Politecnico di Torino

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Autonomous Robot Driving using Sensor Fusion

Supervisors
Prof. Stefano Malan
Prof. Massimo Violante

Candidate
Fabio Marchisio

Intership Tutor
Dott. Leonardo Forese

December 2024

Abstract

In the rapidly evolving landscape of automation, robots and autonomous vehicles have become essential tools, driving innovation, improving efficiency and reliability, while integrating and cooperating with humans.

This thesis presents the development of an Autonomous Mobile Robots (AMRs) system based on the Yahboom ROSMASTER X3, from the assembly phase to code implementation. The system is powered by a NVIDIA Jetson Nano and actuated by a STM32-based board. The robot is equipped with a depth camera and a Light Detection And Ranging (LiDAR) sensor.

The robot primary function is to track a moving target in real time while autonomously avoiding obstacles. The person detection is based on a previous thesis project on neural networks and real-time object tracking. However, the main focus of this project is the sensor fusion of the camera and LiDAR in Robot Operating System (ROS).

Additionally, the thesis explores the architecture of the ROSMASTER X3, the use of Docker and containers, and the communication protocols implemented. Various tests were conducted to assess the system performance in complex environments and under different conditions, focusing on real-time response, obstacle avoidance accuracy, and smoothness in movement transitions.

The results indicate that ROS2 architecture and the integration of sensor fusion techniques significantly enhance the robot autonomous capabilities, making it suitable for dynamic environments. This work contributes to the wider field of autonomous mobile robotics by demonstrating an effective implementation of ROS2-based systems for mobile robots.

Table of Contents

List of Figures

List of Tables

Listings

Acronyms

LiDAR Light Detection And Ranging

AMR Autonomous Mobile Robot

AGV Autonomous Guided Vehicle

ROS Robot Operating System

LKA Lane Keeping Assist

ACC Adaptive Cruise Control

MQTT Message Queuing Telemetry Transport

QoS Quality of Service

DDS Data Distribution Service

SSD Single Shot multiBox Detector

EOL End Of Life

TOF Time Of Flight

CCD Charge Coupled Device

API Application Programming Interface

SDK Software Development Kit

CUDA Compute Unified Device Architecture

GPU Graphics Processing Unit

CPU Central Processing Unit

RAM Random Access Memory

AI Artificial Intelligence

eMMC embedded Multi Media Card

LPDDR Low Power Double Data Rate

MCU MicroController Unit

IMU Inertial Measurements Unit

PWM Pulse Width Modulation

Rviz ROS-Visualization

PID Proportional Integral Derivative

SSH Secure Shell

VScode Visual Studio Code

IDE Integrated Development Environment

FPS Frame per Second

Chapter 1

Introduction

The ROSMASTER X3, developed by Shenzhen Yahboom Technology Co., is an educational robot specifically designed for exploring the ROS environment and advancing robotics research. This Autonomous Mobile Robot (AMR) allows hands-on learning, experimentation with autonomous systems, and exploration of the possibilities of sensor integration and real-time robot control [1].

The initial goal of this project was to autogenerate code for the STM32 microcontroller from a Matlab and Simulink model, with the final objective of developing an autonomous robot capable of following a path, implementing Lane Keeping Assist (LKA), Adaptive Cruise Control (ACC) and recognizing road signs and traffic lights, building upon previous thesis projects. However, after assembling the robot and gaining a deeper understanding of the architecture and connections between the various boards and sensors, it became evident that this approach was not feasible. As a result, the project direction was adjusted to focus on autonomous person-following and dynamic obstacle avoidance, while still aiming to align with the original goal.

1.1 Goals

The goal of this thesis is to develop an autonomous robot capable of following a person in real-time while dynamically avoiding obstacles in different environments. By means of sensor fusion, including data from Light Detection And Ranging (LiDAR) and a camera, the robot aims to demonstrate reliable navigation with various movements thanks to the omnidirectional Mecanum wheels.

1.2 Autonomous Robots

Autonomous robots play a crucial role in modern automation. Two key types of autonomous robots are Autonomous Guided Vehicles (AGVs) and Autonomous Mobile Robots (AMRs).

AGVs have been used since the 1950s and are typically found in controlled environments like warehouses or assembly lines. They rely on predefined paths marked by physical guides such as wires or magnets. While reliable for repetitive tasks, AGVs lack flexibility and are costly to install and maintain due to the necessary supporting infrastructure.

AMRs, on the other hand, are more advanced, offering greater flexibility. They use sensors like cameras and LiDAR to understand their environment in real time, navigating freely without fixed routes. AMRs can adapt to changes in the environment, avoiding obstacles and dynamically optimizing their routes. This makes them ideal for more complex, variable settings [2].

In this project, mapping and path planning are not used due to the lack of computational power, but the principles behind AMRs adaptability and flexibility are followed. The ROSMASTER X3 robot, similar to an AMR, operates in a dynamic environment where it must track and follow a person while avoiding obstacles. Instead of relying on complex mapping or long-term path planning, it uses sensor data to react immediately to changes in its surroundings, ensuring safe movement without predefined paths. This simpler approach mirrors the flexibility of AMRs while focusing on real-time interaction rather than full environmental autonomy.

1.3 Thesis Outline

This thesis is organized into six main chapters, each addressing a critical aspect of the project and its development:

Chapter 1 introduces the objectives and context of the project, providing basic background information on key concepts of autonomous robots.

Chapter 2 reviews the state of the art in the technologies relevant to the project. It covers the Robot Operating System (ROS) and its evolution from ROS1 to ROS2, the use of LiDAR sensors for perception, and the role of Docker and containerization in robotic system development. It also provides basic information about machine learning and neural networks, though they are not the focus of this project.

Chapter 3 covers the hardware components used in the robot, including the Jetson Nano, ROS expansion board (STM32), LiDAR A1, RGB depth camera, and Mecanum wheels. It also describes the architecture of the robot and outlines the steps taken to assemble the system and configure the software environment.

Chapter 4 focuses on the software developed for the project, including a brief overview of the firmware for the STM32 board, the ROS2 scripts, and Python programs used for the robot vision. Additionally, it describes the use of Message Queuing Telemetry Transport (MQTT) for communication between components.

Chapter 5 explains the testing and evaluation of the robot. It discusses the system launch files, activation and deactivation processes, and the performance in terms of movement and obstacle detection, supported by data analysis. LiDAR measurements and velocity data are used to demonstrate the robot ability to dynamically track a person and avoid obstacles.

Chapter 6 concludes the thesis by explaining the results achieved during the project. It also identifies areas for potential improvements and future research, highlighting the limitations and problems encountered in this project.

Chapter 2

State of Art

2.1 Robotic Operating System - ROS

The Robot Operating System is an open-source framework widely used in robotics research and development. Despite its name, it is not an operating system but rather a middleware that provides services such as low-level device control, hardware abstraction, inter-process message passing, commonly-used functionalities, and package management. Additionally, ROS includes tools and libraries for building and running code across multiple computers.

The key concept behind ROS is its modular architecture, where different functionalities are divided into nodes. These nodes are distributed processes grouped in packages that can be shared or published. This design ensures code reuse and allows projects to remain independent from the file system while integrating ROS basic tools.[3]

ROS framework supports multi-language programming, with Python and C++ (via the relatives rospy and roscpp libraries) being the primary languages. These libraries enable programmers to interact with ROS topic, services and parameters. Rospy guarantees fast prototyping, while roscpp can support high performance tasks. The distributed architecture is another important feature that leads to a good scalability. In fact, wrappig each process as a node allows developers to organize larger projects involving multiple nodes through roslaunch.[4][5]

2.1.1 ROS architecture

ROS is built on three levels

- **File-system level**: describes code and executable on disk
- **Computation Graph level**: explains the peer-to-peer communication network between processes
- **Community level**: contains shared code and developers knowledge, promoted by the open-source philosophy of ROS

File-system level

The ROS resources on disk covered in this level are shown in Figure 2.1. [6]

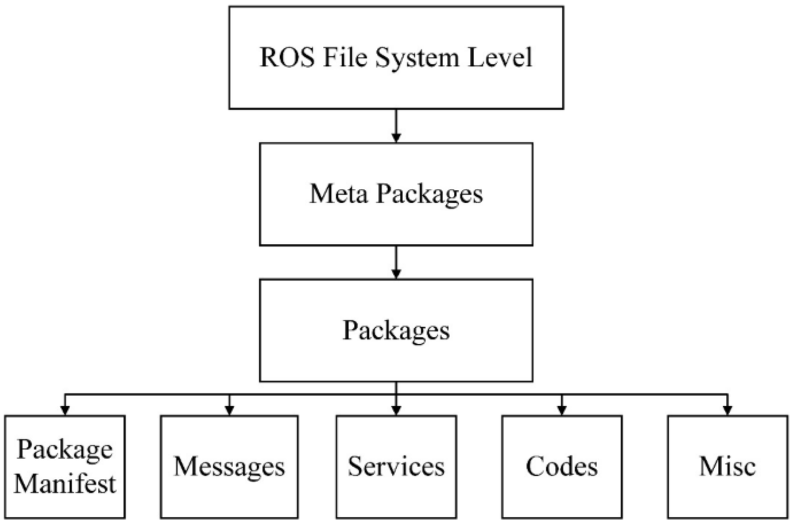


Figure 2.1: ROS File-system level

- **Meta Packages**: A group of related packages in the ROS ecosystem.
- **Packages**: The primary organizational unit for software. A package can include nodes (runtime processes), libraries, configuration files or any other resources that need to be grouped together. Packages are the smallest items that can be individually built and released, making them the fundamental unit for both development and deployment in the ROS ecosystem.
- **Packages Manifest**: A file containing metadata about a package such as name, description, version, license information, etc.
- **Messages**: A description of data structures for messages, defining their layout stored in the file package/msg/MyMessageType.msg.

- **Services**: Specifications that define the structure of request and response messages stored in the file package/srv/MyServiceType.srv

Computation Graph Level

The concept in the computational Graph level are represented in Figure 2.2. [5] [6]

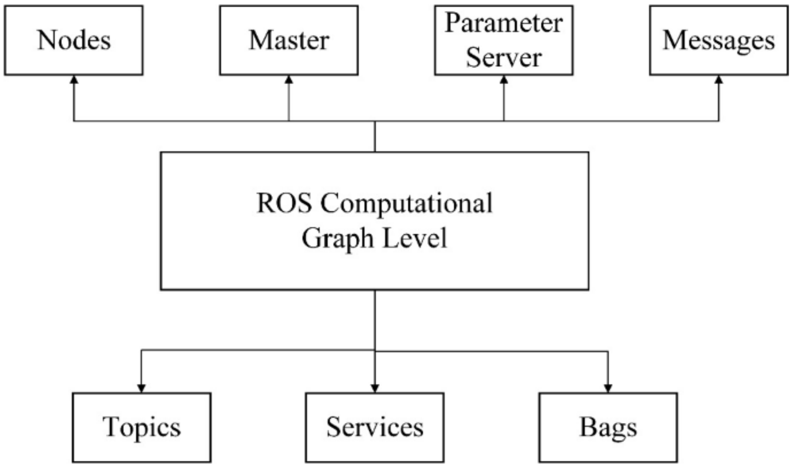


Figure 2.2: ROS Computational Graph level

- **Nodes**: The individual processes responsible for computation, enabling a modular system. Each node typically handles a specific function, such as controlling sensors, motors, localization, or visualization. Nodes are created using ROS libraries, for example Rospy (Python) or Roscpp (C++).
- **Master**: The central node that enable communication by managing name registration and lookup, ensuring a node can reach and interact with the others.
- **Parameter Server**: A centralized storage for data, managed by the Master, allowing nodes to access shared parameters.
- **Messages**: Data structures that can contain the standard primitives types (Boolean, Integer, Floating point, etc) or arrays of these primitives. Used for nodes communication.
- **Services**: Mechanism for request-response interactions. Defined by request and response message structures, services enable one node to provide functionality that others can access as a remote function call.
- **Topics**: Named channels over which messages are transmitted using a publish/subscribe system, decoupling message producers from consumers. Multiple nodes can publish and subscribe to the same topic, as shown in Figure

2.3. ROS topic messages can be transmitted using TCP/IP or UDP. The default transmission method is TCP/IP (TCPROS), which is a long connection method; transmission based on UDP (UDPROS), is a low-latency, high-efficiency transmission method, but it is more prone to lose data.

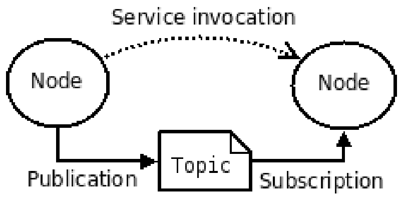


Figure 2.3: ROS Nodes communication

- **Bags**: data format for storing and replaying ROS messages, essential for recording sensor data for algorithm testing and development.

Community Level

The Community level allows software and knowledge exchange through Distributions (as Linux distributions), Repositories, ROS Wiki(forum), Bug Ticket System and many other support options. [6]

In the ROS framework, several fundamental components are essential for effective robotic operation:

- **Launch Files**: These provide a mechanism to start multiple nodes simultaneously, including the Master node, simplifying the initialization of complex robotic systems.
- **RViz**: A 3D visualization tool that enables real-time representation of models in an environment, displaying sensor data and navigation information to facilitate debugging and development.
- **Gazebo**: A 3D physics simulation platform that incorporates the same models as RViz, but with a robust physics engine, allowing for precise simulations that account for physical properties of the robot and its environment.
- **TF Coordinate Transformation**: This package helps track multiple coordinate systems over time and manages transformations between them. Since a robot may have multiple components and poses, this tool is essential for handling posture and movement.
- **Navigation**: A 2D navigation package that computes safe speed commands for robotic navigation, integrating data from various sensors to ensure smooth movement.

These are only a few important tools available with ROS that support researchers, developers and industry in the robot field. [5]

2.1.2 ROS1 and ROS2 comparison

ROS1 was created in 2007 for research support, but being used more and more by companies, this version had certain limitations. The first distribution of ROS2 was released in 2017 and the main goals of this version were to reduce criticalities and limitations of ROS1 and gives more support to companies and their commercial products.

Principal differences between ROS1 and ROS2:

- **Platform**: ROS2 is more extensive and supports the three platforms Linux, MacOS and Windows, while ROS1 is only supporting Linux system.
- **ROS API**: ROS1 uses independent libraries roscpp (C++) and rospy (Python), not guaranteeing to have the same features developed on both libraries. ROS2 relies on a base library Rcl implemented in C that contains the core features. From the rcl library are then built the rclpy and rclcpp for Python and C++.
- **Middelware and Data Distribution Service (DDS)**: ROS1 leveraging on TCP/IP for the middleware communication does not have great flexibility and can be slow for real-time scenarios. ROS2 adopts the Data Distribution Service standard enabling better support for real-time performance and communication policies. In Figure 2.4 the different structure for the two ROS version points out another important difference; the master node of ROS1 is not needed anymore in ROS2 due to the distributed architecture that allows nodes to communicate with each other. This difference ensure a simpler setup in ROS2 with respect to the previous version.

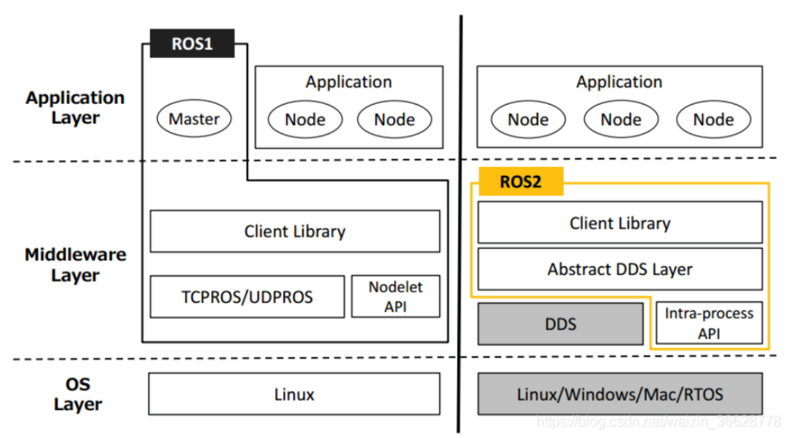


Figure 2.4: ROS1 and ROS2 architecture comparison

- **Quality of Service (QoS)**: ROS1 has low control on reliability and offers only best effort and reliable delivery. ROS2 support a robust QoS framework, enabling custom message delivery and keeping a message history, as well as other policies to adapt to different requirements on communication.
- **Real-Time**: ROS1 lack of real-time support represents a huge limitation. On the other hand, ROS2 with its real-time capabilities, guaranteed by QoS and DDS, is suitable for industrial and autonomous system with stringent timing requirements.
- **Security**: ROS2 includes built-in security features in the DDS (authentication, encryption, access control, etc). ROS1 has limited support for security features making it problematic to use in sensitive applications.
- **Parameter Server**: ROS1 manage parameters through a centralized server, while ROS2 distributes parameters directly to nodes, ensuring more scalability and flexibility
- **Node Life-cycle management**: ROS1 lacks a node lifecycle, which can lead to problematic resources management and result in a less robust system. ROS2 introduces a life-cycle, allowing node to change states (active, inactive, shutting down, etc) and enabling a better resources control.
- **Launch System**: ROS1 uses XML-based file for the launch system, meaning less flexibility and more complexity. Instead, ROS2 launch files are based on Python for improved readability, modularity and complex configuration handling.

All these improvements in ROS2 supporting real-time application and given the fact that ROS1 will reach End Of Life (EOL) in 2025, make ROS2 a good choice for developing the codes for this autonomous driving project.[7] [8]

2.2 LiDAR

The Light Detection And Ranging (LiDAR) is a distance sensing technology based on laser beams. The LiDAR can use a single laser or multi-line laser. The single laser has fast scanning speed, high resolution and high reliability, making it the mainly used in robotics and ensuring accurate measuring and accuracy.

2.2.1 Single Laser LiDAR

The single laser LiDARs are divide in two main categories: Triangular ranging and Time Of Flight (TOF).

Trigonometric ranging method

In the trigonometric ranging a laser beam is used to illuminate with a certain angle the target. The laser, scattered on the target, is reflected with another angle and captured by a Charge Coupled Device (CCD) as shown in Figure 2.5. The laser is focused on the photosensitive CCD sensor by a lens and the movement of the laser light spot on the sensor corresponds to the movement of the target. Therefore, the distance of the target can be obtained from the light spot displacement on the sensor. This displacement is calculated using the geometric triangle theorem, as the incident and reflected light form a triangle.

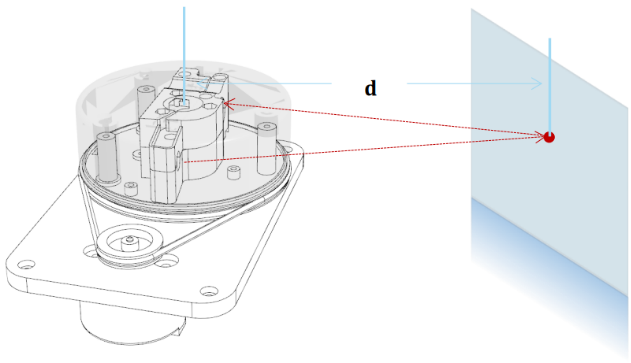


Figure 2.5: RPLIDAR Single Laser Mechanism

Starting from the angular relationship between the two light beam, two type can be derived:

- **Direct Shot type** When the laser beam is vertically incident on the target surface, so it results in the aligned with the normal vector of the surface of the target object as in Figure 2.6

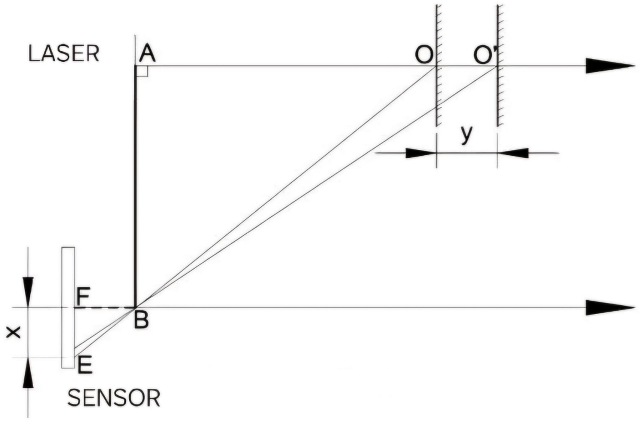


Figure 2.6: Direct shot type triangulation diagram

- **Oblique Shot type** In Figure 2.7 it is shown that in the Oblique shot the laser form an angle (less then 90 degrees) with the normal vector of the target surface.

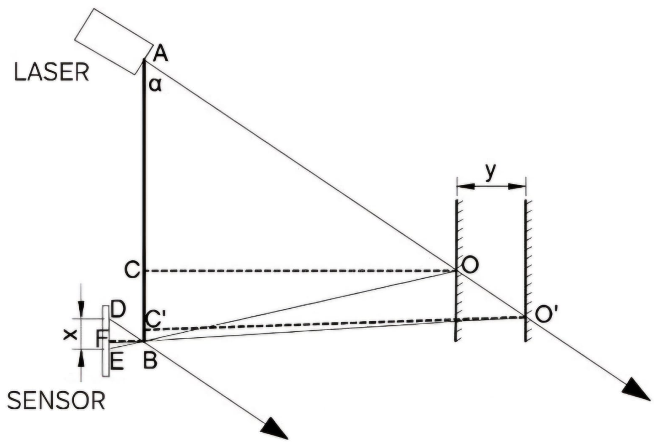


Figure 2.7: Oblique shot type triangulation diagram

Both types of the triangulation ranging method allow to achieve high precision in a non-contact measuring of the distance of the target. However, the resolution of the direct type is lower with respect to the oblique type. [9]

2.2.2 Time Of Flight method

TOF technology is an alternative to the triangular ranging method. The distance of the target is obtained by calculating the time it takes for a light pulse to travel to an object and back. A modulated laser pulse is emitted towards an object; after the laser is reflected, it returns to a sensor that calculates the distance by measuring the time difference between emission and reception (Figure 2.8). This approach is advantageous for accurately measuring large distances while maintaining stability and precision, especially in outdoor environments with strong lighting conditions.

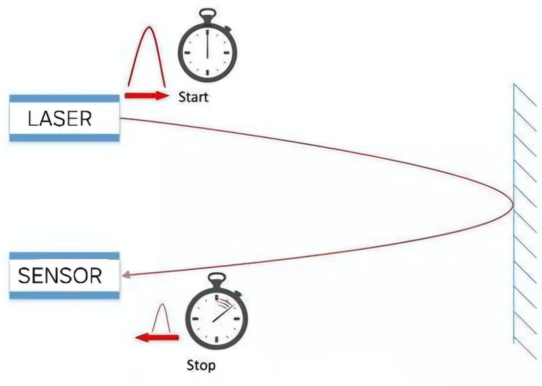


Figure 2.8: Working principle diagram of TOF LiDAR

TOF technology is widely used in applications that require high-precision mapping and real-time obstacle detection, making it particularly valuable in industrial robotics and autonomous navigation. The use of short laser pulses also minimizes interference from external light sources, enhancing the system reliability. However, the lower cost of triangular ranging LiDAR, combined with its sufficient accuracy, meets the requirements of most industrial standards and make the Triangular ranging LiDAR a valid alternative. [9]

2.3 Docker

Docker is a popular containerization platform that packages applications and their dependencies into lightweight, portable containers. This technology has become essential in fields like robotics, where it enhances scalability, consistency, and resource efficiency across various environments. [10]

2.3.1 Docker Engine

The Docker Engine is the core of Docker functionality, acting as a client-server application, responsible for building, running and managing containers. It consists in a server component known as the Docker daemon (dockerd), an Application Programming Interface (API), and a client (docker). The Docker daemon creates, manages, and monitors containers, while the Docker client provides commands to interact with the daemon, such as creating, stopping, or removing containers. This architecture allows developers to manage containerized applications efficiently and ensures that they can be built, tested, and deployed consistently across various systems. [11]

2.3.2 Containers

Containers are the primary units of Docker, encapsulating applications along with their dependencies within self-contained environments. Unlike traditional virtual machines, containers share the host system kernel, making them more lightweight and efficient in terms of resource usage. This efficiency enables the deployment of complex systems, such as those used in robotics, without the overhead of an entire operating system. Docker containers also facilitate version control and reproducibility, which are key for iterative development and testing in robotic applications. [10] [12]

In this project it is used a container to run ROS2 since it is compatible with Ubuntu 20.04 or higher. In fact, the Nvidia Jeston NANO is configured with a Software Development Kit (SDK) based on Ubuntu 18.04 and can only support ROS1 by itself.

2.4 Machine Learning and Neural Networks

Machine learning models, particularly those using deep learning, have become instrumental in robotic vision for recognizing and tracking objects. A common technique in this field is object detection, which uses bounding boxes to mark and identify object locations within an image. Bounding boxes are effective for outlining objects, aiding robots in spatial awareness by providing a simple, computationally efficient representation.

This project vision part uses the SSDMobileNet model, which combines a Single Shot multiBox Detector (SSD) with the lightweight MobileNet architecture. This model is optimized for mobile and embedded systems, allowing a balance between speed and accuracy by using depth-wise separable convolutions. The SSD model processes an image in one pass, making it faster than traditional two-stage detectors. SSDMobileNet is ideal for applications requiring efficient, real-time object detection on resource-constrained devices, such as robotics platforms [13] [14].

These models are crucial for enabling robotic perception in tasks requiring object detection, making them foundational in autonomous systems where real-time tracking and identification are critical aspect.

In this project, the SSD MobileNet model, used for object detection, runs on the Jetson Nano with Compute Unified Device Architecture (CUDA) acceleration. The CUDA Toolkit from Nvidia enables SSD MobileNet to use parallel processing on the Jetson Nano GPU, which significantly improves inference speed, allowing the model to detect and classify objects in real time. [15][16]

Chapter 3

Hardware and Architecture

In this chapter, the hardware components are described, and the architecture of the robot is explained in detail, covering both the configuration and the assembly phases. The overall structure and configuration of the ROSMASTER X3 robot is shown in Figure 3.1.

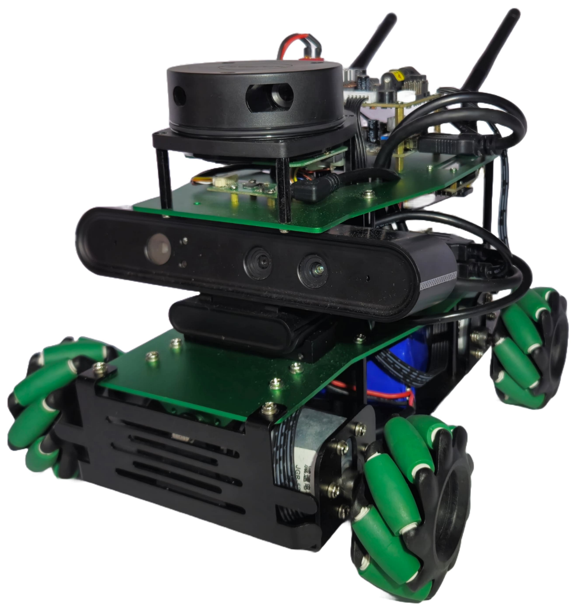


Figure 3.1: Rosmaster X3

3.1 Components

The main components and wiring diagram are illustrated in Figure 3.2. The faded components in Figure 3.2 are optional parts that are not essential and they are not included in the current configuration of the robot. Since the optional screen is not available in the current configuration, the LiDAR and the camera are connected directly to the Jetson Nano. The battery, not present in Figure 3.2, is directly connected to the ROS robot expansion board.

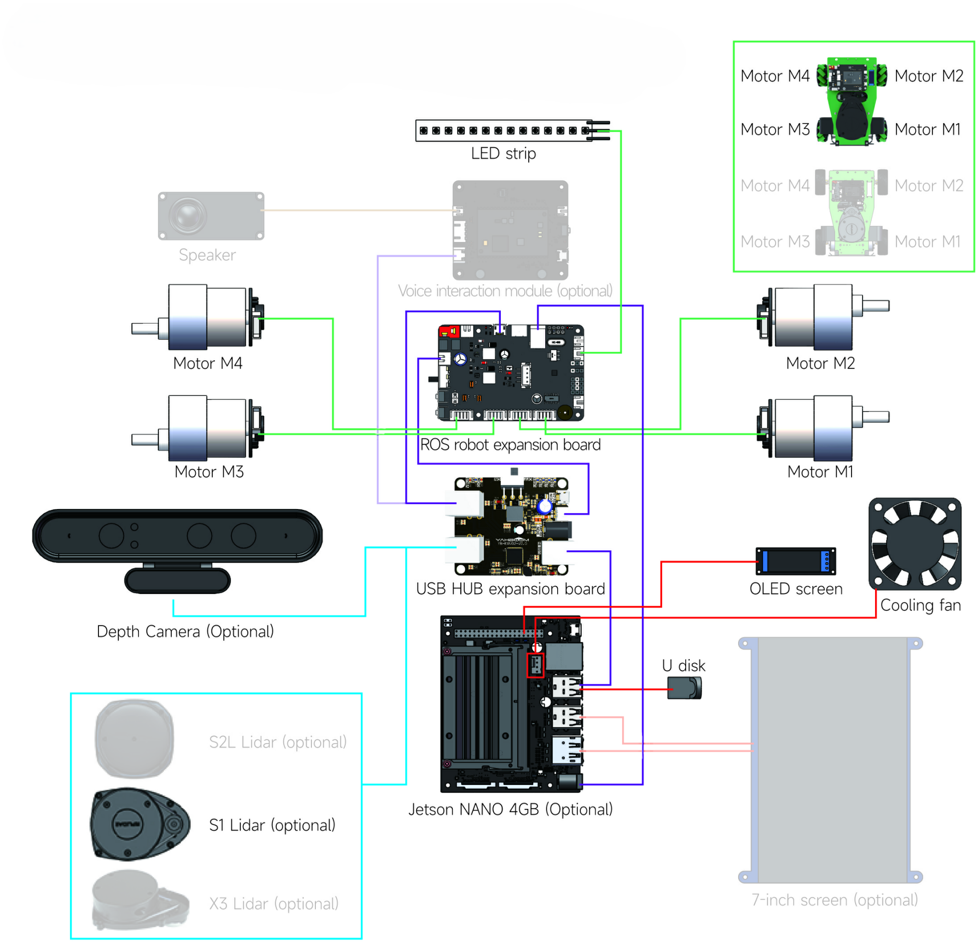


Figure 3.2: Rosmaster X3 wiring diagram and components

3.1.1 Jetson NANO

Figure 3.3: Jetson NANO 4GB

The robot main board is an Nvidia Jetson NANO 4GB SUB version developer kit (Figure 3.3). The Jetson nano is a compact entry level Artificial Intelligence (AI) computing platform, suitable for image processing, object detection and other deep learning and computer vision applications [17]. The AI computation is supported by a Graphics Processing Unit (GPU) with NVIDIA Maxwell architecture (128 NVIDIA CUDA cores), a Central Processing Unit (CPU) Quad-core ARM Cortex-A57 MPCore and 4GB 64-bit of Low Power Double Data Rate (LPDDR)4 Random Access Memory (RAM). Unlike the original B01 version from Nvidia, this SUB version from Yahboom does not have an SD card slot, but it integrates directly an embedded Multi Media Card (eMMC) storage. Since the storage is only 16GB, the operating system runs from a bootable USB disk, known as U-Disk, that allows to extend the system storage capacity. The Jetson NANO operates on 5V power source, making it a good choice for power efficiency. It also offers two power modes, 5W or 10W, to further optimization between power consumption and computational performance. The 5W mode is ideal for lightweight tasks and helps conserve energy, while the 10W mode provides enhanced processing power, beneficial for more demanding AI workloads. This flexibility allows the Jetson Nano to adapt to various application needs, whether prioritizing power savings or maximizing performance for intensive tasks. The board is also equipped with a heat sink and a fan ensuring the correct heat dissipation during high workload, maintaining stable operation even under intensive AI processing tasks. Additionally, it includes a network card, enabling remote control and access, which is essential for real-time monitoring, updates, and control of the machine from a distance. The status of the Jetson is displayed on the OLED display directly connected to it. The basic information are the CPU, RAM and storage utilization and the IP address of the network.

3.1.2 ROS Robot expansion board (STM32)

The ROS robot expansion board V1.0 from Yahboom (Figure 3.4) is the robot drive controller. The board also serves as an STM32-based development platform, equipped with an STM32F103RCT6 MicroController Unit (MCU). Since the STM32 support only serial communication, the board communicates with the Jetson NANO via USB port and it uses a CH340 USB-to-serial chip for the conversion (supporting a baud rate of 115200bps). An alternative communication option available on the board is the reserved CAN bus interface. The expansion board is directly powered by the 12V battery and provides the 5V power to the Jetson NANO as well as the 12V for the USB hub expansion board. It also features an on-board MPU9250 9-axis Inertial Measurements Unit (IMU), which provides data from its 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer. The expansion board can drive four 12V encoder motors, four Pulse Width Modulation (PWM) servos and serial bus servos, providing compatibility with various configurations. Additional peripheral interfaces include an RGB light bar and a buzzer, enhancing its suitability for interactive robotic applications. The board also features control buttons (RESET, KEY1, and BOOT0) for system management and configuration [18].

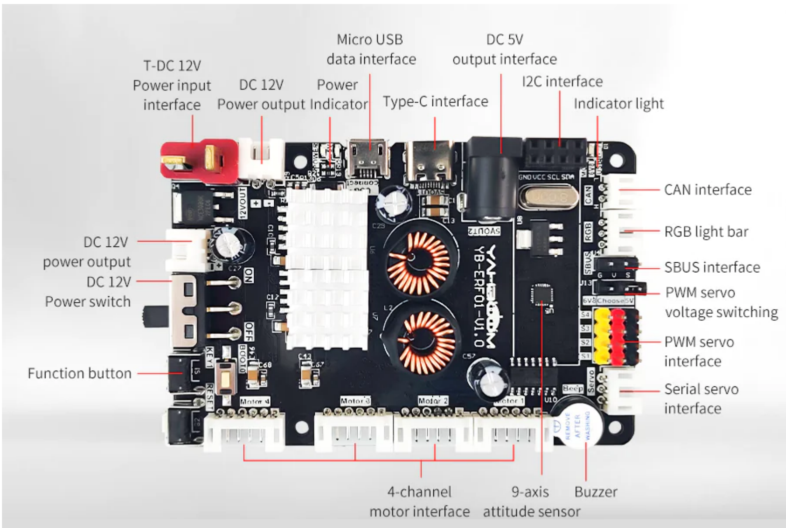


Figure 3.4: Yahboom ROS Expansion board V1.0

3.1.3 LiDAR RPLIDAR A1

The RPLIDAR A1 from Slamtech (Figure 3.5) is a single laser LiDAR sensor based on the triangulation ranging method with an oblique shot type configuration (see section 2.2.1 in Chapter 2 for details on the triangulation method). This sensor, connected diretcly to the Jetson NANO, provides the necessary spatial data for obstacle detection, enabling autonomous navigation. Table 3.1 presents the main technical characteristics of this component.

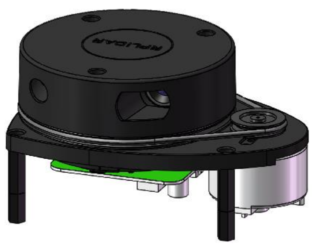


Figure 3.5: Slamtech A1 LiDAR

Parameter	**Value**
Measuring range	0.15m-12m
Sampling Frequency	8000Hz
Rotational speed	5.5Hz
Angular Resolution	≤1°
System Voltage	5V

Table 3.1: SlamTech Sillan RPLIDAR A1M8 technical information

Figure 3.6 shows the output from the LiDAR visualized in ROS-Visualization (Rviz).

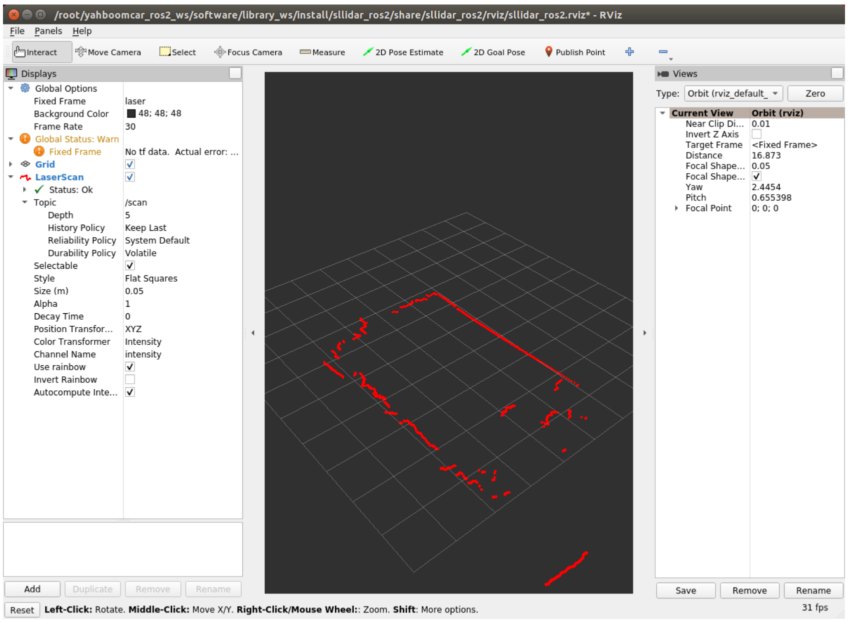


Figure 3.6: Rviz LiDAR visualization

3.1.4 RGB Depth Camera Orbbec

Figure 3.7: Orbbec Astra Pro Plus RGB depth camera

The Astra pro plus developed by Orbbec is a 3D camera based on structured light technology. This cameras project a pattern of light—often grids or dots—onto a surface. By analyzing the distortion of this pattern when it reflects off objects, the camera can calculate the depth and shape of the environment with high precision. This method is commonly used in 3D scanning and mapping applications because it provides accurate depth information even in low-light conditions [19].

Table 3.2 shows the specification of the camera. While the camera is capable of high resolution, it was kept in the default 640x480 configuration to reduce computational load. In this project only the RGB camera is used (Figure3.8a), since the machine learning model is trained on RGB frames. Consequently, the depth (Figure 3.8b) and infrared (Figure3.8c) capabilities of the Astra Pro Plus are not explored.

(a) RGB (b) Depth (c) Infrared

Figure 3.8: Astra Pro Plus camera outputs

Parameter	Value
Depth Technology	Structured Light
Wavelength	850nm
Depth resolution	640x480@30FPS
RGB resolution	1920x1080@30FPS 1280x720@30FPS 640x480@30FPS
Depth FOV	H58.4° V45.5°
RGB FOV	H66.1° V40.2°
Depth Range	0.6m-8m
Precision	±3mm @ 1m
Power Consumption	<2.4W

Table 3.2: Orbbec Astra Pro Plus Specifications and Parameters

3.1.5 Mecanum wheel

Mecanum wheels are designed for omnidirectional movements through a combination of rotational and translational motion. Each wheel consists of a central hub surrounded by rollers set at a 45-degree angle with respect to the hub axis as shown in Figure 3.9. These rollers allow the wheel to exert force in both the forward and lateral directions, depending on the direction of rotation.

Typically, four Mecanum wheels are arranged in pairs, with each pair mounted as mirror images. By controlling the direction and speed of each wheel, the robot can achieve complex motions such as moving forward, backward, and sideways, as well as rotating on a pivot point. This configuration provides flexible movement capabilities, making it ideal for applications that require precision and agility in tight spaces [20].

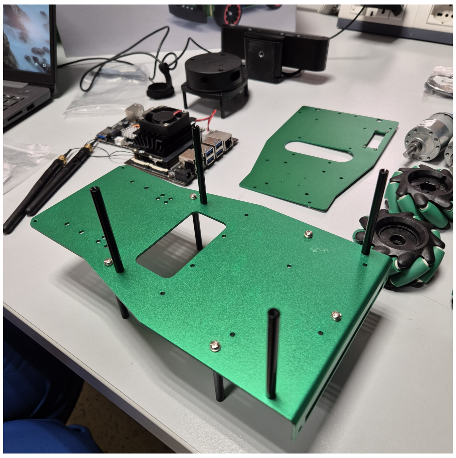
Figure 3.9: Mecanum Wheel

3.2 Assembly steps

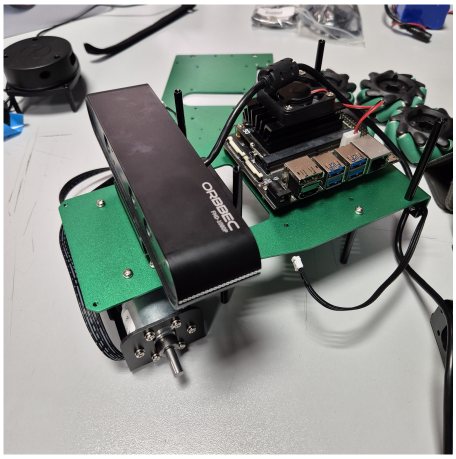
Starting with the components shown in Figure 3.10, the assembly process began with constructing the bottom frame (Figure 3.11a). Next, the camera, the Jetson Nano and the front motors were installed (Figure 3.11b). The bottom frame was then completed by adding the rear motors and the four wheels (Figure 3.11c). Afterwards, the LiDAR, USB hub expansion board, ROS expansion board and OLED display were mounted on the upper frame (Figure 3.11d).

The final steps involved installing the 12V battery, attaching the Wi-Fi antenna, and connecting all necessary cables. Upon completing these steps, the robot was fully assembled (Figure 3.12).

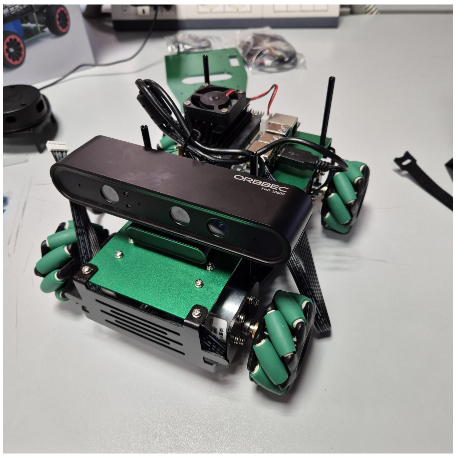
Figure 3.10: Single components



(a) Bottom frame



(b) Jetson Nano and Camera



(c) Motors and Wheels

(d) LiDAR and Expansion Board

Figure 3.11: Assembly steps

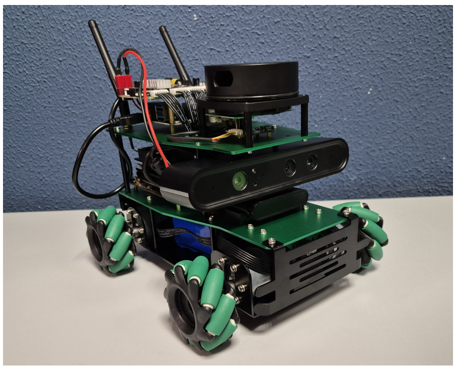


Figure 3.12: ROSMASTER X3 completly assembled

3.3 Architecture of the robot

As mentioned in the beginning of this Chapter, the ROSMASTER x3 has two main boards, the Jetson NANO and the ROS robot expansion board. The expansion board MCU (STM32) contains the low level firmware provided by the manufacturer to send direct signal to the motors, led and buzzer. The commands are sent from the Jetson NANO through ROS thanks to a Python library developed by Yahboom. This library offers function that send via USB port the commands to the expansion board. After being converted the commands are transformed in electrical signal to the motors, while the encoder information are sent from the STM32 to the Jetson NANO.

The Jetson NANO was configured with the image provided by Yahboom, which provides a preconfigured environment and useful material. The Docker container allows to solve the incompatibility between the SDK installed on the U-disk of the Jetson NANO and ROS2.

This project has been developed on this specific architecture, in particular to enable efficient communication and processing across multiple components, each responsible for specific functions in the robot operation. The two primary components are a Python-based vision script running on the Jetson Nano and a ROS node hosted within the container.

- **Python Vision Script**: The vision processing script runs directly on the Jetson Nano, leveraging its computational capabilities for image processing tasks. A machine learning model performs object detection on camera frames and based on the bounding box data, robot motion commands are sent to the ROS node within the container.

- **ROS Node within a Container**: The ROS node operates within a Docker container, providing a modular and isolated environment compatible with ROS2. In this script the data from the LiDAR are fused with the camera commands to achieve autonomous following and obstacle avoidance. The ROS node sends instructions to the STM32 micro-controller using the Python library tools.

- **STM32 Microcontroller**: The STM32 translates commands from the ROS node into motor signals using a Proportional Integral Derivative (PID) algorithm [21]. This technique allows precise and stable motors control, ensuring smoother movements and quick adjustments as needed.

3.4 Environment and System configuration

The main step for the system configuration are as follow:

- Burn the firmware `.hex` file into the STM32 using the MCUISP software
- Write the Jetson NANO image into the U-disk
- Install Docker on the Jetson NANO and download (pull) the desired image with the command:

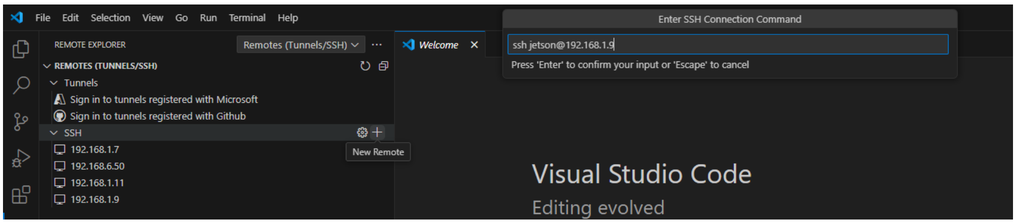
```
docker pull yahboomtechnology/ros-foxy:4.2.0
```

3.4.1 VScode and SSH connection

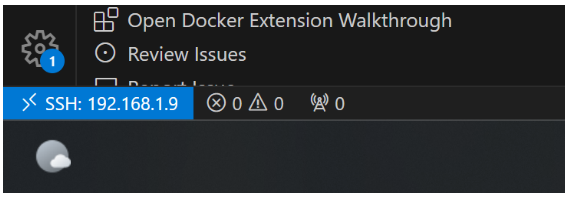
The development environment has been configured using Visual Studio Code. Remote access to the Jetson NANO is achieved creating an Secure Shell (SSH) connection. The ROSMASTER X3, with its factory image, launches a Wi-Fi network at each startup, which can be used if no other network is available. For the remote connection, it is essential that the development computer and the robot are connected to the same network. The IP address can be obtained from the OLED display connected to the Jetson. In Visual Studio Code (VScode), after downloading the extension *Remote Development* and *Remote Explorer* by Microsoft, it is possible to establish the connection with the following command, changing the IP address as in Figure 3.13a:

```
ssh jetson@192.168.1.9
```

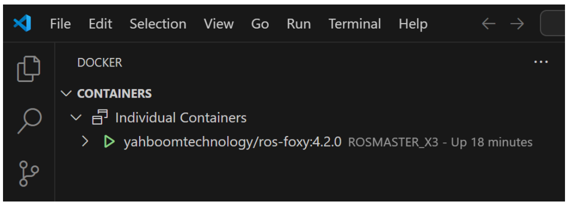
After saving the configuration file and connect to the SSH tunnel, it is possible to check the connection to the Jeston NANO (Figure 3.13b). In the new window are shown the available containers (Figure 3.13c) and starting the container it is possible to attach it to the VScode window (Figure 3.13d). This process allows to modify and develop code inside the container, offering a more complete environment with respect to the terminal editors. After saving the configuration file and connecting to the SSH tunnel, it is possible to verify the connection to the Jetson Nano (Figure 3.13b). In the new window, the available containers are displayed as in Figure 3.13c. By starting a container, it can then be attached to the VScode window (Figure 3.13d). This setup allows code modification and development directly within the container, providing a more complete environment compared to terminal-based editors.



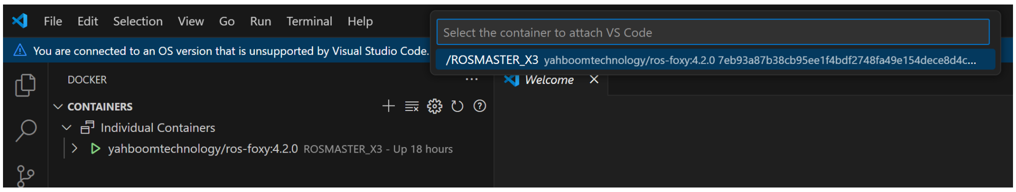
(a) New SSH connection



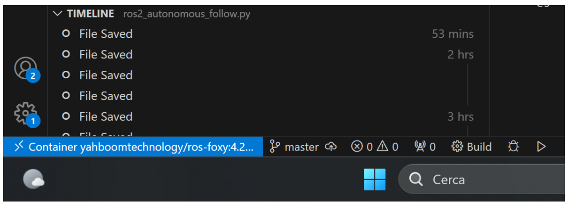
(b) SSH connected



(c) Available Container



(d) Container selection



(e) Container attached

Figure 3.13: VScode development environment configuration

3.4.2 Container configuration

Once the development environment is configured, there are two more configuration to be done in the container:

- **Car type configuration**

  As shown in Figure 3.14, the `.bashrc` file must be modified to specify the type of the robot (X3 in this project) and selecting the appropriate accessories, including A1 LiDAR and Astra Pro Plus camera.

- **Docker container launch file**

  The Bash script `run_docker.sh` (Listing 3.1) allows to run the same container, named *ROSMASTER_X3* in this project, without the need to manually start or create a new container each time. If the container exists but is stopped, the script starts it automatically. The image used for this container is `ros-foxy:4.2.0` from the Yahboom Docker repository, which is based on the Foxy ROS2 distribution. This script also handles the necessary hardware configurations by mounting specific directories and ensuring access to the external devices, including camera, LiDAR, ROS expansion board and USB hub board.

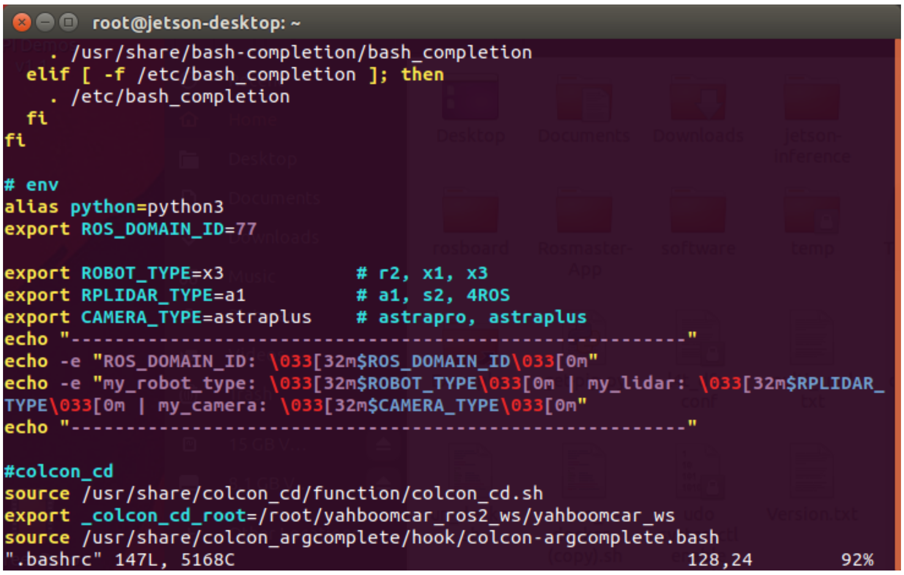


Figure 3.14: Bashrc file configuration

Listing 3.1: Bash script run_docker.sh for managing Docker container

```
#!/bin/bash

CONTAINER_NAME="ROSMASTER_X3"

xhost +

if [ "$(docker ps -aq -f name=$CONTAINER_NAME)" ];
then

    if [ "$(docker ps -q -f name=$CONTAINER_NAME)" ];
    then
        echo "Attaching to the running container..."
        docker exec -it $CONTAINER_NAME /bin/bash
    else
        echo "Starting the existing container..."
        docker start -ai $CONTAINER_NAME
    fi
else
    echo "Creating and running a new container..."
    docker run -it \
    --name $CONTAINER_NAME \
    --net=host \
--env="DISPLAY" \
--env="QT_X11_NO_MITSHM=1" \
-v /tmp/.X11-unix:/tmp/.X11-unix \
-v /home/jetson/temp:/root/yahboomcar_ros2_ws/temp \
-v /home/jetson/rosboard:/root/rosboard \
-v /home/jetson/maps:/root/maps \
-v /dev/bus/usb/001/009:/dev/bus/usb/001/009 \
-v /dev/bus/usb/001/007:/dev/bus/usb/001/007 \
--device=/dev/myserial \
--device=/dev/rplidar \
--device=/dev/input \
--device=/dev/astradepth \
--device=/dev/astrauvc \
--device=/dev/video0 \
-p 9090:9090 \
-p 8888:8888 \
yahboomtechnology/ros-foxy:4.2.0 /bin/bash
```

Chapter 4

Software

In this chapter, the code development is presented; beginning with the initial firmware test, followed by the Python vision script and concluding with the ROS node for autonomous driving.

4.1 Firmware and STM32

The firmware was not developed as part of this project, but was used to test individual components of the robot, such as motors and light bar. The initial component tests were conducted using *STM32CubeIDE*. This Integrated Development Environment (IDE) allows direct configuration of the STM32 pins (Figure 4.1) and setting of all the relevant MCU parameters, including the clock. After configuring the MCU, the STM32CubeIDE generates code that includes the predefined setup, enabling efficient integration of hardware testing functions. The project is then compiled, producing the `.hex` file (hexadecimal data format). The firmware (.hex file) is then flashed onto the STM32 using the MCUISP tool [22]. During the burning phase, the MCU must be set to programming mode, that is entered holding the *BOOT0* button and pressing the *RESET* button of the ROS expansion board.

Uploading the single functions of the firmware on the MCU allows to test the hardware, isolating each component by activating them with the *KEY1* button and understand the firmware.

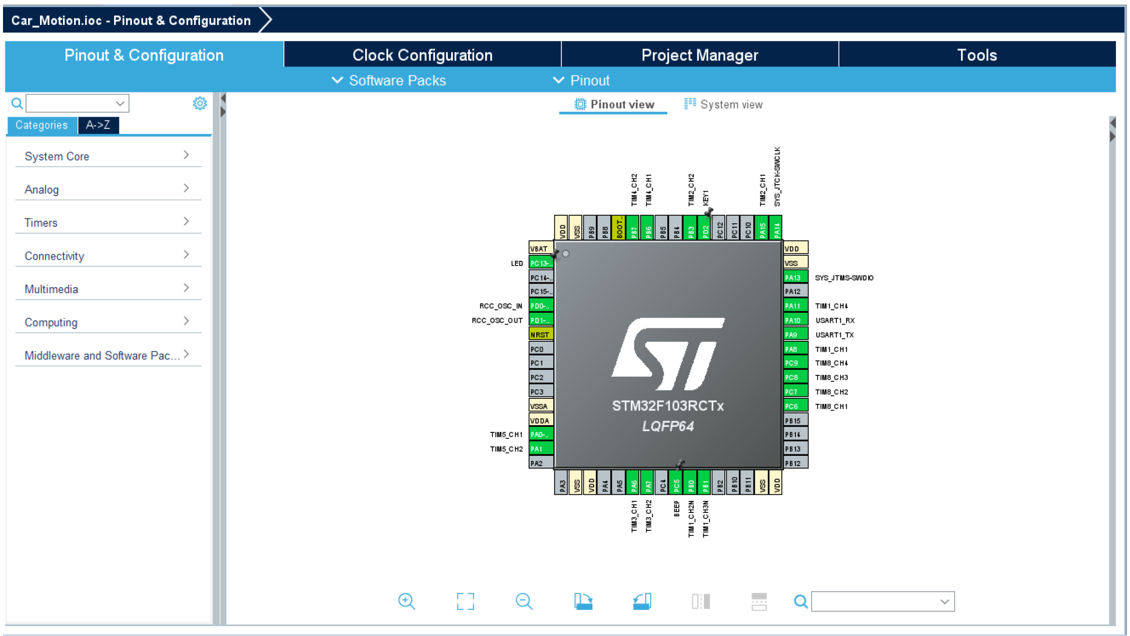


Figure 4.1: Pinout Configuration in STM32CubeIDE

4.2 Rosmaster Library

This library defines the *Rosmaster* class that contains the serial communication definition and functions that enable to send commands to the STM32. In this project four function from the *Rosmaster* class are used:

- **set_car_motion(self, v_x, v_y, v_z)**
  The `set_car_motion` function sets the speeds for the three motion axes: **Vx**, **Vy**, and **Vz**, controlling the robot movement. According to the documentation from Yahboom, **Vx** (linear speed in m/s) determines forward or backward movement, **Vy** (lateral speed in m/s) sets the side-to-side movement, and **Vz** (angular speed in rad/s) controls the turning rate around the robot vertical axis [23]. During testing, it was observed that the **Vy** and **Vz** axes are inverted compared to this documentation. Therefore, adjustments were made in the ROS code accordingly to achieve the correct motion.

- **set_pid_param(self, kp, ki, kd, forever=False)**
  The `set_pid_param` function enables control over motor response and stability by adjusting the primary PID parameters: proportional (Kp), integral (Ki), and derivative (Kd). The function fourth parameter determines the persistence of these settings: if set to *false*, the PID adjustments are temporary, while setting it to *true* writes the parameters to the MCU, making the modification permanent.

- **set_colorful_lamps(self, led_id, red, green, blue)**
  The `set_colorful_lamps` function is designed to control the robot light bar, allowing for customization of the RGB color for either individual LEDs or the entire light bar at once.

- **get_battery_voltage(self)**
  The `get_battery_voltage` function provides the battery voltage, enabling the monitoring of the robot power level.

Each function includes *self* as the first parameter, which is a reference to the instance of the *Rosmaster* class calling the function. This key parameter allows access to the class attributes and methods in Python.

4.3 Python script

This Python script was originally developed in the the Master thesis *Deep Learning-Based Real-Time Multiple-Object Detection on a rover* (2021, [24]). This code was written and configured on a Jetson NANO with a Raspberry PI camera.

The program is based on a graphical menu offering different AI functionalities and the adaption for this project cover only the `Follow Me` function (Appendix A). The code is divided in a main script, `object_detection_module.py`, and other auxiliary modules: `follow_me_module.py`, `safe_rover_module.py` and `centroid_tracking_module.py`.

The **Follow_Me** function in `object_detection_module.py` utilizes the camera and the object detection model (SSD-mobilenet-v2) to detect people, select the target and send commands to the ROS script. The function begin by initializing the camera, detection model and control parameters. In the main loop, the program processes the frames to identify people and it tracks their centroids. The target to follow is selected when an activation sequence is recognized ([24, pp. 43–45]). Movement commands are then sent based on the target bounding box position in the frame. If the target is lost or a switch is detected, additional modules handle the situation.

The first modification involved integrating of the Astra Pro Plus camera and adjusting of the parameters to meet the new camera resolution (640x480) across all scripts. Changing the frame dimensions required redefining the frame parameters relative to the person position as shown in Listing 4.1. Additionally, the new configuration enabled higher Frame per Second (FPS), which involved adjusting the allowable frame count for target disappearance in the `safe_rover_module.py`.

In the original logic, the centroid could only be on either the left or right side of the frame and a stop alert was triggered based on the bottom of the bounding box to indicate a possible collision. The script needed to be adapted to send more information for controlling the robot.

4.3.1 Adaptation of the Follow Me function

In this project, the centroid position is categorized into left, front, and right sections to avoid unnecessary adjustments for very small angles near the center.

The original logic for triggering a stop alert was based on the position of the bounding box bottom edge. However, due to oscillations in the bounding box, the stop command could be activated multiple times, as will be further detailed in the testing chapter. The solution in this project, as shown in Listing 4.2, maintains the bottom of the bounding box as an input, but introduces two distinct thresholds: the *backward* command is only sent if the person is very close, while a *stop* command uses a wider range to prevent intermittent movement. Additionally, a delay is utilized to minimize uneven motion and reduce the impact of rapid, repeated commands.

Listing 4.1: **Follow_me** function initialization

```python
def Follow_Me():

    timeStamp = time.time()
    fpsFilt = 0
    net = jetson_inference.detectNet('ssd-mobilenet-v2', threshold
        =.65)
    dispW = 640
    dispH = 480
    danger_threshold = 465
    close_threshold = 450
    stop_duration_threshold = 0.7
    last_stop_time = None
    flip = 2
    font = cv2.FONT_HERSHEY_SIMPLEX

    cam=cv2.VideoCapture(0)
```

Listing 4.2: **Follow_me** function Main Loop

```python

        for (objectID, centroid) in objects.items():
            # draw both the ID of the object and the centroid of the
                object on the output frame
            text = "ID {}".format(objectID)
            cv2.putText(img, text, (centroid[0] - 10, centroid[1] -
                10),
            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
            cv2.circle(img, (centroid[0], centroid[1]), 4, (0, 255, 0)
                , -1)

            if objectID == target:
                angle = (np.arctan(abs(centroid[0] - (640 / 2)) /
                    (480 - centroid[1] + 0.01)) * 180 / math.pi)
                if target in bounding_boxes:
                    _, _, _, bottom = bounding_boxes[target]
                # If person is too close (stop area)
                if bottom >= close_threshold and bottom <
                    danger_threshold:
                    send_command("stop")
                    cv2.putText(img, "STOP", (200, 260), font, 1,
                        (0, 0, 255), 2)
                    last_stop_time = time.time()  # Record the
                        time when we stopped

                # If person is dangerously close
                elif bottom >= danger_threshold:
```

```
            send_command("backward")
            cv2.putText(img, "MOVE BACKWARD!", (220, 260), font, 1, (0, 0, 255), 2)
            last_stop_time = time.time()  # Reset stop timer when moving backward

        # If person is far enough to resume movement
        else:
            # Only resume if stop time has passed since the last stop message
            if last_stop_time and time.time() - last_stop_time >= stop_duration_threshold or last_stop_time==None:
                if centroid[0] < 640 / 2 - 10:
                    send_command("left", int(angle))
                    cv2.putText(img, "GO LEFT " + str(int(angle)) + "deg", (0, 40), font, 1, (255, 255, 255), 2)
                elif centroid[0] > 640 / 2 + 10:
                    send_command("right", int(angle))
                    cv2.putText(img, "GO RIGHT " + str(int(angle)) + "deg", (440, 40), font, 1, (255, 255, 255), 2)
                else:
                    send_command("forward")
                    cv2.putText(img, "GO STRAIGHT", (200, 40), font, 1, (255, 255, 255), 2)

                # Reset last_stop_time since we're moving again
                last_stop_time = None

            cv2.line(img, (int(640 / 2), 480), (centroid[0], centroid[1] + 150), (255, 255, 255), 2)
```

4.4 ROS2 script

The `ros2_autonomous_follow.py` (Appendix B) contains the main logic and sensor fusion between camera and LiDAR.
The code is organized as follow:

- **Initialization and Setup**
  The *AutonomousFollower* class inherits from *Node* to become a node itself, enabling it to interact with the ROS2 network. The STM32 communication is initialized via the *Rosmaster* library. MQTT is configured for receiving remote commands such as activate, deactivate, stop, and movement directions, allowing the robot behavior to be managed remotely. ROS2 Subscriptions: the script subscribes to the /scan topic to receive LiDAR data for real-time obstacle detection.

- **Control Parameters and Flags**
  Parameters for speed, control angles and obstacle detection range and thresholds are defined to enable fine-tuning of the ROSMASTER X3 behavior. Flags and state variables track whether the target is detected, if obstacles are present, and if the follow-me mode is active.

- **Timers**
  The 0.1-second command timer calls follow_person_and_avoid_obstacles function, ensuring continuous checks for updated LiDAR data and making decisions to follow or avoid obstacles. A battery timer set to trigger every 5 seconds fetches and publishes the battery voltage using publish_battery_info, which allows for constant monitoring. For safety reasons, a watchdog timeout timer is reset with each received command. stop_robot_due_to_inactivity is triggered if no messages are received within the specified threshold (watchdog expired). This ensures the robot stops for safety if communication is interrupted.

- **MQTT Communication**
  The on_message method processes incoming MQTT messages, such as activation commands and movement directions (e.g., left, right, forward, stop, emergency stop and backward), translating them into actions for the robot.

- **LiDAR-Based Obstacle Detection**
  The lidar_callback function processes LiDAR data to detect obstacles on the left, right, and front. It categorizes the warnings based on distance and direction and updates flags accordingly. Dynamic responses are triggered based on obstacle positions, adjusting the robot trajectory or stopping if an obstacle is too close.

- **Follow and Avoid Logic**
  The follow_person_and_avoid_obstacles function manages the robot movements

when the follow-me mode is active. If no obstacles are detected, it follows the person position; while in the presence of obstacles, the robot adapts its motion:

  - Left or Right Obstacle: it moves in opposite direction translating while still going forward.
  - Frontal Obstacle: the robot goes backward.
  - Front-Left or Front-right Obstacle: it translates in opposite direction
  - Multiple Obstacles: it stops completely.

- **Shutdown Routine**
  When the program is interrupted to exit (KeyboardInterrupt), the robot motors are stopped, the LEDs deactivated, the data logs file are close and finally the node is destroyed.

The script logs commands and LiDAR data to CSV files for post-analysis. Additionally, commands and status information are printed to the terminal for real-time monitoring.

4.4.1 Development stages

The code development began without incorporating LiDAR data, focusing first on fine-tuning the movement commands for smoother motion. Once the robot movements were optimized, *LaserScan* data were introduced to stop the robot upon detecting obstacles. After adjusting the detection parameters, the final stage involved enabling dynamic obstacle avoidance while continuing to follow the target.

4.4.2 Robot Motion commands

Messages from the vision script are processed, and for direction commands, the direction and angle are extracted. Listing 4.3 illustrates how the *forward*, *left* and *backward*commands are handled. The *forward* command sets the linear speed to its maximum value if no obstacles are detected within the front clearance range; otherwise, the linear speed is adjusted based on the distance to the nearest object in front as in equation 4.1.

$$\text{percentage} = \frac{\text{self.distance\_front} - \text{self.response\_dist\_front}}{\text{self.response\_clearance\_front} - \text{self.response\_dist\_front}} \tag{4.1}$$

Although the maximum speed of the robot is higher, it has been reduced after testing to achieve smoother motion

The *left* command adjust differently the speed and turning motion depending on the angle:

- below **15 degrees**: the robot translates and moves forward for small angle adjustments.
- from **15 degrees** to **28 degree**: the robot turns on its vertical axis at reduced speed
- over **28 degrees**: the robot turns on its vertical axis at higher speed

The *right* command follow the same logic, but with negative values for speed. In the *backward* command, flags are used to check if an obstacle is close in front, reducing the impact of incorrect backward and stop commands from the vision script. This approach helps prevent unintended stops, allowing the robot to keep moving smoothly. After executing the backward command, a brief delay is introduced before resuming normal operations. The *stop* command follows the same logic, with a similar delay to ensure the robot can only move again after a designated pause.

Listing 4.3: Commands Logic

```
        elif command == "forward" and self.follow_me_active:
            if not self.obstacle_detected and not self.emergency_stopped:
                log_time = datetime.now().strftime("%M:%S.%f")[:-3]
                if self.obstacle_clearance_front:
                    """Full forward speed if no obstacle in front"""
                    self.car.set_car_motion(self.linear_speed, 0, 0)
                    print("Moving forward fast")
                    self.command_writer.writerow([log_time, "forward", 0, self.linear_speed, 0, 0])

                else:
                    """Proportional forward speed wrt to front obstacle/person"""
                    percentage = (self.distance_front - self.response_dist_front) / (self.response_clearance_front - self.response_dist_front)
                    self.car.set_car_motion(self.linear_speed*percentage, 0, 0)
                    print(f"Moving forward proportionally with speed :{self.linear_speed*percentage}")
                    self.command_writer.writerow([log_time, "forward", 0, self.linear_speed*percentage, 0, 0])

```

```python
        elif command == "left" and self.follow_me_active:
            log_time = datetime.now().strftime("%M:%S.%f")[:-3]
            if (angle<=15):
                """Translating when angle below 15 degrees"""
                adjusted_linear_speed = self.linear_speed * 0.7
                if not self.obstacle_detected and not self.emergency_stopped:
                    self.car.set_car_motion(adjusted_linear_speed,0, self.lateral_speed)
                    print(f"Translating left with lateral speed: {self.lateral_speed}, linear speed: {adjusted_linear_speed}")
                    self.command_writer.writerow([log_time, "left", angle, adjusted_linear_speed, self.lateral_speed,0])
            elif(angle>15 and angle<=28):
                """Turning slow when angle below 28 degrees"""
                adjusted_angular_speed = self.angular_speed*0.05
                adjusted_linear_speed = self.linear_speed * 0.6
                if not self.obstacle_detected and not self.emergency_stopped:
                    self.car.set_pid_param(0.1,3,3, 0)
                    self.car.set_car_motion(adjusted_linear_speed, adjusted_angular_speed, 0)
                    print(f"Turning left with angular speed: {adjusted_angular_speed}, linear speed: {adjusted_linear_speed}")
                    self.command_writer.writerow([log_time, "left", angle, adjusted_linear_speed, adjusted_angular_speed])
            else:
                """Turning faster when angle over 28 degrees"""
                adjusted_angular_speed = self.angular_speed*0.08
                adjusted_linear_speed = self.linear_speed * 0.5
                if not self.obstacle_detected and not self.emergency_stopped:
                    self.car.set_pid_param(0.1,3,3, 0) # PID parameter adjustment
                    self.car.set_car_motion(adjusted_linear_speed, adjusted_angular_speed, 0)
                    print(f"Turning fast left with angular speed: {adjusted_angular_speed}, linear speed: {adjusted_linear_speed}")
                    self.command_writer.writerow([log_time, "left", angle, adjusted_linear_speed, adjusted_angular_speed])

        elif command == "backward":
            log_time = datetime.now().strftime("%M:%S.%f")[:-3]
            if self.follow_me_active:
```

```
                    if not self.obstacle_avoid_all and self.front_stop_area:
                        adjusted_linear_speed = self.linear_speed * 0.2
                        self.car.set_car_motion(-adjusted_linear_speed, 0, 0)
                        print("Robot moving backwards")
                        self.set_led_color(255, 0, 0)  # Red LED for moving backwards
                        self.command_writer.writerow([log_time, "backwards", 0, -adjusted_linear_speed, 0, 0])
                        sleep(0.6)
                    else:
                        # Ignore the stop command if the bounding box is faulty but there's no obstacle in front
                        print("Backward command ignored due to clear LiDAR data in front.")
```

4.4.3 LaserScan Data

The LiDAR callback function processes the data from each scan received from the *LaserScan* subscriber. For each scan, points below the threshold distance are counted for the front, left, and right regions based on the angles illustrated in Figure 4.2:

- **0°-60°**: backward area
- **60°-160°**: lateral area
- **160°-180°**: front area

The angles are mirrored, resulting in the following coverage within the 360° range: 40° for front obstacles, 100° for each lateral area (left and right), and 120° for the not scannable backward area. This backward section is not scanned due to the ROSMASTER X3 construction and the position of the WiFi antenna, which obstructs effective backward scanning by the LiDAR. If the number of points in any direction exceeds the threshold, the corresponding flag is activated as illustrated in Listing 4.4, adjusting the robot response accordingly.

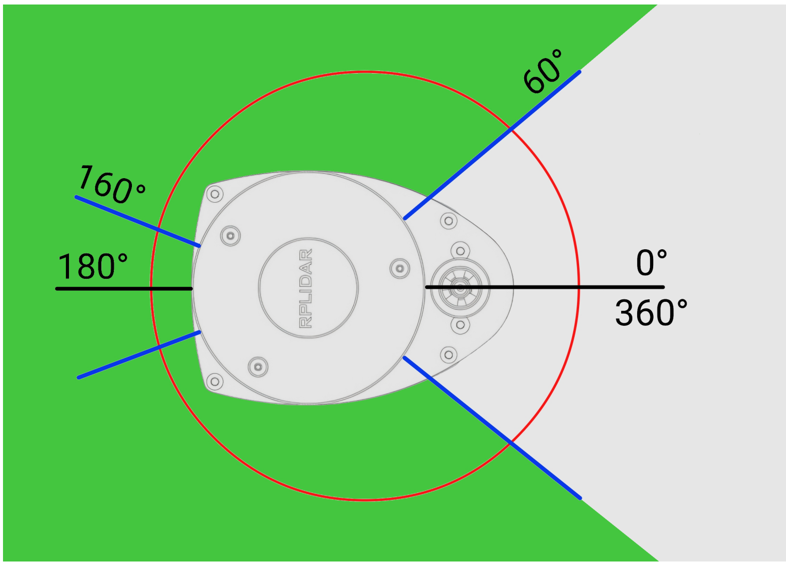


Figure 4.2: LiDAR angles and range

Listing 4.4: LiDAR obstacles detection

```
for i in range(len(ranges)):
                log_time = datetime.now().strftime("%M:%S.%f")[:-3]
                angle = (scan_data.angle_min + scan_data.angle_increment * i) * RAD2DEG
                if 160 > angle > 180 - self.laser_angle:
                    if ranges[i] < self.response_dist_lat:
                        self.right_warning += 1

                if -160 < angle < self.laser_angle - 180:
                    if ranges[i] < self.response_dist_lat:
                        self.left_warning += 1

                if abs(angle) > 160:
                    self.distance_front = ranges[i]
                    self.lidar_writer([log_time, self.distance_front])
                    if ranges[i] <= self.response_dist_front:
                        self.front_warning += 1
                    if ranges[i]  < self.response_clearance_front:
                        self.obstacle_clearance_front = False
                    if ranges[i]  >= self.response_clearance_front:
                        self.obstacle_clearance_front = True
                    if ranges[i]  < self.response_clearance_front -0.5:
                        self.front_stop_area = True
                    if ranges[i]  >= self.response_clearance_front -0.5:
                        self.front_stop_area = False
```

4.4.4 LED light status

The LED light bar is used to show the robot status as shown in Figure 4.3:

- **Blue light**: The robot is initialized and ready for follow-me mode activation.
- **Green light**: Follow me activated with no obstacle or errors detected.
- **Red light**: Obstacles are detected or errors occur due to target loss or a high probability of target switching.

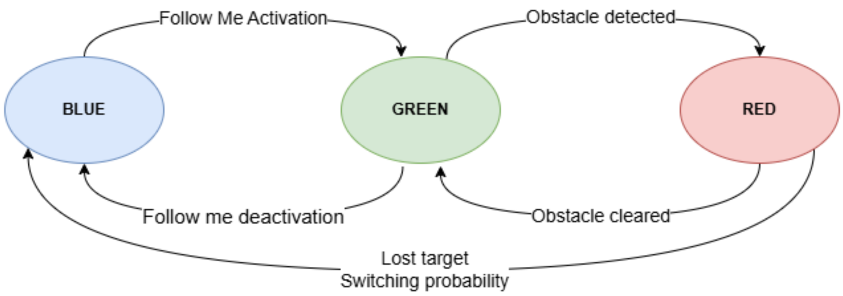


Figure 4.3: LED lights status diagram

4.4.5 Launch file

A launch file is a specialized script in ROS that enables the startup of multiple nodes, simplifying the process of running complex programs. The file `autonomous_follow_launch.py`, Listing 4.5, is used to launch two main components of this project:

- `sllidar_launch.py`
- `ros2_autonomous_follow.py`

The `sllidar_launch.py` script launches the LiDAR node to obtain sensor data, while the second script, `ros2_autonomous_follow.py`, launches the node responsible for the autonomous following logic and sensor fusion.

Listing 4.5: **autonomous_follow_launch.py**

```python
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node

def generate_launch_description():
    # Path to the sllidar launch file
    sllidar_launch_file = '/root/yahboomcar_ros2_ws/software/library_ws/src/sllidar_ros2/launch/sllidar_launch.py'

    # Include the Lidar launch file using the correct path
    lidar_node = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(sllidar_launch_file)
    )

    # Launch of ros2_autonomous_follow executable
    follow_node = Node(
        package='pkg_autonomous',  # Package name
        executable='ros2_autonomous_follow',  # Executable
        name='autonomous_follow',  # Node name
        output='screen',
        parameters=[
            {"mqtt_broker": "localhost"},
            {"mqtt_port": 1883}
        ]
    )

    # LaunchDescription that includes the lidar node and the autonomous follow node
    return LaunchDescription([lidar_node, follow_node])
```

4.5 MQTT comunication

Message Queuing Telemetry Transport (MQTT) is a lightweight communication protocol based on the TCP/IP framework.

It provides a simple network communication mechanism, allowing commands to be sent to the container running ROS2. Listing 4.6 shows the parameters and commands. Since communication occurs on the same machine (the Jetson Nano), broker_ip is set to *localhost*. The topic for message transmission is defined and must be consistent across both scripts. Finally, the MQTT client is initialized, and the connection is established. The messages defined for this project are: *activate*, *deactivate*, *stop*, *emergency_stop* and *direction*, which includes direction information based on the target position.

Listing 4.6: MQTT implementation on Vision script

```python
import paho.mqtt.client as mqtt

# MQTT broker details
broker_ip = "localhost"
broker_port = 1883  # Default MQTT port

# MQTT topic
topic = "robot/control"

def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("Connected to MQTT Broker!")
    else:
        print(f"Failed to connect, return code {rc}")

# Initialize the MQTT client
client = mqtt.Client()

# Set up the connection callback
client.on_connect = on_connect

# Connect to the broker
client.connect(broker_ip, broker_port, 60)

# Start the loop
client.loop_start()


def send_command(direction, angle=None):
    command = {"command": direction}
    if angle is not None:
        command["angle"] = angle
    client.publish("robot/control", json.dumps(command))

```

```
def send_activate():
    command = {"command": "activate"}
    client.publish("robot/control", json.dumps(command))

def send_deactivate():
    command = {"command": "deactivate"}
    client.publish("robot/control", json.dumps(command))

def send_emergency_stop():
    command = {"command": "emergency_stop"}
    client.publish("robot/control", json.dumps(command))

def send_stop():
    command = {"command": "stop"}
    client.publish("robot/control", json.dumps(command))
```

4.5.1 System Architecture and Communication Overview

Following the explanation of individual software components, this section provides an integrated overview of the overall system architecture, as previously detailed in Chapter 3.3. This summary aims to clarify the communication flow between the vision processing, decision-making, and hardware control layers of the system.

Figure 4.4 illustrates the interaction of these components within the established framework:

- Vision Script (blue block) identifies and tracks the target in real-time, sending directional commands.
- ROS2 Node (green block) receives and processes these commands, integrating sensor data for obstacle avoidance and translating them into executable actions.
- STM32 Firmware (gray block) carries out the motion commands on the robot, controlling motor signals and LED lights.

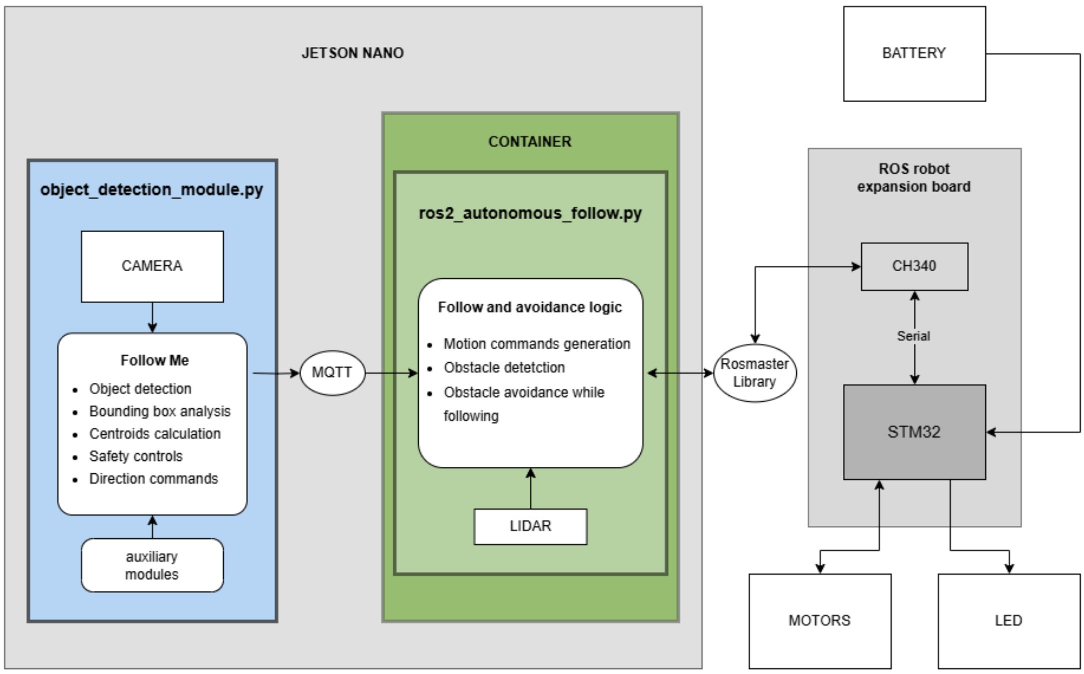


Figure 4.4: System overview diagram

Chapter 5

Evaluation and Testing

After developing the code, it is required to build the ROS package to ensure that all elements within the package are properly compiled, dependencies are effectively handled and the resulting executable is prepared to run. The package developed for this project, `pkg_autonomous`, contains the two main files: `ros2_autonomous_follow.py` and the `autonomous_follow_launch.py`.

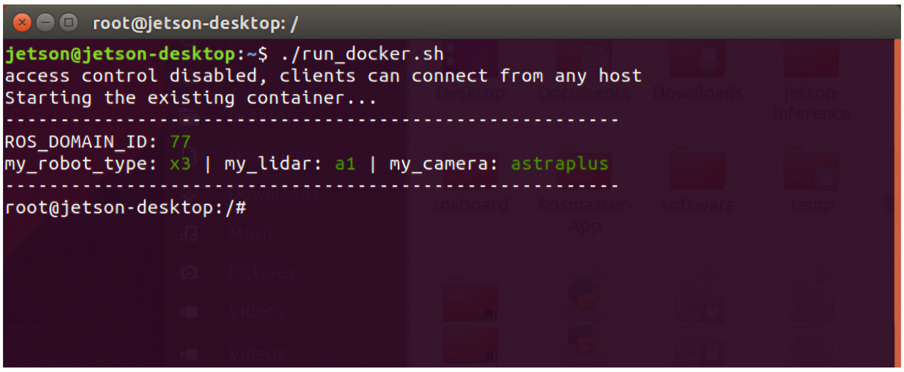


Figure 5.1: Docker container startup

Once the container is run by executing `run_docker.sh` (Figure 5.1), it is necessary to navigate to the ROS2 workspace to initiate the building process. The following ROS2 command, shown in Figure 5.2, allows building only the selected package:

```
colcon build --packages-select pkg_autonomous
```

After the build process, the command below must be executed to update the environment variables, ensuring the newly built packages are included:

```
source ~/yahboomcar_ros_ws/yahboomcar_ws/install/setup.bash
```

This step is crucial for tools like `ros2 launch` and `ros2 run` to locate and execute the packages and their associated files.

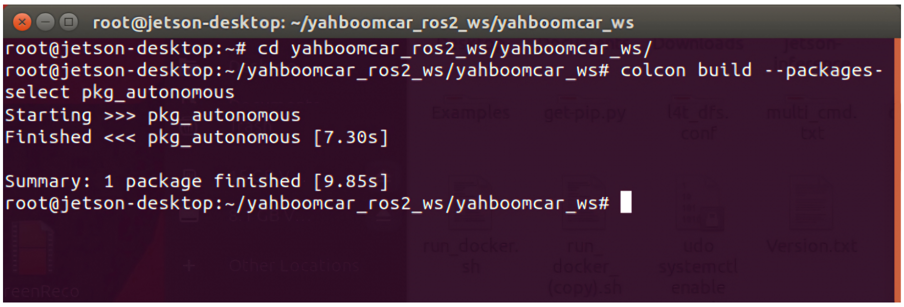


Figure 5.2: ROS2 building command

5.1 System Startup Procedure

The startup procedure is composed of the following steps:

- Start the container by launching the `run_docker.sh` script.
- Activate the ROS2 nodes within the container terminal using the command:

  ```
  ros2 launch pkg_autonomous autonomous_follow_launch.py
  ```

- Execute the Vision script from a different terminal using the command:

  ```
  python3 object_detection_module.py
  ```

- Select the correct mode in the menu of the Vision Script:

  Dynamic Modes → Follow Me

5.2 Initial Adjustments

During the initial testing of the Vision script after its adaptation, a significant performance issue was encountered. The frame rate was excessively low, approximately 3 FPS, even in the Jetson NANO 10 W power mode (Figure 5.3). This low frame rate made the system incapable of reliably detecting and tracking a person. To address this problem, the Jetson NANO clock settings were modified to unlock its full performance capabilities, reaching a frame rate around 16 FPS under no additional computational load.

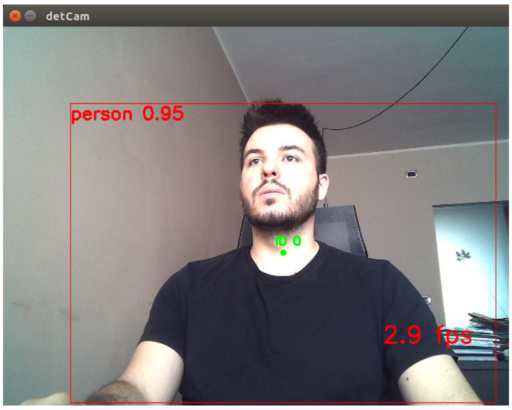


Figure 5.3: Low FPS problem

In the early evaluation of the ROSMASTER X3 movement, an issue was identified with the robot turning motion around its vertical axis. Beyond the inverted axis problem discussed in Chapter 4.2, the motor response was excessively aggressive, causing the robot to spin uncontrollably and lose track of its target. Initial attempts to resolve the issue included lowering the PID parameters and angular speed; however, these changes did not provide any improvements in the robot performance.

The problem was resolved by upgrading the `Rosmaster Library` from version 3.3.6 to version 3.3.9, which resulted in a more effective PID control. After the upgrade, adjusting the proportional Kp parameter of the PID algorithm to a value of 0.1 resulted in smoother motor responses, ensuring accurate turning motion of the robot.

The bounding box inaccuracy is shown in Figure 5.4. The bottom edge of the bounding box determines the stop and backward commands logic. When the bounding box is lower than the actual position of the person and enters the stop range, it results in unintended stops while following the target. As introduce in Chapter 4.4.2, LiDAR data is utilized to verify if the target (treated as an obstacle) is truly close to the front of the robot.

Figure 5.4: Oversized bounding box

Initial tests were conducted over short distances to fine-tune parameters for smoother movements and more balanced LiDAR-based obstacle detection. Vision script parameters were adjusted to ensure the robot maintained a distance of approximately 1 meter from the target. The maximum allowed number of disappearance frames was set to 30, corresponding to 2 seconds based on the Jetson NANO performance of 12–14 FPS under full load.

5.3 Robot behavior and Data analysis

After the system initialization, Figure 5.5, the robot is ready for operation.

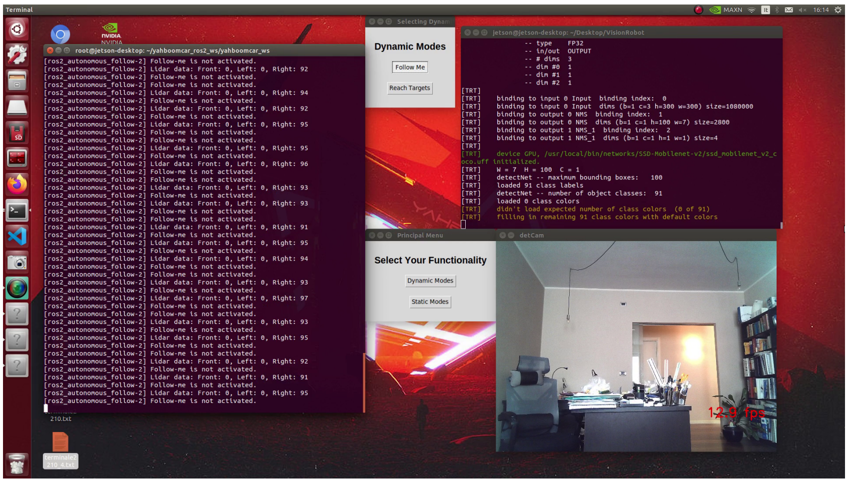
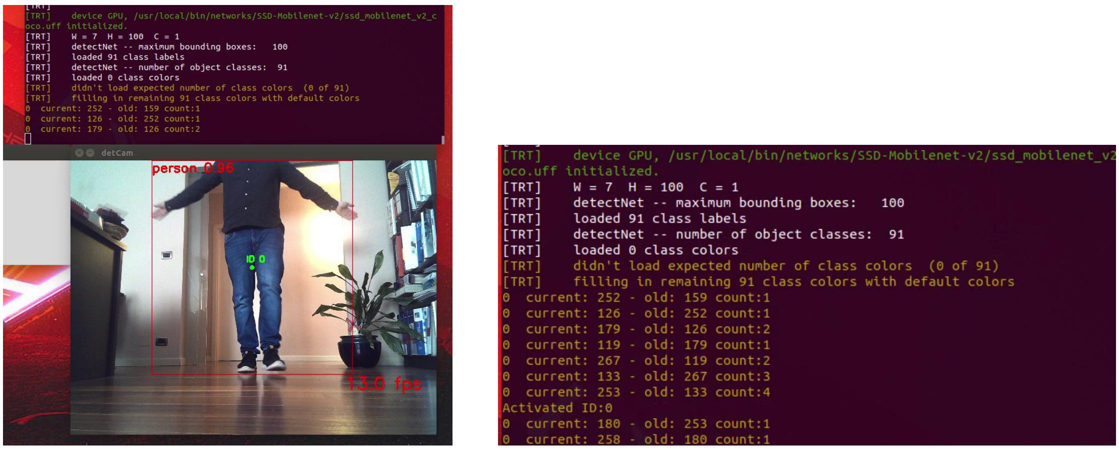


Figure 5.5: System initialized

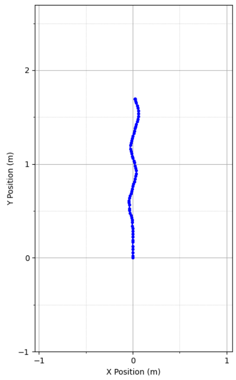
The Follow Me function activation process involves the target widening and closing the arms for three times, as illustrated in Figure 5.6. This method leverages on the bounding box width to activate the robot, allowing it to start following the target.



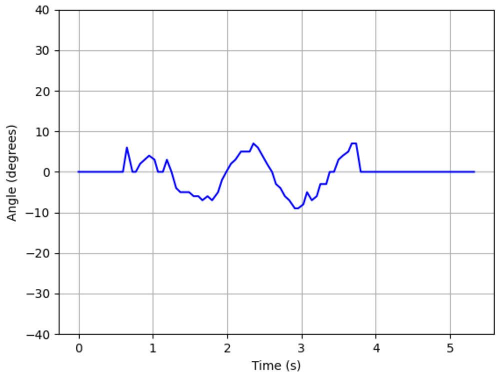
(a) Activation movements (b) Activation counter

Figure 5.6: Follow me activation Procedure

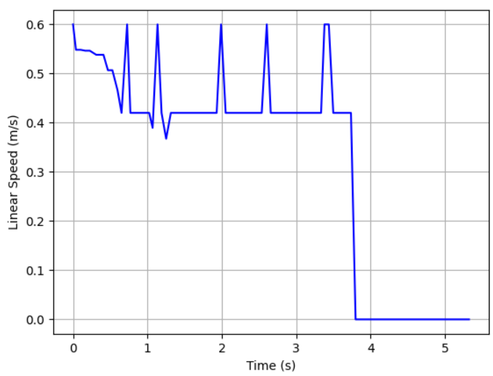
Figure 5.7 shows the trajectory, angles and speeds during the straight path test, consisting in a two meters path with very small angle corrections.



(a) Trajectory



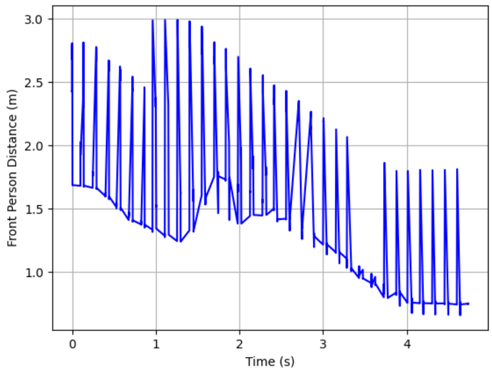
(b) Angle over time



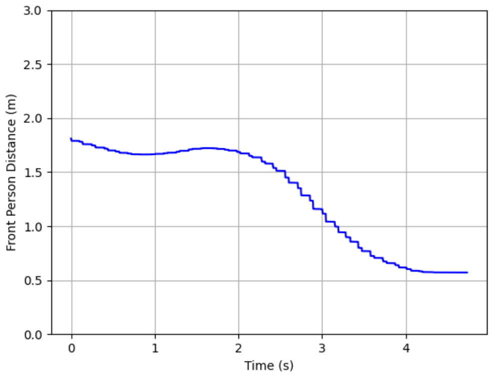
(c) Linear velocity over time

Figure 5.7: Straight trajectory test

The log data from the LiDAR provides insights into the robot behavior. Figure 5.8a shows the raw data, which is affected by high noise levels. This noise is primarily caused by the LiDAR position, being 20 cm above the floor, resulting in the detection of only the target legs. Since the LiDAR is based on single-line technology, its data do not include only the target distance but also the environment, even after selecting only the front area. To enhance the data quality, improbable distances greater than 3 meters are excluded, and a low-pass filter is applied to the signal. As presented in Figure 5.8b, a Butterworth filter is used to isolate the slower-changing trend in the data, which is more relevant for analysis. The filtered signal shows that the robot moves froward to reduce the distance from the target and it stops moving when the target is 0.6-0.7 meters away.



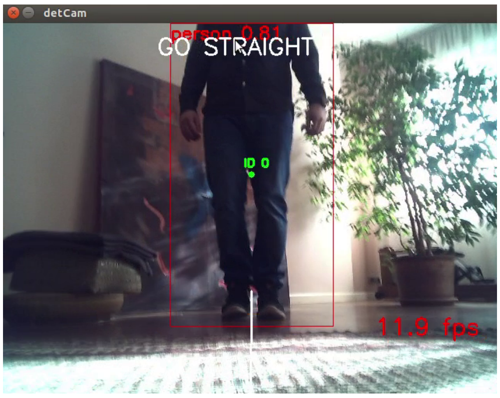
(a) Trajectory



(b) Filtered front distance

Figure 5.8: Front distance of the target

The ROS2 node outputs all information about movements and obstacle detection to the terminal. The ROSMASTER X3 follows the target based on commands from the vision script (Figure 5.9a). In Figure 5.9b the forward command is shown, while Figure 5.9c shows the translating motion to the left.



(a) Forward command (Vision)

```
[ros2_autonomous_follow-2] No obstacles. Following person.
[ros2_autonomous_follow-2] Received MQTT message: {"command": "forward"}
[ros2_autonomous_follow-2] Moving forward proportionally with speed :0.5778461419619046
[ros2_autonomous_follow-2] No obstacles. Following person.
[ros2_autonomous_follow-2] Received MQTT message: {"command": "forward"}
[ros2_autonomous_follow-2] Moving forward fast
[ros2_autonomous_follow-2] Received MQTT message: {"command": "forward"}
[ros2_autonomous_follow-2] Moving forward proportionally with speed :0.5575384360093336
[ros2_autonomous_follow-2] Lidar data: Front: 0, Left: 0, Right: 16
[ros2_autonomous_follow-2] Obstacle detected on the right. Robot moving forward left.
[ros2_autonomous_follow-2] Received MQTT message: {"command": "forward"}
[ros2_autonomous_follow-2] Obstacle detected on the right. Robot moving forward left.
[ros2_autonomous_follow-2] Received MQTT message: {"command": "forward"}
[ros2_autonomous_follow-2] Lidar data: Front: 0, Left: 0, Right: 108
```

(b) Forward motion (Terminal)

```
[ros2_autonomous_follow-2] Turning left with angular speed: 0.025, linear speed: 0.36
[ros2_autonomous_follow-2] Received MQTT message: {"command": "left", "angle": 19}
[ros2_autonomous_follow-2] Turning left with angular speed: 0.025, linear speed: 0.36
[ros2_autonomous_follow-2] No obstacles. Following person.
[ros2_autonomous_follow-2] Received MQTT message: {"command": "left", "angle": 14}
[ros2_autonomous_follow-2] Translating left with lateral speed: 0.4, linear speed: 0.42
[ros2_autonomous_follow-2] No obstacles. Following person.
[ros2_autonomous_follow-2] Received MQTT message: {"command": "left", "angle": 12}
[ros2_autonomous_follow-2] Translating left with lateral speed: 0.4, linear speed: 0.42
[ros2_autonomous_follow-2] Lidar data: Front: 0, Left: 21, Right: 0
[ros2_autonomous_follow-2] Received MQTT message: {"command": "left", "angle": 11}
[ros2_autonomous_follow-2] Obstacle detected on the left. Robot moving forward right.
```

(c) Translating motion (Terminal)

Figure 5.9: Robot movements command

When an obstacle is detected, the vision script continues sending commands related to the target (Figure 5.10a), while the ROS node handles the obstacle avoidance with the proper commands (Figure 5.10b).

The *backward* commands shown in Figure 5.11a is sent by the Vision script. However, if the ROS node determine, through the LiDAR, that in front of the robot there are no obstacles, the command is ignored, as shown in Figure 5.11. In such cases, the ROSMASTER X3 continues moving and following the target.

(a) Left command (Vision)

```
[ros2_autonomous_follow-2] Lidar data: Front: 0, Left: 21, Right: 0
[ros2_autonomous_follow-2] Received MQTT message: {"command": "left", "angle": 11}
[ros2_autonomous_follow-2] Obstacle detected on the left. Robot moving forward right.
[ros2_autonomous_follow-2] Received MQTT message: {"command": "left", "angle": 9}
[ros2_autonomous_follow-2] Lidar data: Front: 0, Left: 20, Right: 0
[ros2_autonomous_follow-2] Received MQTT message: {"command": "left", "angle": 7}
```

(b) Obstacle avoidance command

Figure 5.10: Robot moving forward

(a) Backward command send by Vision script

```
[ros2_autonomous_follow-2] Battery Voltage: 11.7V
[ros2_autonomous_follow-2] No obstacles. Following person.
[ros2_autonomous_follow-2] No obstacles. Following person.
[ros2_autonomous_follow-2] Received MQTT message: {"command": "backward"}
[ros2_autonomous_follow-2] Backward command ignored due to clear LiDAR data in front.
[ros2_autonomous_follow-2] Received MQTT message: {"command": "backward"}
[ros2_autonomous_follow-2] Backward command ignored due to clear LiDAR data in front.
[ros2_autonomous_follow-2] Received MQTT message: {"command": "backward"}
[ros2_autonomous_follow-2] Backward command ignored due to clear LiDAR data in front.
[ros2_autonomous_follow-2] Received MQTT message: {"command": "stop"}
[ros2_autonomous_follow-2] Stop command ignored due to clear LiDAR data in front.
```

(b) Backward command ignored

Figure 5.11: Backward command

If the target is lost, as shown in Figure 5.12, the robot waits for the maximum allowed number of disappearance frame before deactivating the *Follow Me* function. In the event of an error in the vision script, the watchdog mechanism in the ROS node deactivates the ROSMASTER X3 if it is not reset and no commands are received (Figure 5.13).

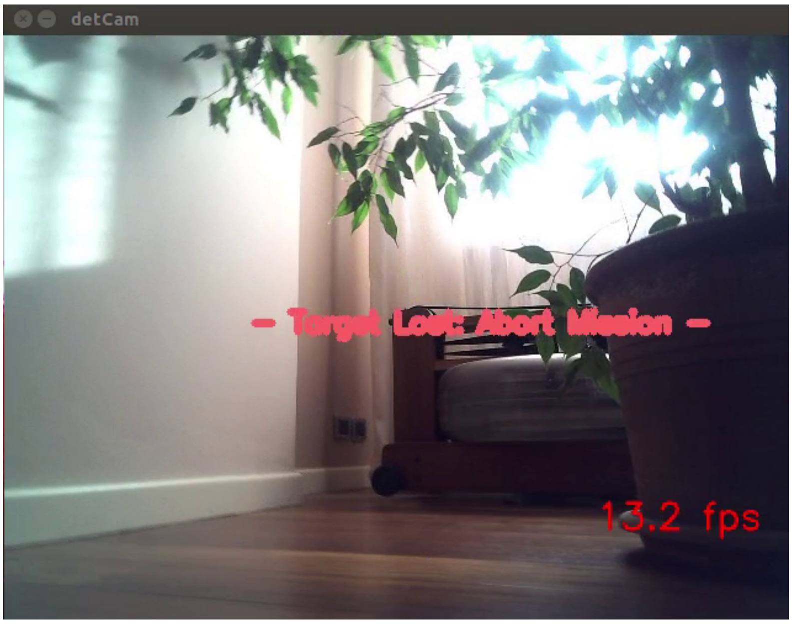


Figure 5.12: Lost Target alert

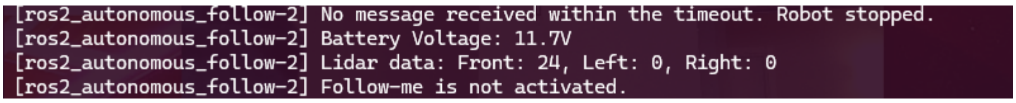


Figure 5.13: Watchdog timeout deactivation

5.4 Trajectory Reconstruction

This section analyzes a comprehensive test where the robot follows a longer path while avoiding obstacles. The map shown in Figure 5.14 was constructed separately, not dynamically during testing. The mapping algorithm used for reconstruct the environment is based on the **gmapping** package from ROS [25]. The robot was controlled manually via keyboard at a slow speed to ensure the creation of an accurate map of the testing area.

Figure 5.14: Rviz map visualization

Using the logged commands, it was possible to reconstruct the robot trajectory as shown in Figure 5.15. The trajectory was calculated point by point using the X and Y displacements. The calculation considers the three different motions of the robot: linear motion, translation and turning.

Equations 5.1 and 5.2 calculate the displacements using the real angle (calculated with Equation 5.3) to determine the robot orientation during turns. Due to variations in PID parameters and the real response of the motors being different from the requested values, a correction factor K was applied. This multiplicative factor take also into account the wheel slip on low friction surfaces. This factor was determined empirically based on the actual path performed.

$$x = x + linear\_speed[i] * cos(real\_angle) * dt \tag{5.1}$$

$$y = y + linear\_speed[i] * sin(real\_angle) * dt \tag{5.2}$$

$$real\_angle = real\_angle + angular\_speed[i] * K * dt \tag{5.3}$$

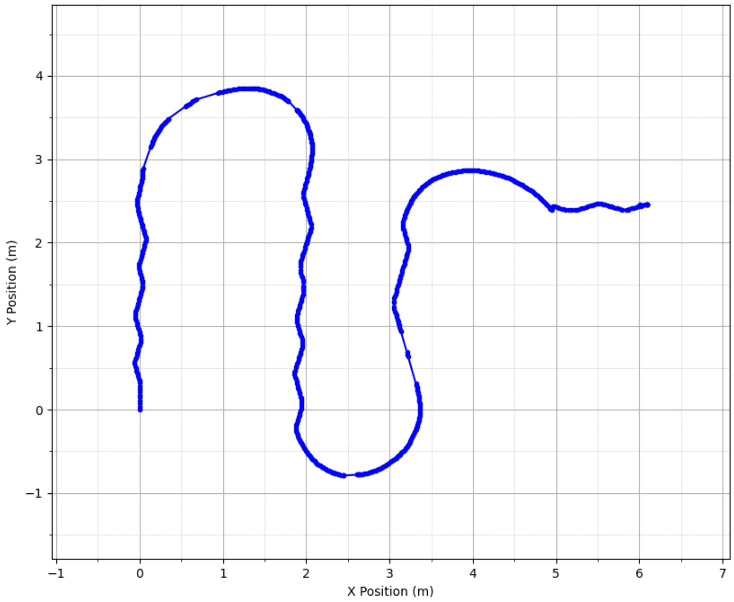
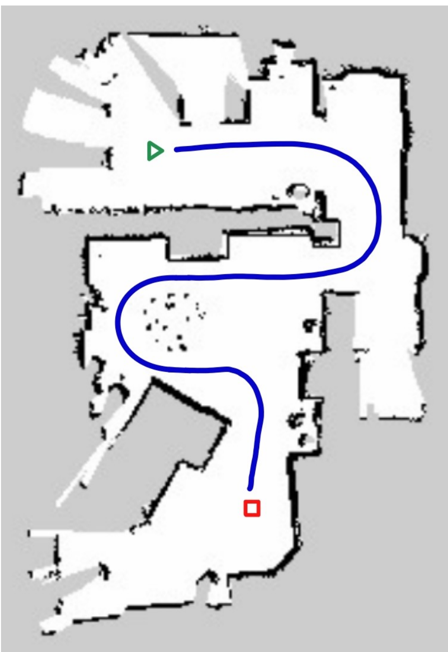
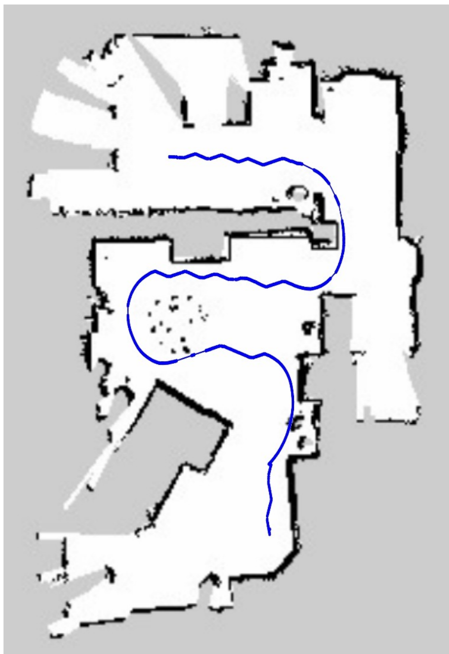


Figure 5.15: Reconstructed trajectory in XY plane



(a) Performed trajectory



(b) Reconstructed trajectory

Figure 5.16: Trajectories comparison

Figure 5.16a shows the performed path drawn on the map, while Figure 5.16b display the reconstructed path of the ROSMASTER X3. During the left turn, the robot navigates around a table (Figure 5.17), which is visible in the map and represented by clusters of dots corresponding to the legs of the table and chairs.

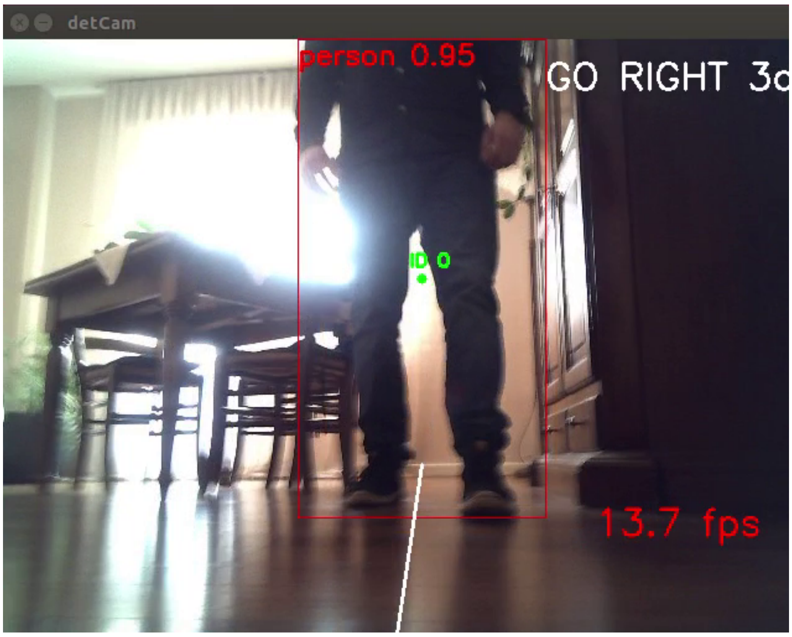


Figure 5.17: Table near the path

The analysis of the plots in Figure 5.18, confirms the trajectory shown in Figure 5.15. Negative angles correspond to right turns, while positive angles represent left turns. As seen in Figure 5.18, the robot performs a right turn, followed by a left turn, and another right turn. From the plots in Figure 5.18 and 5.19, it can be observed that the robot stops during turns, as both velocity and angle drop to zero around 10 and 15 seconds. At approximately 50 seconds, the target stops moving, prompting the robot to stop and reverse to maintain a safe distance from the target. The analysis of these plots and the reconstructed trajectory quantifies the robot behavior during the test, demonstrating its ability to follow a target and navigate in a complex environment.

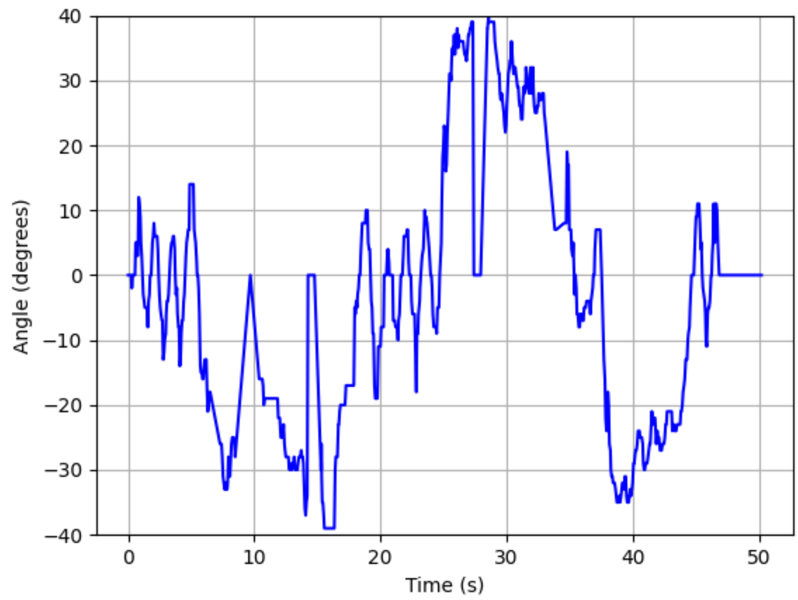


Figure 5.18: Angle over time

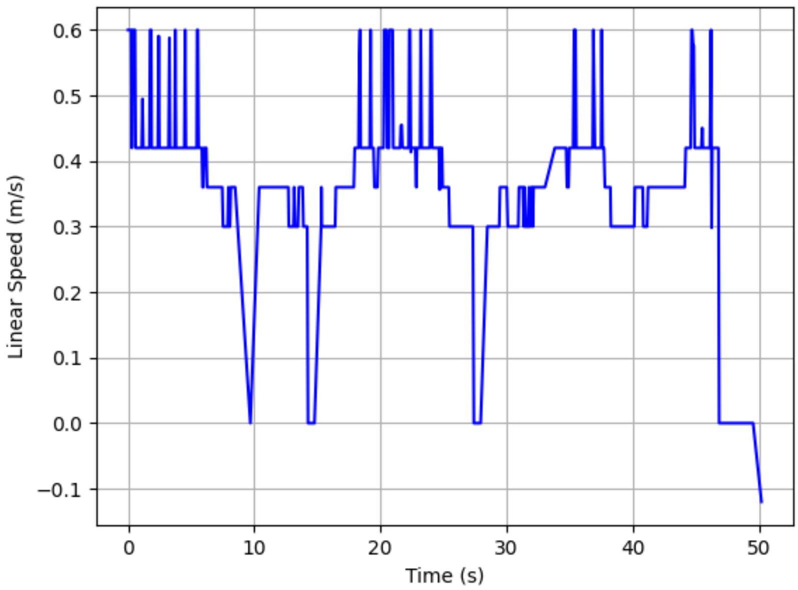


Figure 5.19: Linear Velocity over time

Figure 5.20 shows the uneven pavement on which the robot was tested in another evaluation. Despite the vibration, the ROSMASTER X3 followed the target without significant issues. The only limitation observed was the increased wheels friction, which affected the robot behavior during translating, reducing the translating speed.

Figure 5.20: Outdoor test

Chapter 6

Conclusions and Future works

This project provided a deeper understanding of the ROSMASTER X3 architecture and configuration, laying the groundwork for future developments. While the primary objectives were achieved, the work is not complete. This chapter discusses the key challenges and limitations encountered during the project, along with potential directions for future advancements.

6.1 Problems and Limits of the system

The development of the functionalities was constrained by the computational limitations of the Jetson NANO. These limitations primarily affected the frame rate which needed to exceed the 12 FPS to ensure better tracking and object detection performance. However, during testing, on multiple occasions, the Jetson NANO issued CPU throttling alerts caused by the high computational load.

Real-time mapping of the environment was excluded due to the speed constrain imposed by the mapping algorithm. Slow speeds were required to build an accurate map, which was incompatible with the robot ability to follow a person effectively.

Another significant issue was the robot performance under low light or direct light conditions. In such scenarios, the vision script model often failed to detect people correctly in certain frames, causing the bounding box to disappear. This failure particularly impacted the activation and deactivation phases of the *Follow Me* mode, as the counter would reset, requiring the target keep moving their arms for more than three times. However, during the actual following phase, temporary loss of the bounding box did not affect the robot behavior.

6.2 Future works

Enhancing the robot computational power could unlock new possibilities for development. Upgrading the main board from the Jetson NANO to a more powerful platform, such as the Jetson AGX ORIN, would address current limitations and enable significant improvements.

The two major areas of potential improvement are:

- **New activation method**
  A new activation method based on hand detection could resolve the light-related challenges. Implementing separate machine learning models for activation and tracking would allow for more accurate detection during activation, improving reliability.

- **Trajectory planning**
  Incorporating real-time mapping would improve the robot movement logic. By determining the optimal trajectory, the robot could improve obstacle avoidance and predict the target path in case of temporary disappearance over an extended number of frames.

6.2.1 ADAS development

Another potential development path involves a more automotive focused approach. This includes implementing a lane-keeping algorithm and integrating a machine learning model for road sign and traffic light detection. Such advancements would realign the project with its original goals through a different perspective. While this future work might not necessitate upgrading the Jetson NANO, it may require structural modifications to the robot, such as orienting the camera downward to better detect road lines.

6.3 Final Consideration

In conclusion, this project represented a significant integration challenge, requiring the assembly of the robot from individual parts and the adaptation of the existing code to function within a complete robotic system. The process required an in-depth exploration of the system architecture to ensure seamless communication between components. Adapting the vision script, originally developed without access to the assembled hardware needed extensive testing and refinement. Despite these challenges, the project successfully achieved its primary objective: developing an autonomous robot that tracks and follows a person while dynamically avoiding obstacles using camera and LiDAR sensor fusion. This achievement demonstrates the potential for further advancements and highlights the importance of a robust system architecture in autonomous robotics.

Bibliography

[1] (2015) Company profile. Shenzhen Yahboom Technology Co. Ltd. Last accessed: Oct. 18, 2024. [Online]. Available: http://www.yahboom.net/aboutus

[2] X. Zhao and T. Chidambareswaran, “Autonomous mobile robots in manufacturing operations,” in *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*, 2023, pp. 1–7.

[3] (2018) What is ros? Open Robotics. Last accessed: July 14, 2024. [Online]. Available: https://wiki.ros.org/ROS/Introduction

[4] (2018) Ros introduction. Open Robotics. Last accessed: July 20, 2024. [Online]. Available: https://wiki.ros.org/ROS/Introduction

[5] (2024) Ros introduction. Shenzhen Yahboom Technology Co. Ltd. Last accessed: Oct 24, 2024. [Online]. Available: https://github.com/YahboomTechnology/ROSMASTERX3/blob/main/03.X3-ROS1-Tutorials/08.ROS%20basic%20course/1.ROS%20introduction/1.ROS%20introduction.pdf

[6] (2022) Ros concepts. Open Robotics. Last accessed: Oct 24, 2024. [Online]. Available: https://wiki.ros.org/ROS/Concepts

[7] (2024) Ros 2 introduction. Shenzhen Yahboom Technology Co. Ltd. Last accessed: Oct 26, 2024. [Online]. Available: https://github.com/YahboomTechnology/ROSMASTERX3/blob/main/04.X3-ROS2-Tutorials/08.%20ROS2%20Basic%20Tutorial/1.Introduction%20to%20ROS2/1.%20Introduction%20to%20ROS2.pdf

[8] (2024) Ros 2 design. Open Robotics. Last accessed: Oct 22, 2024. [Online]. Available: https://design.ros2.org/

[9] (2024) Lidar basics. Shenzhen Yahboom Technology Co. Ltd. Last accessed: Oct 20, 2024. [Online]. Available: https://github.com/YahboomTechnology/ROSMASTERX3/blob/main/03.X3-ROS1-Tutorials/12.Lidar%20course/1.Lidar%20basics-Silan/1.Lidar%20basics.pdf

[10] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux Journal*, 2014.

[11] (2024) Docker engine overview. Docker Inc. Last accessed: Oct 28, 2024. [Online]. Available: https://docs.docker.com/engine

[12] S. Singh and N. Singh, "Containers & docker: Emerging roles & future of cloud technology," in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2016, pp. 804–807.

[13] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. Berg, "Ssd: Single shot multibox detector," in *"Computer Vision – ECCV 2016"*, vol. 9905, 10 2016, pp. 21–37.

[14] L. Forese, "Deep learning-based real-time multiple-object detection on a rover." Master's thesis, Politecnico di Torino, 2021. [Online]. Available: https://webthesis.biblio.polito.it/18104/

[15] (2024) About cuda. NVIDIA Corporation. Last accessed: Oct 28, 2024. [Online]. Available: https://developer.nvidia.com/about-cuda

[16] F. Oh. (2012) What is cuda? NVIDIA Corporation. Last accessed: Oct 28, 2024. [Online]. Available: https://blogs.nvidia.com/blog/what-is-cuda-2/

[17] Jetson nano. NVIDIA Corporation. Last accessed: Nov 2, 2024. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano

[18] (2024) Expansion board introduction. Shenzhen Yahboom Technology Co. Ltd. Last accessed: Nov 4, 2024. [Online]. Available: https://github.com/YahboomTechnology/ROSMASTERX3/blob/main/03.X3-ROS1-Tutorials/04.Hardware%20course/1.%20About%20expansion%20board/1.About%20expansion%20board.pdf

[19] J. Wang, C. Zhang, W. Zhu, Z. Zhang, Z. Xiong, and P. A. Chou, "3d scene reconstruction by multiple structured-light based commodity depth cameras," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2012.

[20] (2024) Kinematic analysis of meacanum wheel. Shenzhen Yahboom Technology Co. Ltd. Last accessed: Oct 24, 2024. [Online]. Available: https://github.com/YahboomTechnology/ROSMASTERX3/blob/main/04.X3-ROS2-Tutorials/04.Hardware%20course/14.%20Robot%20kinematics%20analysis%20theory/14.Kinematic%20Analysis%20of%20Mecanum%20Wheel.pdf

[21] (2024) Pid algorithm theory. Shenzhen Yahboom Technology Co. Ltd. Last accessed: Nov 4, 2024. [Online]. Available: https://github.com/YahboomTechnology/ROSMASTERX3/blob/main/04.X3-ROS2-Tutorials/09.Robot%20control%20course/1%E3%80%81PID%20algorithm%20theory/1%E3%80%81PID%20algorithm%20theory/1%E3%80%81PID%20algorithm%20theory.pdf

[22] Eagle Comm.ltd, "MCUISP - MCU in system programmer," 2017, Version 0.993 [Microcontroller programming tool]. [Online]. Available: http://www.mcuisp.com/English%20mcuisp%20web/ruanjianxiazai-english.htm

[23] (2024) Control robot movement. Shenzhen Yahboom Technology Co. Ltd. Last accessed: Nov 08, 2024. [Online]. Available: https://github.com/YahboomTechnology/ROSMASTERX3/blob/main/04.X3-ROS2-Tutorials/05.ROSMASTER%20Basic%20course/8.%20Control%

20robot%20movement/8.%20Control%20robot%20movement.pdf
[24] A. Calio', "Deep learning-based real-time detection and object tracking on an autonomous rover with gpu based embedded device," Master's thesis, Politecnico di Torino, 2021. [Online]. Available: https://webthesis.biblio.polito.it/18081/
[25] (2024) Gmapping mapping algorithm. Shenzhen Yahboom Technology Co. Ltd. [Online]. Available: https://github.com/YahboomTechnology/ROSMASTERX3/blob/main/04.X3-ROS2-Tutorials/10.Lidar%20course/6%E3%80%81gmapping%20mapping%20algorithm/6%E3%80%81gmapping%20mapping%20algorithm.pdf

Appendix A

Follow_Me() function (object_detection_module.py)

```python
def Follow_Me():

    timeStamp = time.time()
    fpsFilt = 0
    net = jetson_inference.detectNet('ssd-mobilenet-v2', threshold
        =.65)
    dispW = 640
    dispH = 480
    danger_threshold = 465
    close_threshold = 450
    stop_duration_threshold = 0.7
    last_stop_time = None
    flip = 2
    font = cv2.FONT_HERSHEY_SIMPLEX
    cam=cv2.VideoCapture(0)

    ct = Centroids()
    sf = SafeRover()
    fm = FollowMe()
    target = -1
    error_message = -1
    counter_message = 0
    bounding_boxes = {}

    while True:
        _,img = cam.read()
        height=img.shape[0]
        width=img.shape[1]
        frame=cv2.cvtColor(img,cv2.COLOR_BGR2RGBA).astype(np.float32)
        frame=jetson_utils.cudaFromNumpy(frame)

        detections=net.Detect(frame, width, height)
        matching_detections= []
        rects = []
        all_objects= []

        for detect in detections:

            ID=detect.ClassID
            confidence=truncate(float(detect.Confidence),2)
            top=int(detect.Top)
            left=int(detect.Left)
            bottom=int(detect.Bottom)
            right=int(detect.Right)
            item=net.GetClassDesc(ID)
            box=(left,top,right,bottom)

            if item == 'person':
                matching_detections.append(detect)
                box_person=(left,top,right,bottom)
```

```python
                rects.append(box_person)
                cv2.putText(img,item+" "+str(confidence),(left,top+20),font,.75,(0,0,255),2)
                cv2.rectangle(img,(left,top),(right,bottom),(0,0,255),1)

            all_objects.append(box)
        objects = ct.centroids_recalculator(rects)

        for (objectID, centroid), bbox in zip(objects.items(), rects):
            bounding_boxes[objectID] = bbox  # Map CentroidID to its bounding box

        if(fm.count_sleep == fm.max_sleep):

            # Waiting for a target
            if(target == -1):
                target = fm.follow_update(objects,rects)

            # Target selected
            else:
                # Control for an eventual lost of target
                if(sf.check_target(objects,target) == 3):
                    send_stop()
                    target = -1
                    print("Target Lost:"+str(target))
                    error_message = 1

                else:
                    # Control if an object is too close
                    ret_command=sf.check_collision(all_objects)
                    if ret_command == 1:
                        cv2.putText(img,"- Stop: probability of collision -",(int(120),int(480/2)),font,.75,(100,100,255),4)

                    # Control for an eventual switching
                    ret_command = sf.check_switch(objects,target,rects)
                    if(ret_command == 2):
                        cv2.putText(img,"- Probability of an object switch in the next frames -",(int(50),int(480/2)),font,.75,(255,255,255),4)
                    if(ret_command == 4):
                        print("Abort Mission")
                        send_stop()
                        target=-1
                        error_message = 2

                    target = fm.unfollow_update(objects,rects,target)
```

```
        else:
            fm.count_sleep += 1

        for (objectID, centroid) in objects.items():
            # draw both the ID of the object and the centroid of the object on the output frame
            text = "ID {}".format(objectID)
            cv2.putText(img, text, (centroid[0] - 10, centroid[1] - 10),
            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
            cv2.circle(img, (centroid[0], centroid[1]), 4, (0, 255, 0), -1)

            if objectID == target:
                    angle = (np.arctan(abs(centroid[0] - (640 / 2)) / (480 - centroid[1] + 0.01)) * 180 / math.pi)
                    if target in bounding_boxes:
                        _, _, _, bottom = bounding_boxes[target]
                    # If person is too close (stop area)
                    if bottom >= close_threshold and bottom < danger_threshold:
                        send_command("stop")
                        cv2.putText(img, "STOP", (200, 260), font, 1, (0, 0, 255), 2)
                        last_stop_time = time.time()  # Record the time when we stopped

                    # If person is dangerously close
                    elif bottom >= danger_threshold:
                        send_command("backward")
                        cv2.putText(img, "MOVE BACKWARD!", (220, 260), font, 1, (0, 0, 255), 2)
                        last_stop_time = time.time()  # Reset stop timer when moving backward

                    # If person is far enough to resume movement
                    else:
                        # Only resume if stop time has passed since the last stop message
                        if last_stop_time and time.time() - last_stop_time >= stop_duration_threshold or last_stop_time==None:
                            if centroid[0] < 640 / 2 - 10:
                                send_command("left", int(angle))
                                cv2.putText(img, "GO LEFT " + str(int(angle)) + "deg", (0, 40), font, 1, (255, 255, 255), 2)
                            elif centroid[0] > 640 / 2 + 10:
                                send_command("right", int(angle))
```

```
                                cv2.putText(img, "GO RIGHT " + str(int(angle)) + "deg", (440, 40), font, 1, (255, 255, 255), 2)
                            else:
                                send_command("forward")
                                cv2.putText(img, "GO STRAIGHT", (200, 40), font, 1, (255, 255, 255), 2)

                            # Reset last_stop_time since we're moving again
                            last_stop_time = None

                        cv2.line(img, (int(640 / 2), 480), (centroid[0], centroid[1] + 150), (255, 255, 255), 2)
        #display an eventuaL error message for N frames
        if(error_message != -1):
            if (error_message == 1):
                    cv2.putText(img,"- Target Lost: Abort Mission -",(int(350),int(480/2)),font,.75,(100,100,255),5)
                    counter_message += 1
                    if(counter_message == 24):
                        error_message = -1
                        counter_message = 0
            if (error_message == 2):
                    cv2.putText(img,"- Switching Avoidance: Abort Mission -",(int(350),int(480/2)),font,.75,(100,100,255),5)
                    counter_message += 1
                    if(counter_message == 24):
                        error_message = -1
                        counter_message = 0

        dt=time.time()-timeStamp
        timeStamp=time.time()
        fps=1/dt
        fpsFilt=.9*fpsFilt + .1*fps

        cv2.putText(img, str(round(fpsFilt,1))+' fps',(480,400),font,1,(0,0,255),2)
        cv2.imshow('detCam', img)
        if cv2.waitKey(1) == ord('q'):
            break
    cam.release()
    # Deactivate the robot when quitting
    send_deactivate()
    cv2.destroyAllWindows()
    client.loop_stop()
    client.disconnect()
```

Appendix B

ros2_autonomous_follow.py

```
#!/usr/bin/env python3

import math
import numpy as np
import paho.mqtt.client as mqtt
import json
import csv
from datetime import datetime
from time import sleep, time


# ROS2 libraries
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan

from Rosmaster_Lib import Rosmaster

# Constants
RAD2DEG = 180 / math.pi

class AutonomousFollower(Node):
    def __init__(self, name):
        super().__init__(name)

        # Initialize STM32 communication
        self.car = Rosmaster()
        self.car.set_car_type(1)
        self.car.create_receive_threading()

        # Initialize MQTT communication
        self.mqtt_client = mqtt.Client()
        self.mqtt_client.on_connect = self.on_connect
        self.mqtt_client.on_message = self.on_message
        self.mqtt_client.connect("localhost", 1883, 60)
        self.mqtt_client.loop_start()

        # Parameters
        self.linear_speed = 0.6
        self.angular_speed = 0.5
        self.lateral_speed = 0.4
        self.distance_front= 2.0  # actual distance front to the nearest object/person
        self.response_dist_lat = 0.3  # Obstacle detection distance
        self.response_dist_front = 0.5  # Obstacle detection distance
        self.response_clearance_front = 1.8 # Clear from obstacle distance for speed control
        self.laser_angle = 120.0  # Angle for laser range scanning

        # Flags and states
```

```python
        self.person_detected = False
        self.obstacle_detected = False
        self.obstacle_clearance_front = True
        self.front_stop_area = False
        self.ready = False
        self.follow_me_active = False
        self.error = False
        self.emergency_stopped = False

        self.obstacle_avoid_front = False
        self.obstacle_avoid_left = False
        self.obstacle_avoid_right = False
        self.obstacle_avoid_all = False

        # Initialize LED to blue
        self.set_led_color(0, 0, 255)

        # Lidar data variables
        self.right_warning = 0
        self.left_warning = 0
        self.front_warning = 0

        # Watchdog-related attributes
        self.watchdog_timeout = 3.0  # Timeout in seconds
        self.watchdog_timer = None  # Placeholder for the watchdog timer

        # Start the watchdog timer when the node is initialized
        self.reset_watchdog()

        # Subscriber
        self.sub_laser = self.create_subscription(LaserScan, '/scan', self.lidar_callback, 10)
        #Publisher
        self.publish_battery_info()

        # Timer for battery voltage publishing
        self.battery_timer = self.create_timer(5.0, self.publish_battery_info)

        # Timer for regular data publishing
        self.timer = self.create_timer(0.1, self.follow_person_and_avoid_obstacles)

        print(f"Autonomous Follow me ROS2 node v2")

        # File logging setup
        self.command_log_file = open('command_log.csv', 'w', newline='')
        self.lidar_log_file = open('lidar_log.csv', 'w', newline='')
```

```python

        # CSV writers
        self.command_writer = csv.writer(self.command_log_file)
        self.command_writer.writerow(['Time', 'Direction', 'Angle', 'Linear Speed','Lateral Speed', 'Angular Speed'])
        self.lidar_writer = csv.writer(self.lidar_log_file)
        self.lidar_writer.writerow(['Time', 'Distance to Nearest Object'])



    def set_led_color(self, r, g, b):
        """Sets the RGB LED color on the robot."""
        self.car.set_colorful_lamps(0xFF, r, g, b)  # 0xFF means all LEDs

    def on_connect(self, client, userdata, flags, rc):
        """MQTT connection callback."""
        print(f"Connected to MQTT with result code {rc}")
        self.ready = True
        self.set_led_color(0, 0, 255)  # Blue LED when ready
        client.subscribe("robot/control")
        print("Subscribed to MQTT topic 'robot/control'")

    def publish_battery_info(self):
        """Fetches and publishes the battery voltage."""
        try:
            battery_voltage = self.car.get_battery_voltage()
            print(f"Battery Voltage: {battery_voltage:.1f}V")
        except Exception as e:
            print(f"Cannot get battery info: {e}")

    def reset_watchdog(self):
        """Resets the watchdog timer to stop the robot if no message is received."""
        if self.watchdog_timer is not None:
            self.watchdog_timer.cancel()  # Cancel any existing watchdog timer

        # Create a new watchdog timer that stops the robot after timeout duration
        self.watchdog_timer = self.create_timer(self.watchdog_timeout, self.stop_robot_due_to_inactivity)
        #print("Watchdog timer reset.")

    def log_command(self, direction, angle, linear_speed, lateral_speed, angular_speed):
        log_time = datetime.now().strftime("%M:%S.%f")[:-3]
        self.command_writer.writerow([log_time, direction, angle, linear_speed, lateral_speed, angular_speed])
```

```python

    def log_lidar_data(self, distance):
        log_time = datetime.now().strftime("%M:%S.%f")[:-3]
        self.lidar_writer.writerow([log_time, distance])

    def stop_robot_due_to_inactivity(self):
        """Stops the robot due to inactivity (no messages received)."""
        if self.follow_me_active:
            self.set_led_color(255, 0, 0)  # Red LED for inactivity
            print("No message received within the timeout. Robot stopped.")
            self.follow_me_active = False
        self.car.set_car_motion(0, 0, 0)  # Stop the robot
        log_time = datetime.now().strftime("%M:%S.%f")[:-3]
        self.command_writer.writerow([log_time, "--- Deactivation inactivity ---", '-', '-', '-', '-'])
        self.lidar_writer.writerow([log_time, "--- Deactivation inactivity ---", '-', '-', '-', '-'])



    def on_message(self, client, userdata, msg):
        """MQTT message callback for follow-me activation."""
        print(f"Received MQTT message: {msg.payload.decode('utf-8')}")
        try:
            # Decode the payload into a string and then parse the JSON data
            message_str = msg.payload.decode('utf-8')
            message = json.loads(message_str)  # Convert the JSON string to a dictionary
            self.reset_watchdog()
            # Extract the command and angle from the message
            command = message.get("command", "")  # Get the command or default to an empty string
            angle = message.get("angle", 0)  # Get the angle or default to 0

            if command == "activate":
                self.follow_me_active = True
                self.emergency_stopped = False
                self.set_led_color(0, 255, 0)  # Green LED when follow-me is activated
                print("Follow-me activated")
                log_time = datetime.now().strftime("%M:%S.%f")[:-3]
                self.command_writer.writerow([log_time, "--- Activation Start ---", '-', '-', '-', '-'])
                self.lidar_writer.writerow([log_time, "--- Activation Start ---", '-', '-', '-', '-'])

```

```python

        elif command == "deactivate":
            self.follow_me_active = False
            self.car.set_car_motion(0, 0, 0) #Stop the robot
            self.set_led_color(0, 0, 255)  # Blue LED when deactivated
            print("Follow-me deactivated")
            log_time = datetime.now().strftime("%M:%S.%f")[:-3]
            self.command_writer.writerow([log_time, "--Deactivation Standard --", '-', '-', '-', '-'])
            self.lidar_writer.writerow([log_time, "--Deactivation Standard--", '-', '-', '-', '-'])


        elif command == "emergency_stop":
            self.handle_emergency_stop()

        elif command == "forward" and self.follow_me_active:
            if not self.obstacle_detected and not self.emergency_stopped:
                log_time = datetime.now().strftime("%M:%S.%f")[:-3]
                if self.obstacle_clearance_front:
                    """Full forward speed if no obstacle in front"""
                    self.car.set_car_motion(self.linear_speed, 0, 0)
                    print("Moving forward fast")
                    self.command_writer.writerow([log_time, "forward", 0, self.linear_speed, 0, 0])

                else:
                    """Proportional forward speed wrt to front obstacle/person"""
                    percentage = (self.distance_front - self.response_dist_front) / (self.response_clearance_front - self.response_dist_front)
                    self.car.set_car_motion(self.linear_speed*percentage, 0, 0)
                    print(f"Moving forward proportionally with speed :{self.linear_speed*percentage}")
                    self.command_writer.writerow([log_time, "forward", 0, self.linear_speed*percentage, 0, 0])



        elif command == "left" and self.follow_me_active:
            log_time = datetime.now().strftime("%M:%S.%f")[:-3]
```

```
                if (angle<=15):
                    """Translating when angle below 15 degrees"""
                    adjusted_linear_speed = self.linear_speed * 0.7
                    if not self.obstacle_detected and not self.emergency_stopped:
                        self.car.set_car_motion(adjusted_linear_speed,0, self.lateral_speed)
                        print(f"Translating left with lateral speed: {self.lateral_speed}, linear speed: {adjusted_linear_speed}")
                        self.command_writer.writerow([log_time, "left", angle, adjusted_linear_speed, self.lateral_speed,0])
                elif(angle>15 and angle<=28):
                    """Turning slow when angle below 28 degrees"""
                    adjusted_angular_speed = self.angular_speed*0.05
                    adjusted_linear_speed = self.linear_speed * 0.6
                    if not self.obstacle_detected and not self.emergency_stopped:
                        self.car.set_pid_param(0.1,3,3, 0)
                        self.car.set_car_motion(adjusted_linear_speed, adjusted_angular_speed, 0)
                        print(f"Turning left with angular speed: {adjusted_angular_speed}, linear speed: {adjusted_linear_speed}")
                        self.command_writer.writerow([log_time, "left", angle, adjusted_linear_speed, adjusted_angular_speed])
                else:
                    """Turning faster when angle over 28 degrees"""
                    adjusted_angular_speed = self.angular_speed*0.08
                    adjusted_linear_speed = self.linear_speed * 0.5
                    if not self.obstacle_detected and not self.emergency_stopped:
                        self.car.set_pid_param(0.1,3,3, 0) # PID parameter adjustment
                        self.car.set_car_motion(adjusted_linear_speed, adjusted_angular_speed, 0)
                        print(f"Turning fast left with angular speed: {adjusted_angular_speed}, linear speed: {adjusted_linear_speed}")
                        self.command_writer.writerow([log_time, "left", angle, adjusted_linear_speed, adjusted_angular_speed])

            elif command == "right" and self.follow_me_active:
                log_time = datetime.now().strftime("%M:%S.%f")[:-3]
                if (angle<=15):
                    """Translating when angle below 15 degrees"""
                    adjusted_linear_speed = self.linear_speed * 0.7
```

```python
                    if not self.obstacle_detected and not self.emergency_stopped:
                        self.car.set_car_motion(adjusted_linear_speed,0, -self.lateral_speed)
                        print(f"Translating right with lateral speed: {self.lateral_speed}, linear speed: {adjusted_linear_speed}")
                        self.command_writer.writerow([log_time, "right", -angle, adjusted_linear_speed, -self.lateral_speed,0])
                elif(angle>15 and angle<=28):
                    """Turning slow when angle below 28 degrees"""
                    adjusted_angular_speed = self.angular_speed*0.05
                    adjusted_linear_speed = self.linear_speed * 0.6
                    if not self.obstacle_detected and not self.emergency_stopped:
                        self.car.set_pid_param(0.1,3,3, 0)
                        self.car.set_car_motion(adjusted_linear_speed, -adjusted_angular_speed, 0)
                        print(f"Turning right with angular speed: {adjusted_angular_speed}, linear speed: {adjusted_linear_speed}")
                        self.command_writer.writerow([log_time, "right", -angle, adjusted_linear_speed, -adjusted_angular_speed])

                else:
                    """Turning faster when angle over 28 degrees"""
                    adjusted_angular_speed = self.angular_speed*0.08
                    adjusted_linear_speed = self.linear_speed * 0.5
                    if not self.obstacle_detected and not self.emergency_stopped:
                        self.car.set_pid_param(0.1,3,3, 0) # PID parameter adjustment
                        self.car.set_car_motion(adjusted_linear_speed, -adjusted_angular_speed, 0)
                        print(f"Turning fast right with angular speed: {adjusted_angular_speed}, linear speed: {adjusted_linear_speed}")
                        self.command_writer.writerow([log_time, "right", -angle, adjusted_linear_speed, -adjusted_angular_speed])

            elif command == "stop":
                log_time = datetime.now().strftime("%M:%S.%f")[:-3]
                if self.follow_me_active:
                    if self.front_stop_area:   # Avoid stopping for errors in the bounding box
                        self.car.set_car_motion(0, 0, 0)  # Stop motors
```

```
                        print("Robot stopped")
                        self.set_led_color(255, 0, 0)  # Red LED for stop
                        self.command_writer.writerow([log_time, "stop", 0, 0, 0, 0])
                        sleep(0.5)
                    else:
                        # Ignore the stop command if the bounding box is faulty but there's no obstacle in front
                        print("Stop command ignored due to clear LiDAR data in front.")


            elif command == "backward":
                log_time = datetime.now().strftime("%M:%S.%f")[:-3]
                if self.follow_me_active:
                    if not self.obstacle_avoid_all and self.front_stop_area:
                        adjusted_linear_speed = self.linear_speed * 0.2
                        self.car.set_car_motion(-adjusted_linear_speed, 0, 0)
                        print("Robot moving backwards")
                        self.set_led_color(255, 0, 0)  # Red LED for moving backwards
                        self.command_writer.writerow([log_time, "backwards", 0, -adjusted_linear_speed, 0, 0])
                        sleep(0.6)
                    else:
                        # Ignore the stop command if the bounding box is faulty but there's no obstacle in front
                        print("Backward command ignored due to clear LiDAR data in front.")

        except json.JSONDecodeError as e:
            print(f"Failed to decode MQTT message: {e}")
        except KeyError as e:
            print(f"Key error: {e}")
        except Exception as e:
            print(f"Unexpected error: {e}")

    def handle_emergency_stop(self):
        """Handles an emergency stop by stopping the robot """
        self.follow_me_active = False
        self.emergency_stopped = True
        self.car.set_car_motion(0, 0, 0)
        self.set_led_color(255, 0, 0)  # Red LED for emergency stop
        print("Emergency stop received. Robot stopped and follow function deactivated.")
```

```python

    def lidar_callback(self, scan_data):
        """Processes LiDAR data for obstacle detection."""
        ranges = np.array(scan_data.ranges)
        self.right_warning = 0
        self.left_warning = 0
        self.front_warning = 0


        for i in range(len(ranges)):
                log_time = datetime.now().strftime("%M:%S.%f")[:-3]
                angle = (scan_data.angle_min + scan_data.angle_increment * i) * RAD2DEG
                if 160 > angle > 180 - self.laser_angle:
                    if ranges[i] < self.response_dist_lat:
                        self.right_warning += 1

                if -160 < angle < self.laser_angle - 180:
                    if ranges[i] < self.response_dist_lat:
                        self.left_warning += 1

                if abs(angle) > 160:
                    self.distance_front = ranges[i]
                    self.lidar_writer([log_time, self.distance_front])
                    if ranges[i] <= self.response_dist_front:
                        self.front_warning += 1
                    if ranges[i]  < self.response_clearance_front:
                        self.obstacle_clearance_front = False
                    if ranges[i]  >= self.response_clearance_front:
                        self.obstacle_clearance_front = True
                    if ranges[i]  < self.response_clearance_front -0.5:
                        self.front_stop_area = True
                    if ranges[i]  >= self.response_clearance_front -0.5:
                        self.front_stop_area = False



        #obstacle left
        if self.left_warning > 10:
            self.obstacle_detected = True
            self.obstacle_avoid_left = True
            print(f"Lidar data: Front: {self.front_warning}, Left: {self.left_warning}, Right: {self.right_warning}")

        #obstacle right
        elif self.right_warning > 10:
            self.obstacle_detected = True
            self.obstacle_avoid_right =  True
```

```
            print(f"Lidar data: Front: {self.front_warning}, Left: {self.left_warning}, Right: {self.right_warning}")

        #obstacle front
        elif self.front_warning > 10:
            self.obstacle_detected = True
            self.obstacle_avoid_front = True
            print(f"Lidar data: Front: {self.front_warning}, Left: {self.left_warning}, Right: {self.right_warning}")


        #obstacle in all directions
        elif self.front_warning > 10 and self.left_warning > 10 and self.right_warning > 10:
            self.obstacle_detected = True
            self.obstacle_avoid_all = True
            print(f"Lidar data: Front: {self.front_warning}, Left: {self.left_warning}, Right: {self.right_warning}")


        else:

            if self.follow_me_active:
                self.set_led_color(0, 255, 0)  # green LED for obstacle cleared and follow me active
            else:
                self.set_led_color(0, 0, 255)  # blu LED for obstacle cleared and follow me not active

            self.obstacle_detected = False
            self.obstacle_avoid_front = False
            self.obstacle_avoid_left = False
            self.obstacle_avoid_right = False
            self.obstacle_avoid_all = False

    def follow_person_and_avoid_obstacles(self):
        """Logic for following the person and avoiding obstacles."""

        # Wait if the system is not ready
        if not self.ready:
            print("Waiting for MQTT connection...")
            return

        # If follow-me is active and there are no obstacles, follow the person
        if self.follow_me_active and not self.obstacle_detected:
            self.set_led_color(0, 255, 0)  # Green LED when follow-me is active and no obstacles
            print("No obstacles. Following person.")
            return
```

```python

        # If follow-me is not active but no obstacles detected, ensure robot stops
        if not self.follow_me_active and not self.obstacle_detected:
            self.set_led_color(0, 0, 255)  # Blue LED for follow-me not active and no obstacles
            self.car.set_car_motion(0, 0, 0)  # Ensure the robot fully stops if inactive
            return

        # Stop if follow-me mode is not activated
        if not self.follow_me_active:
            print("Follow-me is not activated.")
            self.car.set_car_motion(0, 0, 0)  # Ensure the robot fully stops if not active
            self.set_led_color(0, 0, 255)  # Blue LED for follow-me not active
            return

        if self.follow_me_active and self.obstacle_detected:
            self.set_led_color(255, 0, 0)  # Red LED for obstacle detected
            if self.obstacle_avoid_all:
                """Obstacle in all directions"""
                print("Obstacle detected. Robot is stopped.")
                self.car.set_car_motion(0, 0, 0)  # Ensure the robot fully stops on obstacle detection

            elif self.obstacle_avoid_right and self.obstacle_avoid_left:
                """Obstacles on both sides"""
                self.car.set_car_motion(self.linear_speed * 0.5, 0, 0)  # Obstacle both left and right slowly move forward
                print("Obstacles detected on both sides. Moving slowly forward.")

            elif self.obstacle_avoid_left and self.obstacle_avoid_front:
                """Obstacle on the left and front"""
                self.car.set_car_motion(0, 0, -self.angular_speed * 1.2)  # If obstacle also in front, rotate to the right
                print("Obstacle detected on the left. Robot moving right.")

            elif self.obstacle_avoid_right and self.obstacle_avoid_front:
                """Obstacle on the right and front"""
```

```
                    self.car.set_car_motion(0, 0, self.angular_speed * 1.2) # If obstacle also in front, rotate to the left
                    print("Obstacle detected on the right. Robot moving left.")

                elif self.obstacle_avoid_left:
                    """Obstacle on the left"""
                    self.car.set_car_motion(self.linear_speed * 0.5, 0, -self.angular_speed * 1.2)  # Move forward-right
                    print("Obstacle detected on the left. Robot moving forward right.")

                elif self.obstacle_avoid_right:
                    """Obstacle on the right """
                    self.car.set_car_motion(self.linear_speed * 0.5, 0, self.angular_speed * 1.2)   # Move forward-left
                    print("Obstacle detected on the right. Robot moving forward left.")

                elif self.obstacle_avoid_front:
                    """Obstacle in front"""
                    self.car.set_car_motion(-self.linear_speed * 0.3, 0, 0) # Move backwards
                    print("Obstacle detected in front. Robot moving backwards.")

                return


def main(args=None):
    rclpy.init(args=args)
    node = AutonomousFollower("autonomous_follower")
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.car.set_car_motion(0, 0, 0) # Stop the robot motors
        node.set_led_color(0, 0, 0)  # Turn off LEDs on shutdown
        node.command_log_file.close()
        node.lidar_log_file.close()
        node.destroy_node()
    finally:
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```