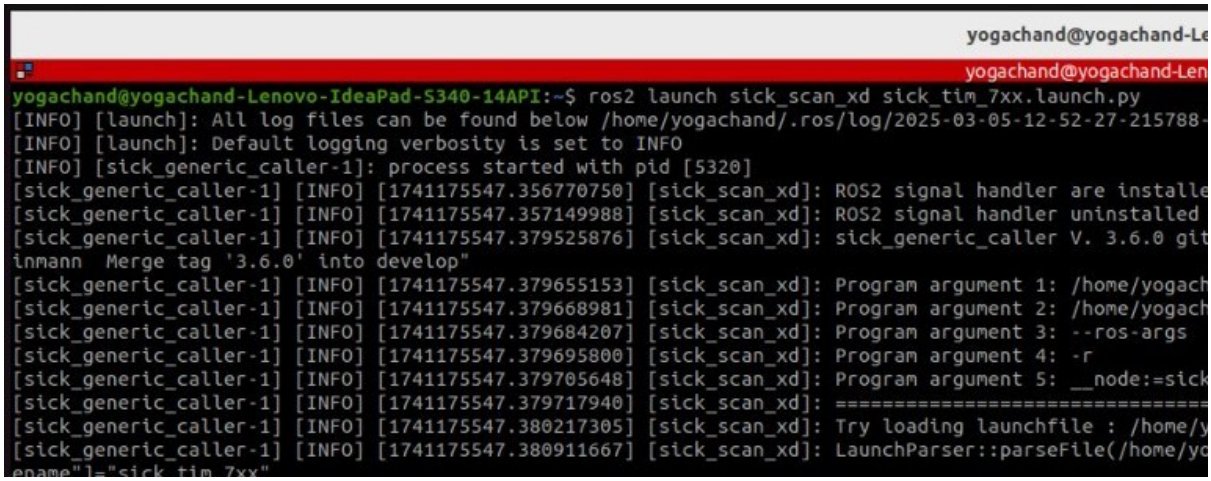

Figure 30. Launch File for TiM781 LiDAR Integration

ROS services: -

OS services are utilized to send COLA commands to the sensor, a feature invaluable for diagnostic purposes by enabling device status queries. As depicted in Figure 26, the LiDAR driver leverages ROS services during its launch sequence to exchange COLA commands. This exchange allows for the retrieval of crucial device status information, ensuring proper initialization and facilitating troubleshooting. This integration of ROS services with COLA commands is a critical component for maintaining sensor health and operational integrity within the ROS environment.

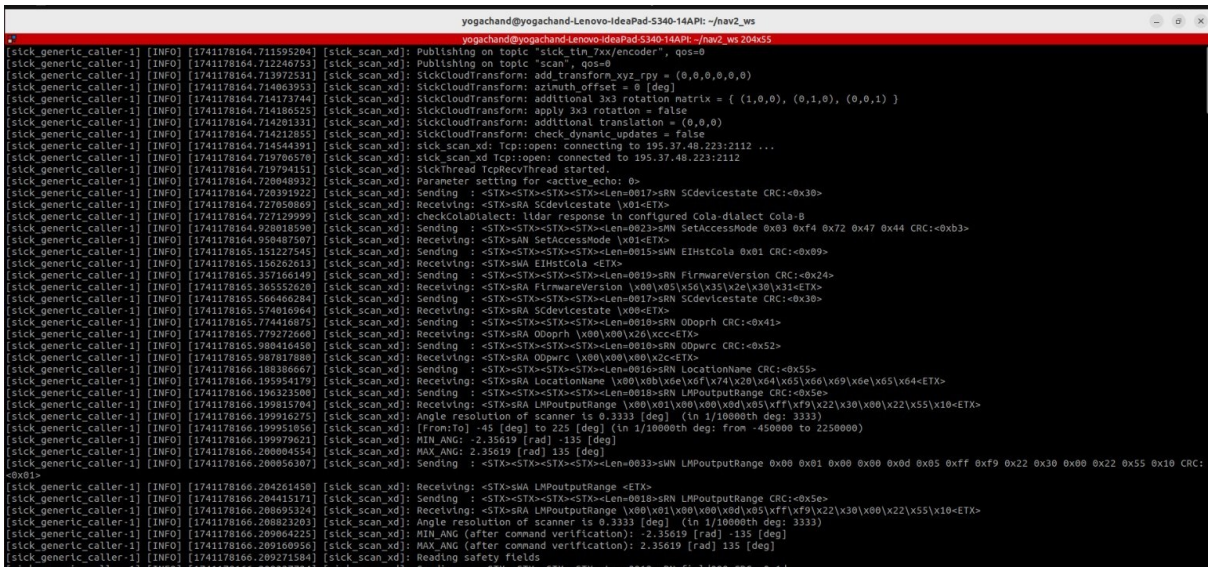

Figure 31. LiDAR Driver COLA Command Exchange via ROS Services

Driver states, timeouts:

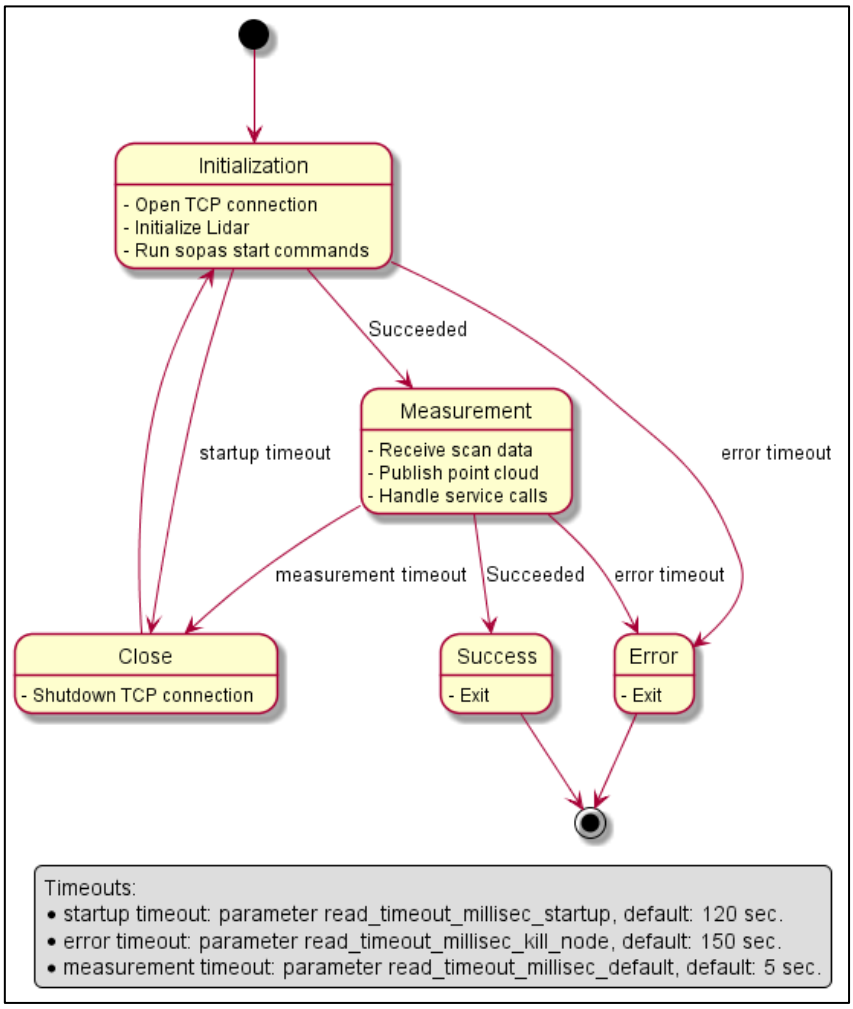

Figure 32. The transition between states and Timeouts [20]

The driver operates in two distinct states: Initialization and Measurement. Upon startup, the driver enters Initialization mode as shown in above figure 27, where it configures the scanner using a predefined list of SOPAS commands. Successful completion of this phase triggers an automatic transition to Measurement mode, during which the scanner transmits scan data, and the corresponding point cloud is published. To ensure robust operation, communication between the driver and scanner is continuously monitored. Three distinct timeouts are implemented to handle communication disruptions. In Measurement mode, a 5-second inactivity period (Timeout 0) results in the closure of the TCP/IP connection, followed by a brief delay, reconnection, and re-initialization. During Initialization, a 120-second inactivity (Timeout 1) triggers the same reconnection and re-initialization process. Irrespective of the operating mode, a 150-second inactivity period (Timeout 2) leads to the termination of the driver. Importantly, the internal timer tracking these timeouts is reset upon successful communication, with the timeout durations reflecting the time elapsed since the last received message from the LiDAR. If no message has been received since program start, the timeout calculations begin from the program's initiation time.

Common Parameters: -

For the launch-file settings and the tag/values pairs the following keywords are supported:

Keyword	Meaning	Default value	Hint
min_ang	Start scan angle in [rad]	-2.3998277	We can change as per requirement
max_ang	End scan angle in [rad]	+2.3998277	We can change as per requirement
hostname	Ip address of scanner	195.37.48.223	change to scanner ip address in your network (see faq)
port	port number	2112	do not change, check firewall rules if there is blocking traffic
timelimit	Timelimit in [sec]	5	do not change

Table 11. LiDAR Launch File Parameters

Software PLL:

The software PLL plays a crucial role in converting the LiDAR's internal tick timestamps to the ROS 2 system time. Many sensor devices, including LiDARs, provide data with their own timestamps. While these timestamps are often accurate, they may have a different time base and a bias relative to the PC's system's time or other sensor clocks. Without specialized hardware like GPS for synchronization, this software PLL estimates and compensates for the discrepancy. Initially, we encountered issues with the PLL implementation on Windows, but these were successfully resolved by utilizing a ROS 2 service. This solution ensured accurate synchronization between the LiDAR's timestamps and the ROS 2 system time, facilitating reliable point cloud data processing. The software component responsible for this PLL functionality is a critical part of the overall software architecture, ensuring accurate time synchronization and data integrity within the ROS 2 environment.

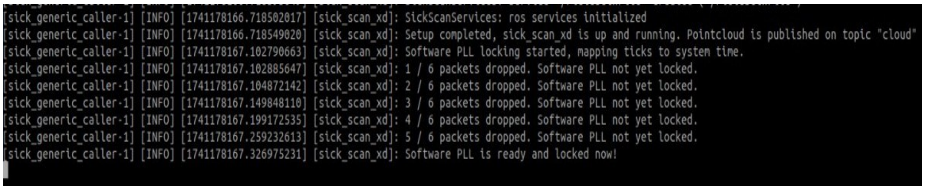

Figure 33. Terminal shows LIDAR PLL is ready and locked

7 Software Implementation

7.1 ROS2 Workspace and Package Development – {*Krish. Shah* (41564)}

A custom ROS2 workspace was created to facilitate organized software development. Custom packages were built to handle motor control, sensor data processing, and communication between different subsystems. Within these packages, publisher nodes were implemented to send messages for motor execution, while subscriber nodes processed sensor feedback for real-time decision-making. The software architecture was structured to allow modular and scalable expansion in future iterations.

Additionally, debugging tools such as ROS2 logging and real-time visualization in Rviz were leveraged to improve development efficiency. Separate packages were made for NAV2 and Sensors which helped to locate the configuration and launch files to easily modify for flexible testing. [10]

Figure 34. ROS2 Packages

Detailed Explanation: 5.3.1 ROS2 Package Overview

7.2 ROS2 Custom Node Implementation – {*Krish. Shah* (41564)}

The system was designed to support different levels of automation, ranging from advanced navigation techniques to simpler motion control methods. A motor control node was implemented to handle /cmd_vel messages and interact with the RoboClaw motor controller via serial communication, ensuring precise control of speed and direction. This was done to implement a simple solution

Despite tuning the PID parameters and refining encoder feedback, challenges arose in implementing an effective closed-loop control system due to lack of feedback from the IMU or the encoders.

The open-loop control method was able to generate movement but lacked precision due to reliance on time-based execution rather than direct feedback integration.

- Detailed Explanation: 7.10.6 Simple Solution Implementation – {Krish. Shah (41564) & Nirzer Gajera (41638)}

7.3 URDF Model for Rover Simulation – {*Krish. Shah* (41564)}

A **URDF (Unified Robot Description Format) model** was developed to replicate the physical properties of the rover within a simulated environment. The model was detailed to accurately represent joint configurations, dimensions, weight distribution, and sensor placement.

However, a significant challenge arose in replicating the movement of the rover's caterpillar tracks, as URDF does not natively support continuous track simulation.

Attempts to approximate the behaviour using multiple linked wheels or simplified friction models did not fully capture the real-world dynamics of tracked movement. While the physics-based model performed well in basic scenarios, real-world execution presented discrepancies, primarily due to the lack of precise odometry integration, the limitations of open-loop motor control, and the difficulty in simulating track-based propulsion. Despite these challenges, the URDF model was instrumental in validating theoretical behaviours before physical testing and provided a useful framework for sensor placement and collision detection. [10]

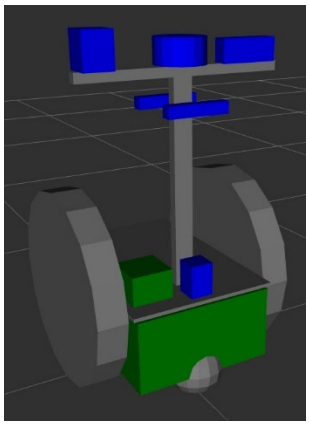

Figure 35. URDF model

Detailed Implementation: 7.10.2 URDF Making – {Krish. Shah (41564)}

7.4 Python based GUI- {*Nirzer Gajera (41638)*}

The **Path Planner** GUI component is developed using **PyQt6**, a Python framework for building cross-platform graphical user interfaces. The main purpose of this GUI is to allow users to **select up to 5 waypoints for navigation while avoiding obstacles and manual control of the rover**. The selected waypoints are then converted to real-world coordinates and published to the **ROS2 navigation stack** via a **DDS-based communication system**.

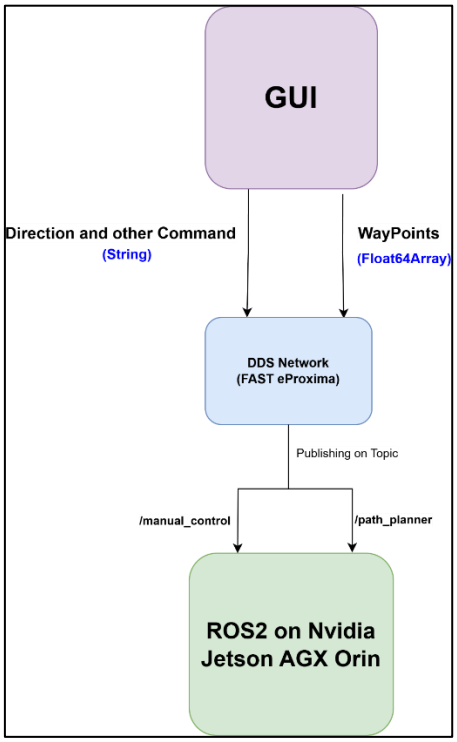

Figure 36. Communication structure for GUI

- **Key Functionalities for the project**

1. **Waypoint Selection:**
    - The user can click anywhere in the GUI to select up to 5 waypoints.
    - If more than 5 points are selected, the selection resets, allowing the user to define a new path.
    - Points cannot be placed inside obstacles.

2. **Obstacle Avoidance:**
    - The GUI contains **predefined obstacles**, represented as black rectangles.
    - If the user clicks inside an obstacle, the point is rejected, and an alert is printed.
3. **Path Visualization:**
    - Selected waypoints are drawn as small red circles.
    - Lines are drawn to connect selected points, forming a path.
4. **Path Conversion & Publishing:**
    - The waypoints are **converted from pixel coordinates to real-world meter values** using a predefined scale factor (**PIXEL_TO_METER**).
    - The converted waypoints are packaged into a **ROS2 Float64MultiArray message** and published to the robot via DDS communication.
    - The ROS2 node logs the published path for verification.

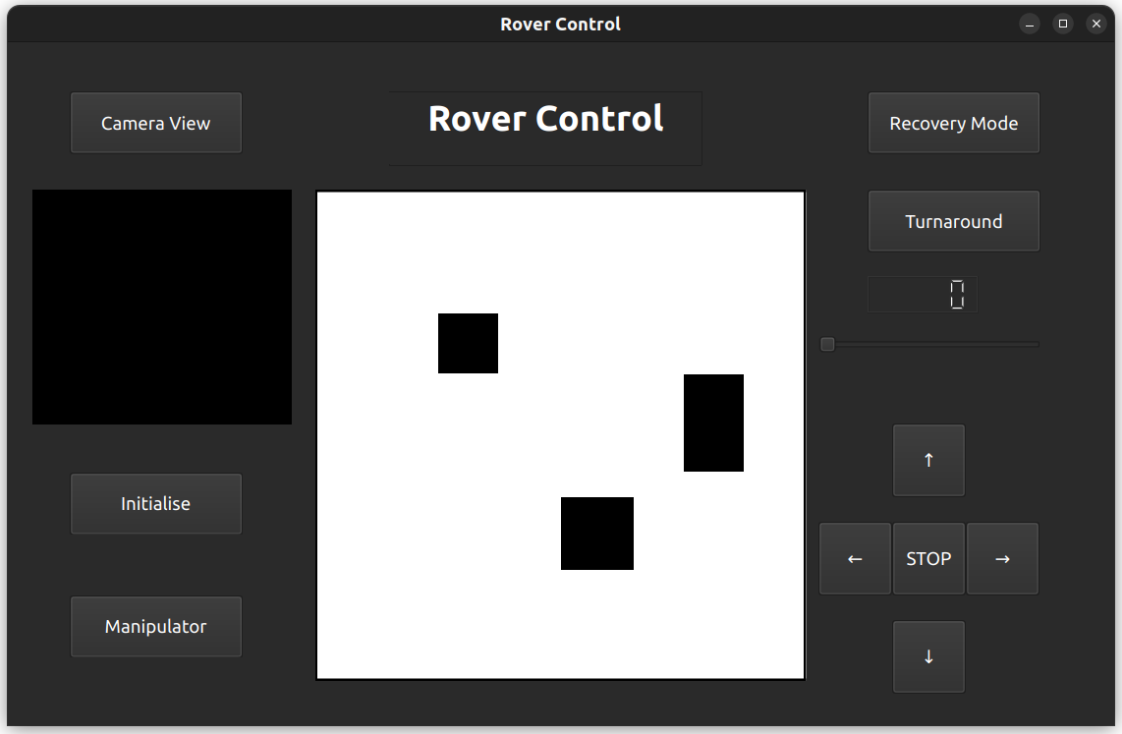

Figure 37. GUI Snapshot

7.4.1 Important Libraries for ROS2- *{Nirzer Gajera (41638)}*

1. rclpy (ROS Client Library for Python)

rclpy is the Python client library for **ROS2**, allowing developers to create nodes, publish/subscribe to topics, call services, use actions, and work with parameters. It provides bindings to the **ROS2 core middleware (rcl)** and serves as an abstraction layer to work with **DDS-based communication** in Python.

Key Features of rclpy

- **Node Management** – Create and manage ROS2 nodes.
- **Topics** – Publish and subscribe to messages.
- **Services** – Implement and call services.
- **Actions** – Support for long-running tasks.
- **Parameters** – Configure node settings dynamically.
- **Timers** – Schedule periodic tasks.
- **Executor Model** – Manages spinning and execution of callbacks.

Configuration of rclpy in GUI

```
self.publisher = self.create_publisher(String, 'manual_control', qos_profile)
             self.path_publisher   =   self.create_publisher(Float64MultiArray,
'path_planner', qos_profile)
```

2. os Library in Python (Used in ROS2)

The os library in Python provides functionality for interacting with the operating system. In ROS2 applications, it is often used to:

- Manage environment variables (e.g., ROS_DOMAIN_ID)
- Run system commands (e.g., launching ROS nodes, executing shell scripts)
- Handle file and directory operations (e.g., reading and writing logs)
- Retrieve system information (e.g., checking CPU/GPU resources)

7.4.2 Major Functions and Classes of the GUI- *{Nirzer Gajera (41638)}*

Class: `PathPlanner(QtWidgets.QLabel)`

This class extends QLabel to function as a **canvas for drawing waypoints and obstacles**.

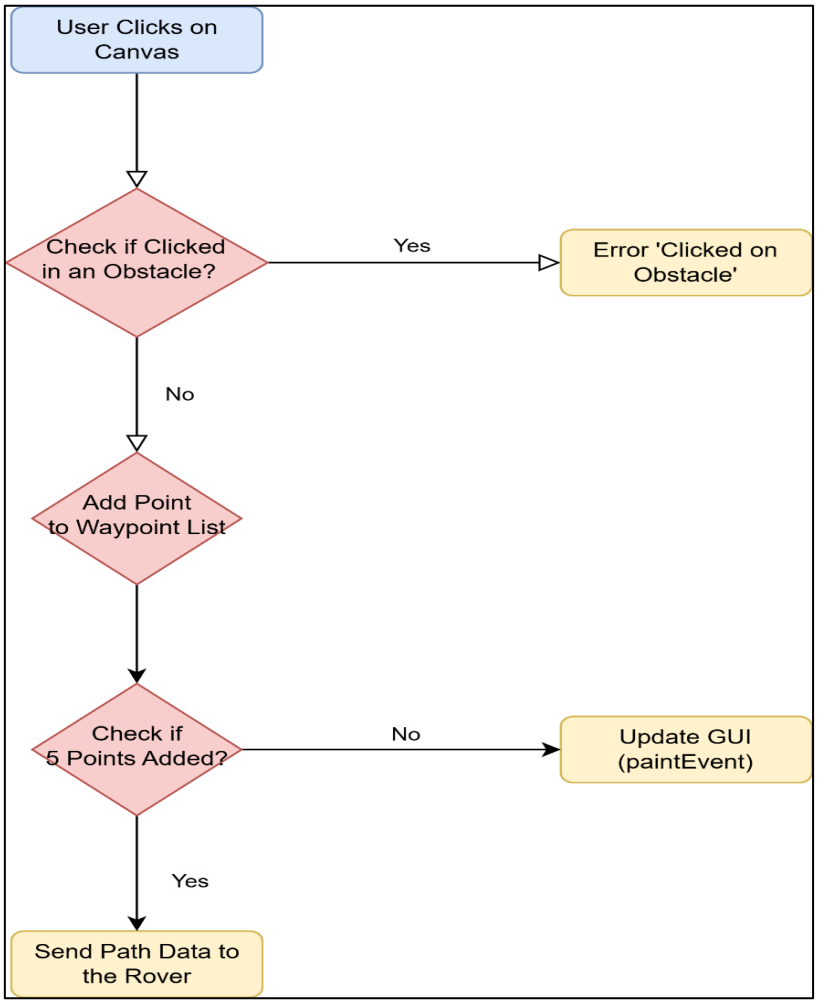

Figure 38. Flowchart for user Click

Constructor: `__init__`

Key Functionalities

- Sets **window size** and **background colour**.
- Initializes an empty list (self.points) to store **user-selected waypoints**.
- Defines a **list of obstacles** using QRect, which stores their positions and sizes.

```
def __init__(self, parent=None):
        super().__init__(parent)
        self.points = []  # Store up to 5 points
        self.setFixedSize(400, 400)
        self.setStyleSheet("background-color: white; border: 2px solidblack;")
        self.setAlignment(QtCore.Qt.AlignmentFlag.AlignCenter)

        # Define Obstacles (Static List of Rectangles)
        self.obstacles = [
            QtCore.QRect(100, 100, 50, 50),  # Obstacle 1
            QtCore.QRect(200, 250, 60, 60),  # Obstacle 2
            QtCore.QRect(300, 150, 50, 80)   # Obstacle 3
        ]
```

Function: `mousePressEvent(self, event)`

Key Functionalities

- Detects **mouse clicks** and retrieves the **clicked position**.
- **Prevents** users from selecting points **inside obstacles**.
- Allows users to select up to **5 waypoints**. If exceeded, it **resets** the waypoints.
- Calls self.update() to trigger paintEvent, which updates the GUI.
- Calls send_path_to_rover() **when 5 waypoints are selected**.

```
    def mousePressEvent(self, event):
        """Click to select up to 5 points while avoiding obstacles."""
        clicked_point = event.pos()

        # Check if Clicked Inside an Obstacle
        for obstacle in self.obstacles:
            if obstacle.contains(clicked_point):
                print("Cannot place point inside an obstacle!")
                return

        if len(self.points) < 5:
            self.points.append(clicked_point)
        else:
            # Reset selection if more than 5 points are clicked
            self.points = [clicked_point]

        self.update()  # Redraw the GUI
```

Function: `paintEvent(self, event):`

Key Functionalities

- Uses **QPainter** to draw **obstacles** (black rectangles).
- Draws **waypoints** as red circles.
- **Connects waypoints** with red lines to visualize the **selected path**.

```
    def paintEvent(self, event):
        """Draw selected points and obstacles."""
        super().paintEvent(event)
        painter = QtGui.QPainter(self)

        # Draw Obstacles
        painter.setBrush(QtGui.QColor(0, 0, 0))  # Black obstacles
        for obstacle in self.obstacles:
            painter.drawRect(obstacle)

        # Draw Points and Connect with Lines
        if self.points:
            pen = QtGui.QPen(QtGui.QColor(255, 0, 0), 3)
            painter.setPen(pen)

            for i, point in enumerate(self.points):
                painter.drawEllipse(point, 5, 5)  # Draw points
                if i > 0:
                      painter.drawLine(self.points[i - 1], point)  # Connect
points
```

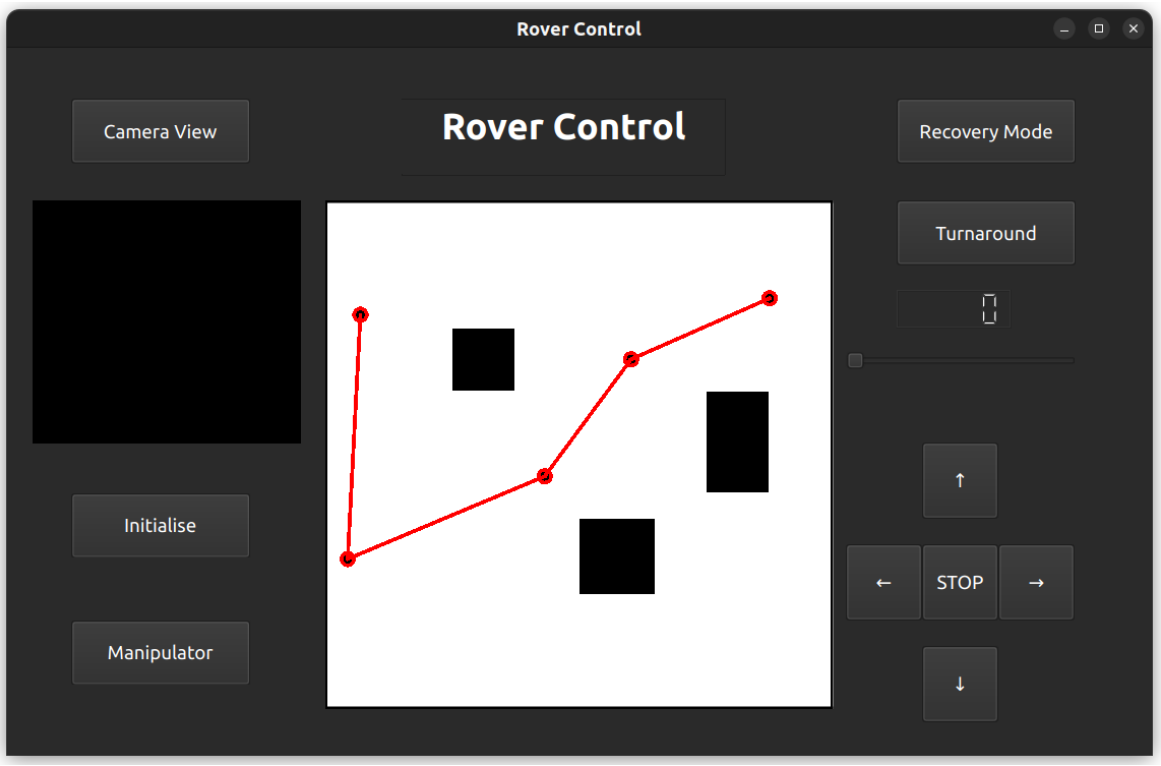

Figure 39. Waypoints selected in GUI

Function: `handleSpeedChange(self, value)`

- Updates the speed display.
- Sends a command "SET_SPEED <value>" to the rover.

```
def handleSpeedChange(self, value):
        """Update speed value."""
        self.SpeedDisplay.display(value)
        self.send_command(f"SET_SPEED {value}")
```

Function: `send_path_to_rover(self)`

Key Functionalities

- Converts **pixel coordinates to real-world coordinates** using PIXEL_TO_METER.
- Creates a Float64MultiArray message containing the **(x, y) positions** of all waypoints.
- Publishes the message using **ROS2 DDS communication** (self.window().ros_node.path_publisher.publish(msg)).
- Logs the **sent path data**.

```
    def send_path_to_rover(self):
        """Send selected points to the rover."""
        if len(self.points) == 5:
            PIXEL_TO_METER = 0.05  # Scale Factor

            path_data = []
            for point in self.points:
                path_data.append(float(point.x()) * PIXEL_TO_METER)
                path_data.append(float(point.y()) * PIXEL_TO_METER)

            msg = Float64MultiArray()
            msg.data = path_data

            self.window().ros_node.path_publisher.publish(msg)
            self.window().ros_node.get_logger().info(f"Sent Path: {msg.data}")
```

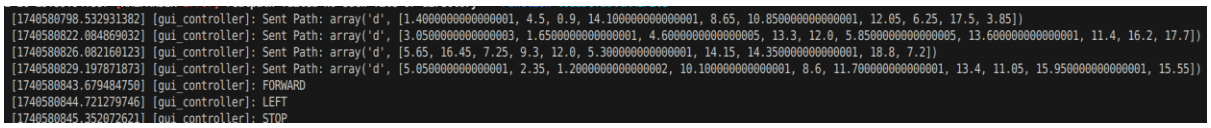

Figure 40. Log Results in Terminal for Waypoints

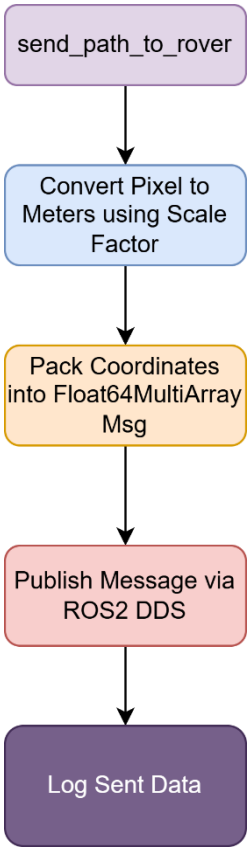

Figure 41. send_path_to_rover Funtion flow diagram

Function:`cleanup_ros(self)` [Clean Up Resources]

- Destroys the ROS2 node and shuts down ROS2.
- Terminates the motor control process.

```
def cleanup_ros(self):
        """Clean up ROS."""
        if hasattr(self, 'ros_node') and self.ros_node is not None:
            self.ros_node.destroy_node()
        rclpy.shutdown()

        # Terminate the motor control process if it's running
        if self.motor_control_process is not None:
            self.motor_control_process.terminate()
            self.motor_control_process = None
            print("Motor control node terminated.")
```

Function: `setupUi(self)` **(UI Layout and Components)**

This method creates all the UI components and places them in the window.

- **Speed Slider & Display:**
  - self.Speed: A horizontal slider for controlling speed.
  - self.SpeedDisplay: A digital display showing the selected speed value.
  - handleSpeedChange() updates the speed when the slider moves.
- **Directional Buttons:**
  - Forward (↑), Backward (↓), Left (←), Right (→), and Stop.
  - Each button is connected to send_command(), sending movement commands to the rover.
- **Additional Buttons:**
  - "Manipulator" – Reserved for additional functionalities.
  - "Initialise" – Calls launch_motor_control_node() to start the motor control script.
  - "Camera View" – Reserved for integrating a camera feed.
  - "Recovery Mode" – Sends a "RECOVER" command.
  - "Turnaround" – Sends a "Turn" command.
- **Screens:**
  - self.Camera_Screen: An OpenGL widget to display a camera feed.
  - self.path_planner: A PathPlanner instance for point selection.
- **Title:**
  - self.T_roverctrl: A text browser displaying "Rover Control".

7.5 Lidar Initialization: *{Yogachand pasupuleti (41588)}*

To initialize and launch the TiM781 LiDAR sensor within the ROS 2 environment, an existing launch file, sick_tim_7xx.launch, provided within the sick_scan_xd package, was utilized. The process involved creating a ROS 2 workspace and leveraging the package's built-in launch capabilities. The implementation steps were as follows:

- **Workspace Creation and Package Setup:** A ROS 2 workspace was created, and, as shown in Figure 32, the sick_scan_xd package was sourced. This provided access to the necessary driver, executable (sick_generic_caller), and launch files for the LiDAR.

- **Launch File Identification and Configuration:** The desired launch file, sick_tim_7xx.launch, was identified within the launch directory of the sick_scan_xd package. This file was designed to automate the process of starting the sick_generic_caller node, which is essential for interfacing with the sensor. Furthermore, during the hardware setup, LiDAR's IP address was discovered. To ensure this IP address was used by default for the LiDAR, it was configured within the sick_tim_7xx.launch XML launch file. This configuration is detailed in Figure 4.2.6, which illustrates how the IP address was discovered when initializing the device in network.

- **Launch File Execution:** The launch file was executed using the ros2 launch sick_scan_xd sick_tim_7xx.launch command. This command initiated the ROS 2 launch system, which parsed the launch file and started the specified nodes.

- **LiDAR Data Visualization and Verification:** Upon successful execution of the launch command, the sick_generic_caller node was initiated, and the LiDAR sensor began publishing data. To visualize and verify this data, the RViz2 tool was utilized. By adding a display option and changing the display type to "LaserScan," the LiDAR data could be observed in real-time. This visualization, as shown in the figure below, confirmed proper sensor initialization and communication within the ROS 2 environment.

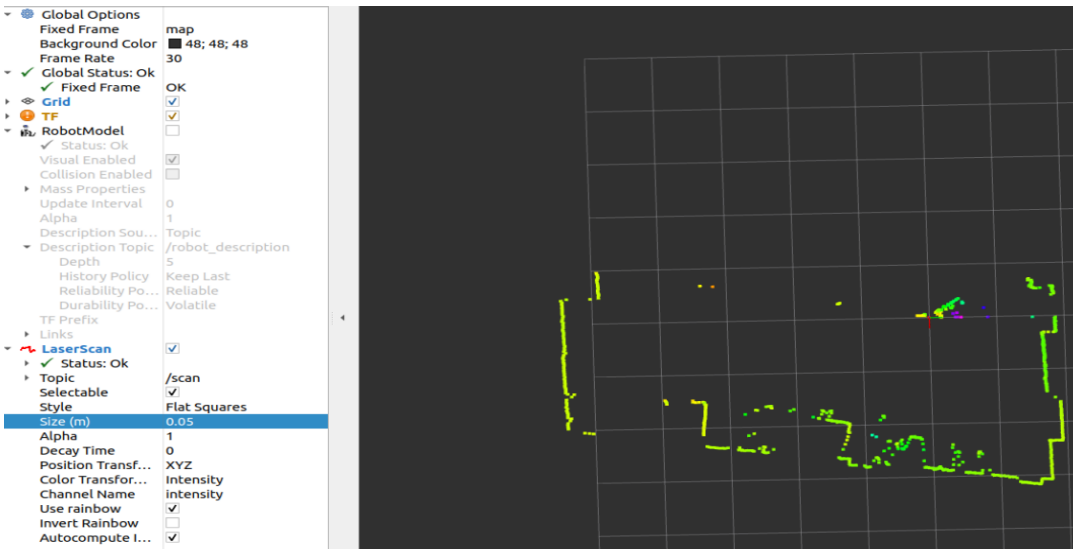

Figure 42. Rviz LaserScan Visualization of LiDAR Data

- The resulting visualization in RViz2 offers a real-time representation of the LiDAR's perception of the Hochschule Bremerhaven lab environment where we implemented the LiDAR. As shown in the image above, this is real-time LiDAR scan data visualized in RViz, enabling interpretation of the scan data.

7.5.1 Static Map Generation: -*{Yogachand pasupuleti (41588)}*

To facilitate early visualization and sensor verification, a static map was generated. This was achieved by applying a static transformation between the map, base_link, and odometry frames, populating the space with LiDAR sensor data without requiring robot movement. Specifically, the command **"ros2 run tf2_ros static_transform_publisher 0 0 0 0 0 0 map base_link"** was added to the sick_tim_7xx.launch file. After the transform was applied and the launch file executed, the fixed frame in RViz was set to 'map'. To visualize the map data generated by SLAM, the topic 'map' was added to the RViz display. As shown in the figure below, this allowed for a preliminary spatial representation, enabling basic assessment of LiDAR functionality.

To validate the mapping approach further, full SLAM functionality, as discussed in section 7.9, was employed to generate a dynamic map within a simulated environment.

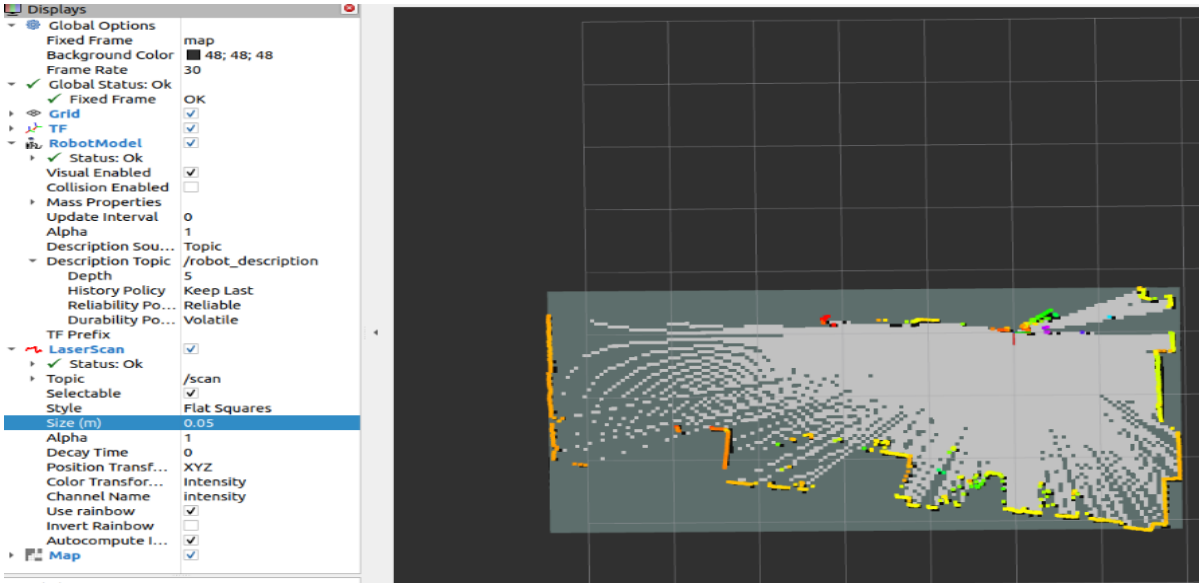

Figure 43. Static map visualization in RViz

7.6 Safety Stop Node for Robot Obstacle Avoidance. *{yogachand pasupuleti (41588)}*

The software implementation of the safety_stop_node, a ROS 2 node designed to enhance robot safety by detecting and responding to potential collisions using laser scan data. The node monitors the robot's immediate surroundings, specifically within a configurable angular range, and triggers safety stop signals based on proximity to obstacles. It also provides visual feedback through ROS 2 visualization tools, aiding in debugging and understanding the robot's environment.

Node Functionality

The safety_stop_node employs a straightforward architecture centered around a laser scan subscriber, safety stop and object distance publishers, and visualization publishers. The core functionality is implemented through a state machine, transitioning between FREE, WARNING, and DANGER states based on configurable distance thresholds.

- **Data Flow:** The node subscribes to the LaserScan topic, processes the data to identify the closest obstacle within a defined angular range, and publishes safety stop signals (Bool), object distance (Float32), and visualization markers (Marker, MarkerArray).
- **State Machine:** The state machine provides a clear and robust method for managing safety responses.

- The node starts in the FREE state, transitions to WARNING when an obstacle is detected within the warning distance, and enters DANGER when an obstacle is within the danger distance. The state dictates the safety stop signal and visualization updates.
- **Parameterization:** The node is highly configurable through ROS 2 parameters, allowing users to adjust safety thresholds, angular range, and topic names. This flexibility enables adaptation to different robot platforms and environments.
- **Visualization:** The node utilizes ROS 2 visualization tools to display safety zones (warning and danger) as cylinders and the closest detected object as a sphere. The color and transparency of these markers change based on the node's state, providing intuitive feedback to the user.

Implementation Details

The safety_stop_node is implemented in Python using the rclpy library for ROS 2 communication. Key aspects of the implementation include:

- **Laser Scan Processing:** The laser_callback function processes incoming LaserScan messages. It calculates the indices corresponding to the defined angular range, iterates through the scan data, filters out invalid and close-range readings, and determines the minimum distance to an obstacle.
- **State Transition Logic:** The laser_callback function implements the state machine logic. It compares the minimum distance to the configured warning and danger thresholds to determine the appropriate state.
- **Safety Stop and Object Distance Publishing:** The safety_stop_pub and object_distance_pub publishers transmit the safety stop signal (boolean) and the closest object distance (float), respectively. If no object is detected, -1.0 is published for the distance.
- **Visualization Marker Creation and Update:** The create_markers function initializes the visualization markers. The update_markers function dynamically updates the marker properties based on the node's state. The color and transparency of the markers are adjusted to reflect the current safety level. The object marker is created at the location of the closest object, based on the distance and angle data from the LaserScan. As shown in the figure below, these markers can be visualized in RViz by adding the display setting and setting it to MarkerArray. This allows for real-time visual feedback of the safety zones and detected obstacles."

Key Changes and Additions:

- "As shown in the figure below": This directly refers to the figure depicting the visualization.

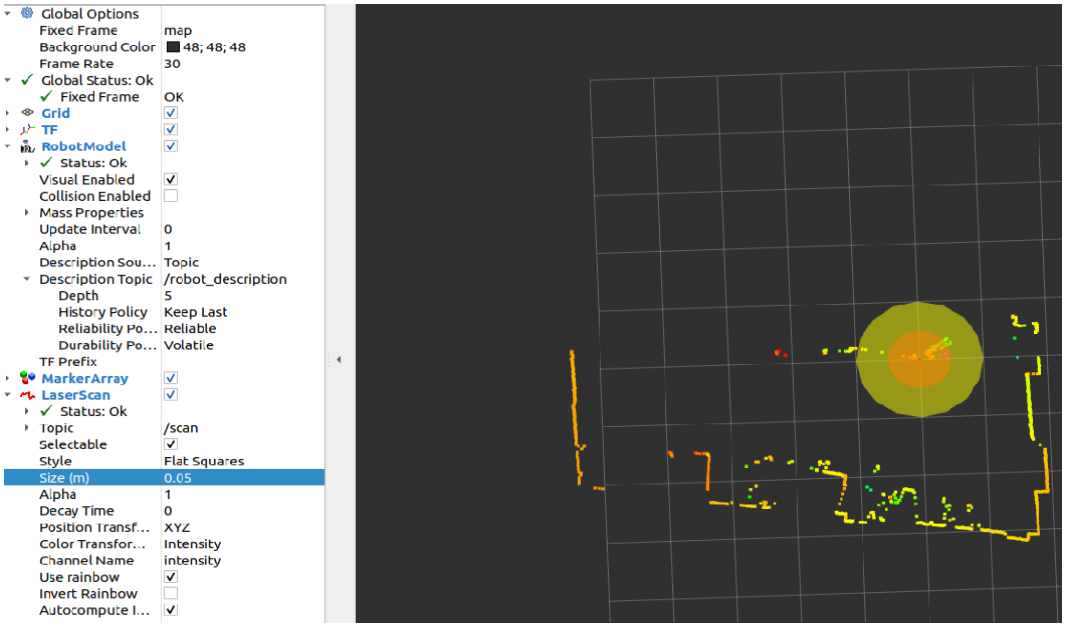

Figure 44. RViz visualization of safety markers

- "These markers can be visualized in RViz by adding the display setting and setting it to MarkerArray": This provides explicit instructions on how to view the markers in RViz, which is very helpful for the reader.
- "This allows for real-time visual feedback of the safety zones and detected obstacles.": This reinforces the purpose of the visualization and its practical application.
- This revised section now provides a clear connection to the figure and offers practical guidance for visualizing the markers in RViz.

- **Error Handling:** The code includes checks for invalid and infinite laser scan ranges, ensuring robust operation in various environments. The code will ignore laser scan readings that are below the robot_clearance_distance to stop the robot from stopping because of its own chassis or other close by robot parts.

Code Structure and Best Practices

The code is organized into a modular structure, with separate functions for specific tasks. This promotes code readability, maintainability, and reusability. The use of clear variable names and comments enhances code understanding. The node adheres to ROS 2 conventions for node creation, parameter handling, and message publishing.

Testing and Validation

The safety_stop_node underwent thorough testing and validation to ensure its reliability and effectiveness.

Simulated Environment Testing:

- Initial testing was conducted in a simulated environment using Gazebo, allowing for controlled experimentation and rapid iteration.
- This included verifying obstacle detection, state transitions, output accuracy, and visualization.

Real-Robot Validation:

- Following successful simulation testing, the node was validated on a physical robot platform.
- The node was checked in real world scenarios and the safety stop functionality was successful.
- This validation confirmed the node's performance and suitability for real-world deployment.

7.7 Simulated SLAM Implementation and Launch Configuration in Gazebo: *{yogachand pasupuleti 41588 & Shamanth B Adiga (41527)}*

This section details the implementation and configuration of the Simultaneous Localization and Mapping (SLAM) system within a Gazebo simulation environment. The slam_toolbox node was utilized for map generation and robot localization, and a custom launch file automated the node's startup and parameterization. This approach allowed for controlled testing and validation of the SLAM system before real-world deployment. To initiate SLAM, first load the robot's URDF, as detailed in section 7.3, defining its physical model within the simulation. Once the robot model is loaded, execute the SLAM launch file to begin map generation, leveraging the slam_toolbox node to simultaneously create a map of the environment and determine the robot's location within it.

Dynamic TF frame: -

The SLAM process relies heavily on dynamic TF (Transform Frame) frames, which are crucial for accurately representing the robot's pose and sensor data relative to the map. These dynamic TF frames, as detailed in Section 4.1.8, are essential for the ongoing localization and mapping performed by the slam_toolbox node. This ensures that the robot's position and sensor data are consistently updated and aligned with the generated map. The figure below shows the dynamic TF frames of a robot within the simulation environment.

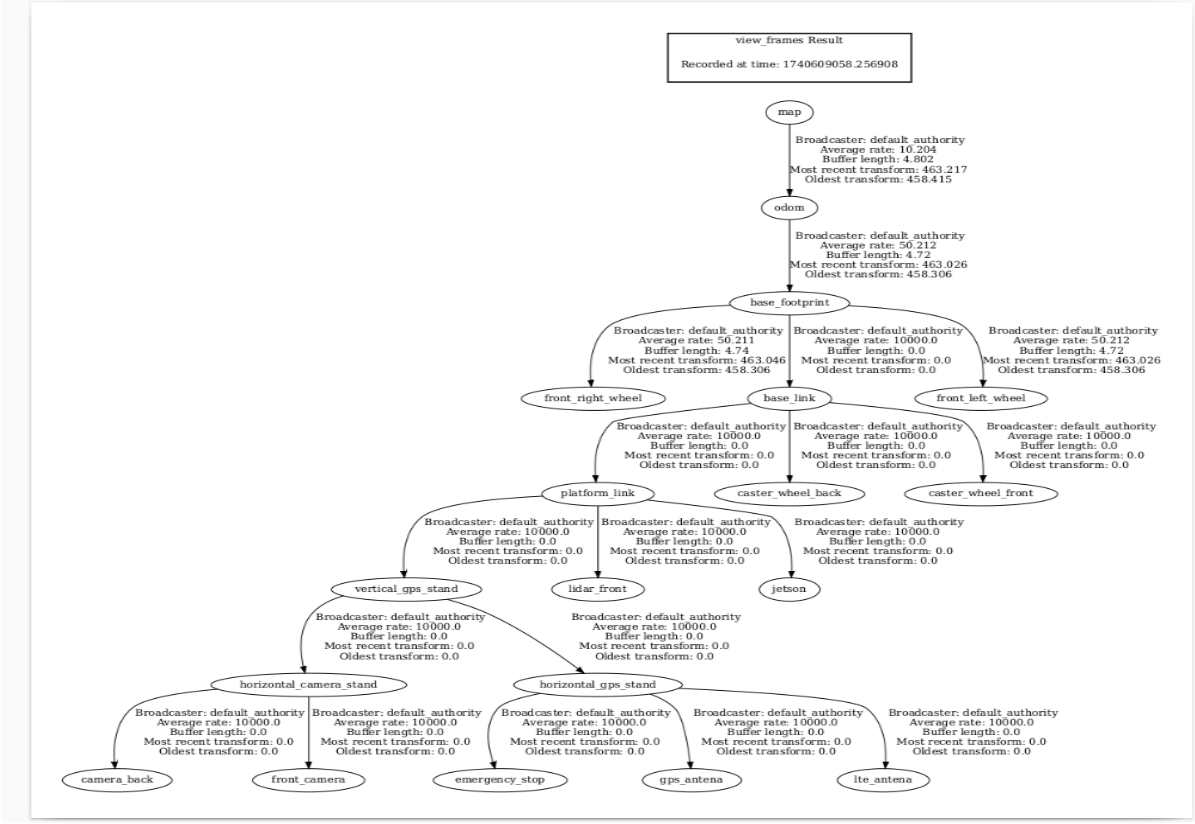

Figure 45. Tree of simulated robot

The structure of the TF tree used in the mapping process is presented in, with explanations provided for some of the main frames.

• **map frame:** A global frame that represents a stationary reference on the map where all static objects are assumed to be fixed.

• **odom frame:** A local frame that represents the position and orientation of the robot based on odometry, which can move over time and provides relative motion information.

• **base_footprint frame:** This is usually a two-dimensional projection of the robot on the ground. It's the reference for the robot's contact with the floor or the ground.

• **base_link frame:** This is the robot's main body frame, from which positions and motions of all other parts of the robot are defined.

• **platform_link frame:** This frame is associated with a laser scanner or LIDAR sensor on the robot. It's the reference point from which scan data is measured.

Teleop_twist_keyboard:

To generate a map during SLAM in simulation, the teleop_twist_keyboard ROS package, launched in another terminal, employed keyboard inputs translated into Twist messages for robot control.

- i: Forward
- k: Stop
- j: Left (turn left)
- l: Right (turn right)

Simulation Environment and SLAM Implementation:

A realistic Gazebo simulation environment was established, incorporating a robot model equipped with a simulated laser scanner. This environment allowed evaluation of the slam_toolbox node's performance under controlled conditions

SLAM Toolbox Configuration:

- slam_toolbox node configured for "mapping" mode (2D occupancy grid map generation).
- Simulated laser scan data from Gazebo (/scan topic) as primary sensor input.
- Map resolution set to 0.05 meters (balance of detail and computational load).
- Loop closure functionality enabled and tuned (map drift correction).
- Ceres Solver used for map and robot pose optimization.
- Scan matching, correlation, and loop detection parameters adjusted for maximum mapping accuracy.

Visualization in RViz:

- RViz was used to visualize the generated 2D occupancy grid map, the robot's pose, and the laser scan data in real-time. This provided valuable feedback on the SLAM algorithm's performance.
- Global frame set to "map" in RViz.
- "map" topic added to visualize the generated map.
- "scan" topic added to visualize the laser scanner's perception.
- Map, robot pose, and laser scan data visualized.

Launch File Automation:

- To simplify the startup and configuration of the slam_toolbox node, a ROS 2 launch file was developed.

Automated Node Startup and Configuration:

- The launch file automates the startup of the async_slam_toolbox_node as a lifecycle node, enabling controlled state transitions.
- Launch arguments configure the node, including autostart, use_lifecycle_manager, use_sim_time, and slam_params_file.
- The launch file automatically loads the necessary parameter file, ensuring consistent configuration.
- The launch file enables the use of simulation time.

Lifecycle Management:

- Conditional logic was implemented to manage the node's lifecycle, automatically configuring and activating the node based on launch arguments.

Simulation Results and Validation:

Within the Gazebo simulation, the slam_toolbox node successfully achieved:

- Accurate generation of 2D occupancy grid maps.
- Reliable robot localization within the simulated map.
- Effective loop closure detection and correction.

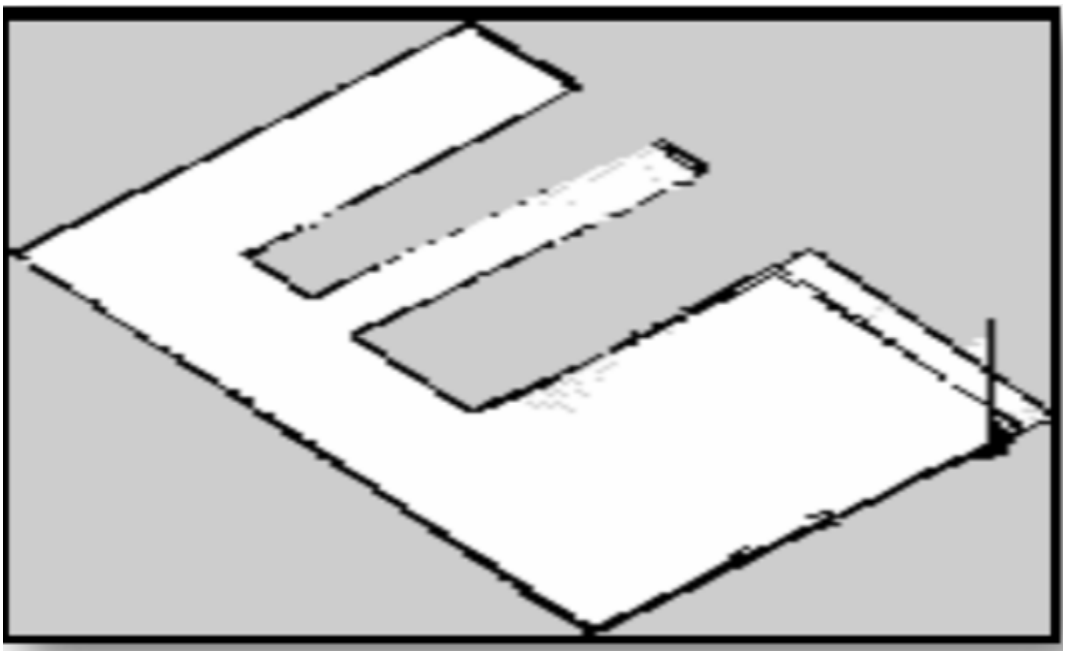

Figure 46. 2D map of lab generated using SLAM

7.8 Simulated Navigation Stack Implementation and Configuration

{Yogachand pasupuleti & Shamanth B Adiga(41527)}

This section details the implementation and configuration of the robot's navigation stack within our Gazebo simulation environment. We utilized the nav2_bringup package to launch and configure the navigation nodes, and a comprehensive YAML parameter file to fine-tune the navigation behavior. This setup allowed for thorough testing and validation of the navigation system in a controlled, simulated environment.

Launch File Automation:

To streamline the startup and configuration of the Nav2 navigation stack, we employed a ROS 2 launch file. This file automates the following key processes:

Node Launch and Lifecycle Management:

- The launch file manages the lifecycle of various Nav2 nodes, including the controller server, planner server, behavior server, and more.
- It supports both composed and non-composed node execution, providing flexibility in system architecture.
- It uses lifecycle nodes, which allow for a controlled startup, configuration, and shutdown of the navigation stack.

Figure 47. Lifecycle Nodes

Parameter Loading:

- The launch file loads the navigation parameters from a specified YAML file, ensuring consistent configuration across all nodes.
- It allows for the remapping of topics.
- It allows to use simulation time.

Simulation Time Integration:

- The launch file is configured to use simulation time from Gazebo, ensuring accurate timing and synchronization within the simulated environment.

Configuration of the Nav2 Navigation Stack (YAML File)

The nav2_params.yaml file serves as the central configuration repository for the Nav2 navigation stack within our Gazebo simulation. It allows for granular control over each component's behavior, ensuring optimal performance for our robotic platform. While the structure appears similar across sections, each addresses a distinct aspect of navigation.

Key Configuration Sections and Their Roles:

1. **amcl (Adaptive Monte Carlo Localization):**
   Purpose: Configures the robot's localization system, enabling it to estimate its position and orientation within the map.
   a. **Key Parameters:**
      i. alpha1-5: Motion model noise parameters, influencing how the robot's movement is modeled.
      ii. laser_model_type, sigma_hit, z_hit, etc.: Laser scan likelihood parameters, defining how laser measurements are used to update the position estimate.
      iii. max_particles, min_particles: Particle filter parameters, controlling the accuracy and computational cost of localization.
      iv. scan_topic: the laser scan topic to subscribe to.
   b. **Significance:** Accurate localization is fundamental for reliable navigation. This section ensures that the robot can accurately determine its position in the map.
2. **bt_navigator (Behavior Tree Navigator):**
   a. **Purpose:** Configures the behavior tree-based navigation system, defining the robot's high-level navigation strategies.
   b. **Key Parameters:**
      i. navigators: Defines the available navigation plugins (e.g., navigate_to_pose).
      ii. Plugin configurations: Parameters specific to each navigation plugin.
      iii. error_code_names: Defines custom error codes for the behavior tree.
   c. **Significance:** Behavior trees provide a flexible and modular way to define complex navigation behaviors. This section allows us to customize the robot's decision-making process.
3. **controller_server:**

a. **Purpose:** Configures the robot's motion controller, responsible for executing planned paths.
b. **Key Parameters:**
    i. controller_frequency: Defines the control loop frequency.
    ii. goal_checker_plugins, progress_checker_plugins: Specifies plugins for checking goal completion and progress.
    iii. MPPI Controller parameters: time steps, batch size, noise parameters, cost functions parameters.
c. **Significance:** The controller server ensures that the robot follows planned paths accurately and smoothly. The MPPI controller is finely tuned for optimal performance.

4. **local_costmap and global_costmap:**
    a. **Purpose:** Configures the local and global costmaps, which represent the robot's environment as a grid of cost values.
    b. **Key Parameters:**
        i. resolution, width, height: Defines the costmap's dimensions and resolution.
        ii. plugins: Specifies the costmap layers (e.g., voxel_layer, inflation_layer).
        iii. Layer-specific parameters: Defines the behavior of each costmap layer, such as inflation radius and sensor observation sources.
        iv. scan: configuration for the scan topic.
    c. **Significance:** Costmaps are essential for path planning and obstacle avoidance. These sections ensure that the robot has an accurate and up-to-date representation of its surroundings.

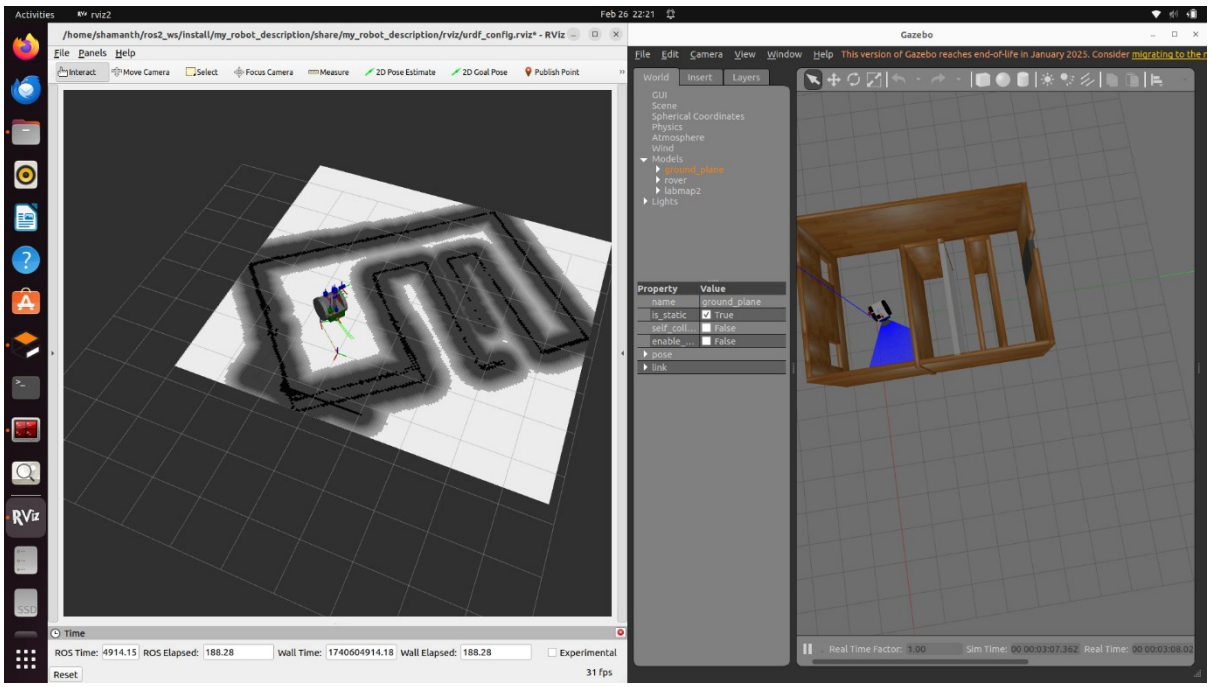

Figure 48. Point to point navigation using a preloaded map to NAV2 stack

5. **planner_server:**
    a. **Purpose:** Configures the path planner, which generates paths for the robot to follow.
    b. **Key Parameters:**
        i. planner_plugins: Specifies the path planning algorithms (e.g., GridBased).
        ii. Plugin-specific parameters: Defines the behavior of each planner plugin, such as tolerance and search heuristics.
    c. **Significance:** The planner server determines the optimal path for the robot to reach its goal.
6. **smoother_server:**
7.
    a. **Purpose:** Configures the path smoother, which refines the planned paths to improve smoothness.
    b. **Key Parameters:**
        i. smoother_plugins: Specifies the smoothing algorithms.
        ii. Plugin-specific parameters: Defines the behavior of each smoother plugin.
    c. **Significance:** Smoother paths improve the robot's motion and reduce wear and tear.
8. **behavior_server:**
    a. **Purpose:** Configures the recovery behaviors, which are used to handle navigation failures.
    b. **Key Parameters:**
        i. behavior_plugins: Specifies the available recovery behaviors (e.g., spin, backup).
        ii. Plugin-specific parameters: Defines the behavior of each recovery plugin.
    c. **Significance:** Recovery behaviors ensure that the robot can handle unexpected situations and continue navigating safely.

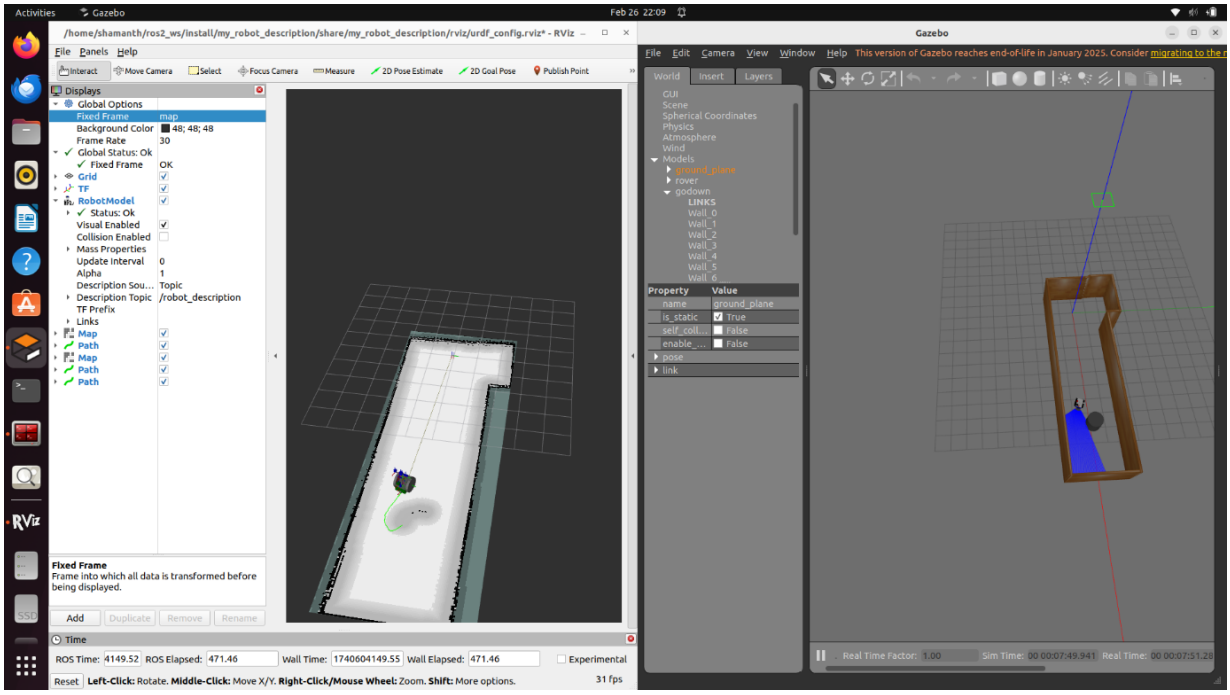

Figure 49. Observing Simultaneous Mapping by SLAM and autonomous driving feature by NAV2

9. **waypoint_follower:**
    a. **Purpose:** Configures the waypoint follower, which allows the robot to navigate to a sequence of waypoints.
    b. **Key Parameters:**
        i. loop_rate, action_server_result_timeout: parameters for the waypoint following action.
        ii. waypoint_task_executor_plugin: plugin for the waypoint follower.
    c. **Significance:** Allows the robot to follow a set of predefined poses.
10. **velocity_smoother:**
    a. **Purpose:** Configures the velocity smoother, which smooths the robot's velocity commands.
    b. **Key Parameters:**
        i. smoothing_frequency, max_velocity, max_accel: parameters for the velocity smoother.
    c. **Significance:** Smooth velocity commands improve the robot's motion and reduce wear and tear.
11. **collision_monitor:**
    a. **Purpose:** Configures the collision monitor, which detects and prevents collisions.
    b. **Key Parameters:**
        i. polygons: parameters for the collision zones.
        ii. observation_sources: sensor topic to use.
    c. **Significance:** Prevents the robot from colliding with obstacles.

12. **docking_server:**
    a. **Purpose:** Configures the docking server, which enables the robot to autonomously dock.
    b. **Key Parameters:**
        i. Parameters for the docking process.
        ii. Plugin configurations.
        iii. Controller configuration.
    c. **Significance:** Allows the robot to autonomously dock.
13. **loopback_simulator:**
    a. **Purpose:** Configures the loopback simulator.
    b. **Key Parameters:**
        i. Frame ids.
        ii. Update duration.
    c. **Significance:** Allows for testing of the navigation stack.

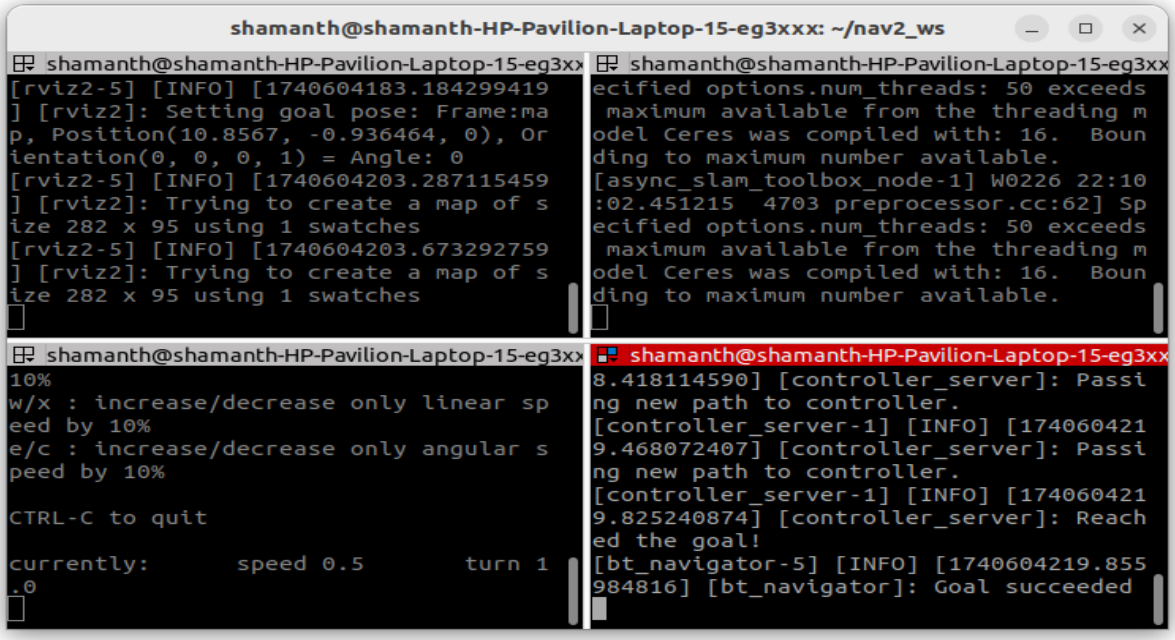

Figure 50. A screenshot representing GOAL SUCCEEDED info giving by NAV2

7.9 Implementing ROS 2 driver for Swift Navigation's GNSS/INS receivers – *{Niranjan Channayya (41559)}*

Using Swift Navigations GNSS standalone position system and DGPS RTK positioning system helps us in determining the Global positioning of our Autonomous rover. The latter, gives us centimetre-level accuracy in positioning the rover with respect to the Base antenna.

- **GNSS** (Global Navigation Satellite System) provides absolute positioning using satellite signals. This is the standalone positioning referred above.
- **RTK (Real-Time Kinematic)** positioning enhances GNSS data by using a base station to provide correction data, enabling centimetre-level accuracy.

We already have a ROS2 driver to implement this system in ROS2 environment. We have built and installed this driver. [34]

The driver receives Swift binary (SBP) messages and publishes the following ROS topics:

1. **GpsFix** – (*gps_msgs/msg/GPSFix)*, This topic publishes comprehensive and high-precision GPS data necessary for precise localization of rover such as- RTK Positioning Data, Orientation Estimates, Velocity Measurements and Error and Diagnostic Metrics.

2. **NavSatFix** – *(sensor_msgs/msg/NavSatFix)*, This topic publishes **GNSS** data including latitude, longitude, and altitude which can be used for Localization, Path planning and sensor fusion.

3. **TwistWithCovarianceStamped** – *(geometry_msgs/msg/TwistWithCovarianceStamped),* This message type is used to represent velocity data (both linear and angular) along with an associated covariance matrix that indicates the uncertainty in the measurements. This message is commonly used in applications such as localization, navigation, and sensor fusion, especially when working with systems like IMU, odometry, and GNSS/INS.

4. **Baseline** – *(swiftnav-ros2/msg/Baseline)*, This is specifically used for publishing baseline information between the two antennas. The baseline in this context refers to the relative distance and orientation vector between a base station and a rover receiver, typically represented in the North-East-Down (NED) coordinate system. This is essential to achieve highly precise RTK positioning.

5. **TimeReference** – *(sensor_msgs/msg/TimeReference),* This is used to synchronize the system clock with an external time source. This message is particularly useful when integrating systems that rely on external time synchronization.

6. **Imu –** *(sensor_msgs/msg/Imu),* This topic publishes the data such as Linear acceleration, angular velocity and orientation from the IMU sensor.

By using the data obtained from above topics with combination of wheel odometry data, it is possible to achieve an autonomous navigation rover in outside environment.

Below is shown the implementation of ROS2 driver for Swift piksi on the processor-

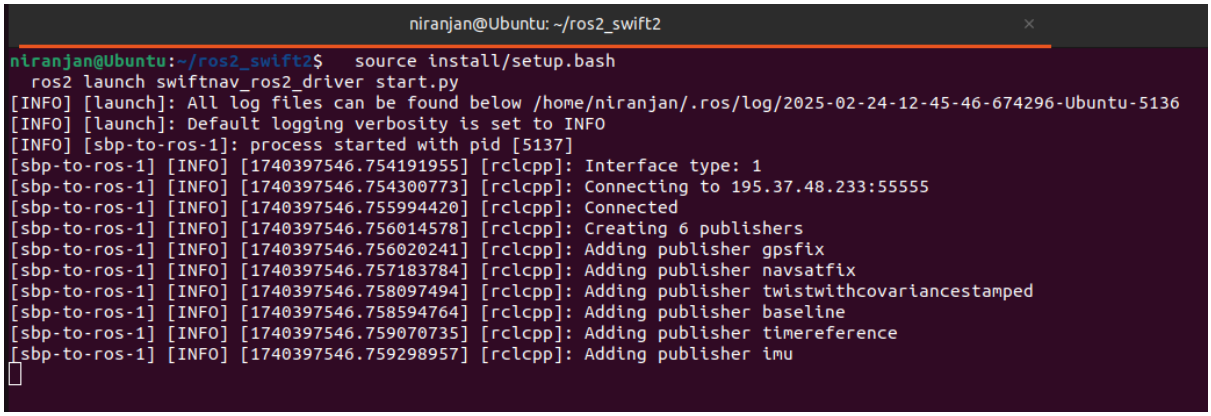

Figure 51. Launching the ROS2 driver for swift navigation

After launching the ROS2 driver, we can observe that the computer establishes connection with the Swift hardware unit via TCP/IP. The IP address is *192.37.48.233* and the port is *55555*.

After the connection is successful, the package creates six publishers as discussed before. Now the topics are ready to be used.

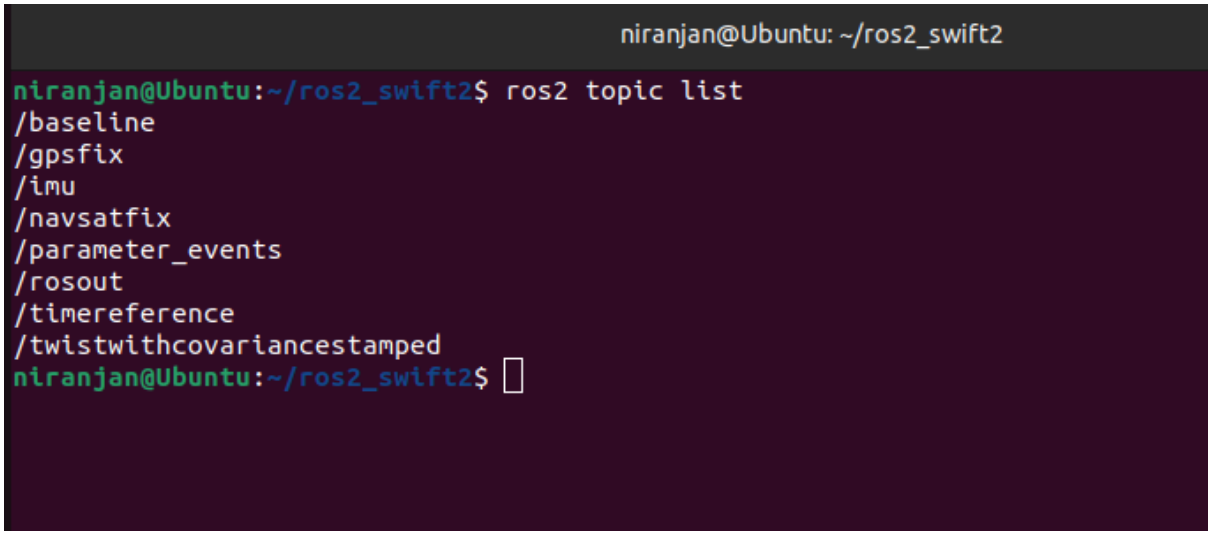

Figure 52. Topic list of ROS2 driver for swift navigation

Now we can use these published data for Autonomous rover applications.

Getting Swift Piksi Standalone Position (GNSS)–

This is using GPS data – Latitude, Longitude and Altitude data to determine the Rover position which is required for path planning. The *GpsFix* can give us this data.

We have obtained the Standalone data of two paths we traced around the campus of Hochschule Bremerhaven.

The visualization of the collected GPS data showcases the potential of this approach for outdoor navigation tasks. The results validate the capability of a rover to autonomously follow a predefined path using the GNSS data but with few downsides. Which is discussed further.

Below are the plots of GPS data gathered while tracing paths in and around the Hochschule.

The obtained coordinates are plotted using MATLAB on Google maps using *geoplot* function.

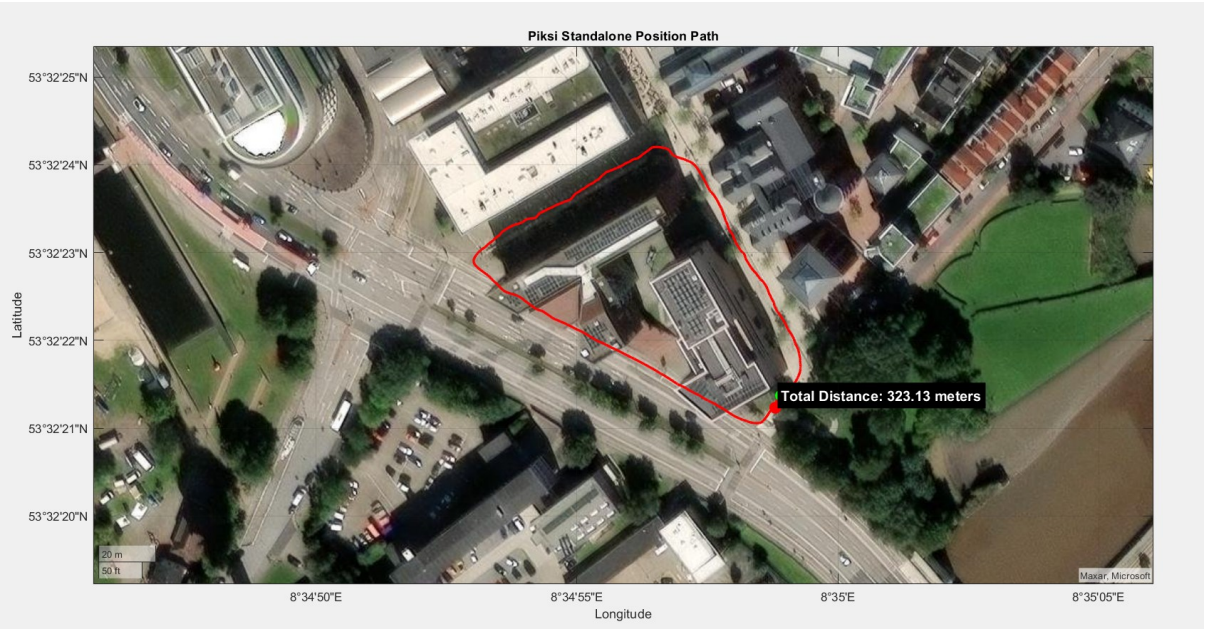

Figure 53. Swift module traced around T building, Hochschule Bremerhaven

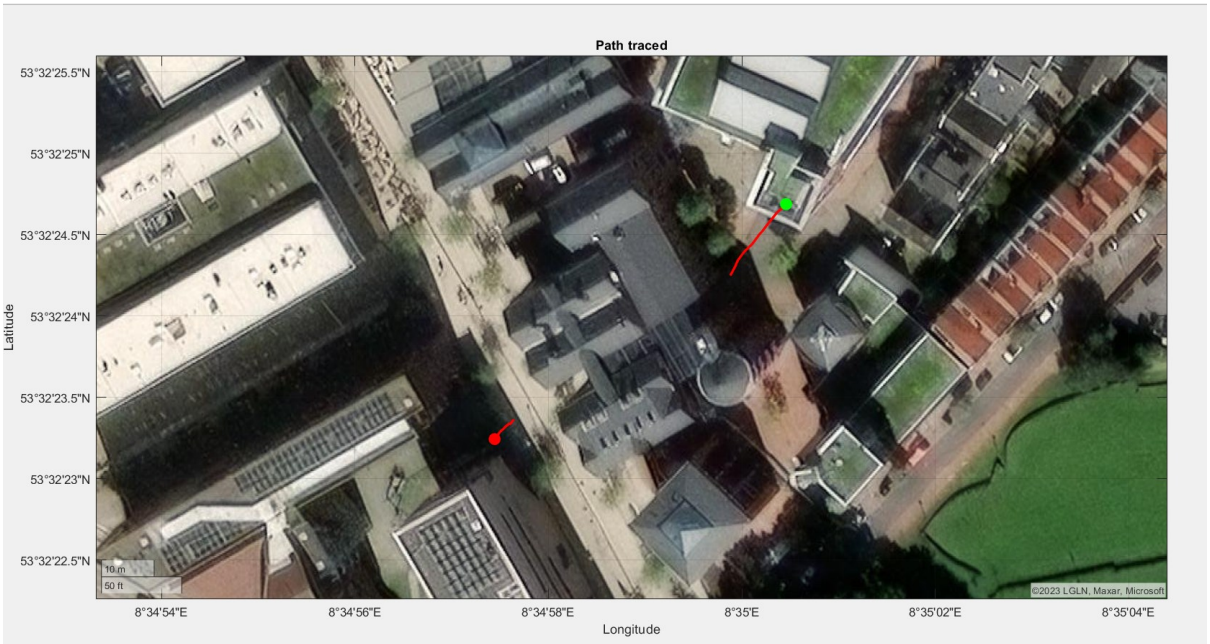

Figure 54. Swift module traced path from Z building to T building through K building at Hochschule Bremerhaven

The path closely matches the actual geographical features seen in the satellite image. Sharp turns and corners are captured, reflecting the GNSS receiver's ability to provide real-time positional updates. Minor deviations from the expected path can be attributed to GNSS signal noise and multipath errors due to reflections from nearby buildings.

But, in the second plot where we go from Z building to T building through K building, we can clearly see that no data has been captured since it was in an indoor environment. In these kinds of scenarios where we will have to go through paths where there is no GNSS signal, we must use IMU for dead-reckoning. Another, downside of using only GPS data is there is some deviations from actual path as well. This problem can be addressed by using the RTK technology provided by Swift Piksi where we will be able to obtain centimetre-level accuracy.

Limitations and solutions for using Swift Piksi Standalone position can be summarized as follows-

Limitation	Description	Solution
GNSS Signal Drift	Signal noise can cause minor deviation	Implement GNSS + Odometry fusion for better accuracy
Path Errors	Signal reflections from buildings can affect accuracy	Use RTK corrections for centimetre-level precision
Signal Loss	Poor signal quality in dense urban areas	Integrate IMU for dead-reckoning during outages
Lack of Orientation data	GNSS doesn't provide orientation information	Combine with wheel odometry or magnetometer

Table 12. Limitations and Solutions in using Piksi Standalone position

Swift Piksi Real-time Kinematics (RTK positioning)

We were not able to use Base antenna due to technical problems, so the simulator optionality of Swift piksi is been used to demonstrate the RTK approach.

The *baseline* topic publishes all the data related to RTK positioning.

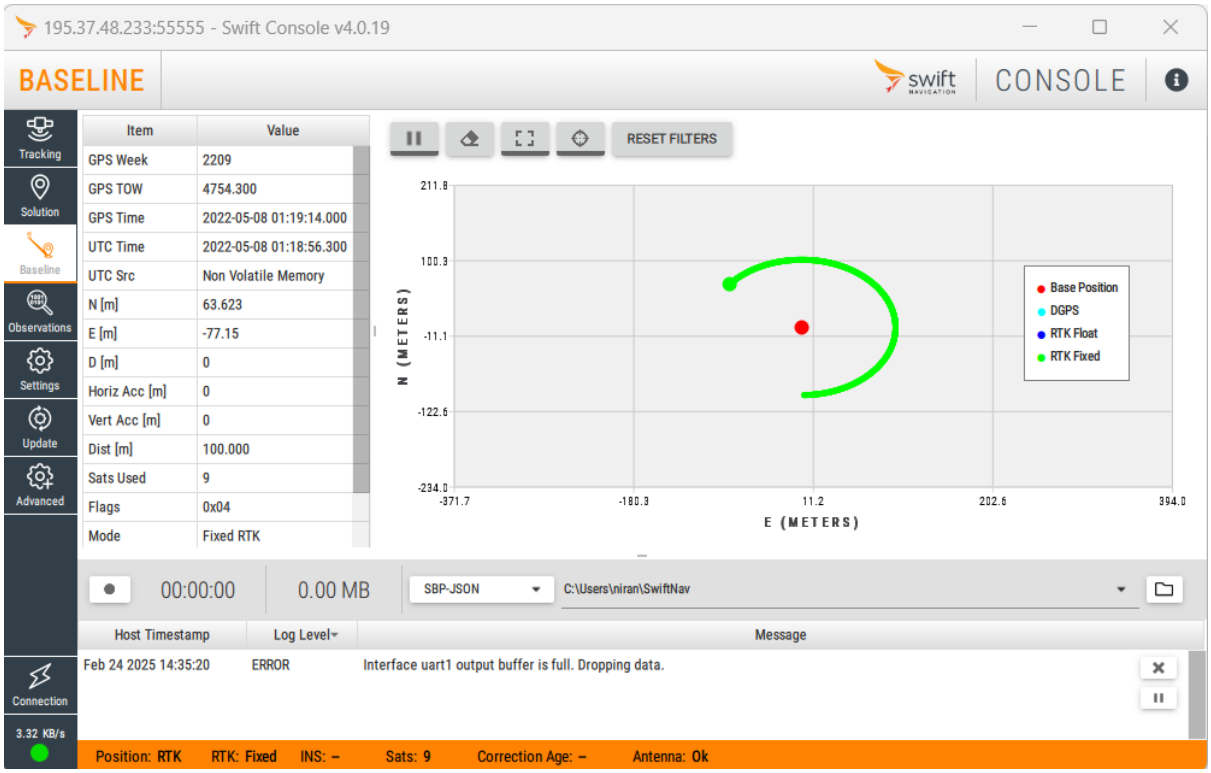

Figure 55.Simulated RTK position in swift console

```
---
header:
  stamp:
    sec: 1651972236
    nanosec: 0
  frame_id: swiftnav-gnss
mode: 4
satellites_used: 9
baseline_n_m: 96.139
baseline_e_m: 27.52
baseline_d_m: 0.0
baseline_err_h_m: 0.0
baseline_err_v_m: 0.0
baseline_length_m: 100.00028860458353
baseline_length_h_m: 100.00028860458353
baseline_orientation_valid: true
baseline_dir_deg: 15.973885873594108
baseline_dir_err_deg: 0.0
baseline_dip_deg: 0.0
baseline_dip_err_deg: 0.0
---
```

Figure 56. The Baseline Publisher

The information from *baseline* topic gives us significant information such as Baseline N, Baseline E, Baseline D which is the distance in northward direction, eastward direction and downward direction from the base station.

The rover can use this data to compute its exact position relative to the base station using the following formula:

$$Baseline\ length = \sqrt{(North)^2 + (East)^2 + (Down)^2}$$

7.10 Detailed Procedure & Troubleshooting

7.10.1 Motor Testing – {*Krish. Shah* (41564)}

The motor testing phase was one of the most critical steps in the development process as it formed the basis of the rover's motion control system. The motors, controlled by the **RoboClaw 2x60A Motor Controller**, do not **natively support the ROS2 architecture**. Instead, **RoboClaw provides a Python3 library for direct serial communication**, meaning the motors had to be interfaced using custom solutions or adapted drivers.

❖ **Initial Setup and Testing**

The first step was to validate the **basic functionality** of the motors and the motor controller outside ROS2. This was conducted on a **Windows PC** using **RoboClaw Motion Studio**, an official configuration and debugging tool provided by **RoboClaw**. Motion Studio allows:

- Direct connection to the RoboClaw controller over USB.
- Real-time adjustment and tuning of **PID parameters**.
- Monitoring of motor current, encoder counts, and battery voltage.
- Sending manual speed and position commands to the motors to confirm responsiveness.
- Diagnosing errors such as **encoder faults** and **motor overcurrent protection**.

❖ Configuring the Motors in Motion Studio

Using Motion Studio, the following configurations were performed:

- Setting the **motor type and gear ratio**.
- Configuring the **encoder type** (quadrature encoders in this case).
- Defining **PID parameters** for speed control.
- Configuring the **deadband (response threshold)** to eliminate motor jitter.
- Enabling **closed-loop speed control** (though later reliability issues with encoders reduced reliance on this).

This allowed initial validation that both motors rotated correctly, encoders counted accurately, and motor currents were within safe operating ranges. [25]

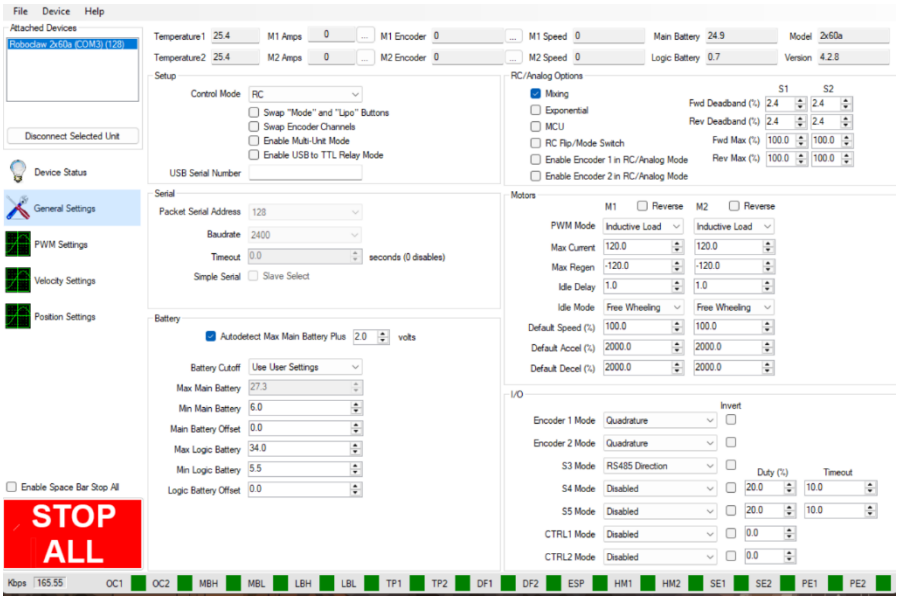

Figure 57. Motion Studio Snapshot

❖ Attempt to Use Third-Party ROS2 Driver

After initial hardware validation, an effort was made to use a **third-party ROS2 package** developed by **Wimble Robotics** to directly integrate RoboClaw into the ROS2 ecosystem. The intention was to avoid writing a custom node from scratch by leveraging this prebuilt driver. [33]

However, several issues were encountered with the **Wimble Robotics driver**:

- **CRC Miscalculation Errors:** The driver performed cyclic redundancy checks (CRC) on serial data packets received from RoboClaw, but these checks repeatedly failed, resulting in communication breakdown. This was traced to incorrect implementation of the CRC calculation logic within the driver, which had to be manually corrected.

- **Serial ReadByteWidth Error:** After rectifying the CRC logic, the driver continued to throw a low-level serial error related to ReadByteWidth, which could not be consistently reproduced or resolved. Despite several debugging attempts involving buffer size adjustments, read timeout tuning, and manual retries, this error persisted, making the driver unreliable for continuous motor control.
- **Inconsistent Encoder Reads:** Even in cases where the driver appeared to initialize correctly, encoder data was often read as corrupt or out-of-bounds, introducing additional instability.

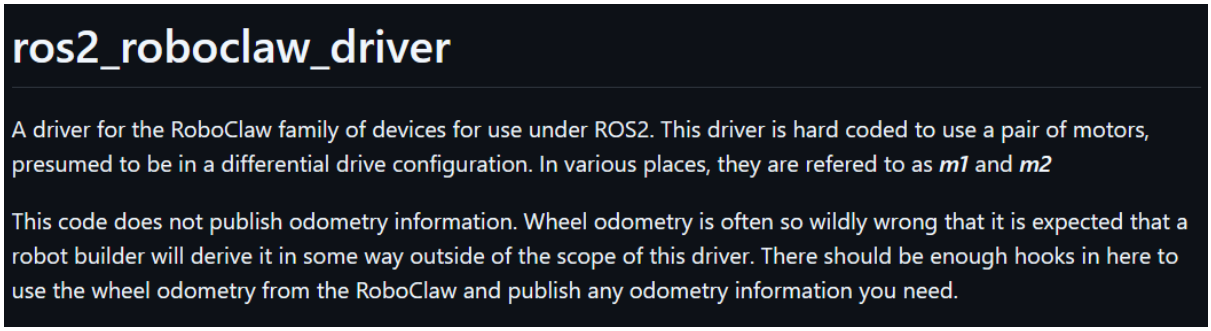

Figure 58. 3rd part Roboclaw Driver [33]

❖ **Debugging Timeline and Process**

To resolve these issues, the following debugging actions were undertaken:

- **Comparing Driver Code with RoboClaw Documentation:** The CRC routine was rewritten directly using the official RoboClaw communication protocol documentation.
- **Direct Serial Monitoring:** Data packets were captured directly using tools like minicom to verify actual communication at the byte level, comparing it with the expected packet format.
- **Isolation Tests:** The same driver was tested on a clean ROS2 installation on Ubuntu 22.04 to rule out environmental issues (Jetson-specific quirks). The error persisted.
- **Reverting to Baseline:** Attempts were made to roll back the driver to earlier versions, but similar issues were encountered, suggesting a persistent flaw either in the serial handling or the low-level timing assumptions.

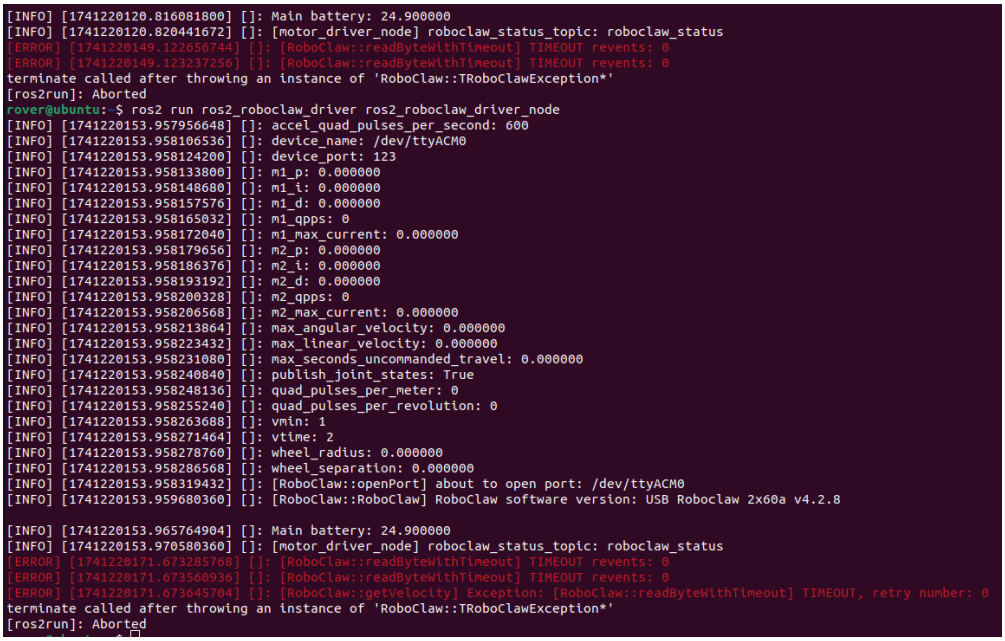

Figure 59. ReadWidthByte Error Message Indicating Serial Problem

Conclusion

Due to these unresolved serial communication problems, the **Wimble Robotics driver was deemed unreliable for production use**. This experience highlighted the risks of relying on **third-party drivers for hardware not officially supported by ROS2**.

This extensive debugging effort ultimately resulted in the decision to shift away from this driver, and the need for an **alternative motor control approach** was established.

7.10.2 URDF Making – {Krish. Shah (41564)}

The process of creating the URDF (Unified Robot Description Format) model for the autonomous rover was a **step-by-step, measurement-driven process** aimed at building a modular and realistic digital twin of the actual rover.

❖ **Measurements and Structural Mapping**

The first step involved **measuring the physical dimensions** of the rover and its individual components directly from the hardware platform. This included:

- **Overall chassis length, width, and height.**
- Dimensions and mounting points for the **motors** and **tracks.**
- Positions and orientations of **sensors** such as LiDAR and cameras.
- The vertical and lateral offset between the rover's base and its sensors.

These measurements were recorded and formed the basis for the **link dimensions** in the URDF.

❖ **Modular URDF Creation**

Rather than creating a monolithic URDF file, the design followed a **modular approach**, where each subsystem (platform, support structure, sensor mounts) was defined as a separate **Xacro macro**. These modular pieces were then assembled into the complete rover description.

This modularity ensured:

- **Easy modification:** Individual subsystems could be adjusted independently.
- **Reusability:** The same Xacro files could be adapted if hardware changed (e.g., swapping LiDAR for a different model).
- **Clarity:** Each file focused on a single physical section, improving readability and reducing error-prone edits.

❖ **Upgrading to Xacro**

Once the basic structure was created using static URDF, the file was **refactored into Xacro macros** to enable parameterization. This allowed:

- Changing component dimensions from a **single place**.
- Toggling sensor mounts on/off without restructuring the entire file.
- Reusing the same part definitions across simulation and real-world models.

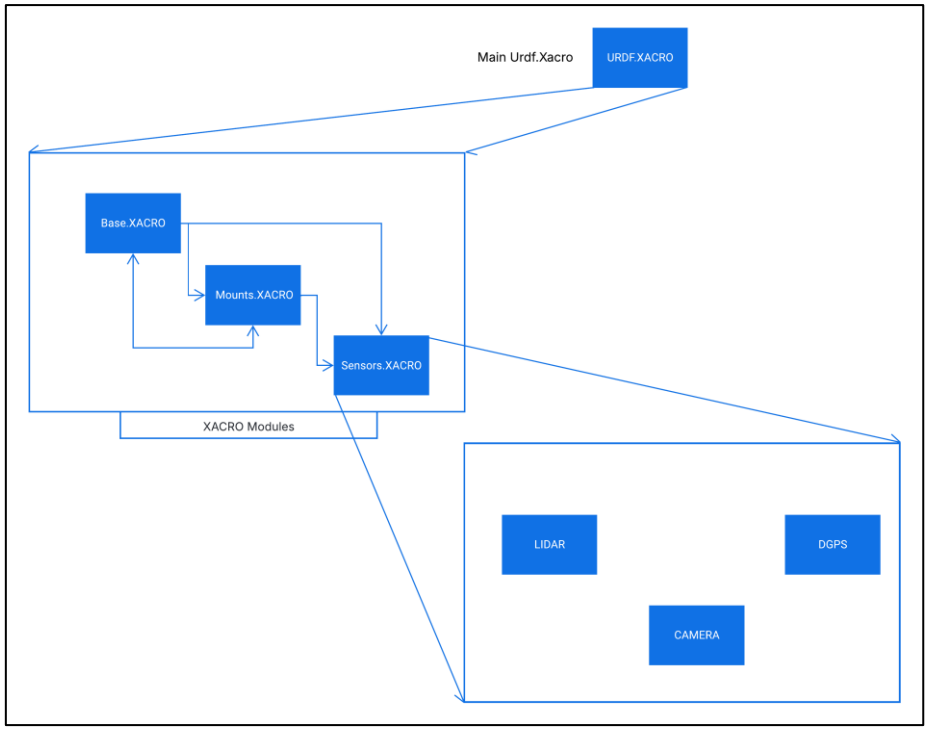

Figure 60. Modular Xacro Files

❖ **Debugging Challenges – Tracked Locomotion**

The most significant challenge was the fact that **URDF natively does not support continuous tracks** (like tank tracks). ROS2 URDF and Gazebo simulations are **wheel-based**, meaning the tracks could not be directly modelled.

To overcome this, a **mathematical equivalence approach** was applied:

- The full length of the **track's contact patch (ground-contact length)** was calculated.
- This length was then **mapped into a single large wheel** with an equivalent circumference.
- This ensured that **one full rotation of the wheel** would represent **one complete movement cycle of the track**, preserving realistic translation distance.

❖ **Additional URDF Debugging**

- **Visual alignment issues:** Sensor frames were often misaligned initially, requiring trial-and-error adjustments to parent/child relationships and origin offsets.
- **Incorrect inertia values:** Default inertia caused instability in Gazebo physics, which was gradually tuned for stable behaviour.

- **Sensor frame naming consistency:** Matching the frame names in URDF with the expected frame names in sensor drivers was essential to ensure correct TF tree generation.

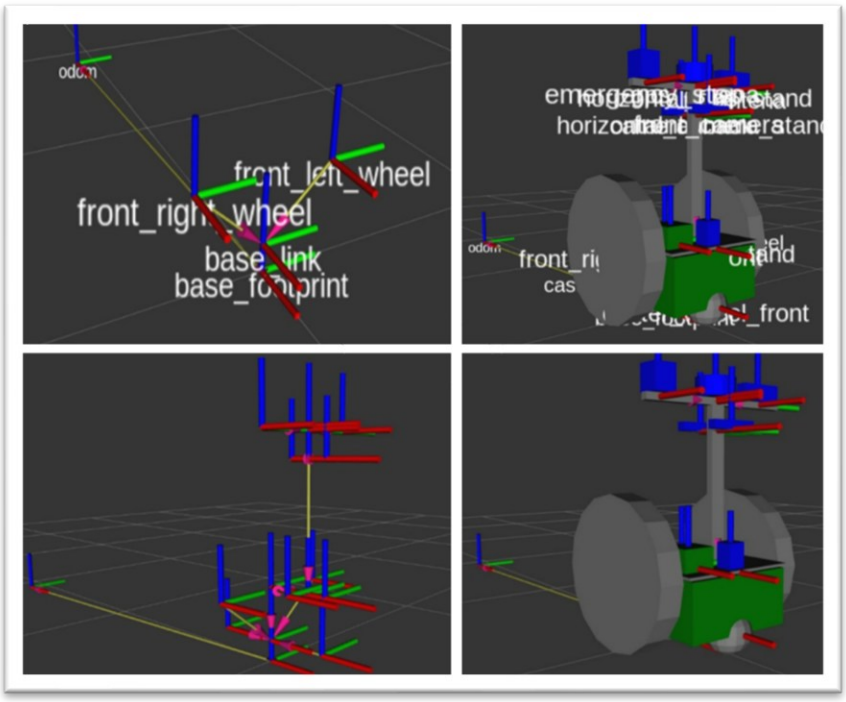

Figure 61. URDF with Transforms

Conclusion

The final URDF was tested by:

- Loading into **Rviz2** to check for correct visualization.
- Running in **Gazebo** to validate physics behaviour.
- Performing static transform sanity checks to ensure all mounted sensors had correct frame relationships to base_link.

This systematic and modular approach resulted in a flexible, **easy-to-maintain URDF/Xacro model** that could evolve alongside hardware changes, while faithfully representing the rover's physical structure and sensor placements.

7.10.3 Replicating the Dynamics of the Rover – {*Krish. Shah* (41564)}

Replicating the dynamics of the autonomous rover inside Gazebo was a crucial part of the simulation process, ensuring that the simulated rover behaved as closely as possible to the real hardware platform.

- ❖ **Custom Inertia Calculation**

The first step involved defining realistic physical properties for the rover's chassis and components. To do this, a **custom script** was written to compute the **inertia matrix** for the chassis using the actual 3D model (STL file) of the rover body. This allowed the URDF to include accurate mass, center of gravity, and rotational inertia values for the chassis, rather than relying on overly simplistic box approximations.

For individual components (sensors, mounts, and platforms), **approximate mass and shape-based inertia values** were calculated manually using known dimensions and material properties. These were iteratively tuned during simulation testing to ensure stability in Gazebo.

❖ **Physical Instability in Gazebo**

Once initial physical properties were set, early simulations showed that the rover was **unstable in Gazebo**. The tracked locomotion could not be directly implemented because URDF and Gazebo lack native support for **continuous tracks**. This led to unrealistic slipping, jittering, and rotational instability in the simulation.

❖ **Experimenting with Different Wheel Configurations**

To mitigate these issues, several configurations were tested:

- Adding **4 individual wheels**, attempting to mimic track behaviour through closely spaced wheels.
- Adding **support caster wheels** to stabilize the rover’s body.
- Tweaking **friction coefficients** between the wheels and ground plane.

These approaches improved basic stability but failed to correctly emulate the motion dynamics of a tracked vehicle.

❖ **Solution: Virtual Caster Wheels**

The final approach involved adding **two imaginary caster wheels**, one at the front and one at the rear of the rover's base, **hidden inside the body**. These caster wheels provided stable contact points, significantly improving stability without affecting the physical footprint.

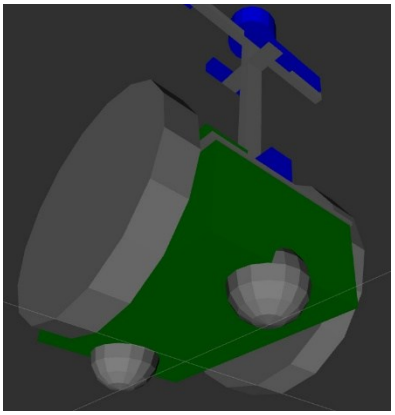

Figure 62. Caster Wheels for Dynamic Stability

❖ **Preserving Track Length**

Since actual continuous tracks could not be simulated, the tracks were replaced with a single **large virtual wheel** on each side. The circumference of these wheels was set to match the **actual track length**, ensuring that each full rotation of the virtual wheel corresponded to **one full movement cycle of the physical track**.

This mathematical substitution ensured that the rover covered realistic distances per wheel rotation while benefiting from Gazebo's built-in wheel physics, which are well-supported.

❖ **Debugging Process**

- **Physics instability was repeatedly tested by driving the simulated rover over uneven terrain** to check for tipping or unnatural behaviour.
- **Mass and inertia tuning** was performed iteratively by observing excessive bounciness, drift, or unrealistic acceleration.
- The **contact between wheels and ground was fine-tuned** using collision parameters to balance grip and slide.
- Each modification was validated by comparing **real-world drive tests with simulated drive paths**.

Conclusion

Through **custom inertia calculation**, **virtual wheel substitution**, and **addition of stabilizing caster wheels**, a stable and physically realistic simulation of the tracked rover was achieved in Gazebo. This process ensured that simulated motion closely approximated real-world behaviour, providing a usable testing platform for sensor integration and control algorithm validation.

7.10.4 Using ZED-X Camera – {*Krish. Shah* (41564)}

The **ZED X stereo camera** was integrated into the rover primarily to provide **visual odometry**, allowing the system to estimate its movement based on visual features detected by the camera.

Process

- The camera was physically mounted at the front of the rover and calibrated for stereo depth perception.
- The official **ZED ROS2 wrapper** was used to bring the camera feed into the ROS2 environment.
- This wrapper provided both **RGB-D image streams** and real-time **visual odometry**, published on the topic /zed/zed_node/odom.

Objective

The objective was to **replace unreliable wheel encoder-based odometry** with **visual odometry**, especially on uneven surfaces where track slippage would render encoder readings inaccurate.

Attempted Implementation

- Visual odometry was fed directly into the **TF tree**, providing a transform between odom and base_link.
- This transform was consumed by Nav2 and other localization nodes, aiming to achieve **more reliable localization**.
- Integration with the **robot_localization** package was also attempted, fusing joint_states and visual odometry.

Challenges Encountered

- Visual odometry suffered from **drift in low-feature environments** (e.g., smooth floors with minimal texture).
- The **ZED node's odometry update rate** was lower than ideal, introducing minor lag into the system.
- Synchronization with Joint_states data was not fully reliable due to **timestamp mismatches**.

Conclusion

Although visual odometry improved localization in textured environments, it was not sufficient as a standalone localization source. It was ultimately used as a **supplementary input**, rather than fully replacing wheel odometry.

7.10.5 NAV2 Implementation on Hardware– *{Krish. Shah (41564) & Yogachand Pasupuleti (41588)}*

➢ **Context**

- Nav2 importance and functionality previously discussed in section 4.1.7 & section 5.3.2, covering features like path planning, obstacle avoidance.
- Shift from simulation to real-world hardware implementation.

➢ **Objective**

- Demonstrate Nav2's capabilities on a physical robot.
- Validate simulated planning and control in a real-world environment.
- Achieve autonomous navigation.

➢ **Requirement**

- Map input for Nav2 implementation.
- Acceptable map types: pre-existing map of the test environment or real-time generated map.

➢ **Method for map generation**

- SLAM (Simultaneous Localization and Mapping), which was already explained in earlier sections.

➢ **Process**

- The initial process involved launching the robot's URDF file and ensuring the **robot_state_publisher** was active. This node is responsible for publishing the correct **TF tree** by reading the **joint_states** published by the motor driver node and calculating the poses of all robot links based on these states.
- During setup, we attempted to replace **encoder-based odometry** with **visual odometry** from the ZED-X camera. This required configuring the **robot_localization package** to consume the **visual odometry** topic (/zed/zed_node/odom) instead of the traditional wheel odometry from the motor encoders. This change was intended to improve localization accuracy by relying on camera-based motion estimation rather than mechanical encoder readings, which were unreliable due to slippage and mechanical noise.
- To complete the TF tree, a static transform between the **camera frame** (zed_camera_center) and the **base_link** was published, ensuring proper data fusion in the localization pipeline.

- After setting up the above processes, the remaining steps included running the motor control node, launching the LiDAR node for scan data visualization, and configuring RViz to show the map and global frame.
- Launch the robot's URDF file.
- Run the motor control node.
- Launch the lidar launch file for scan data visualization.
- In RViz, add the map and global frame topics, setting them to 'map'.

➢ **Attempted Implementation**

To implement Nav2 on hardware, all necessary launch files were executed according to the previously established process. The resulting TF frame visualization, as shown in the provided image, illustrates the system's state following the launch.

During the initial tests, the TF tree revealed significant discrepancies between simulation and hardware. In simulation, odometry was directly provided by the simulated environment, ensuring smooth updates. However, on hardware, the physical encoder data from the tracked chassis was unreliable due to slippage, low-speed noise, and calibration issues.

The LiDAR node produced a **separate TF frame** and published data to the /scan topic. However, **the connection between odometry (odom) and the map frame remained broken**. Without correct odometry, Nav2 was unable to update its internal localization, resulting in a stationary robot from the planner's perspective, regardless of actual movement.

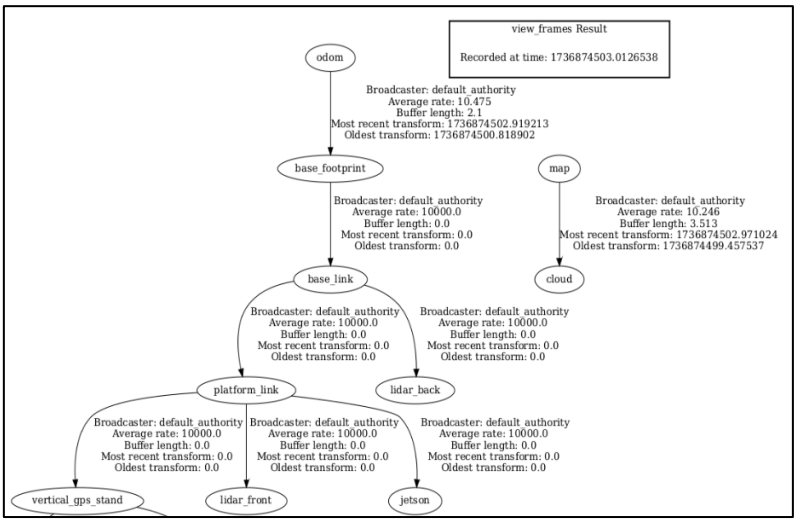

Figure 63 Broken TF Tree

The team attempted to mitigate this by generating a new **map1** topic, remapping LiDAR scan data and adding artificial transform data. A custom **'mapping with known poses' technique**, based on the **Bresenham algorithm**, was applied. This

approach generated a distorted and highly unstable map, with very slow updates. This was traced to the **incorrect or missing static transform between odom and ****map**, further compounded by noisy encoder data.

Additionally, **ZED-X camera visual odometry** was tested to replace wheel odometry. However, as documented in section 7.10.4, visual odometry from ZED-X was found unreliable in low-texture environments, such as flat outdoor areas with grass or dirt surfaces. Combined with inconsistent time synchronization between ZED data and LiDAR data, this approach also failed to produce reliable localization.

Further debugging revealed that the **URDF model of the rover itself** contributed to the issue. The tracked locomotion system was represented using large wheels with approximate circumferences matching the tracks. However, this simplified representation introduced **unrealistic motion artifacts** in the transform tree. These artifacts fed into SLAM and Nav2, corrupting pose estimation and leading to drift and incorrect localization.

Conclusion

The primary root causes of Nav2 failure in hardware were:

- Unreliable encoder-based odometry due to track slippage and mechanical noise.
- Inaccurate URDF kinematics, particularly for the tracked system.
- **Lack of proper static transform connections between map, odom, and base_link.**
- Poor performance of visual odometry from ZED-X in low-feature outdoor environments.
- Timing and synchronization issues between LiDAR, camera, and encoder data.

To achieve functional autonomous navigation with Nav2, the following steps are recommended for future iterations:

- Replace or significantly upgrade the encoder system to provide more accurate wheel odometry.
- Explore hybrid odometry techniques, combining visual, wheel, and inertial odometry for better robustness.
- Use a more accurate physical URDF, possibly adding additional contact points or better modelling track-ground interaction.
- Ensure the complete and correct TF tree is continuously published, including all necessary static and dynamic transforms.

- Investigate improved time synchronization between sensors, especially ensuring all devices share a common time source.

Ultimately, achieving **stable and reliable odometry** is essential for closing the localization loop in SLAM and enabling Nav2's planner to correctly compute paths. Without accurate odometry, the system cannot transition from waypoint-based command execution to full autonomous navigation.

7.10.6 Simple Solution Implementation – {*Krish. Shah* (41564) & Nirzer Gajera (41638)}

This section details the **practical steps, actual implementation process, and troubleshooting efforts** undertaken to build the simplified motion control system, referencing the architecture already discussed in Section 5.3. This approach was adopted after the **Nav2-based autonomous navigation failed to function reliably on the real hardware**, as discussed in section 7.10.5.

Step 1 - Initial Hardware and Software Setup

- After abandoning the unreliable Nav2 implementation, the team decided to create a **basic, reliable motion control system** that could operate the rover using **direct motor commands** and **simple obstacle detection**.
- The hardware setup included the existing **RoboClaw 2x60A Motor Controller**, **TIM781 LiDAR**, and **Jetson Nano** running ROS2 Humble.
- To enable this, we created a new **ROS2 package** specifically for this custom solution, named motor_control, which included the following custom nodes:
    - **motor_command_publisher** (for sending predefined movement commands)
    - **keyboard_override_NODE** (for manual control via wireless keyboard)
    - **TIM_7XX_NODE** (for obstacle data collection from LiDAR)
    - **Safety_stop_NODE** (for emergency stop logic)
    - **Master_motor_NODE** (for central command processing)
    - **Custom Motor Control Node** (for direct serial communication with RoboClaw)

Step 2 - ROS2 Node Creation

- Each node was implemented as a standalone Python script using **rclpy**, following standard ROS2 node lifecycle practices.

- Nodes communicated via standard **topics** (e.g., /motor_command, /object_distance).
- Specific logic for **serial communication with RoboClaw** was written using the **official Python RoboClaw library**, since there was no reliable ROS2 driver available.
- The motor_command_publisher contained basic motion sequences like:
    - Move forward for 5 seconds
    - Turn left for 3 seconds
    - Stop
- The keyboard_override_NODE listened for **manual inputs** sent via DDS from a simple remote GUI.

Step 3 - Integration and Testing

- Nodes were launched together using a **launch file**, ensuring proper startup order.
- Real-time debugging was performed using **ros2 topic echo** to verify correct message passing.
- **Motion Studio** was used in parallel to check live motor current, encoder counts, and voltage to ensure proper hardware response.
- Extensive testing was performed **indoors first (flat surface)**, followed by field testing in outdoor environments.

Step 4 - Obstacle Detection and Emergency Stop

- The TIM781 LiDAR was configured using Sick_lib, and its output was processed in TIM_7XX_NODE.
- This node published raw distance data, which was consumed by Safety_stop_NODE.
- Based on **predefined safety thresholds**, Safety_stop_NODE would either allow movement or issue a **stop command**.
- This created a **basic safety mechanism**, ensuring the rover would stop if it detected obstacles too close.

Step 5 - Debugging Process

- Multiple rounds of **debugging were required**, particularly in the following areas:

- **Serial Timeouts:** Initially, the serial port would occasionally hang, requiring the team to add **error handling and auto-reconnection logic**.
- **Encoder Read Instability:** As discussed in Section 7.10.1, encoder readings were unreliable, especially at low speeds. This required the system to **operate purely in open-loop control**.
- **Message Dropping:** Due to high-frequency message publishing (from LiDAR), the Jetson Nano's limited processing power occasionally dropped messages. The publishing rate was reduced to mitigate this.
- **Timing Conflicts:** The combination of manual control, automatic sequences, and safety overrides required careful message priority handling, ensuring emergency stop commands always took precedence.

Process Summary

- Launch the **URDF model** to maintain correct frames.
- Start the **motor control node** to establish serial communication.
- Start the **LiDAR node** to gather obstacle data.
- Start the **safety node** to continuously monitor and enforce safety.
- Start either the **predefined command publisher** or the **manual control node**.
- Observe all topics using **ROS2 CLI** to verify data flow.
- Continuously monitor hardware status via **Motion Studio**.

Comparison with Nav2 Approach

- Unlike Nav2, this approach **did not rely on odometry feedback**. All movements were time-based and assumed ideal ground conditions.
- There was **no map**, and the rover had no memory of its environment. It simply reacted to immediate sensor data.
- This greatly **simplified the software stack**, reducing computational load on Jetson Nano.

Conclusion

This **simple motion control system** provided a practical, working solution suitable for basic movement and safety demonstration. While it lacked advanced capabilities like autonomous path planning and mapping, it served as a **functional fallback**

ensuring the rover could move safely under either **predefined** or **manual commands**. This **basic fallback approach** allowed the team to focus on hardware and sensor testing, deferring complex autonomy for future work.

8 Testing

8.1 Test Plan – {*Krish. Shah* (41564)}

The testing of the autonomous rover followed a **structured, multi-stage approach**, ensuring that each hardware and software component was thoroughly validated before full integration. The test plan was designed to identify potential issues early, allowing for debugging and optimization before deploying the complete system on the target Jetson onboard the rover. This approach was shaped by multiple challenges and discussions throughout the project, as previously documented in testing logs and problem-solving sessions.

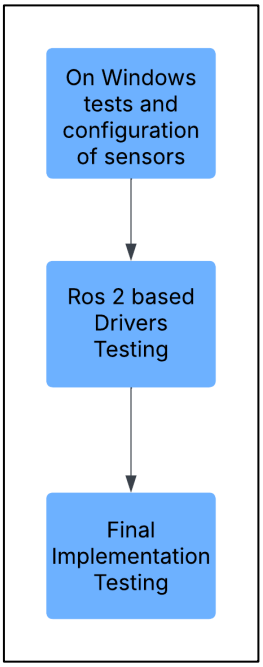

Figure 64. Testing Workflow

8.1.1 Testing Stages – {*Krish. Shah* (41564)}

1. Individual Hardware and Sensor Testing on Windows/Linux

The first stage focused on validating the functionality of individual hardware components and sensors before their integration with ROS2. This phase was performed on Windows PCs or separate Linux systems, depending on manufacturer software support. Our previous tests confirmed that running initial validation on these platforms helped isolate hardware-related issues before introducing ROS2 complexities.

- **Motor Controller Testing (Windows/Linux)**: The RoboClaw motor controller was tested using **Motion Studio** on Windows and custom serial communication scripts on Linux to ensure correct motor operation and encoder feedback. Previous discussions highlighted **challenges in low-speed encoder feedback**, which were addressed during this phase.
- **LiDAR Testing (Windows/Linux)**: The Sick TIM781 LiDAR was tested using manufacturer-provided software and later using ROS2-compatible drivers to confirm point cloud data accuracy. Past testing logs indicated that **point cloud visualization was successful**, though integration into a mapping pipeline required additional debugging.
- **GPS/INS Testing (Windows/Linux)**: The Swift DGPS/INS was tested by logging raw data and verifying accuracy against external GPS sources. Earlier discussions noted **difficulty in synchronizing IMU data with the rest of the system**, which was considered in this phase.
- **ZED X Stereo Camera Testing (Windows/Linux)**: Camera drivers were tested to ensure stereo vision functionality and depth map generation. Prior chats highlighted **latency issues in real-time depth computation**, which were observed during testing.

2. ROS2 Package and Driver Testing on Development Systems

Once the individual hardware components were verified, the next phase focused on testing their respective ROS2 packages and drivers in a **controlled development environment** before deployment on the rover. This followed from our discussions on **incremental debugging** to isolate failures in each subsystem.

- **Motor Control ROS2 Package**: The motor_control package was tested with predefined movement commands using /cmd_vel topics and monitored for motor response. Our previous troubleshooting efforts helped resolve **open-loop execution inconsistencies**, though precise closed-loop control remained a challenge.
- **LiDAR Integration in ROS2**: The sick_scan_xd package was tested for correct LiDAR data publishing and visualization in **Rviz**. Earlier tests showed that **sensor noise had to be filtered for better mapping results**.
- **TF Broadcasting**: The my_tf_broadcaster package was tested to ensure correct frame transformations between base_link, odom, and sensor coordinate frames. Past discussions pointed out **misaligned transforms affecting localization**, which was addressed here.
- **GPS/INS Data Processing**: The GPS/INS ROS2 package was tested for real-time position updates and synchronization with IMU data. Previous testing logs showed **drift issues**, requiring tuning of filter parameters.

- **SLAM Testing**: The my_slam_config package was tested using ROS2 SLAM Toolbox for map generation using LiDAR data. Past discussions noted **Nav2 integration challenges**, influencing how mapping tests were designed.

3. Full System Integration on Jetson (Target Platform)

After verifying individual ROS2 packages, the final testing phase focused on full system integration on the **Jetson** running Ubuntu 20.04 with ROS2 Humble.

- **System Bringup and Launch File Testing**: The rover_bringup package was tested to ensure all nodes launched correctly without conflicts. Debugging history showed **dependency issues between launch files**, which were resolved.
- **Motor and Sensor Synchronization**: LiDAR, GPS, IMU, and motor control data were synchronized to verify real-time system operation. Previous tests highlighted **timing inconsistencies**, which were mitigated through this stage.
- **Navigation and Path Planning Tests**: The Nav2 stack was initially tested but encountered **odometry integration issues**, leading to reliance on the simpler open-loop motor control approach. Prior discussions detailed **failed localization attempts due to incomplete transform tree**.
- **Real-World Rover Movement Tests**: Final tests included running movement sequences, obstacle detection, and map validation in a controlled environment. Previous real-world test results informed **adjustments in open-loop control parameters**.

8.2 Test Results

8.2.1 Navigation and Path Planning – {*Krish. Shah* (41564) & Nirzer Gajera (41638)}

Efforts were made to utilize the **Nav2 stack** for full autonomy, incorporating waypoint navigation and real-time obstacle avoidance using LiDAR and GPS data.

However, the implementation was unsuccessful due to multiple issues—such as incorrect odometry integration and unreliable localization. The inability to establish a stable map and refine real-world coordinates prevented reliable path planning.

Instead, testing continued with **basic movement sequences** in an open-loop format, where the rover executed pre-set motion commands without real-time corrections. This approach partially succeeded in achieving movement but lacked dynamic adaptability.

8.2.3 Simple Time-based Control Navigation using Wireless GUI.

Test Setup

The rover was tested in an indoor environment on a flat surface. The test scenario involved sending a sequence of coordinates to the rover through the **/path_planner** topic. The rover executed the received commands using the implemented **open loop control system**. The test path included straight movements along the X and Y axes, with gradual acceleration and deceleration applied to improve motion smoothness.

Observations

The following observations were recorded during multiple test runs:

Observation Point	Description
Initial Movement Response	Rover responds promptly to the first movement command after receiving the coordinates.
Acceleration Phase	Gradual acceleration works effectively, reducing wheel slip at the start.
Straight Line Movement	Movement along straight paths was generally smooth, but accuracy degraded over longer distances.
Deceleration Phase	Gradual deceleration reduced abrupt stops but showed sensitivity to surface friction.
Movement Accuracy	There was **noticeable drift** over longer distances due to cumulative slippage and minor motor imbalances.
Repeated Path Consistency	Running the same path multiple times led to different final stopping positions each time, with deviations of up to 15-20 cm.
Turn Performance	Turns were **time-based**, so turn angles were approximate, resulting in small heading errors after each turn.
Wireless Delay Impact	Occasional minor lag was observed between sending coordinates from the GUI and the rover starting its movement (less than 0.5 seconds).
Real-time Feedback Absence	Due to the lack of encoder or IMU feedback, the system had **no way to detect or correct drift or slippage** during execution.
Environment	Changes in surface friction (smooth tile vs carpet) significantly

Sensitivity	affected accuracy. On smoother surfaces, drift was lower. On higher friction surfaces, drift increased.

Table 13 Simple Implementation Observations

Quantitative Results

Test Case	Target Distance (meters)	Actual Distance (meters)	Error (cm)
Move forward 2m	2.00	1.88	12 cm short
Move forward 3m	3.00	2.85	15 cm short
Turn 90 degrees (right)	90°	~86°	4° error
Full path (square)	4m x 4m square	Drifted outside by ~18 cm on final point	

Table 14 Quantitative Results

Key Findings

- Open loop control using time-based execution is **too sensitive to surface type, motor calibration, and battery voltage**.
- The lack of encoder feedback makes **drift correction impossible**.
- Gradual acceleration/deceleration improved **smoothness** but did not fully solve accuracy issues.
- Wireless GUI transmission worked reliably, but even small communication delays affected synchronization.
- Repeated runs of the same path produced **non-repeatable results**, showing cumulative error accumulation.

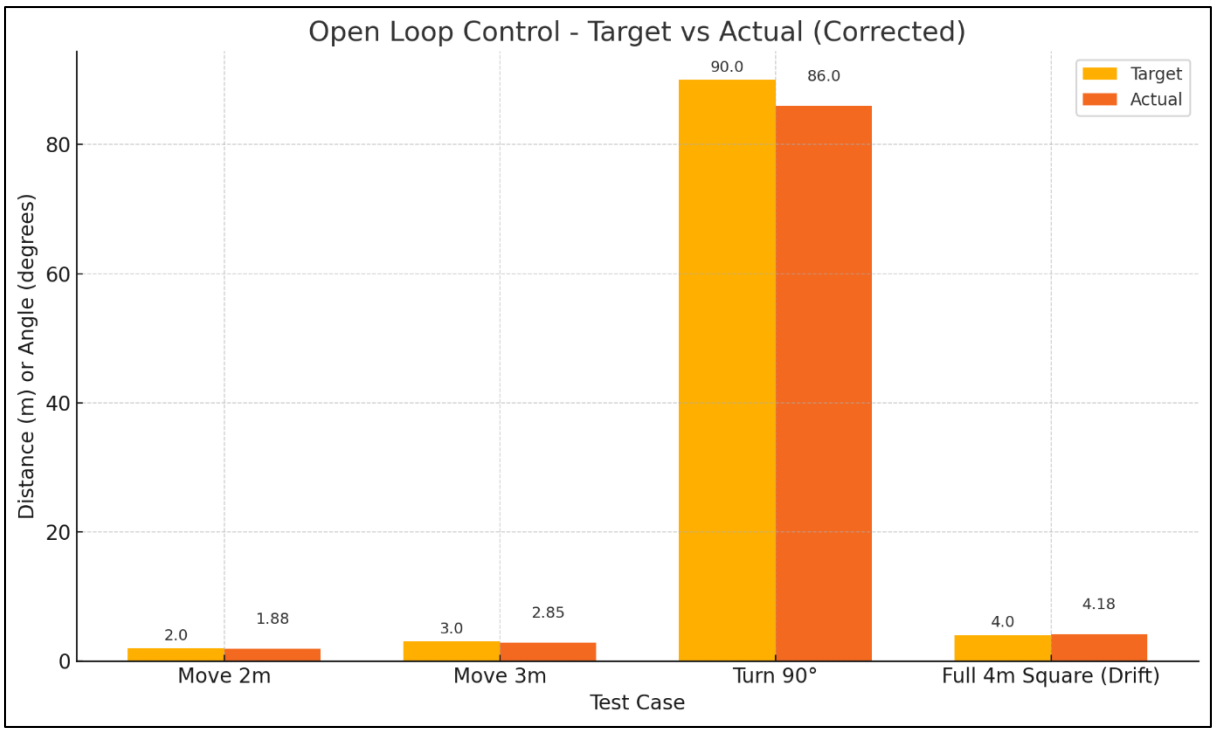

Figure 65. Plot for Drift

Conclusion

The **open loop time-based control system** provided a quick and simple method to test basic movement capabilities. However, it is **unsuitable for reliable autonomous navigation** in real-world environments. The absence of feedback (encoders, IMU) leads to significant accuracy loss, especially over longer distances or on variable surfaces. Future work will focus on transitioning to a **closed loop control system** using real-time encoder feedback for position estimation and drift correction.

8.2.2 Swift Piksi testing for Navigation in ROS2 environment – *{Niranjan Channayya (41559)}*

Below is the summary of Piksi Multi testing for Navigation in ROS2 environment-

Tested Component	Description	Observation	Future Approaches
Standalone GPS Positioning	GNSS-based localization without RTK corrections	Accuracy within 1-3 meters, but signal drift and building reflections cause path deviations	Implement RTK positioning for centimetre-level accuracy
ROS2 integration	Built and installed Swift Piksi ROS 2 driver	Successfully published ROS topics required for navigation	Optimize sensor fusion with Extended Kalman Filter (EKF) for improved localization
RTK Simulation	Simulated RTK mode in Swift Piksi console	Baseline vectors were correctly computed in RTK mode	Deploy actual RTK base station instead of simulation for real-world testing
Baseline Topic Testing	Verified baseline length and orientation in ROS 2	Data was successfully received, indicating relative positioning worked	Fine-tune RTK corrections for consistent baseline orientation
IMU Data Processing	Processed acceleration from Bosch BMI160 for displacement calculation	Observed significant drift in distance due to sensor noise and bias errors	Apply low-pass filtering, sensor calibration, and zero velocity update (ZUPT)
Signal Loss in Indoor Areas	Tested movement through indoor areas	GNSS signal completely lost in indoors	Implement dead-reckoning using IMU and wheel odometry in GNSS-denied environments

Table 15. Testing of Piksi Multi for navigation and Future approaches

8.3 Test Summary – {*Krish. Shah* (41564)}

The test plan ensured a **gradual transition from individual component testing to full system deployment**, allowing for **early debugging and refinement** at each stage. The structured approach enabled effective troubleshooting, leading to a **partially successful** motion control system due to open-loop execution and an **unsuccessful** Nav2-based navigation attempt due to localization issues. This testing process was continuously refined based on previous challenges encountered during hardware validation, ROS2 debugging, and real-world movement experiments.

Ultimately, the system **achieved basic motion control in open-loop mode**, while autonomous navigation using **Nav2 remained unachieved** due to unresolved localization and mapping challenges. The testing cycle was continuously improved based on iterative learning from hardware validation, ROS2 debugging, and real-world movement tests.

9 Summary, Conclusion & Outlook – *{Niranjan Channayya (41559)}*

9.1 Wrap up of achieved goals – *{Niranjan Channayya (41559)}*

During the project, progress was made in multiple areas related to autonomous navigation, mapping, localization, and motion control. The architecture for the project was successfully designed, providing a structured framework for system integration. A custom Roboclaw library-based ROS 2 driver was developed to control the mobile platform efficiently.

Motion control was successfully implemented using /cmd_vel messages published to the ROS 2 motor control node. The platform demonstrated basic forward, backward, and turning movements controlled through this interface. Encoder feedback was partially tested to estimate wheel odometry, though further refinement is needed for reliable localization.

A GUI-based wireless manual control system was also developed, allowing remote command transmission using a PyQt and PySide6 interface. This allows operators to manually control the robot and test basic motion capabilities from a distance.

The URDF model of the robot was created, visualized in RViz, and loaded into Gazebo for initial simulation testing. Differential drive configuration was successfully applied using the libgazebo_ros_skid_steer_drive plugin.

In terms of perception and mapping, extensive testing was conducted using the SICK TIM LiDAR sensor, both within and outside the ROS 2 environment. The LiDAR data was successfully visualized in RViz, and modifications were made to the provided launch files for SICK TIM 781 to enhance its compatibility with ROS 2 for SLAM-based mapping.

The safety stop functionality was also implemented using real-time obstacle detection from the LiDAR sensor, ensuring a safer navigation experience.

Both static and dynamic maps were generated using SLAM based on LiDAR data, but dynamic mapping exhibited inaccuracies due to odometry errors in the hardware setup. To address these challenges, a simulation environment was utilized where odometry and TF frames were self-created using a self-built URDF file, providing more reliable sensor data. This allowed successful implementation of accurate map generation using the SLAM Toolbox. Moreover, point-to-point autonomous navigation was achieved by integrating the Nav2 stack, enabling the robot to navigate to predefined goal points while avoiding obstacles.

In terms of GNSS-based localization, the Swift Piksi GNSS receiver was tested for standalone positioning and RTK-based navigation. In standalone mode, the receiver provided positioning accuracy within 1-3 meters, while RTK corrections improved it to centimetre-level accuracy. The baseline vector between the base station and the rover was successfully measured and integrated into the navigation system. The ROS 2 driver for Swift Piksi was installed and tested, publishing key topics such as /fix, /baseline, and /odometry/filtered. However, odometry drift in standalone GNSS mode was observed, necessitating improvements through sensor fusion with IMU and wheel odometry.

To improve accuracy, RTK corrections were simulated and validated, confirming that the system could effectively achieve high-precision localization. The integration of GNSS with other sensors, such as IMU and LiDAR, was explored for robust navigation in environments with temporary GNSS outages. Future improvements will focus on deploying a real RTK base station, fine-tuning sensor fusion algorithms using Extended Kalman Filter (EKF), and optimizing path-planning algorithms for better performance in real-world environments.

This accomplishment demonstrates a strong understanding of ROS 2, SLAM-based mapping, autonomous navigation using Nav2, and GNSS-based localization with Swift Piksi in addition to using IMU making it a foundation for further improvements in real-world robotic applications.

9.2 Wrap up of limitations - *{Niranjan Channayya (41559)}*

During the implementation of the autonomous navigation system, several challenges and limitations were encountered, affecting the accuracy and efficiency of localization, motion control, and sensor integration. Odometry readings from the encoders were unreliable due to system drift and inconsistent data from the Roboclaw motor controller, making it difficult to derive accurate position estimation. Attempts to use visual odometry as a fallback approach also did not succeed, primarily because of poor feature tracking in low-texture indoor environments and fluctuating lighting conditions. Additionally, the full integration of LiDAR, GPS, and camera systems into the ROS 2 framework was not completed, which restricted the system to basic motion control without real-time environmental awareness. Another limitation arose from the uneven weight distribution of the rover platform, which caused imbalanced traction, leading to deviations in straight-line movement and turning inaccuracies. The power and control systems also presented challenges, with observed motor behaviors including sudden starts and inconsistent speed control, potentially caused by power management issues and command transmission delays between the Jetson Nano and the Roboclaw motor controller. These limitations emphasize the need for further development in key areas such as sensor fusion, odometry filtering, power system stabilization, and more reliable localization techniques to enhance the overall autonomy and performance of the rover in challenging environments

From a software and usability perspective, the Pyati-based GUI could not be successfully converted into an Android application, leaving the system without a tablet-compatible interface. In terms of GNSS localization, Swift Piksi's standalone mode exhibited positioning errors of 1–3 meters, which was inadequate for precise autonomous navigation. RTK corrections were only tested in simulation but were not fully deployed, limiting the achievable localization accuracy and GNSS signal loss occurred indoor environments, impacting continuous navigation reliability. Moreover, sensor fusion between GNSS, IMU, and odometry needs to be implemented.

To address these limitations, future work can focus on improving odometry accuracy through IMU-based dead reckoning and advanced sensor fusion, deploying a real RTK base station to achieve centimetre-level GNSS accuracy, and optimizing motion control algorithms to mitigate sudden motor movements. Additionally, an Android-based GUI can be developed to enhance user accessibility, and EKF parameters will be further refined for better fusion of GNSS, IMU, and odometry data. Testing visual odometry in controlled environments can also be explored to improve feature tracking and enable localization in GNSS-denied scenarios. These improvements will contribute to a more robust and reliable autonomous navigation system for real-world applications

9.3 Required steps to close the gaps - *{Niranjan Channayya (41559)}*

The autonomous navigation system implementation faced challenges in localization, motion control, and sensor integration. Encoder-based odometry was unreliable due to drift and inconsistent data, and visual odometry failed in low-texture environments. Sensor integration for LiDAR, GPS, and cameras was incomplete, limiting real-time awareness. Imbalanced platform weight caused movement inaccuracies, while unstable power delivery and command delays affected motor control.

Improving odometry accuracy through IMU-based dead reckoning and sensor fusion with wheel encoders will help mitigate localization errors. Deploying a real RTK base station instead of relying on simulations will provide centimetre-level positioning accuracy, reducing GNSS drift and improving baseline stability. Fine-tuning the Extended Kalman Filter (EKF) parameters is essential to optimize GNSS, IMU, and odometry integration, ensuring smoother localization transitions. The motor control algorithms must be refined to eliminate sudden jumps and improve power distribution for stable movement. Additionally, developing an Android-based GUI will enhance usability and enable wireless control via tablets. Finally, testing visual odometry in controlled environments and integrating LiDAR-based SLAM with accurate TF frame transformations will further strengthen the system's ability to operate reliably in GNSS-denied environments. To overcome these issues, key improvements include better encoder filtering, sensor fusion using IMU and LiDAR, full sensor integration in ROS 2, hardware balancing, power stabilization, and optimized motor control timing. Real-world testing will be essential to validate and refine these solutions.

Works Cited

[1] M. G. B. &. S. W. D. Quigley, Programming Robots with ROS: A Practical Introduction to the Robot Operating System, USA: O'Reilly Media, 2015.

[2] S. F. T. G. B. W. W. &. C. J. Macenski, “Navigation 2: The ROS2 Navigation Framework and System,” in *arXiv preprint*, 2022.

[3] N. Corporation, “ Jetson AGX Orin Developer Kit Technical Overview,” [Online]. Available: https://developer.nvidia.com/.

[4] “Sick TIM781 LiDAR Datasheet,” [Online]. Available: https://www.sick.com/.

[5] “Swift DGPS/INS System,” [Online]. Available: https://www.swift.com/.

[6] W. Garage, “ROS: An Open-Source Robot Operating System,” in *ICRA Workshop on Open Source Software*, 2010.

[7] S. B. W. &. F. D. Thrun, Probabilistic Robotics, MIT Press, 2005.

[8] O. (. M. Group), “Data Distribution Service (DDS) for Real-Time Systems Specification,” 2023. [Online]. Available: https://www.omg.org/.

[9] R. B. Review, “Trends in Autonomous Robotics for Logistics, Agriculture, and Security Applications,” Robotics Business Review, 2023.

[10] “ROS2 Humble Documentation,” [Online]. Available: https://docs.ros.org/en/humble/index.html.

[11] T. Q. Company, “QT6 Documentation,” 2024. [Online]. Available: https://doc.qt.io/qt-6/.

[12] S. R. ,. D. T. ,. A. N. ,. G. B. M. A. G. Mikael Arguedas, “ROS 2 Releases and Target Platforms,” [Online]. Available: https://ros.org/reps/rep-2000.html.

[13] ROS2, “Quality of Service settings,” Open Robotics, [Online]. Available: https://docs.ros.org/en/humble/Concepts/Intermediate/About-Quality-of-Service-Settings.html.

[14] E. W. José L. Millán, “foxglove,” [Online]. Available: https://foxglove.dev/blog/how-to-use-ros2-lifecycle-nodes.

[15] [Online]. Available: https://index.ros.org/p/nav2_lifecycle_manager/.

[16] S. Macenski, ""On Use of SLAM Toolbox, A fresh(er) look at mapping and localization for the dynamic world",” ROSCon 2019.

[17] S. J. I. Macenski, ""SLAM Toolbox: SLAM for the dynamic world", Journal of Open Source Software, 6(61), 2783,,” 2021.

[18] T. Horelican, “Utilizability of Navigation2/ROS2 in Highly Automated and Distributed Multi-Robotic Systems for Industrial Facilities,” May 2022.

[19] “ROS2 Navigation Stack Documentation,” [Online]. Available: https://navigation.ros.org/.

[20] S. S. Intelligence, “Github Repository for SICK Scan LiDARs (sick_scan_xd),” [Online]. Available: https://github.com/SICKAG/sick_scan_xd?tab=readme-ov-file.

[21] “ZED X Stereo Camera System, Stereolabs Documentation,” [Online].

[22] D. S. a. F. Fraundorfer, ""Visual Odometry [Tutorial]," in IEEE Robotics & Automation Magazine, vol. 18, no. 4, pp. 80-92, Dec. 2011, doi: 10.1109/MRA.2011.943233.,” [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6096039&isnumber=6096005.

[23] C. C. e. al, “Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age," in IEEE Transactions on Robotics, vol. 32, no. 6, pp. 1309-1332, Dec. 2016, doi: 10.1109/TRO.2016.2624754,” [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7747236&isnumber=7747233.

[24] G. F. P. J. D. &. E.-N. A. Franklin, Feedback control of dynamic systems. Pearson., 2019.

[25] “BasicMicro. (n.d.). RoboClaw 2x60A Motor Controller,” [Online]. Available: https://www.basicmicro.com/RoboClaw-2x60A-Motor-Controller_p_15.html.

[26] Ulrich, “Ulrich Robotics. (n.d.). Tracked Robotic Chassis Platform,” [Online]. Available: https://www.ulrichrobotics.com/products/tracked-robot-platform.

[27] J. Y. Wong, Theory of Ground Vehicles. John Wiley & Sons, 2011.

[28] “NVIDIA Jetson Nano Developer Kit,” [Online]. Available:

] https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[29 “RoboClaw Motor Controller,” [Online]. Available: https://www.basicmicro.com/.
]

[30 S. H. o. R. Caterpillar Track Systems in Robotics.
]

[31 R. E. J. Serial Communication Protocols.
]

[32 “ROS2 Navigation Stack Documentation,” [Online]. Available:
] https://navigation.ros.org/.

[33 “Wimble Robotics,” [Online]. Available:
] https://github.com/wimblerobotics/ros2_roboclaw_driver.

[34 “ROS 2 driver for Swift Navigation's GNSS/INS receivers and Starling
] Positioning Engine software,” [Online]. Available: https://github.com/swift-nav/swiftnav-ros2.

[35 PID Control Algorithms in Robotics, IEEE Transactions.
]

[36 S. R. Motion Control for Autonomous Vehicles.
]

[37 M. P. Locomotion Design for Mobile Robots.
]

[38 J. o. F. R. Terrain Adaptability in Robotics.
]

[39 D. o. t. S. TOOLBOX. [Online]. Available:
] https://docs.ros.org/en/humble/p/slam_toolbox/#.

[40 “CLI-Tools,” [Online]. Available:
] https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html.

[41 “Google Images”.
]

Code Listings

GUI Repository: -

The source code for this project GUI and related files is available on **GitHub**. One can visit the repository to access the latest code, documentation, and updates.

The packages built by {Krish. Shah (41654)} are available on GITHUB. All the names of the package's names simply define the purpose of the packages and are separated as per different functionalities. Also, a readme section is planned, which will soon be implemented for better understanding regarding the packages and nodes. In all the code written by this author there are comments for better understanding and configuration of important parts.

Additionally https://github.com/ESDBOT/esdbot24/tree/navigation has all the project files and contains packages like nav2_config and sick_scan_xd which was contributed by - {Yogachand Pasupuleti (41588) & Nirzer Gajera (41638)}.