Politecnico di Torino

Master's Degree in Computer Engineering

Master's Degree Thesis

Sensor Fusion for Autonomous Driving

Supervisor
Prof. Stefano Malan

Candidate
Maria Francesca Merangolo

Internship Tutor
Ing. Angelo Borneo

April 2025

Abstract

In a context where robotics and autonomous vehicles are becoming increasingly central in industry and research, this thesis presents the development of an autonomous driving system for Yahboom ROSMASTER X3 educational robot.

The main objective is the implementation of a rover capable of autonomously following a line and recognising road signs by integrating sensor fusion techniques between Light Detection And Ranging (LiDAR) and a depth camera. This combination allows the robot to obtain a more accurate perception of its surroundings, enhancing navigation adaptability to complex scenarios.

The system is based on Robot Operating System (ROS)2 and exploits a hardware platform consisting of an NVIDIA Jetson Nano for image processing and autonomous navigation and Mecanum wheels that provide omnidirectional mobility, improving the robot manoeuvring capabilities. Computer vision plays a key role in traffic sign recognition, implemented via a MobileNetV2 Single Shot multiBox Detector (SSD) deep learning model, suitably trained on a customised dataset. The software pipeline includes the implementation of navigation algorithms, Message Queuing Telemetry Transport (MQTT) communication for remote management, and integration with ROS2 to ensure modularity and efficiency.

After the assembly and configuration phase of the rover, the system was subjected to several tests to evaluate performance in real navigation scenarios for signal recognition, stability in line tracking, and responsiveness to environmental variations.

The results obtained demonstrate that the integration of sensor fusion and machine learning within a ROS2 architecture allows a significant improvement in the robot autonomous capabilities. This work represents a contribution to research on robotic navigation systems, highlighting the potential of advanced perception technologies for real-world applications in dynamic environments.

Acknowledgements

I would like to express my sincere gratitude to MCA Company, especially Angelo Borneo, for providing me with this opportunity for professional growth.

Special thanks are due to my thesis advisor, Professor Stefano Malan, for his knowledge, expertise and unwavering availability to offer valuable guidance and feedback.

Table of Contents

List of Figures

List of Tables

Listings

Acronyms

ROS	Robot Operating System
LiDAR	Light Detection And Ranging
SSD	Single Shot multiBox Detector
VS Code	Visual Studio Code
SSH	Secure Shell Protocol
MQTT	Message Queuing Telemetry Transport
DDS	Data Distribution Service
ToF	Time-of-Flight
CUDA	Compute Unified Device Architecture
GPU	Graphic Processing Unit
CPU	Central Processing Unit
AI	Artificial Intelligence
USB	Universal Serial Bus
FPS	Frames Per Second
mAP	Mean Average Precision
HSV	Hue Saturation Value

Chapter 1

Introduction

This thesis is developed as part of a collaborative effort with Fabio Marchisio who worked on the Master thesis Autonomous Robot Driving using Sensor Fusion [1].

1.1 Objectives

The main goal of this thesis is to develop an autonomous driving system for a robot, which allows it to autonomously follow a line while recognising and responding to traffic signs. The system integrates sensor fusion, combining data from LiDAR and a camera, along with computer vision techniques. This integration enhances the perception of the robot in its environment, allowing it to navigate reliably and perform complex manoeuvres facilitated by its omnidirectional Mecanum wheels. The fusion of LiDAR and camera data compensates for the limitations inherent in individual sensors, providing a more comprehensive understanding of the surroundings [2].

In addition to sensor fusion, the system incorporates computer vision algorithms to detect and interpret traffic signs, improving the decision-making process of the robot and enabling it to adjust its trajectory accordingly.

The robot used in this thesis is the ROSMASTER X3, an educational robot developed by Shenzhen Yahboom Technology Co., specifically designed for exploring the ROS environment and advancing robotics research [3]. The ROSMASTER X3 is equipped with high-performance hardware modules, including LiDAR, a depth camera, and a voice interaction module. Its aluminium alloy chassis and Mecanum wheels facilitate 360° omnidirectional movement, rendering it an ideal platform for developing and testing autonomous driving systems [4].

1.2 Overview and Motivations

Autonomous navigation has become a fundamental topic in the field of robotics, with applications spanning industrial automation, smart transportation, and assistive robotics. The capacity of a robot to move autonomously on a path and make

dynamic adjustments for obstacles is a significant advancement towards developing more sophisticated autonomous systems.

The demand for intelligent robotic systems capable of sensing, comprehending, and responding appropriately is increasing. Standard line following robots rely on vision or sensor-based tracking, but often encounter challenges in diverse environments, occlusions, and real-world conditions. The integration of intelligent artificial technology aims to enhance the robustness of autonomous robots, demonstrating the potential for combining machine learning and robotics to develop advanced navigation systems [5].

Beyond navigation, autonomous robotics hold significant potential in various industrial sectors. In the field of manufacturing, autonomous robots facilitate the efficient transportation of goods within warehouses and production lines. In the context of urban mobility, self-driving vehicles employ advanced navigation principles to enhance transportation safety and efficiency [6]. Furthermore, assistive robots equipped with intelligent navigation systems can support individuals with disabilities, facilitating movement in controlled environments and improving overall accessibility.

The integration of Machine Learning and Robotics will continue to evolve, driving further advancements in autonomy. The development of more robust and adaptive autonomous robots will further the safer, faster and smarter automation in other research areas as research progresses.

1.3 Thesis Outline

This thesis is divided into seven chapters, each covering a specific aspect of the research and development process:

Chapter 1 introduces the goals and the context of the project, providing a background information on key concepts of autonomous driving and the thesis structure.

Chapter 2 presents a comprehensive review of the existing technologies pertinent to the project, encompassing an introduction to ROS, a comparative analysis of ROS1 and ROS2, an overview of LiDAR and depth camera technologies, and the application of machine learning in object detection, specifically utilising SSD MobileNetv2 for traffic sign recognition.

Chapter 3 describes the hardware components used in the project, including the Jetson Nano, LiDAR, sensors, motors, and camera. It also covers the assembly of the rover and the system configurations required for development, such as Visual Studio Code (VS Code), Secure Shell Protocol (SSH), and Docker setup.

Chapter 4 explains the process of preparing a custom dataset, training the SSD MobileNetv2 model, and evaluating the performance of the model using standard metrics.

Chapter 5 provides an in-depth discussion of the software framework, including the ROSMaster library, Python scripts for line following and traffic sign detection,

ROS2 scripts, and MQTT communication. The chapter also explains the environment setup and launch files.

Chapter 6 presents the testing scenarios, evaluates the performance of the rover, and discusses encountered issues and possible improvements.

Chapter 7 summarises the results, highlights system limitations, and suggests potential extensions of the project. The chapter also explores practical applications and the broader impact of this research.

Chapter 2

State of the Art

2.1 ROS

Robot Operating System (ROS) is an open source framework, used in the field of robotics, that defines the components, interfaces and tools for building advanced robots. Initially created in 2007 by Willow Garage, ROS has revolutionised the approach to the development of robotic systems by providing a flexible, modular and scalable platform that facilitates the design, implementation and testing of complex robotic algorithms [7] [8]. It supports a federated system reminiscent of code repositories, thus facilitating collaboration and dissemination of projects. This means that individual projects can be developed and executed independently, while being fully integrated with the core tools provided by ROS.

Currently, ROS is designed to run exclusively on Unix-based platforms. Although its software is mainly tested on Ubuntu and macOS systems, the ROS community has helped extend support to other Linux distributions, including Fedora, Gentoo and Arch Linux [8].

2.1.1 Architecture of ROS

ROS has three levels of concepts: the filesystem level, the computing graph level and community level [9].

Filesystem level

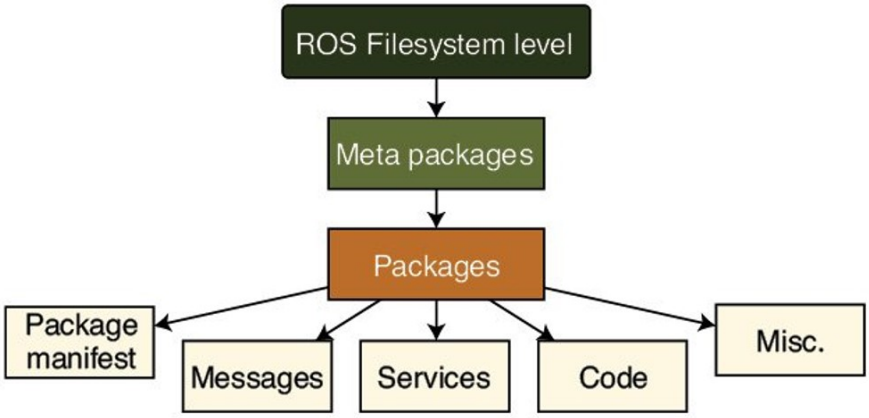


Figure 2.1: Filesystem Level of ROS.

The Filesystem layer, shown in Figure 2.1 [10], comprises the primary resources stored on disk, which are essential for organising and managing software components. These resources include:

- **Meta packages:** A specialised type of package that serves as a logical grouping of related packages. Meta packages are often used to maintain backward compatibility for projects that have moved from older ROS build systems, such as rosbuild stacks.
- **Packages:** The fundamental unit of software organisation in ROS. A package may contain runtime processes (nodes), libraries, data sets, configuration files or any other related resource. Packages are the smallest unit that can be built and released within ROS, making them the most atomic element of the system.
- **Package manifests:** Each package includes a manifest file (package.xml), which provides essential metadata such as package name, version, description, licence details, dependencies and additional exported information.
- **Messages:** Message files define the structure of data exchanged between nodes in ROS. These descriptions are stored in the package directory, following the format `my_package/msg/MyMessageType.msg`.
- **Services:** Services in ROS enable synchronous communication between nodes by defining request and response structures [10] [8].

Computation Graph Level

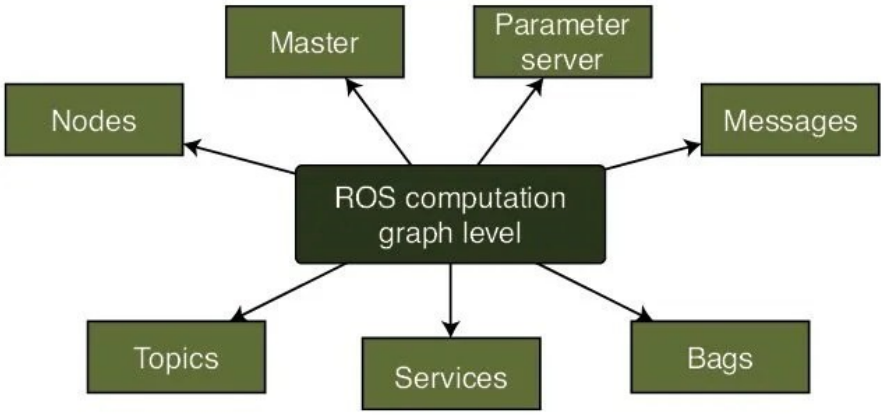


Figure 2.2: The computation Graph Level of ROS.

The ROS Computation Graph, in Figure 2.2 [10], is a peer-to-peer network of ROS processes that work together to process data. The main concepts of the ROS Computation Graph are nodes, the Master, the Parameter Server, Messages, Services, Topics and Bags. Each of these components plays a role in the flow of data through the system [9].

- **Nodes:** Nodes are the individual processes responsible for computation in ROS. A robotic control system usually consists of several nodes, each of which performs a specific task, such as controlling a sensor (e.g. a laser rangefinder), motors, localisation or path planning. The nodes are implemented using ROS client libraries such as `roscpp` or `rospy`.
- **Master:** The ROS Master is responsible for registration and name look-up, enabling nodes to find each other, exchange messages and invoke services. The Master facilitates dynamic connections between nodes when they are added or removed. Although the Master helps nodes find each other, it does not directly manage communication between them. Nodes that subscribe to a topic connect to nodes that publish that topic. This connection is made via a protocol based on standard TCP/IP sockets.
- **Parameter Server:** The Parameter Server stores data in a central location, identified by a key. It is part of the Master and is used to share configuration or parameter data between the various nodes.
- **Messages:** Messages are the fundamental unit of communication between nodes. They are data structures composed of fields of specific types, such as integers, floating-point numbers, Booleans and arrays. Messages can include complex structures and nested arrays, similar to C structures.

- **Topics:** They are used for communication via a publish/subscribe model. A node sends messages by publishing them on a topic and other nodes can subscribe to the topic to receive the data. Multiple publishers and subscribers may exist for a single topic. This model decouples the production of information from its consumption, providing flexibility to the system.
- **Services:** While the publish/subscribe model is ideal for many-to-many communication, it is not suitable for request/reply interactions. For such interactions, ROS uses services, which involve a pair of messages: one for the request and one for the response. A service provider node offers a service with a name and a client node sends a request message and waits for a response.
- **Bags:** Bags are used to store and replay ROS message data. They are essential for storing data that might be difficult to collect but are important for algorithm development and testing, such as sensor data.

This structure allows flexibility and decoupling. For example, a node that publishes data does not need to know which nodes subscribe to it and vice versa. This decoupling makes it easier to modify or expand the system. For example, if we add another sensor, we only have to re-attribute the names of the topics to which nodes subscribe or publish, instead of making changes to the nodes themselves [8] [9].

Community Level

At the community level, several resources facilitate the exchange of software and knowledge within the ROS ecosystem. These include:

- **Distributions:** Much like Linux distributions, ROS distributions group specific sets of software versions together, facilitating installation and ensuring consistent versions across multiple components.
- **Repositories:** ROS uses a federated network of code repositories where different institutions can develop and release their own robot software components.
- **ROS Wiki:** The community Wiki is the main platform for documenting information on ROS. It is an open space for anyone who wants to contribute by creating tutorials, providing corrections or updating existing documentation.
- **Bug reporting system:** This is the place where tickets can be submitted to report problems or bugs of ROS.
- **Discussion lists:** The ros-users mailing list is the main means of communication for updates on ROS and a forum to discuss software-related questions.
- **ROS Answers:** A dedicated Q&A platform where users can ask and answer ROS-related questions.

- **Blog:** The official ROS blog provides regular updates, including pictures and videos, offering insights into new developments and community events.

2.1.2 ROS2

ROS2 represents the second generation of the Robot Operating System. It is an enhanced version of ROS1, specifically designed to address several of the issues and limitations present in the previous system. This version introduces a more robust and flexible framework for creating robotic applications, providing improved functionality and addressing major challenges:

- **Platform Support:** ROS1 primarily supports Linux, with some support for Mac OS and Windows. ROS 2 extends support to multiple platforms, including Windows 10 and Mac OS X, broadening its applicability.
- **Distributed architecture:** In ROS1, the communication relies on a central master node that manages the registration of nodes and facilitates their communication. This master-slave architecture can become a bottleneck in large-scale systems. In contrast, ROS2 adopts a distributed architecture based on the Data Distribution Service (DDS), that plays a central role in the ROS2 system and enabling direct communication between nodes without the need for a central master. This model is somewhat similar to the broadcast model, where all nodes can publish and subscribe to messages on the DataBus. However, its key improvement is that communication involves multiple parallel paths. Each node only needs to focus on the messages it cares about and can ignore those it does not need (it is like a rotating hot pot where various dishes are transmitted on the DataBus). In the architecture of ROS2, illustrated in Figure 2.3 [11], the blue and red sections represent DDS. By incorporating DDS into the four major components of ROS, it significantly enhances the overall capabilities of the distributed communication system. This means that when developing robots, communication issues no longer need to be a concern, allowing more focus on other aspects of application development.

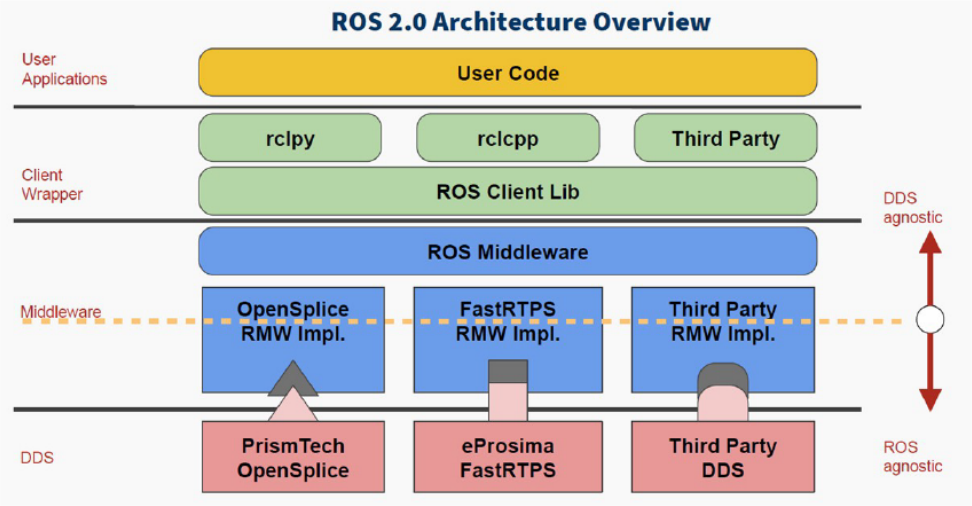


Figure 2.3: ROS2 architecture

Another crucial feature of DDS is its Quality of Service (QoS), a network transmission strategy where the application specifies the required quality of network communication. It ensures that these requirements are met as much as possible, aiming to satisfy the communication quality needs of the customer.

This concept can be regarded as an agreement between the data provider and the receiver, ensuring that the transmission adheres to the specified requirements [11].

- **Real-Time Capabilities:** A key feature of ROS2 is the support of real-time functionality. This is especially important for robotic applications that require timely responses, such as hardware control or real-time sensor data processing. This real-time support is a fundamental upgrade from ROS1, where real-time performance was often difficult to achieve.
- **Support for modern programming languages:** ROS2 also embraces modern programming languages, supporting C++11 and Python 3.5 or later. This allows developers to take advantage of the latest features and improvements of the languages, making their code more efficient and maintainable.
- **Support for modern programming languages:** In addition, ROS2 introduces a new compilation system called Ament, which replaces the Catkin system of ROS1. Ament provides a more flexible and powerful compilation system that better supports the needs of modern robotic applications.
- **Communication:** ROS1 and ROS2 can still communicate with each other through a bridge known as the rosbridge. This allows developers to integrate existing ROS1 systems with ROS2, easing the transition for projects already built on ROS1.

2.1.3 Comparison of ROS1 and ROS2

ROS2 improvements and the main differences between ROS1 and ROS2 are summarised in the Table 2.1. Figure 2.4 [12] illustrates the main architectural differences between ROS1 and ROS2.

Feature	**ROS1**	**ROS2**
Architecture	Centralised Master node for communication	Distributed architecture without a Master node
Platform Support	Ubuntu, limited support for other platforms	Ubuntu, macOS, Windows 10
Node Communication	Synchronous (via Master node)	Asynchronous and more flexible
Real-Time Support	Limited or no real-time support	Full real-time support
Node Writing Style	No specific convention for writing nodes	Uses Object-Oriented Programming (OOP) conventions
Programming Languages	C++03, Python 2.7	C++11, Python 3.5+
Cross-compatibility	ROS1-only	ROS1 and ROS2 can communicate through rosbridge
Middleware	Custom protocol, not DDS-based	Built on DDS

Table 2.1: Comparison between ROS1 and ROS2

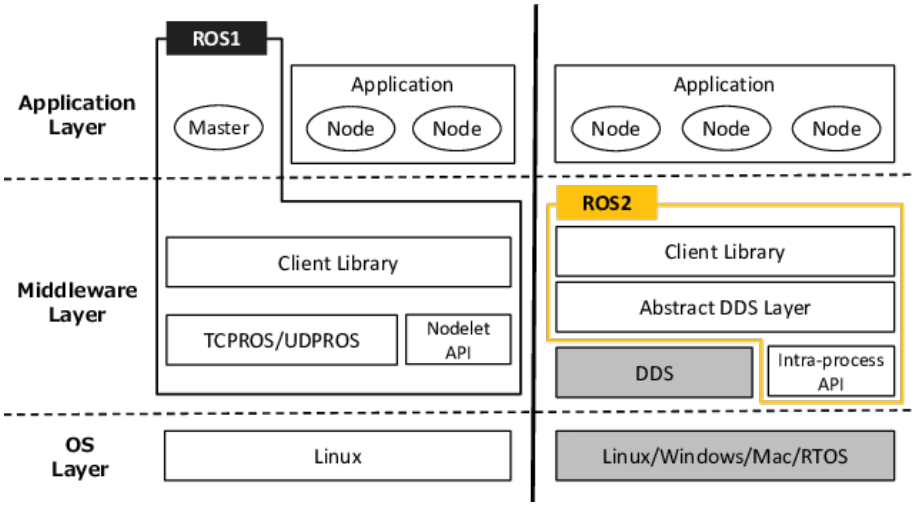


Figure 2.4: ROS1 and ROS2 architecture comparison

2.2 LiDAR Technology

The LiDAR is a remote sensing method that uses laser pulses to measure distances to objects. This systems emit laser beams and measure the time it takes for the reflected light to return, calculating distances with high accuracy. This technology is widely used in autonomous vehicles and robotic systems for spatial awareness and can be classified into single-line and multi-line configurations, depending on the number of laser beams used for scanning.

2.2.1 Single-line LiDAR

A single-line LiDAR emits a single laser beam and is commonly used in robotics due to its high scanning speed, high resolution and reliable performance. It operates with a higher angular frequency and sensitivity than multi-line LiDARs, which makes it particularly accurate in measuring distances and detecting obstacles.

Considering a single-line LiDAR, developed by SLAMTEC, the system consists of four main components: the laser, the receiver, the signal processing unit and the rotation mechanism, as shown in Figure 2.5.

- The **laser** serves as the emission source of the LiDAR and operates in pulsed mode.
- Once the laser beam hits an obstacle, the reflected light is captured by the **receiver** through a lens system that focuses the signal for further processing.
- The **signal processing unit** monitors the laser emission and processes the received signals. Based on these signals, it calculates the distance to the target object with great accuracy.

- These three main components are mounted on a **rotating mechanism**, which ensures continuous scanning by rotating at a stable speed. This enables LiDAR to generate a 2D plan of the environment in real time [13].

Figure 2.5: System composition of RPLIDAR

Two methods of single-line LiDAR, to compute the distance, are the triangulation measurement and the time of flight (ToF) [13].

Triangulation method

The laser triangulation technique, shown in Figure 2.6 [14], determines the distance by projecting a laser beam at an angle onto a target. The reflected laser is then captured by a lens and focused on a Charge-Coupled Device (CCD) position sensor. As the target moves along the path of the laser, the position of the reflected dot on the sensor shifts accordingly. The displacement of the dot is proportional to the movement of the target and allows the distance to be calculated using an appropriate algorithm.

This method is based on the principles of trigonometry, as the incident and reflected beams form a triangle. By applying geometric calculations, the system accurately determines the distance between the target and the LiDAR sensor.

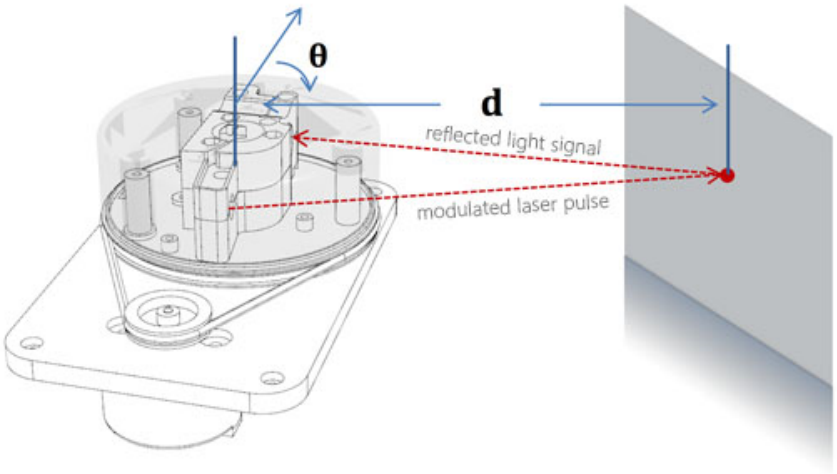


Figure 2.6: RPLIDAR based on laser triangulation.

Time-of-Flight

The Time-of-Flight (ToF) technique, illustrated in Figure 2.7 [15], determines distance by measuring the time it takes for a laser pulse to reach a target and return. The system emits a modulated laser beam that hits the target and is partially reflected by the LiDAR sensor. By analysing the phase shift between the transmitted and received signals, the system accurately calculates the distance. The method is based on directing a laser beam towards a target. A portion of the photons in the beam is reflected upon impact and detected by the sensor. By recording the time taken for this round trip, the system calculates the distance using the following formula:

$$\text{Distance} = \frac{\text{Photon travel time}}{2} \times \text{Speed of light}$$

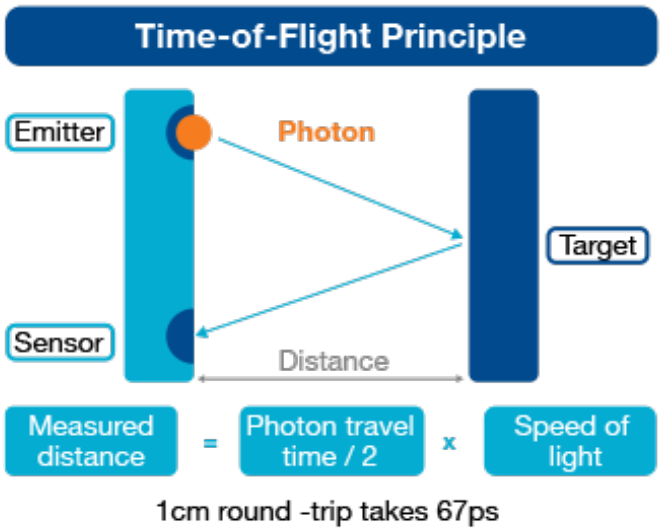


Figure 2.7: ToF Method

2.3 Depth Camera: Astra

Depth cameras are designed to capture and record three-dimensional information from a scene. Unlike traditional 2D cameras, they are able to sense depth, allowing detailed 3D representations of objects and environments to be generated [16].

The Astra series by Orbbec is a line of depth cameras that provide 3D sensing capabilities. These cameras are compatible with ROS 2, allowing for seamless integration into robotic systems. The Astra cameras utilize structured light technology to capture depth information, offering high-resolution depth data suitable for various applications, including object detection and environment mapping [17].

Figure 2.8: The components of a depth sensor

Depth cameras, illustrated in Figure 2.8 [18], use IR (Infra-Red) technology to

generate depth maps of the environment. Unlike standard RGB (Red Green Blue) cameras, depth sensors require two essential components: an IR projector and an IR camera. Although some depth sensors may include an RGB camera, this is not a fundamental requirement for depth detection. The IR projector emits a structured pattern of infrared light that illuminates objects in the scene. Although invisible to the human eye, this pattern appears as a dense grid of dots when observed by an IR camera. The IR camera, which functions like a standard camera but in the infrared spectrum, captures this pattern as it is distorted by objects struck at different distances. The depth sensor processor then analyses these distortions to calculate depth information. Objects closer to the sensor show a more distorted pattern of points, while those further away appear with a denser distribution. By evaluating the displacement of these points, the system constructs a depth map representing the spatial geometry of the scene. This depth map can be viewed directly from the sensor and processed by a computer for applications such as 3D reconstruction, object recognition and augmented reality [18].

2.4 Machine Learning and Object Detection

Machine learning has revolutionised the field of computer vision, allowing systems to perform complex visual recognition tasks with remarkable precision. One of the most important applications of machine learning in computer vision is object detection, which involves not only identifying objects within images or video frames but also determining their location. Unlike traditional image classification, which only labels an entire image, object detection offers both classification and localisation of objects [19].

In recent years, deep learning has transformed object detection, making it faster and more accurate than ever before. Thanks to powerful neural networks and GPU acceleration, advanced models can detect and track objects in real time, paving the way for advanced applications based on artificial intelligence.

In automotive technology, it plays a key role in Advanced Driver Assistance Systems (ADAS), helping vehicles detect pedestrians, lane markings and other cars, thereby improving overall road safety. Modern object detection is based on deep learning and Convolutional Neural Networks (CNNs). Some of the most widely used models are YOLO (You Only Look Once), SSD and R-CNN (Region-based CNN), all of which are designed to quickly and accurately identify and locate objects in an image. [20] [21]

There are two main ways to implement object detection using deep learning:

- **Using pre-trained models:** these are models that have already been trained on huge datasets and are able to instantly detect common objects (such as people, vehicles and text) without additional training.

- **Training a Custom Model:** when a specific type of object needs to be detected, a technique called transfer learning allows an existing model to be fine-tuned for a specific application. This saves time compared to training a model from scratch, as the underlying network has already learned useful visual models. In this project, a custom model was trained to detect road signs [20].

2.4.1 SSD MobileNet

Single Shot multiBox Detector SSD MobileNet is an object detection model designed for real-time inference on devices with limited computational resources, such as smartphones and embedded systems. This models achieve a balance between speed and accuracy by using depth-wise separable convolutions and they are vital for robotic perception tasks that involve object detection [22].

In this project, the SSD MobileNet model is used for object detection and runs on the Jetson Nano with Compute Unified Device Architecture (CUDA) acceleration, that allows the model to use parallel processing on the Jetson Nano GPU, significantly improving the inference speed, which enables real-time detection and classification of objects.

Architecture

The architecture of SSD MobileNet consists of two main components: the base MobileNetV2 convolutional neural network and the SSD layer. The MobileNetV2 network serves as a feature extractor, providing feature maps that are subsequently processed by the SSD layer to classify detected objects [22]. It employs an inverted residual structure, where the residual connections are positioned between the bottleneck layers. The intermediate expansion layer utilises efficient depthwise convolutions to process features, introducing non-linearity. In its entirety, the MobileNetV2 architecture consists of an initial fully convolutional layer with 32 filters, followed by 19 residual bottleneck layers, as shown in Figure 2.9 [23].

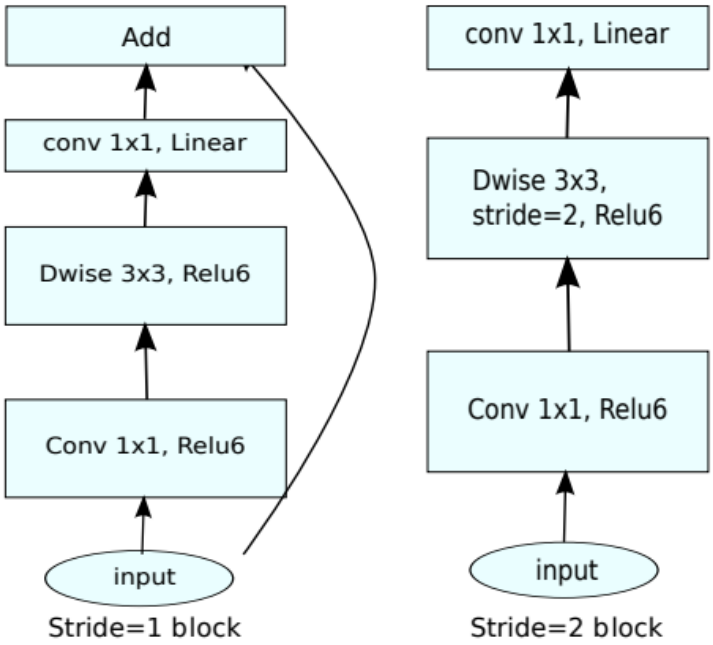


Figure 2.9: MobileNet V2 Architecture

SSD Networks

SSD networks work by dividing the output space into a series of predefined bounding boxes with varying aspect ratios and scales. During inference, confidence scores are assigned to each class for each selection rectangle, followed by non-maximum suppression to discard redundant detections. In addition, predictions of multiple levels are merged to improve the detection of objects at different scales [24].

The architecture of SSD, illustrated in Figure 2.10 [25], is built upon the well-established VGG-16 model, though it omits the fully connected layers. VGG-16 was chosen as the base network due to its proven effectiveness in high-quality image classification tasks and its widespread use in problems where transfer learning enhances performance. Instead of the original fully connected layers, SSD incorporates a series of auxiliary convolutional layers (beginning from conv6). This modification facilitates the extraction of features at multiple scales and progressively reduces the input size at each subsequent layer [25].

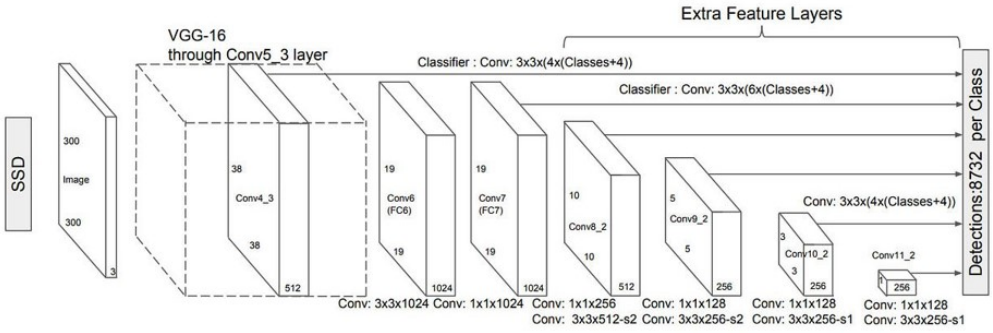


Figure 2.10: Architecture of SSD

Chapter 3

ROSMASTER X3 Hardware

In this chapter, the hardware components and structure of the robot are detailed, with a particular emphasis on the step-by-step assembly process.

The ROSMASTER X3, as introduced in Chapter 1.1, is an advanced robotic platform designed for research and development in autonomous navigation and robotic applications. It offers a versatile design that is compatible with both the Raspberry Pi and Jetson series controllers, ensuring that users can choose the hardware that best suits their performance requirements. All configurations operate on the Ubuntu system.

The choice of main controller primarily influences the performance of the robot, for example, differences in processing power can affect the speed of data handling and the efficiency of executing complex algorithms. However, the course materials, product features, and control software provided remain consistent regardless of the selected hardware. This consistency allows researchers and developers to concentrate on optimising their applications without the need to adapt to different software ecosystems.

The structure of the robot is shown in Figure 3.1 [26]. It is constructed of aluminium alloy, ensuring durability while maintaining a lightweight frame.

Figure 3.1: ROSMASTER X3

3.1 Main components of the robot

The heart of the system is the Jetson Nano 4GB, which serves as the primary processing unit. It is connected to a dedicated ROS expansion board that acts as the central hub for interfacing with all peripheral components. A high-performance LiDAR sensor is strategically mounted on the chassis to provide comprehensive environmental scanning, essential for 3D mapping and obstacle detection. Complementing the LiDAR is a depth camera, connected via a Universal Serial Bus (USB) interface to the Jetson Nano, which supplies real-time depth information critical for navigation and object recognition tasks.

Additionally, the platform incorporates a voice interaction module, allowing users to control robot movement and execute functions through voice commands. This module is integrated with the system through appropriate interfacing, via USB or GPIO connections, ensuring seamless communication with the Jetson Nano. Its 360° omnidirectional Mecanum wheels, each driven by independent DC motors, grant the robot exceptional manoeuvrability.

Furthermore, the ROSMASTER X3 supports multiple remote control methods, including a mobile phone application, a handheld controller, the ROS system itself, and even a computer keyboard.

3.1.1 Jetson Nano

The Jetson Nano 4GB SUB (Figure 3.2 [27]) is the main board of ROSMASTER X3 and it is configured with the image provided by Yahboom, which provides a pre-configured environment and useful material. Developed by NVIDIA, the board is an embedded computing platform designed for edge AI applications, providing a balance between performance and power efficiency.

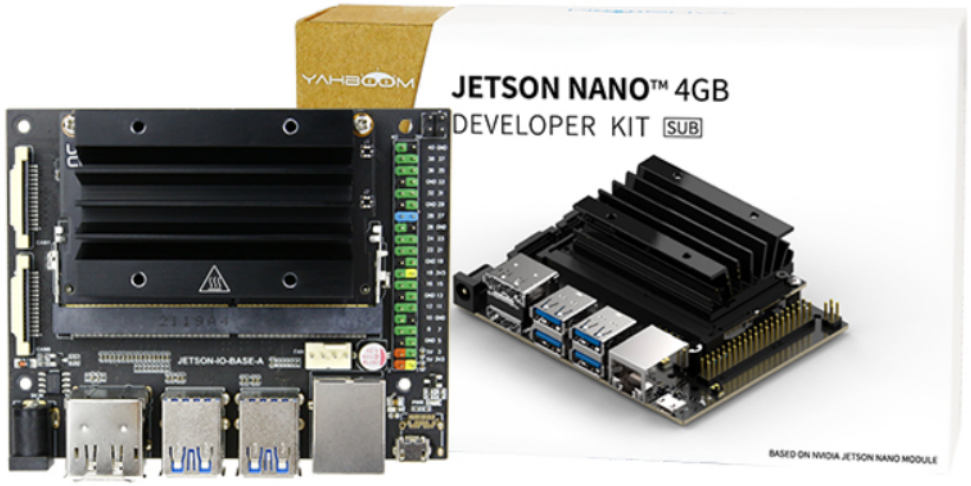


Figure 3.2: Jetson Nano 4GB SUB

The status of the board is shown on the OLED display, which is directly connected to it, and provides key system information, including:

- **Central Processing Unit (CPU) Utilisation:** Displays the current processing load, allowing for monitoring of computational resource usage.
- **RAM Usage:** Indicates the amount of memory that is being used, helping to assess system performance and potential bottlenecks.
- **Storage Utilisation:** Provides insights into available disk space, ensuring sufficient capacity for data logging and software execution.
- **Network Status:** Shows the IP address assigned to the Jetson Nano, facilitating remote access and ROS communication over a network.

The Jetson Nano possess better computing power, thanks to its NVIDIA Maxwell Graphic Processing Unit (GPU) with 128 CUDA cores, and can be combined with NVIDIA TensorRT to accelerate deep learning. Compared with Raspberry Pi 4B, it can run basic Artificial Intelligence (AI) deep learning algorithms more efficiently and with greater stability. Despite having a faster CPU, the latter is not designed for heavy AI workloads, as it lacks a dedicated GPU acceleration unit for deep learning.

The detailed specifications of the Jetson Nano are presented in Table 3.1 [28], where they are compared with those of the Raspberry Pi 4B.

Feature	Jetson Nano 4GB SUB	Raspberry Pi 4B
CPU	Quad-core ARM Cortex-A57 MPCore (1.43 GHz)	Quad-core ARM Cortex-A72 (1.5 GHz)
GPU	NVIDIA Maxwell, 128 CUDA cores	Broadcom VideoCore VI
RAM	4GB LPDDR4	2GB, 4GB, 8GB LPDDR4
Storage	MicroSD, eMMC (optional), USB	MicroSD
AI Acceleration	Yes, CUDA, TensorRT, Deep Learning optimised	No AI acceleration
Video Output	HDMI + DisplayPort	2x micro-HDMI
USB Ports	4x USB 3.0, 1x USB 2.0 Micro-B	2x USB 3.0, 2x USB 2.0
Ethernet	Gigabit Ethernet	Gigabit Ethernet
Wireless	Requires USB Wi-Fi module	Wi-Fi 802.11ac, Bluetooth 5.0
Power Consumption	5W - 10W	Max 6.7W
OS Support	Ubuntu-based NVIDIA JetPack	Raspberry Pi OS, Ubuntu
ROS Compatibility	Optimised for ROS/ROS2	Supports ROS, but less optimised
Edge AI Applications	Yes, supports TensorFlow, PyTorch, OpenCV	Limited, lacks hardware acceleration

Table 3.1: Comparison between Jetson Nano 4GB SUB and Raspberry Pi 4B

3.1.2 ROS robot expansion board

The ROS Expansion Board V1.0, shown in Figure 3.3 [29], serves as the primary interface between the Jetson Nano and the various hardware components of the ROSMASTER X3, acting as both a communication bridge and a power management hub. Communication between the Jetson Nano and the expansion board occurs via

a USB serial connection, where the Jetson Nano transmits serial data to the onboard microcontroller (MCU). This microcontroller, an STM32F103RCT6, is responsible for interpreting and executing commands, enabling seamless control over multiple peripherals [30].

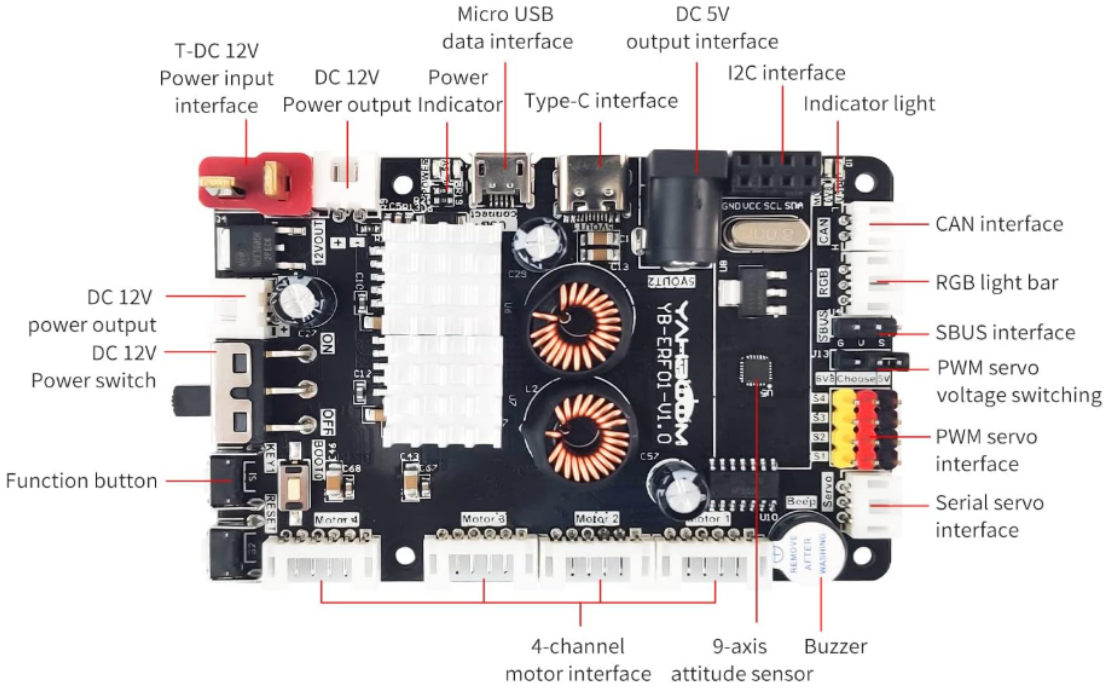


Figure 3.3: ROS expansion board

The expansion board plays a crucial role in power management. It connects to a 12V battery via a T-type DC interface, with an onboard voltage converter ensuring a stable 5V supply to the Jetson Nano. This also powers essential peripherals such as the USB hub, motors and sensors.

The board supports four 12V encoder motors (Figure 3.4 [29]), allowing for precise closed-loop control, which is critical for autonomous navigation. In addition, it features four PWM-controlled and serial bus servos, providing flexible actuation. For real-time localisation and stability, an integrated 9-axis attitude sensor provides essential motion and orientation data.

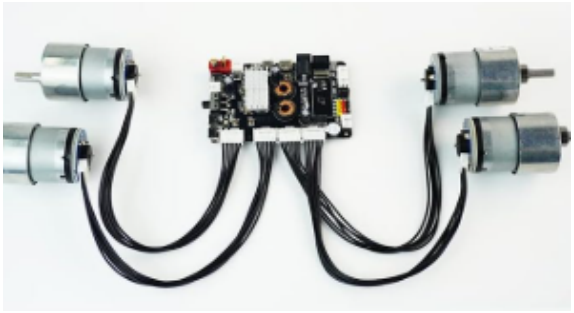


Figure 3.4: 4-Channel motor with encoder

Beyond power and motion control, the board includes various interactive interfaces and components, such as an RGB light bar, an active buzzer and control buttons (RESET, KEY1, and BOOT0) for firmware management and system configuration. These elements are managed by the onboard microcontroller, which responds to commands from the Jetson Nano.

3.1.3 RPLIDAR A1

This type of LiDAR (Figure 3.5) is based on laser triangulation ranging principle, already discussed in Section 2.2.1, and uses high-speed vision acquisition and processing hardware developed by Slamtec. The system measures distance data in more than 8000 times per second.

The core of RPLIDAR A1 runs clockwise to perform a 360 degree omnidirectional laser range scanning for its surrounding environment and then generate an outline map for the environment. It improves the internal optical design and algorithm system to make the sample rate up to 8000 times,which is the highest in the current economical LiDAR industry.

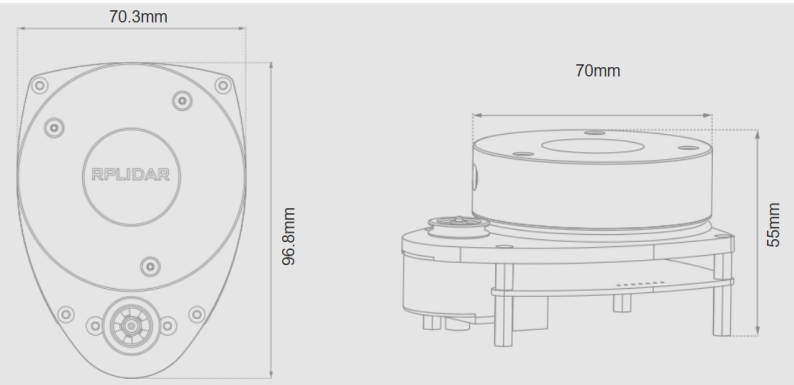


Figure 3.5: RPLIDAR A1 dimensions

The performance of this type of sensor is defined by several key parameters [13]:

- **Ranging Radius:** The maximum distance the LiDAR can measure.
- **Ranging Sample Rate:** The number of distance measurements performed per second.
- **Scanning Frequency:** The number of full scans the LiDAR performs per second.
- **Angular Resolution:** The angle between two consecutive measurements.
- **Measurement Accuracy:** The smallest detectable change in distance.

A summary of the LiDAR specifications is presented in Table 3.2 [31].

Measuring Range	0.15m - 12m
Sampling Frequency	8K
Rotational Speed	5.5Hz
Angular Resolution	$\leq 1^{\circ}$
Dimensions	96.8 x 70.3 x 55mm
System Voltage	5V
System Current	100mA
Power Consumption	0.5W
Output	UART Serial (3.3V voltage level)
Temperature Range	0℃ - 40℃
Angular Range	360°
Range Resolution	$\leq 1\%$ of the range ($\leq$ 12m) $\leq 2\%$ of the range (12m - 16m)
Accuracy	1% of the range ($\leq$ 3m) 2% of the range (3m - 5m) 2.5% of the range (5m - 25m)

Table 3.2: Specifications of the RPLIDAR A1 Sensor

3.1.4 Astra Pro Plus

The Orbbec Astra Pro Plus is an advanced 3D depth camera designed for robotics, computer vision, and various AI applications. It integrates depth sensing, colour

imaging, and infrared technology, making it suitable for applications requiring precise spatial perception.

This type of camera uses structured light technology to capture depth information, enabling high-precision 3D mapping. In this project, the model of machine learning is trained on RGB frames, so only RGB camera is used.

Astra Pro Plus operates efficiently in a depth range of 0.6 m to 8 m, providing a resolution of 640 × 480 pixels at 30 frames per second for depth data and 1920 × 1080 pixels for RGB images.

The Table 3.3 shows the technical features of the camera [32].

Specification	**Details**
Depth Technology	Structured Light
Depth Range	0.6m - 8m
Depth Resolution	640 × 480 @30fps
RGB Resolution	1920 × 1080 @30fps
Field of View (FOV)	Horizontal: 60° / Vertical: 49.5° / Diagonal: 73°
Operating Temperature	0℃ - 40℃
Interface	USB 2.0
Power Consumption	2.25W
Operating System Compatibility	Windows, Linux, Android
Dimensions	165mm × 30mm × 40mm
Weight	200g

Table 3.3: Technical specifications of the Orbbec Astra Pro Plus

3.1.5 Mecanum Wheels

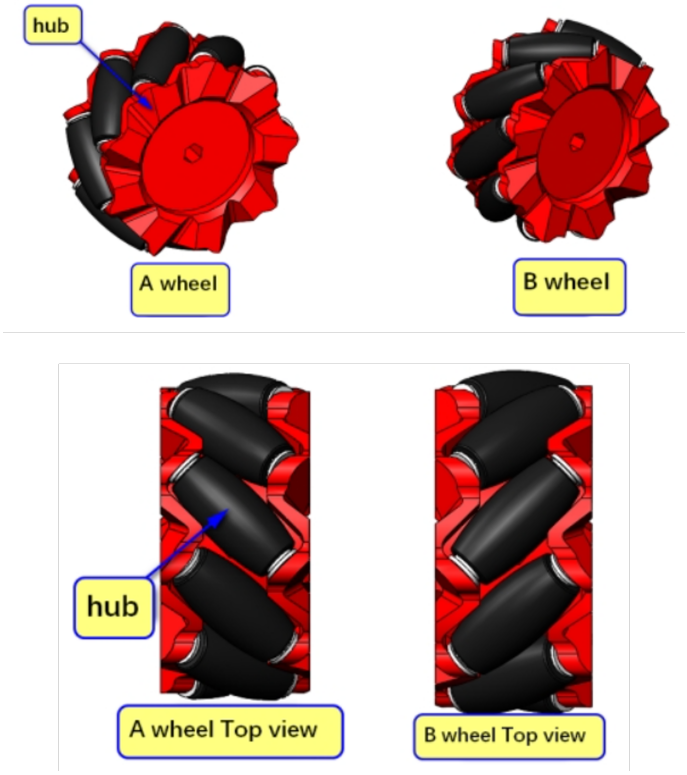


Figure 3.6: Wheels hub

A Mecanum wheel consists of a central hub, shown in Figure 3.6, and several rollers arranged around it. These rollers are passive, i.e. they are not directly powered but rotate freely. The axis of each roller is positioned at an angle of 45° to the axis of the hub. The Mecanum wheel assembly comprises two distinct types, designated A and B, which are the mirror image of each other.

Mecanum wheels are classified into two types based on their directional movement. When a type A wheel moves forward, it simultaneously moves to the right, creating an oblique forward-right movement. In contrast, when moving backward, it moves to the left, resulting in a back-left oblique movement. Similarly, the type B wheel follows the opposite pattern, allowing an oblique forward-left or back-right movement.

Throughout this study, the front of the carriage is considered the positive direction. The forward movement of a wheel corresponds to standard motor rotation, while the backward movement indicates motor reversal.

The correct configuration for installing Mecanum wheels in ROSMASTER X3 should follow the diagram [ABBA] (Figure 3.7 [33]).

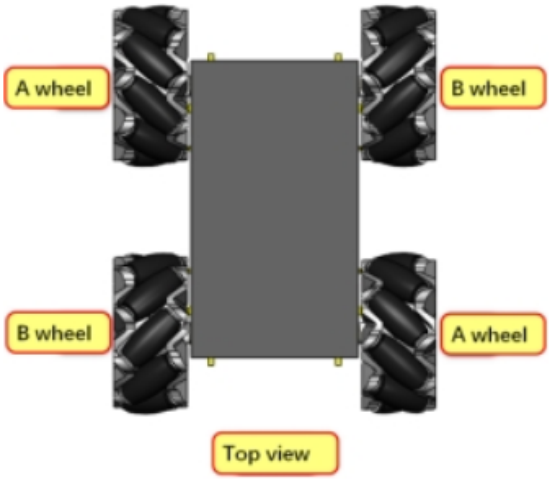


Figure 3.7: Mecanum wheels configuration

3.2 Rover Assembly

The ROSMASTER X3 robot was provided by MCA Engineering S.r.l., which supplied a kit containing all the necessary components, requiring full assembly, before proceeding with the implementation phase.

Figure 3.8: ROSMASTER X3 components

Starting with the components illustrated in Figure 3.8, the assembly process began with the construction of the bottom frame, which serves as the structural foundation of the robot (Figure 3.9). This stage involved securely fastening the base components to ensure stability and alignment. Following this, Jetson Nano (Figure 3.10), the camera (Figure 3.11) and front motors were carefully mounted onto the frame. The positioning of these elements was crucial, as the Jetson Nano acts as the central processing unit of the system, while the camera plays a key role in autonomous navigation. The assembly of the bottom frame was then completed

by installing the 12V battery, ensuring a reliable power supply for the entire system, the rear motors and attaching the four wheels (Figure 3.12). Special attention was given to aligning the motors correctly to guarantee smooth and efficient movement.

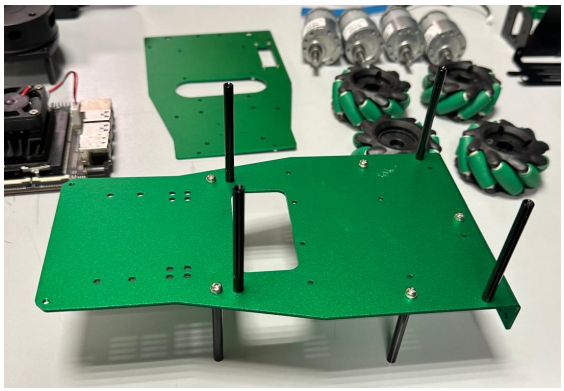


Figure 3.9: Bottom frame

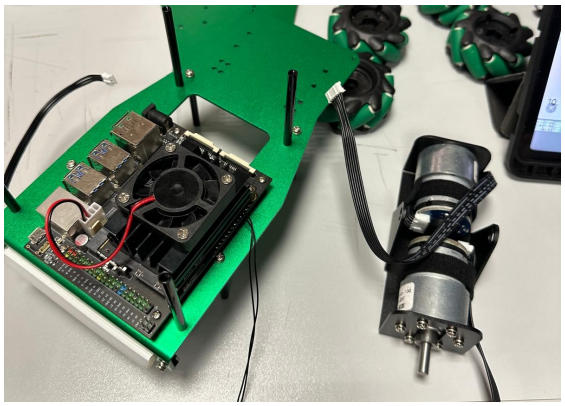


Figure 3.10: Jetson Nano

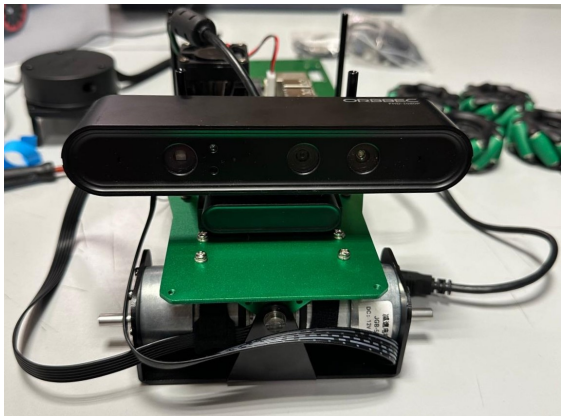


Figure 3.11: Depth Camera

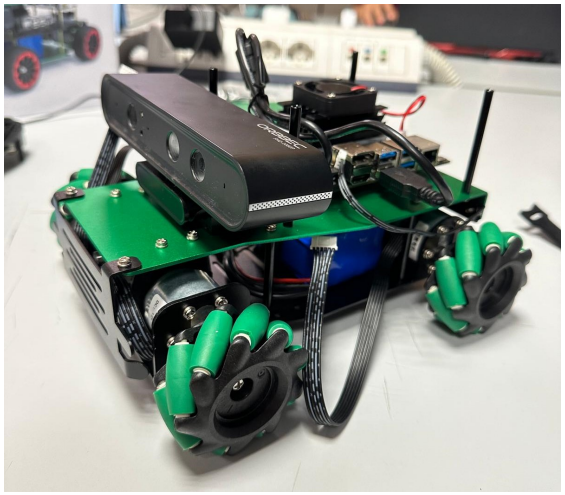


Figure 3.12: Mecanum Wheels

With the base structure fully assembled, the focus shifted to integrating the upper components, which included mounting the LiDAR sensor, ROS expansion board, and OLED display onto the upper frame (Figure 3.13). The LiDAR was positioned at an optimal height to provide a clear 360-degree scan of the surrounding environment, essential for mapping. The USB hub and ROS expansion board were securely fixed to facilitate seamless communication between various electronic modules. The Wi-Fi antenna was then attached to enable wireless connectivity, which is essential for remote control and data transmission. All necessary cables were connected, with care taken to secure them in place using cable ties and protective sleeves to prevent tangling and accidental disconnections.

Upon completion of these steps, the assembly process was finalised, resulting in a fully constructed and operational robot (Figure 3.14).

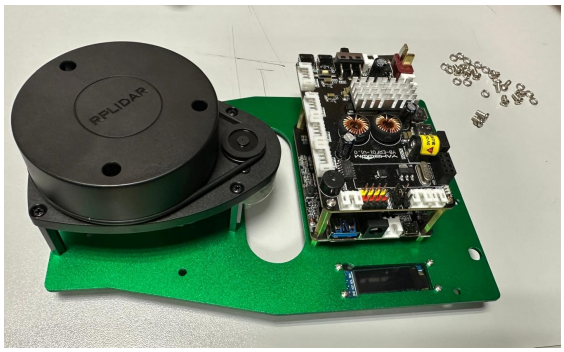


Figure 3.13: LiDAR and Expansion Board

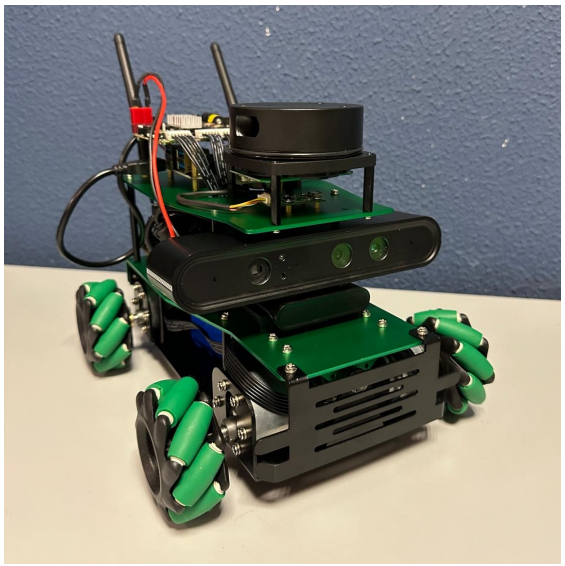


Figure 3.14: Robot fully assembled

Chapter 4

Traffic Sign Recognition System

This chapter discusses the object detection process for traffic sign recognition, covering the entire pipeline from dataset preparation to model training and real-time inference on the Jetson Nano.

The training phase was conducted on a Lenovo IdeaPad 3 15ITL6 (technical specifications are shown in Table 4.1) running Windows 11 Home, as the computational demands of the training exceeded the memory limits of the Jetson Nano. Once the model was successfully trained, it was subsequently deployed on the board, leveraging its built-in processing capabilities for real-time inference in a resource-limited environment.

Component	**Value**
CPU	11th Gen Intel(R) Core(TM) i7-1165G7 @2.80GHz
Number of Cores	4
RAM Capacity	16 GB
RAM Speed	3200 MHz
Graphics Card	Intel(R) Iris(R) Xe Graphics
Video Memory	128 MB

Table 4.1: Details of hardware used for training

4.1 Preparation of the Custom Dataset

The decision to create a custom dataset was driven by several key factors:

- **Pre-trained datasets were unsuitable**: While various publicly available object detection datasets exist, they often contain images that are too small. In fact, the selected SSD MobileNet model operates on 300×300 pixel images, requiring dataset images of appropriate resolution to ensure optimal feature extraction and detection accuracy.

- **Lack of specific traffic sign data**: Large-scale datasets, such as COCO (Common Objects in Context), do not contain the particular traffic signs chosen for this application, making them inadequate for model training.

Consequently, a custom dataset was created that included four traffic signs: mandatory straight ahead, advance warning of the left turn, advance warning of the right turn and stop. The goal was to train an SSD MobileNet model tailored to the specific objects of interest, enabling robust and reliable detection performance.

Examples of images are presented in Figure 4.1.

Figure 4.1: Example images from the dataset.

4.1.1 Dataset generation

The initial stage of the dataset preparation process leverages the functionalities provided by the *jetson-inference* repository, developed by NVIDIA itself for educational and instructional purposes. The repository serves as a comprehensive inference and real-time computer vision Deep Neural Network (DNN) library specifically designed for Jetson devices.

In this phase a custom video was recorded specifically for dataset creation. This video featured the four selected traffic signs in various environments to ensure a diverse range of lighting conditions and viewing angles. Capturing signs at different illumination levels and perspectives was essential to improving the robustness of the model, enabling it to generalise more effectively to real-world scenarios. By incorporating variations in brightness, shadowing, and orientation, the dataset better represents the challenges encountered in real-time traffic sign detection.

Once the video was recorded, the next step involved preparing the dataset for training, which required generating the necessary image samples and organising the dataset structure. To achieve this, a dedicated Python script (Figure 4.2) was executed and prompts the user to enter a model name, in this case `signals`, which is then used to create a dedicated directory within the `data` folder.

```
C:\Users\margh\Desktop\jetson-train>python prepare_dataset.py

Please enter model name: signals
Making directory structure
Total FPS 30.878
Total Img saved 1389
```

Figure 4.2: Dataset creation

Three subdirectories are generated within this folder:

- **ImageSets/Main**: This directory contains text files that define the train-validation split for the dataset. These files are used to specify which images will be used for training and which will be reserved for validation and testing, ensuring proper evaluation of the performance of the model.
- **JPEGImages**: This folder holds the extracted image frames, from the recorded video in JPEG format, which are used as input for training the object detection model. Each image is stored at a resolution of 300 × 300 pixels, ensuring compatibility with the SSD MobileNet architecture.
- **Annotations**: This directory is initially empty, as the images must first be manually annotated before they can be used for training. Annotation involves defining bounding boxes around the traffic signs to create ground-truth labels for the object detection model.

Once the images are generated, the script automatically partitions the dataset: by default, 10% of the images are allocated to the validation and test sets, while the remaining 90% is designated for training. The filenames of these images are stored in corresponding text files within the `ImageSets/Main/` directory.

FPS

Frames Per Second (FPS), in the previous Figure 4.2, refers to the number of frames processed or displayed per second.

In the context of this project, FPS represents the speed at which frames are extracted from the video and saved as images. The value is calculated as:

$$FPS = \frac{\text{Total number of processed frames}}{\text{Total time elapsed (in seconds)}}$$

A higher value indicates greater efficiency in the frame extraction process, whereas a lower value may suggest a slowdown due to hardware limitations or a high-resolution input video.

In the field of computer vision and deep learning, FPS is also used to measure the performance of inference models in real-time applications. For example, on the

Jetson Nano, an object detection model with a high value can process more images per second, ensuring a faster response in real-time scenarios.

4.1.2 Image Annotation Process

Since the images extracted from the video do not come with pre-existing labels, manual annotation is required to specify the locations of traffic signs within each image. To achieve this, the LabelImg software was utilised. It is an open-source annotation tool, developed by Tzutalin, that allows users to create bounding boxes around objects of interest and save the annotations in Pascal VOC format (XML files), which is compatible with the SSD MobileNet training pipeline [34].

By using LabelImg, each image in the JPEGImages directory was carefully labelled, and the corresponding annotation files were stored in the Annotations folder. This step was critical in ensuring that the dataset contained high-quality ground-truth labels, enabling the object detection model to learn accurate feature representations of the selected traffic signs.

An example of how the images are labelled can be seen in Figure 4.3

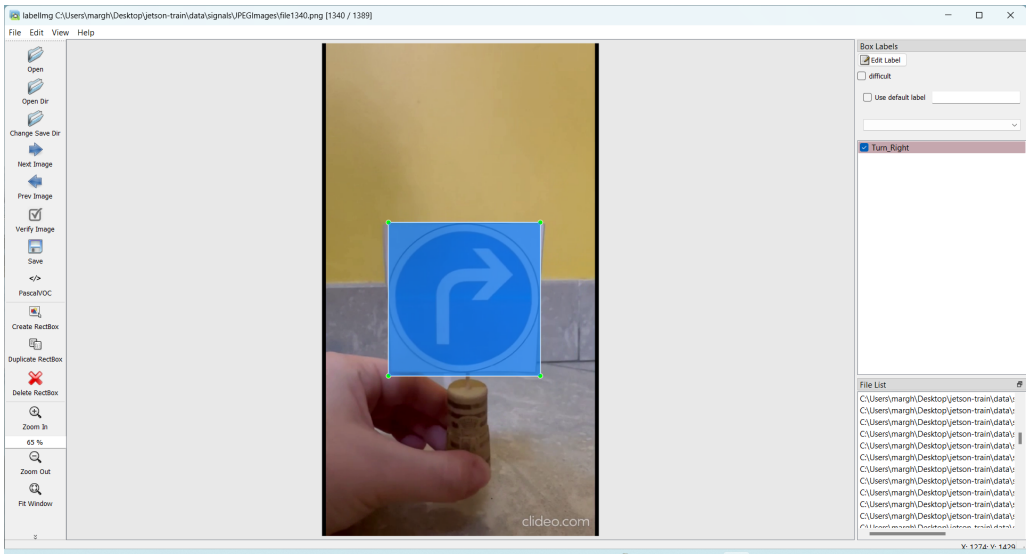


Figure 4.3: LabelImg utilization

Each annotated image corresponds to an individual XML file (an example is shown in Figure 4.4) containing detailed information about the objects present in the image. The format of the XML file is designed to be machine-readable and provides essential metadata for each object, including:

- **File Information**: The XML file begins with general information about the image, such as its file name and the size of the image (height and width). This ensures that the model knows the exact dimensions of the image when it is being processed.

- **Object Details**: For each traffic sign or object in the image, there is a corresponding `<object>` tag that contains specific information about that object. This includes:
  - **Name**: The class label (in this case, the traffic sign type) for the object.
  - **Bounding Box Coordinates**: The coordinates of the bounding box that surrounds the object, expressed in terms of the object top-left and bottom-right corners (`xmin`, `ymin`, `xmax`, `ymax`). These coordinates define the region in the image that contains the object.
- **Additional Metadata**: Some annotation files may also contain extra information, such as the pose or the truncation level of an object. However, in this specific case, only the class label and the bounding box coordinates are typically required.

```
<annotation>
    <folder>JPEGImages</folder>
    <filename>file1.png</filename>
    <path>C:\Users\margh\Desktop\jetson-train\data\signals\JPEGImages\file1.png</path>
    <source>
        <database>Unknown</database>
    </source>
    <size>
        <width>800</width>
        <height>1422</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>Go_Straight</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>251</xmin>
            <ymin>507</ymin>
            <xmax>533</xmax>
            <ymax>770</ymax>
        </bndbox>
    </object>
</annotation>
```

Figure 4.4: Pascal VOC format XML file

After labelling all, a file named `labels.txt` is created within the `data/signals` folder. This file contains the names of all the labels used in the dataset:

```
Stop
Turn_Right
Turn_Left
Go_Straight
```

4.2 Training the SSD MobileNetV2 Model

Once the dataset preparation is complete, the training process can commence using the following command, as illustrated in Figure 4.5:

```
python train_ssd.py --dataset-type=voc --data=data/signals/ \
--model-dir=models/signals --resolution=300 --batch-size=4 \
--workers=0 --epochs=500 --validation-mean-ap 1 --use-cuda False
```

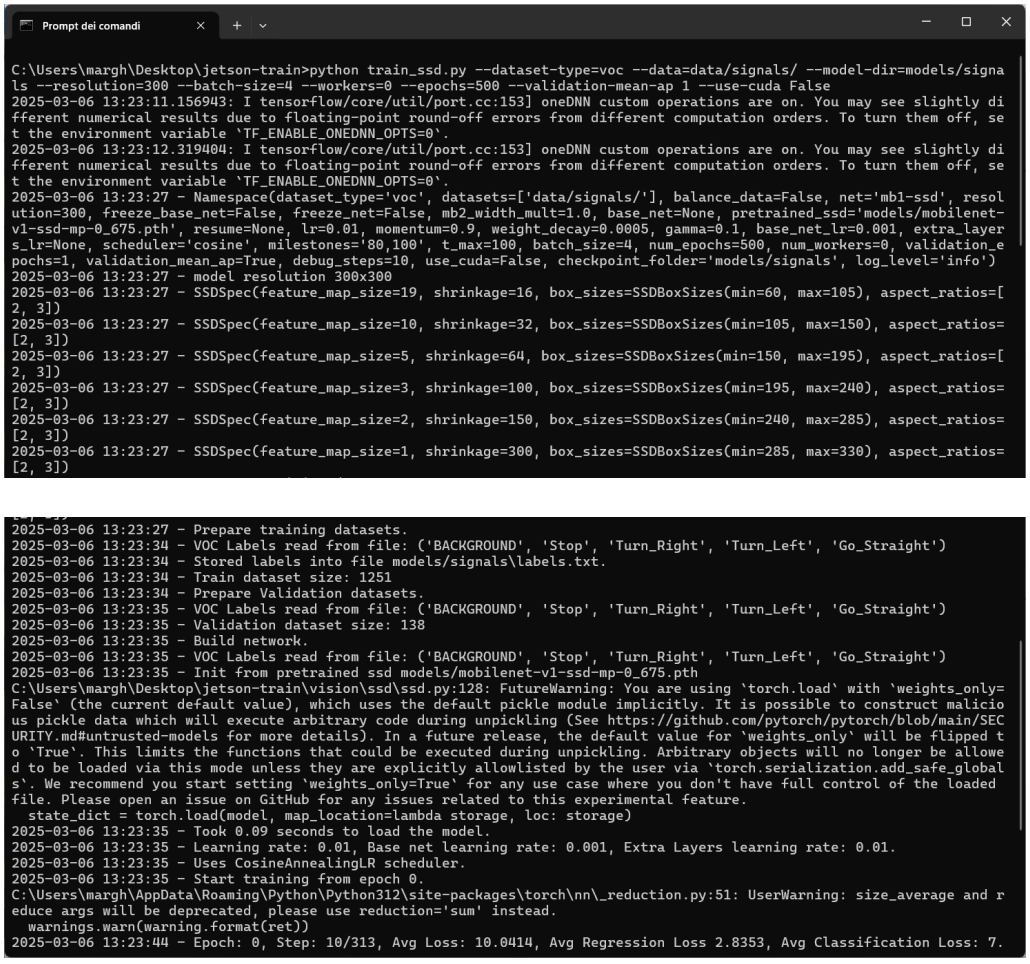


Figure 4.5: Training phase

The parameters used in the training command are detailed as follows:

- `--dataset-type=voc`: Specifies the dataset format. The VOC format refers to the PASCAL VOC dataset structure, which is commonly used for object detection tasks.

- `--data=data/signals/`: Defines the path to the dataset directory containing training and validation images along with their annotations.
- `--model-dir=models/signals`: Specifies the directory where the trained model checkpoints and related files will be stored.
- `--resolution=300`: Defines the input image resolution for the model. A value of 300 indicates that images will be resized to 300×300 pixels before being processed.
- `--batch-size=4`: Sets the number of images processed in a single training iteration. A batch size of 4 is chosen to balance memory constraints and training stability.
- `--workers=0`: Specifies the number of worker threads used for data loading. A value of 0 means that data loading is handled by the main process, which can be beneficial for resource-constrained hardware.
- `--epochs=500`: Defines the total number of training epochs. An epoch represents one complete pass through the training dataset.
- `--validation-mean-ap=1`: Enables the computation of the Mean Average Precision (mAP) on the validation dataset. The mAP metric is a standard evaluation measure for object detection models.
- `--use-cuda=False`: Determines whether to utilise CUDA for GPU acceleration. Setting this to `False` forces training to run on the CPU, which may be necessary for environments with limited GPU resources.

At regular intervals, the model is saved in a checkpoint that includes the state dictionary of the model (weights), as well as epoch and loss information to track training progress. This enables the model to be resumed or tested at a later stage. In addition, the checkpoint facilitates systematic evaluation and comparison of different training stages. Since the loss function serves as a key metric in assessing the performance of the model, each checkpoint is associated with a specific loss value recorded at the time of saving.

At the end of the training process, the best-performing model is selected based on the checkpoint that exhibits the lowest loss value. This approach ensures that the final deployed model is the most optimised version, having achieved the best balance between under-fitting and over-fitting during training. Using this checkpoint strategy, not only training can be made more robust against potential failures, but it also allows for fine-grained model selection based on empirical performance metrics.

4.3 Evaluation Metrics

In the field of object detection, particularly in critical applications such as traffic sign recognition, it is essential to assess the performance of the model using metrics that effectively summarise both the accuracy of object localisation and the reliability of classification. The evaluation of an object detection model goes beyond simple accuracy measurements, as it involves determining how well the model detects objects, how precise the bounding box predictions are, and how confidently the model classifies each object. The key metrics used in this study are discussed below, along with their significance and expected values.

4.3.1 Loss Function

The SSD model employs the MultiboxLoss function, which is specifically designed for Single-Shot object detection. The loss function consists of two key components:

- **Classification Loss**: Measures how accurately the model predicts the class of each detected object. It is computed using a cross-entropy loss function, which measures the divergence between the predicted class probabilities and the actual class labels. A lower classification loss indicates better performance in distinguishing different object categories. Figure 4.6 provides a visual representation of this loss function. The horizontal axis represents the number of epochs, while the vertical axis indicates the classification loss value. In this context, an **epoch** refers to a complete pass through the entire training dataset by the model. During each epoch, the model processes all available training samples and updates its parameters accordingly. Multiple epochs are typically required for the model to learn meaningful patterns and improve its performance. The downward trend in the curve suggests that the model is progressively improving in distinguishing object classes as training proceeds.

- **Regression Loss**: Quantifies the accuracy of the bounding box predictions, assessing how well the predicted bounding boxes align with the ground truth. It is typically computed using a smooth L1 loss function, which penalises large deviations while remaining less sensitive to minor errors to ensure stability during training. Figure 4.7 illustrates the regression loss over time. The x-axis represents the number of epochs, and the y-axis denotes the loss value. A decreasing regression loss suggests that the model is becoming more precise in localising objects within the image.

The classification and regression losses are computed separately and then combined with their respective weights to form the total loss, which is minimized during training. So, the total loss represents the overall discrepancy between the predictions of the model (both class labels and bounding box coordinates) and the true values. In Figure 4.8, after 500 epochs, the loss decreases as training progresses,

reflecting that both the classification and regression errors are being reduced. The curve exhibits a characteristic pattern often observed in deep learning models. Initially, the loss is significantly high due to the random initialisation of the network weights. As training progresses, the loss rapidly decreases, indicating that the model is learning meaningful features from the dataset. However, after the initial rapid decline, the loss stabilises and presents minor oscillations. These oscillations indicate that the performance of the model varies across different batches. This behaviour could be attributed to variations in the dataset, where certain samples are more challenging for the model. The presence of these fluctuations suggests the necessity of fine-tuning hyperparameters, such as the learning rate, batch size, and data augmentation strategies, to enhance model stability.

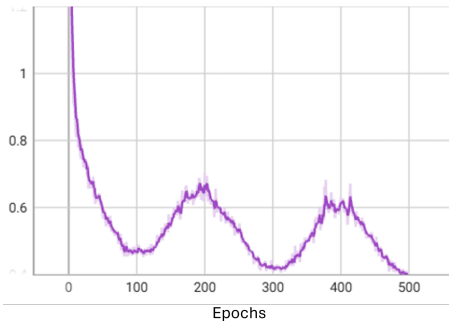


Figure 4.6: Classification Loss

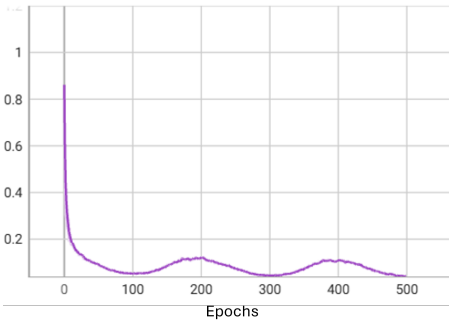


Figure 4.7: Regression Loss

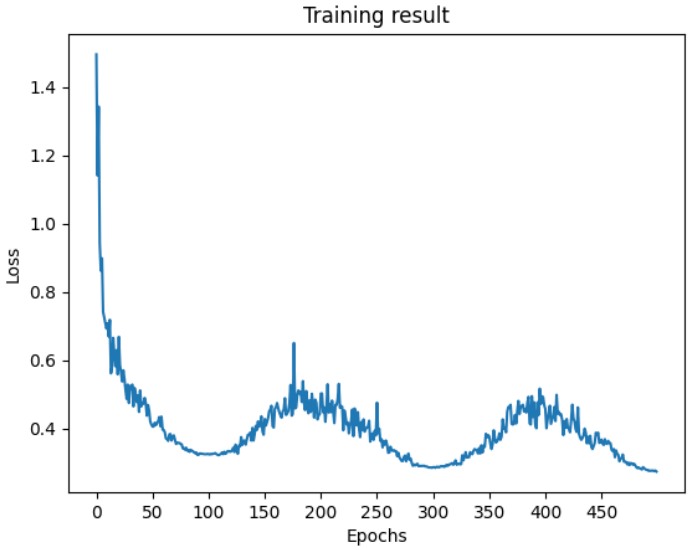


Figure 4.8: Training Loss

For example, the Stochastic Gradient Descent (SGD) optimiser was used to minimise the loss. This optimiser is well-suited for object detection tasks as it helps improve convergence and stability. The primary hyperparameters that affect the learning process include:

- **Learning rate (lr)**: Determines the step size at each iteration while moving towards a minimum of the loss function.
- **Momentum**: Helps accelerate gradient descent in relevant directions and dampens oscillations.
- **Weight decay**: Regularises the model to prevent over-fitting by penalising large weights.

4.3.2 Mean Average Precision

The **Mean Average Precision (mAP)** is one of the most widely used metrics for evaluating object detection models. It measures the overall accuracy of object detection across all classes. The mAP is computed as follows:

1. **Precision**: Defined as the ratio of correctly detected objects (true positives) to the total number of predicted objects (true positives + false positives). Precision indicates how many of the detections of the model are correct.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.1}$$

2. **Recall**: Defined as the ratio of correctly detected objects to the total number of actual objects present in the dataset (true positives + false negatives). Recall represents the ability of the model to detect all relevant objects.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.2}$$

3. **Precision-Recall Curve**: By varying the confidence threshold for detections, a precision-recall curve is generated, showing the trade-off between precision and recall at different thresholds.
4. **Average Precision (AP)**: The area under the precision recall curve for each class. A higher AP indicates better performance in detecting objects of a specific class.
5. **mAP**: The mean of the AP values across all object classes, providing a single summarised metric for overall model performance.

$$\text{mAP} = \frac{1}{N} \sum_{i=1}^{N} AP_i \tag{4.3}$$

where N is the total number of classes.

A higher mAP trend, illustrated in Figure 4.9, indicates that the model demonstrates strong performance in accurately detecting and classifying traffic signs. The dark purple line indicates the mean value of the mAP, illustrating how the accuracy of the model evolves over the training epochs. By contrast, the light pink line likely represents fluctuations or individual precision measurements for specific batches, highlighting transient performance variations during training.

At the beginning of the training process, the mAP increases rapidly, suggesting that the model is effectively learning to recognise traffic signs. However, there are sudden drops in mAP, which may be attributed to variations in the data distribution or batch size effects. Despite these fluctuations, the mAP remains close to 1, indicating a robust adaptation of the model.

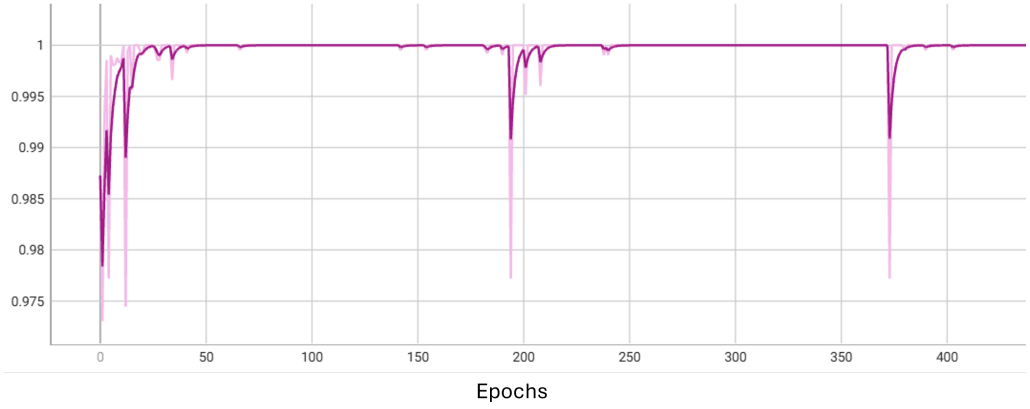
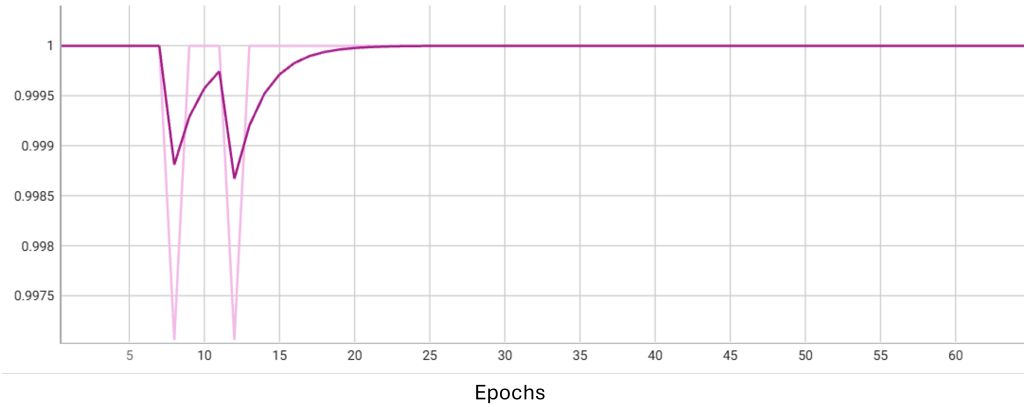
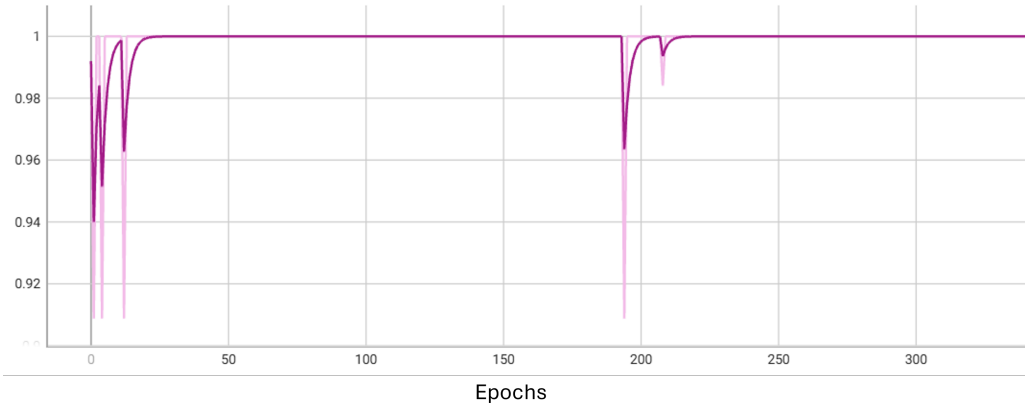


Figure 4.9: mAP

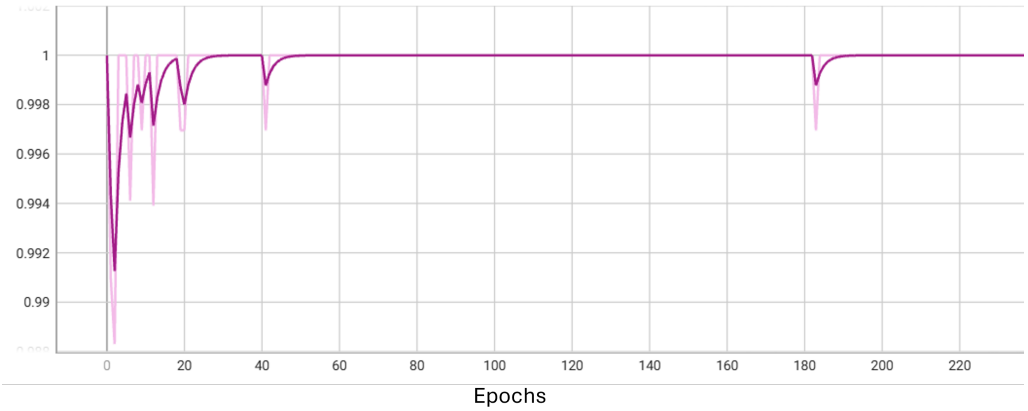
Nonetheless, it is essential to analyse class-wise AP values to ensure that the model does not exhibit disproportionately high performance on certain classes while underperforming on others. The graphs corresponding to the four classes are presented in Figure 4.10.



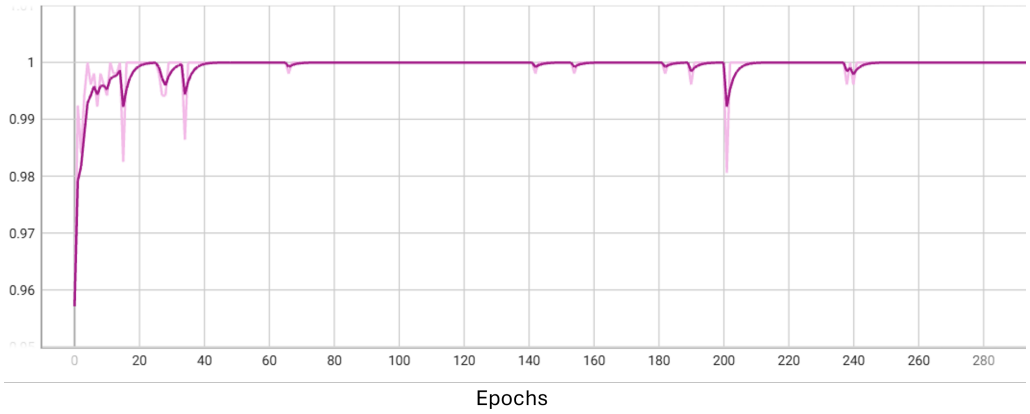
(a) Go Straight



(b) Stop



(c) Turn Left



(d) Turn Right

Figure 4.10: Average Precision for each traffic sign class

The AP remains close to 1 for most of the training process, suggesting that the model is highly effective at recognising all traffic signs. However, intermittent sharp declines to lower values are observed from variations in the dataset, changes in lighting conditions, occlusions, or similarities between certain traffic signs. The consistently high AP values across all four classes suggest that the model has successfully learned to generalise across different traffic sign types.

By evaluating these metrics, it is possible to refine the training process, adjust hyperparameters, and improve the dataset to achieve better real-time performance in traffic sign recognition.

4.4 Inference and Testing on Jetson Nano

4.4.1 Exporting the Model to ONNX Format

The trained model, originally stored in the `.pt` format, was exported to the `.onnx` format to facilitate deployment on the Jetson Nano. The ONNX (Open Neural Network Exchange) format enables interoperability between different deep learning frameworks and optimizes inference performance when used with TensorRT, the high-performance deep learning inference SDK developed by NVIDIA. The model conversion was performed using the following command:

```
python onnx_export.py --net ssd-mobilenet --input best_model.pth \
--output best_model.onnx --labels labels.txt --width 300 --height 300
```

4.4.2 Inference Execution

After exporting the model to ONNX format, inference was conducted using the `detectnet` utility, shown in Figure 4.11:

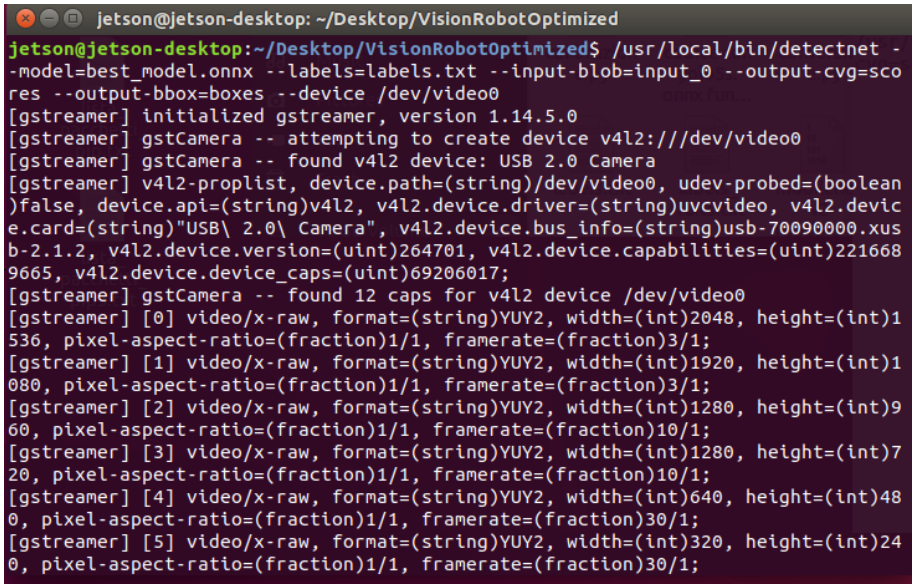
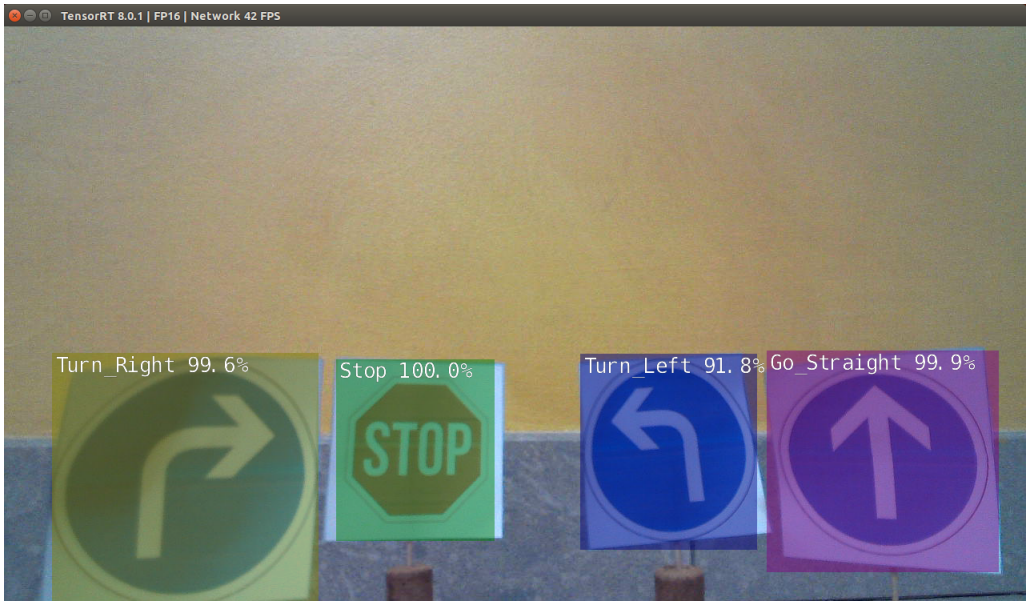


Figure 4.11: Inference on Jetson Nano

The parameters used in the command are as follows:

- `--model`: Specifies the path to the ONNX model file.
- `--labels`: Defines the label file associated with the model.
- `--input-blob`: Names the input tensor of the model.
- `--output-cvg`: Defines the output tensor for confidence scores.
- `--output-bbox`: Specifies the output tensor for bounding box coordinates.
- `--device`: Identifies the video input source (e.g., a camera at `/dev/video0`).

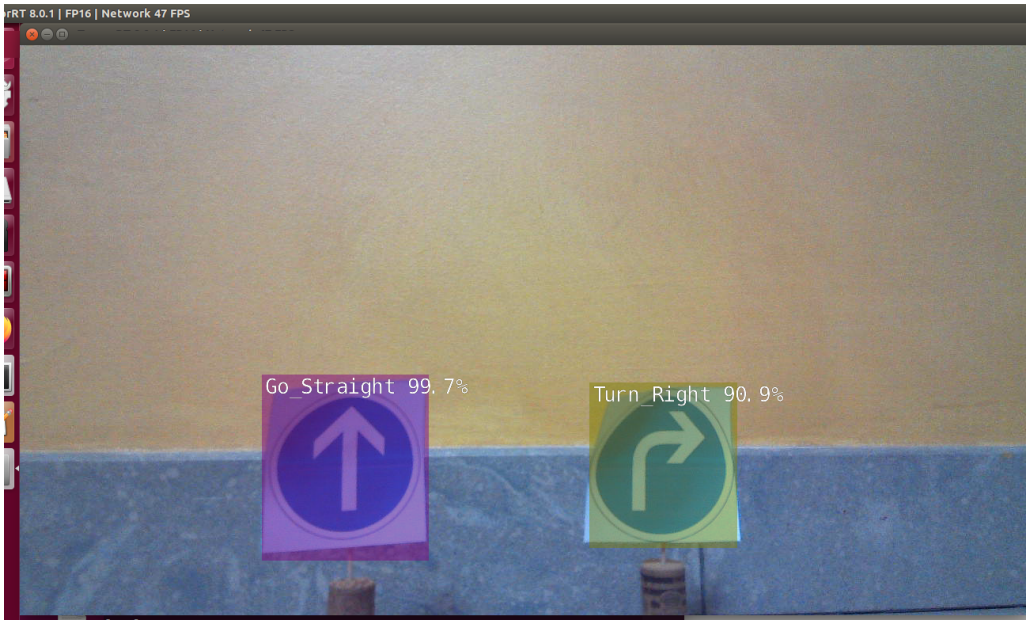
The objects recognised along with their corresponding confidence scores are presented in Figure 4.12. The confidence score is a numerical value, ranging from 0 to 100%, indicating the probability that the predicted object belongs to a given class. In this context, confidence scores reflect how confidently the model identifies traffic signs within the given input images, with higher values indicating stronger certainty in the detection.


TensorRT 8.0.1 | FP16 | Network 42 FPS
Turn_Right 99.6%
Stop 100.0%
STOP
Turn_Left 91.8%
Go_Straight 99.9%

(a)


TensorRT 8.0.1 | FP16 | Network 32 FPS
Go_Straight 99.8%
Stop 99.9%
STOP

(b)



(c)

Figure 4.12: Object detections

It is noteworthy that the confidence scores for most traffic signs are above 99%. However, for the "Turn Left" and "Turn Right" signs, the scores are slightly lower, although still above 90%. This small discrepancy might seem to come due to greater diversity in their look - caused by variances such as differing viewpoints, occlusions or diagens. Nevertheless, a confidence level above 90% shows robust performance and indicates that the model is reliable in detecting these signs.

4.4.3 Inference Latency and FPS Evaluation

For real-time applications, such as Advanced Driver Assistance Systems (ADAS), evaluating inference time and FPS is critical. These metrics significantly impact system responsiveness. The inference time for each frame can be measured using the profiling option:

```
/usr/local/bin/detectnet --model=best_model.onnx \
--labels=labels.txt --input-blob=input_0 \
--output-cvg=scores --output-bbox=boxes \
--device /dev/video0 --profile
```

This command provides execution times for preprocessing, inference, and post-processing stages (Figure 4.13).

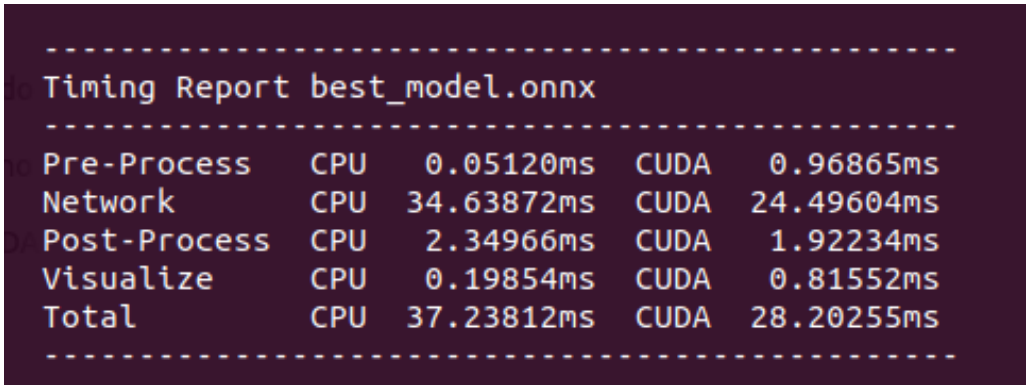


Figure 4.13: Inference time

The timing report for `best_model.onnx` shows what is the time needed for what in the processing. The total inference time on the CPU is **37.23812 ms**, however, it is much lower at **28.20255 ms** for the CPU (CUDA). This suggests that offloading computations to the GPU improves performance.

- **Pre-processing**: The GPU takes approximately **0.97 ms**, which is considerably higher than the CPU **0.05 ms**. However, this difference is relatively small in the overall computation.
- **Network inference**: This is longest operation. The CPU takes **34.64 ms** while the GPU speeds it up to **24.49 ms**, which is considerable.
- **Post-processing**: The GPU takes **1.92 ms** compared to the CPU **2.35 ms**, showing a moderate improvement.
- **Visualisation**: GPU performance (**0.82 ms**) is better than the CPU (**0.20 ms**), but this step is not a major contributor to the total time.

Overall, using CUDA decreases the whole processing time by about **25%**, therefore the GPU acceleration is profitable in this model. However, further optimisations, such as model pruning or quantisation, could enhance efficiency further, especially given the resource constraints of the Jetson Nano.

4.4.4 Benchmarking FPS using `trtexec`

To evaluate the frames per second (FPS) performance of a given model, the `trtexec` benchmarking tool in TensorRT can be employed. This tool facilitates the execution and performance measurement of TensorRT engines, providing valuable insights into the inference speed and efficiency of the model under various configurations.

The command used to run the benchmark is as follows:

```
/usr/src/tensorrt/bin/trtexec \
--loadEngine=best_model.onnx.1.1.8001.GPU.FP16.engine \
--batch=1
```

This command loads the pre-compiled TensorRT engine and performs inference with a batch size of one, which is typical for real-time applications where the model processes individual frames in sequence.

The output generated by the tool contains several performance metrics, including the inference time per batch and the corresponding FPS. The relevant section of the output, which provides the inference performance data, is shown in Figure 4.14.

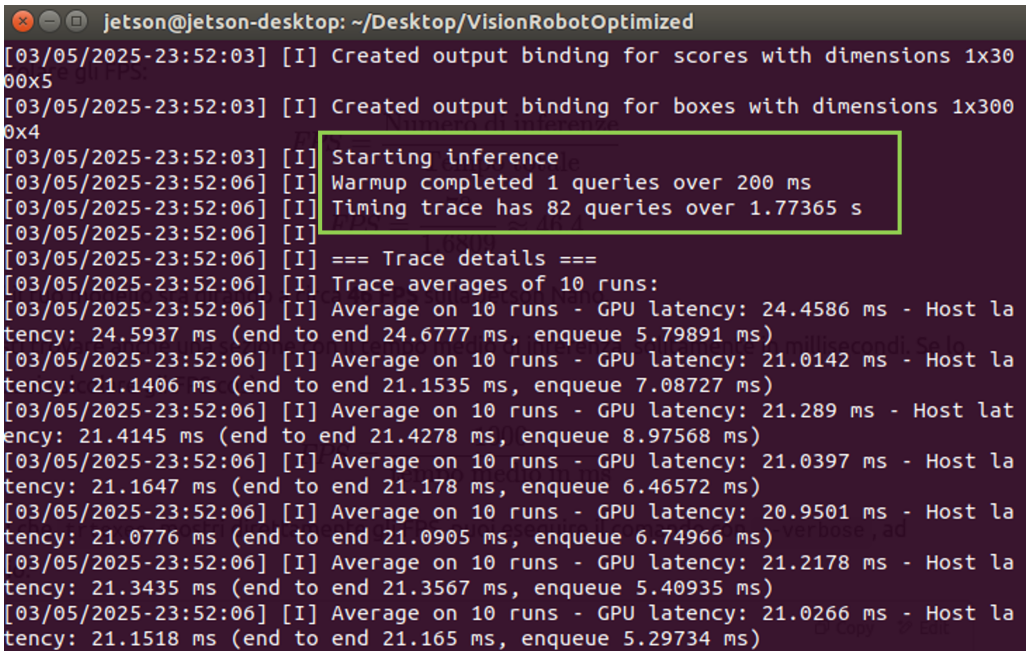


Figure 4.14: Inference performance metrics.

From this output, FPS can be calculated by dividing the number of iterations by the total time taken for inference. Specifically, in this case, the FPS is computed using the following formula:

$$FPS = \frac{82}{1.77365} \approx 46.23 \tag{4.4}$$

In this calculation, 82 represents the number of frames processed, and 1.77365 seconds is the total time taken for the inference process. The resulting FPS value indicates that the model achieves a frame rate of 46.23 frames per second on the Jetson Nano.

This performance demonstrates that the model is capable of real-time processing, making it well-suited for applications requiring high throughput, such as real-time object detection.

Chapter 5

Software Implementation

This chapter presents the code development process for the autonomous driving system. The process begins with the initial configuration of the environment, ensuring that all required dependencies and settings are properly established. It then details the implementation of a Python-based vision script and concludes with the adaptation of the ROS node to support autonomous driving, incorporating MQTT communication, a feature that was partially developed in the previous master's thesis [1].

5.1 System Architecture

The system has been developed within a structured architectural framework to ensure efficient communication and seamless processing between multiple components. Each component is responsible for distinct tasks that contribute to the autonomous operation of the robot. The architecture revolves around two primary elements: a Python-based vision script running on the Jetson Nano and a ROS2 node encapsulated within a Docker container. The communication between these components is handled via the MQTT protocol, ensuring reliable message exchange.

The main functionalities of the system can be divided into two stages:

- **Python Script:** Initially, the vision script focuses on detecting and tracking a coloured line, enabling the robot to follow a predefined path. This functionality is implemented using OpenCV-based image processing techniques. Subsequently, the system is extended to incorporate traffic sign recognition through a machine learning model capable of detecting specific road signs. The extracted information, including the coordinates of the boundary box and the classification results, is then used to generate high-level motion commands. These commands are transmitted via MQTT to the ROS2 node, which handles execution.

- **ROS2 Node within a Docker Container:** The ROS2 node, running inside a Docker container, is responsible for interpreting the commands received from the vision script and translating them into precise movement instructions. It achieves this by integrating data from multiple sensors, including the onboard camera and LiDAR, allowing for real-time obstacle detection and autonomous navigation. The node ensures that the robot reacts appropriately to the detected road signs, executing manoeuvres such as turning or stopping as required. Additionally, it facilitates low-level motion control by communicating with the STM32 microcontroller (not covered in this thesis) via dedicated Python libraries.

This architecture, illustrated in Figure 5.1, separates the vision and motion control tasks while ensuring smooth interaction between them through MQTT communication. The modular design allows for future extensions, such as enhanced navigation algorithms or additional sensor integrations.

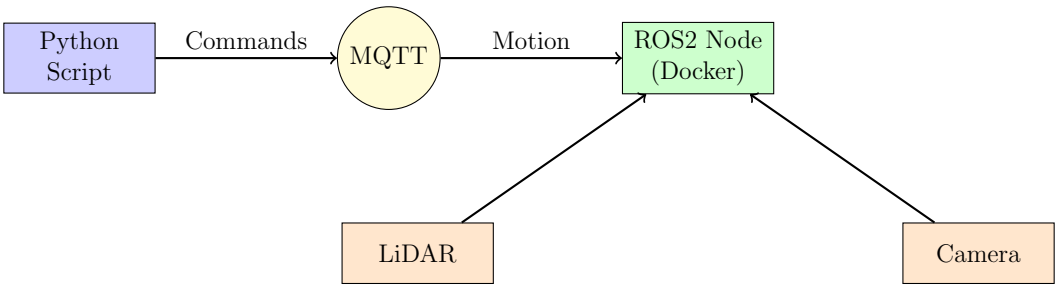


Figure 5.1: System architecture

5.2 Python Script

This Python script is part of a broader project initially developed in the Master's thesis Deep Learning-Based Real-Time Detection and Object Tracking on an Autonomous Rover with GPU-based Embedded Device [35]. It was later extended in a subsequent thesis focusing on the development of the Follow Me functionality for target person tracking [1]. The programme is structured around a graphical menu that provides various features and has undergone significant enhancements. In particular, the interface has been completely overhauled to deliver a more visually appealing and intuitive experience, thereby facilitating smoother navigation. Additionally, improvements in window management ensure that all menu windows are closed gracefully when the programme terminates, preventing resource leaks and ensuring a clean exit. These enhancements not only contribute to an improved aesthetic but also bolster the robustness and reliability of the overall system.

This thesis specifically builds upon this work by improving the Follow Line functionality. The initial focus of the code was the implementation of vision-based line tracking. At a later stage, traffic sign recognition was integrated using the pre-trained **SSD MobileNet V2** neural network discussed in Chapter 4.

5.2.1 Line Following

In the context of computer vision systems for autonomous driving, the correct segmentation of the line to be followed is crucial. The presented module is based on two main components:

- **Colour Mask Calibration**: Allows the user to interactively set the threshold parameters in the Hue Saturation Value (HSV) colour model to isolate the line (assumed to be green in this case) from the background.
- **Follow Line Core**: The main function (`Follow_Line_Core`) that, after acquiring the calibrated threshold parameters, executes the autonomous line tracking procedure.

Mask Calibration

The `maskCalibration()` function (Listing 5.1) provides an interactive graphical interface using trackbars, allowing the user to adjust the HSV parameters in real time for line segmentation. The video stream is initialised using OpenCV, in particular `cv2.VideoCapture(0)`, enabling real-time video capture from the webcam. By utilising `cv2.namedWindow("Trackbars")` and `cv2.createTrackbar()`, a window is created containing six sliders, each dedicated to adjusting the lower and upper threshold values for each HSV channel. Default values are set for a green-coloured line. The frame is then converted from BGR to HSV using `cv2.cvtColor()`, as the HSV colour space is more robust for analysis under varying lighting conditions. The `cv2.inRange()` function creates a binary mask that isolates pixels within the defined minimum and maximum values, highlighting the area of interest and a `while` loop continuously updates the frame and mask, allowing the user to observe the effect of adjustments in real time. The loop terminates when the `'q'` key is pressed.

Listing 5.1: Mask calibration function

```
1  def maskCalibration():
2
3      # Start video stream from the webcam
4      cap = cv2.VideoCapture(0)
5
6      # Create the window for sliders
7      cv2.namedWindow("Trackbars")
8
9      # Set default values for the green line
10     default_low_b = np.uint8([38, 71, 125])
```

```
11      default_high_b = np.uint8([77, 255, 255])
12
13      # Create sliders for Hue, Saturation, and Value channels
14      cv2.createTrackbar("Low H", "Trackbars", default_low_b[0],
            179, nothing)
15      cv2.createTrackbar("High H", "Trackbars", default_high_b[0],
            179, nothing)
16      cv2.createTrackbar("Low S", "Trackbars", default_low_b[1],
            255, nothing)
17      cv2.createTrackbar("High S", "Trackbars", default_high_b[1],
            255, nothing)
18      cv2.createTrackbar("Low V", "Trackbars", default_low_b[2],
            255, nothing)
19      cv2.createTrackbar("High V", "Trackbars", default_high_b[2],
            255, nothing)
20
21      # Initialise sliders with default values
22      cv2.setTrackbarPos("Low H", "Trackbars", default_low_b[0])
23      cv2.setTrackbarPos("High H", "Trackbars", default_high_b[0])
24      cv2.setTrackbarPos("Low S", "Trackbars", default_low_b[1])
25      cv2.setTrackbarPos("High S", "Trackbars", default_high_b[1])
26      cv2.setTrackbarPos("Low V", "Trackbars", default_low_b[2])
27      cv2.setTrackbarPos("High V", "Trackbars", default_high_b[2])
28
29      while True:
30          ret, frame = cap.read()
31          if not ret:
32              break
33
34          # Convert the frame from BGR to HSV colour space
35          hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
36
37          # Read the current slider values
38          low_h = cv2.getTrackbarPos("Low H", "Trackbars")
39          high_h = cv2.getTrackbarPos("High H", "Trackbars")
40          low_s = cv2.getTrackbarPos("Low S", "Trackbars")
41          high_s = cv2.getTrackbarPos("High S", "Trackbars")
42          low_v = cv2.getTrackbarPos("Low V", "Trackbars")
43          high_v = cv2.getTrackbarPos("High V", "Trackbars")
44
45          # Define arrays for the HSV range
46          low_b = np.uint8([low_h, low_s, low_v])
47          high_b = np.uint8([high_h, high_s, high_v])
48
49          # Create the mask to isolate the line
50          mask = cv2.inRange(hsv_frame, low_b, high_b)
51
52          # Display the mask and the original frame
53          cv2.imshow("Mask", mask)
54          cv2.imshow("Frame", frame)
55
```

```
56          # Exit loop when 'q' key is pressed
57          if cv2.waitKey(1) & 0xFF == ord('q'):
58              break
59
60      cap.release()
61      cv2.destroyAllWindows()
62
63      # Check if the user modified the values from the default
            settings
64      if (low_b == default_low_b).all() and (high_b ==
            default_high_b).all():
65          print("No changes detected, using default green line
                values")
66          return default_low_b, default_high_b
67
68      return low_b, high_b
69
70  def nothing(x):
71      # Empty callback function for sliders.
72      pass
```

Follow Line

The function `Follow_Line()` (Listing 5.2) integrates the calibration phase with the tracking algorithm. With the parameters obtained, the function `Follow_Line_Core` is invoked, which manages the autonomous line tracking algorithm by sending commands to the control system. The use of a `try/except` block allows the handling of any interruptions (e.g. via `KeyboardInterrupt`) and the activation of a clean-up routine (`exit_handler`), thus guaranteeing a safe program stop.

Listing 5.2: Follow Line function

```
1   def Follow_Line():
2       print("Starting mask calibration before Follow Line...")
3
4       # Start calibration to obtain updated HSV parameters
5       low_b, high_b = maskCalibration()
6
7       print(f"Using HSV range: Low={low_b}, High={high_b}")
8       try:
9           # Start line tracking with the calibrated parameters and
                control functions
10          return Follow_Line_Core(send_activate, send_command,
                send_stop, truncate, low_b, high_b)
11      except KeyboardInterrupt:
12          # Handle keyboard interruption
13          exit_handler(None, None)
```

Follow Line Core

The function `Follow_Line_Core` (Listing 5.3) implements the actual control algorithm for tracking the visible line in the video stream.

Specifically, the script starts by opening the camera using `cv2.VideoCapture(0)` and configuring the image resolution to improve computational handling. Each frame captured by the camera is converted from the BGR color space to HSV using `cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)`. The HSV color space is more robust for object detection under varying lighting conditions. At this point, the function `cv2.inRange(hsv_frame, low_b, high_b)` creates a binary mask that isolates the line based on the HSV color values defined by the arguments `low_b` and `high_b`, already set during calibration.

Using `cv2.findContours`, the contours in the binary mask are identified, and the largest contour is selected, which is assumed to represent the line to follow. The moments of the contour (`cv2.moments`) are then used to calculate the centroid of the contour, providing the horizontal (X) position of the line. The position of the centroid is compared with the center of the image to calculate the horizontal error (`error_x`).

Depending on the horizontal error, the robot will perform one of the following actions, sending the correct command through the MQTT protocol:

- Move forward if the line is centered
- Turn left if the line is to the left of the center
- Turn right if the line is to the right of the center

The maximum steering angle (`max_steering_angle`) is set to 30° to prevent sharp turns.

Listing 5.3: Follow_Line_Core function

```
def Follow_Line_Core(send_activate, send_command, send_stop,
    truncate, low_b, high_b):

    # Initialize video stream
    cap = cv2.VideoCapture(0)
    cap.set(3,160)  # Set image width
    cap.set(4,120)  # Set image height

    # Define the maximum steering angle to prevent excessive
        movement
    max_steering_angle = 30  # Limits the maximum steering angle

    try:
        # Main loop to read the video stream and track the line
        while True:
            ret, frame = cap.read()
```

```
15              hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)  #
                    Convert to HSV color space
16
17              # Create the mask to detect the line
18              mask = cv2.inRange(hsv_frame, low_b, high_b)
19
20              # Find contours in the mask
21              contours = cv2.findContours(mask, cv2.RETR_EXTERNAL,
                    cv2.CHAIN_APPROX_SIMPLE)[-2]
22              if len(contours) > 0:
23                  c = max(contours, key=cv2.contourArea)  # Select
                        the largest contour (presumably the line)
24                  M = cv2.moments(c)
25
26                  if M["m00"] != 0:  # Check if the contour has a
                        non-zero area
27                      cx = int(M["m10"] / M["m00"])  # Calculate the
                            contour centroid in X
28                      cy = int(M["m01"] / M["m00"])  # Calculate the
                            contour centroid in Y
29
30                      # Calculate the horizontal error with respect
                            to the image center
31                      image_center_x = frame.shape[1] / 2
32                      error_x = cx - image_center_x
33
34                      # Calculate the steering angle based on the
                            horizontal error
35                      steering_angle = (error_x / image_center_x) *
                            max_steering_angle
36
37                      # Based on the horizontal error, send commands
                            to move the robot
38                      if abs(error_x) < 10:  # Line is centered
39                          print("On track forward!")
40                          send_activate()  # Activate movement
41                          send_command("forward")  # Move forward
42                      elif error_x < -10:  # Line is to the left
43                          print("Turning left!")
44                          send_activate()  # Activate movement
45                          send_command("left", int(-1 *
                                steering_angle))  # Turn left
46                      elif error_x > 10:  # Line is to the right
47                          print("Turning right!")
48                          send_activate()  # Activate movement
49                          send_command("right", int(steering_angle))
                                # Turn right
50
51                      # Draw the centroid on the frame to visualize
                            the position of the line
```

```
52                      cv2.circle(frame, (cx, cy), 5, (255, 255,
                            255), -1)
53
54                  # Draw the contour of the line
55                  cv2.drawContours(frame, [c], -1, (0, 255, 0), 1)
56
57              # Show the mask and the original frame
58              cv2.imshow("Mask", mask)
59              cv2.imshow("Frame", frame)
60
61              # Exit if the user presses 'q'
62              if cv2.waitKey(1) & 0xFF == ord('q'):
63                  break
64
65      finally:
66          # Release the camera and close the windows
67          cap.release()
68          cv2.destroyAllWindows()
69
70      return
```

5.2.2 Integration of traffic signals

In the new function `Follow_Road_Core`, the script loads and integrates the pre-trained neural network for traffic sign recognition, including *"Stop"*, *"Go Straight"*, *"Turn Left"*, and *"Turn Right"*, modifying the behaviour of the rover.

The main integrations (Listing 5.4) consist of the initialisation of the object detection model, using the `best_model.onnx` model, with a detection threshold of 90% to avoid false positives. The `threshold` parameter sets the confidence level required for the detection to be considered valid. The `labels.txt` file contains the labels for the traffic signs.

The robot, as before when only tracking the line, uses the webcam to continuously capture frames, which are now processed to detect traffic signs. For each captured frame, the function `detection(frame, net, truncate)` is called, which processes the frame and returns a list of detected objects (traffic signs). If a sign has not been detected recently or the cooldown time has passed, the system will handle the sign accordingly.

Listing 5.4: Follow_Road_Core function

```
1  def Follow_Road_Core(send_activate, send_command, send_stop,
       truncate, low_b, high_b):
2
3      print("Follow Road started")
4
```

```
5       net = jetson_inference.detectNet("best_model.onnx",
            threshold=0.85, input_blob="input_0", output_cvg="scores",
            output_bbox="boxes", labels="labels.txt")
6
7       cap = cv2.VideoCapture(0)
8
9       max_steering_angle = 30
10      stop_detected = False
11      stop_start_time = 0
12      stop_cooldown = 15
13
14      last_detection = None  # Last detected sign
15      detection_cooldown = 2  # Minimum time between two detections
            of the same sign
16      last_detection_time = 0  # Timestamp of the last detection
17
18      try:
19          while True:
20              ret, frame = cap.read()
21              if not ret:
22                  continue
23
24              detections = detection(frame, net, truncate)
25
26              # Traffic sign handling
27              if not stop_detected and detections:
28                  for detect in detections:
29                      item = net.GetClassDesc(detect.ClassID).strip()
30
31                      # If the sign is new or at least 2 seconds
                            have passed, process it
32                      if item != last_detection or (time.time() -
                            last_detection_time > detection_cooldown):
33
34                          if item == "Stop":
35                              print("Stop Sign Detected!")
36                              send_stop()
37                              stop_detected = True
38                              stop_start_time = time.time()
39                              last_detection = "Stop"
40                              last_detection_time = time.time()
41                              break
42
43                          elif item == "Go_Straight":
44                              print("Go_Straight Detected!")
45                              send_activate()
46                              send_command("forward")
47                              last_detection = "Go_Straight"
48                              last_detection_time = time.time()
49                              break
50
```

```
                        elif item == "Turn_Left":
                            print("Turn_Left Detected!")
                            send_activate()
                            send_command("left",
                                max_steering_angle)
                            last_detection = "Turn_Left"
                            last_detection_time = time.time()
                            break

                        elif item == "Turn_Right":
                            print("Turn_Right Detected!")
                            send_activate()
                            send_command("right",
                                max_steering_angle)
                            last_detection = "Turn_Right"
                            last_detection_time = time.time()
                            break

            # Cooldown for stop sign
            if stop_detected:
                if time.time() - stop_start_time < stop_cooldown:
                    cv2.imshow("Frame", frame)
                    if cv2.waitKey(1) & 0xff == ord('q'):
                        break
                    continue
                else:
                    print("Stop cooldown finished, resuming line
                        tracking.")
                    stop_detected = False
                    send_activate()
                    ....
```

The `detection` (Listing 5.5) function is responsible for detecting traffic signs within the frame. For each detected sign, a rectangle is drawn around the area of interest, and the name of the sign along with its confidence score is displayed.

Listing 5.5: Detection function

```

def detection(frame, net, truncate, ct):

    height = frame.shape[0]
    width = frame.shape[1]

    frame_color = cv2.cvtColor(frame,
        cv2.COLOR_BGR2RGBA).astype(np.float32)
    frame_cuda = jetson_utils.cudaFromNumpy(frame_color)

    detections = net.Detect(frame_cuda, width, height)

    matching_detections = []
```

```
13      rects = []
14      all_objects = []
15
16      for detect in detections:
17
18          ID = detect.ClassID
19          confidence = detect.Confidence
20          top = int(detect.Top)
21          left = int(detect.Left)
22          bottom = int(detect.Bottom)
23          right = int(detect.Right)
24          item = net.GetClassDesc(ID)
25          item = item.strip()  # Removes spaces and invisible
                characters like \r and \n
26          box = (left, top, right, bottom)
27
28          matching_detections.append(detect)
29          box_stop = (left, top, right, bottom)
30          rects.append(box_stop)
31          cv2.putText(frame, f"{item} ({confidence:.2f})", (left,
                top - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0),
                2)
32          cv2.rectangle(frame, (left, top), (right, bottom), (0,
                255, 0), 2)
33
34      return matching_detections
```

For each detected traffic sign, the system compares it with the last detected sign. If a different sign is detected or the cooldown has expired, the appropriate command is issued:

- `Stop`: The robot will stop and enter a cooldown phase.
- `Go_Straight`: The robot will continue moving forward.
- `Turn_Left` and `Turn_Right`: The robot will turn left or right, respectively, with a maximum steering angle defined by `max_steering_angle`.

After handling the traffic signs, the system proceeds to track the road line as before.

5.3 MQTT communication

Message Queuing Telemetry Transport (MQTT) is a lightweight communication protocol built on the TCP/IP framework. It serves as an efficient mechanism for transmitting commands to the container running ROS2.

The process begins with the initialisation of an MQTT client that connects to a broker configured to use `localhost` and the default port `1883` ensuring that all

communication occurs within the same machine. A dedicated topic, `"robot/control"`, is used to standardise message transmission.

The system provides various functions to transmit commands via MQTT:

- `send_command(direction, angle=None)`: Sends a movement command specifying a direction. If an angle is provided, it is included in the message.
- `send_activate()`: Transmits an `"activate"` command to initialise the system.
- `send_deactivate()`: Sends a `"deactivate"` command to disable the system.
- `send_stop()`: Dispatches a `"stop"` command to pause the movement of the robot.

Each command is structured in JSON format and published to the defined MQTT topic, ensuring a uniform communication protocol.

The MQTT client is configured with an `on_connect` callback to verify successful connection, printing a confirmation message upon a successful connection. Once the connection is established, the `loop_start()` function is invoked to handle incoming and outgoing messages continuously in the background and ensures that the system remains responsive to commands without requiring manual intervention.

5.4 ROS2 workspace

5.4.1 Docker

Docker is an open source technology developed in the Go programming language for creating, managing, and deploying applications in containerised environments. The concept of Docker is to provide an isolated environment for each application, preventing conflicts due to software dependencies and optimising system resource utilisation. In the past, running multiple applications on a single server could cause compatibility issues such as network port conflicts or interference between various libraries. The use of containers, on the other hand, provides each application with its own isolated and distinct environment, thereby enhancing stability and security [36]. This approach provides a modular and isolated execution framework, enhancing system stability and maintainability.

In this thesis, Docker is used to execute ROS2 in a suitable environment, thereby bypassing the constraints imposed by the native Jetson Nano operating system. Specifically, the board is set up with a Software Development Kit (SDK) based on Ubuntu 18.04, which only natively supports ROS1. However, ROS2 requires Ubuntu version 20.04 or higher, making it necessary to use a Docker container to implement a compatible environment without directly altering the board operating system. This provides a convenient way to utilise the features of ROS2 while ensuring compatibility with the Jetson Nano software framework.

5.4.2 Visual Studio Code and SSH Configuration

VS Code provides a development experience across remote environments, enabling developers to edit, debug, and execute code on different machines. One of the most effective ways to achieve this is through the *"Remote Development"* extension, which allows secure connections to remote hosts via SSH.

In this project, this feature proves particularly useful, as it enables direct access to the robot computing resources from a local development machine. By ensuring that both the host (ROSMASTER X3) and the client machine are connected to the same Wi-Fi network, VS Code can establish a stable SSH connection. This eliminates the need for direct physical access to the embedded system while facilitating real-time code modifications and debugging.

To configure remote SSH access in VS Code, the process begins by pressing the shortcut key Ctrl + Shift + P to reveal the command palette. In the input window, the term "remote" should be typed to filter and display the available remote commands, after which the option to log into the designated remote host must be selected.

Subsequently, the option **remote-SSH: Add New SSH Host** should be chosen. At this stage, a prompt will appear requesting the connection command, which should be entered as follows:

```
ssh jetson@192.168.1.80
```

In this command, "jetson" designates the username for the remote system, while "192.168.1.80" specifies its IP address. This particular IP address is used as both the robotic system and the local computer are connected to the same network, ensuring efficient communication between the two devices.

Furthermore, to streamline remote SSH access within VS Code, the ROSMASTER X3 can be added as a remote host in the `~/.ssh/config` file. This file should include the IP address of the remote host (in this case, `192.168.1.80`), the username (`jetson`) and hostname (yahboom), as illustrated in Figure 5.2. This configuration facilitates future connections while also enhancing security and manageability.

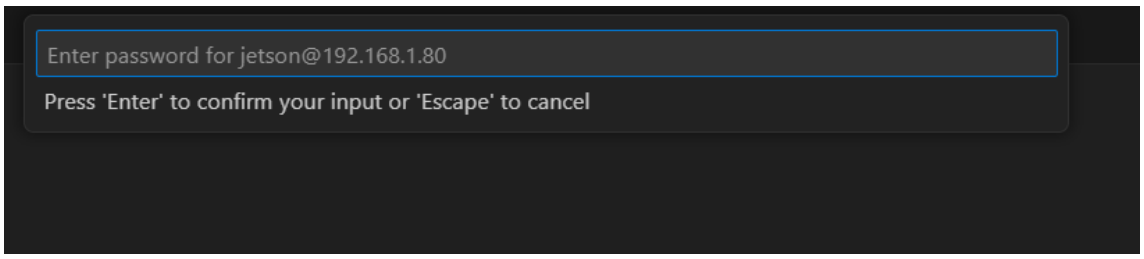


Figure 5.2: New SSH connection

Once connected (Figure 5.3), VS Code provides a fully functional development environment, enabling interaction with the remote filesystem, execution of terminal

commands, and seamless integration of debugging tools.

For Docker integration, the extension for VS Code, installed from the Marketplace, ensuring that a running container is available on the robot. Using the *Attach to Running Container* feature within the extension grants full access to the container's filesystem, including the workspace directory `/root/yahboomcar_ros2_ws` (Figure 5.4). This approach isolates the development environment, mitigating conflicts with system-wide dependencies while ensuring consistency across different setups through a pre-configured containerised workspace.

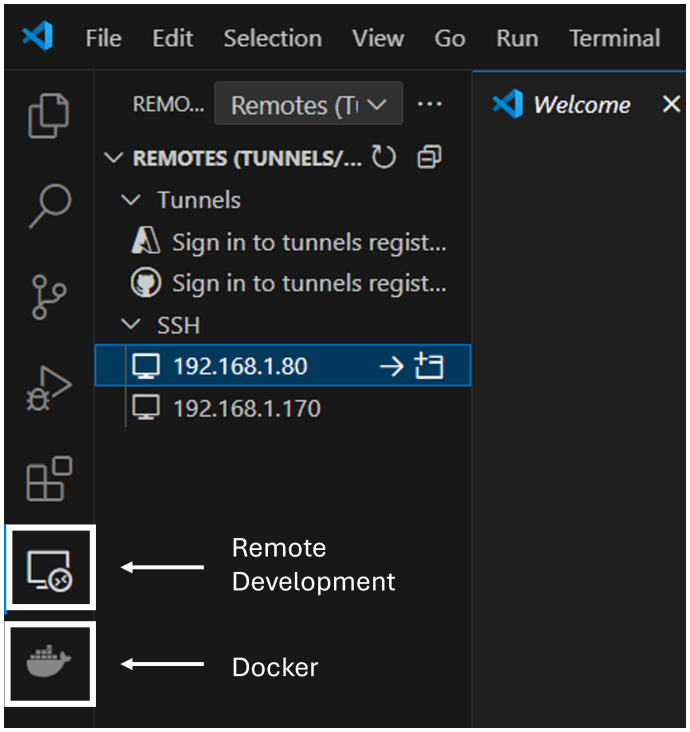


Figure 5.3: SSH connection

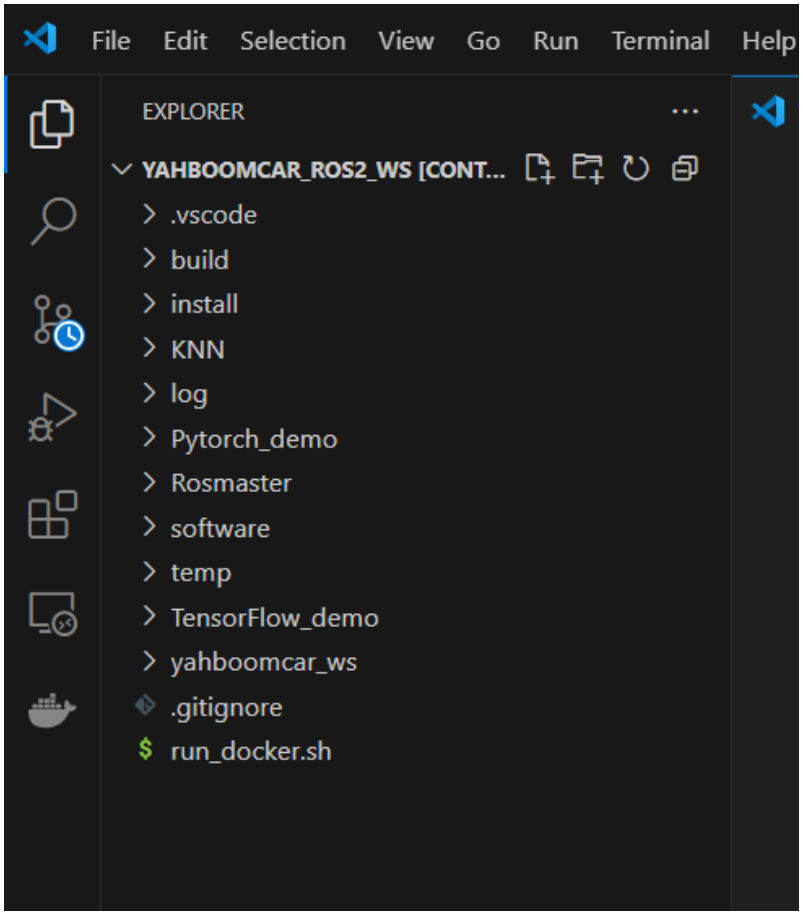


Figure 5.4: Workspace directory

5.4.3 Development Environment and Launch Files

In ROS2 a launch file is an essential component that facilitates the simultaneous execution of multiple nodes while also enabling the configuration of runtime parameters. ROS2 supports three primary formats for launch files: XML, YAML, and Python. Each of these formats provides a structured approach to defining node execution, parameter settings and additional runtime configurations.

To organise launch files within a ROS2 package, a new package is created and a dedicated directory is added within the package source using the following commands:

```
cd ~/yahboomcar_ros2_ws/yahboomcar_ws/src/pkg_autonomous_follow_road
mkdir launch
```

Typically, launch files follow a naming convention: `LaunchName_launch.py`, where `LaunchName` is a user-defined identifier and the suffix `_launch.py` is standardised. Consequently, the file `autonomous_follow_road_launch.py` is created in the launch

folder. This file integrates an external launch script for the LiDAR sensor and defines a node for autonomous road-following behaviour, as shown in Listing 5.6.

Listing 5.6: Autonomous follow road launch file

```
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():
    # Define the path to the LiDAR launch file
    sllidar_launch_file =
        '/root/yahboomcar_ros2_ws/software/library_ws/src/
        sllidar_ros2/launch/sllidar_launch.py'

    # Include the LiDAR launch file
    lidar_node = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(sllidar_launch_file)
    )

    # Define the autonomous navigation node
    follow_node = Node(
        package='pkg_autonomous_follow_road',  # Package name
        executable='follow_road',  # Executable name
        name='autonomous_follow_road',  # Node name
        output='screen',
        parameters=[
            {"mqtt_broker": "localhost"},
            {"mqtt_port": 1883}
        ]
    )

    # Return the launch description containing both nodes
    return LaunchDescription([lidar_node, follow_node])
```

This launch file ensures that the LiDAR sensor is properly initialised before the autonomous navigation node begins execution. The `IncludeLaunchDescription` function allows for seamless integration of pre-existing launch scripts, thereby enhancing modularity and maintainability. Furthermore, the configuration parameters specified within the `Node` object allow for customisation of the MQTT broker and port settings, which are essential for real-time data communication.

To ensure that a launch file is correctly included in the ROS2 package structure, it is necessary to modify the `setup.py` file. This step involves specifying the location of the launch file within the package and ensuring that it is correctly registered upon compilation (Listing 5.7).

Listing 5.7: Modifications to setup.py for launch file integration

```
from setuptools import setup
import os
from glob import glob

package_name = 'pkg_autonomous_follow_road'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'),
           glob(os.path.join('launch',  '*launch.py')))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='root',
    maintainer_email='1461190907@qq.com',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'follow_road =
               pkg_autonomous_follow_road.follow_road:main'
        ],
    },
)
```

5.4.4 Rosmaster Library and ROS2 Script

The ROS2 node `follow_road.py`, within the `pkg_autonomous_follow_road` package, implements the core logic for sensor fusion between the camera and LiDAR. Central to this architecture is the *Rosmaster* library, which provides a robust communication framework between various nodes. This library simplifies node registration, topic discovery, service management, and parameter handling.

Key aspects of the code include:

- **Hardware Communication via Rosmaster**: The node establishes communication with the STM32 board by creating an instance of the `Rosmaster` class (i.e., `self.car = Rosmaster()`). It configures the car type and initiates a receiving thread, ensuring smooth integration between the hardware and the higher-level control software.

- **Sensor Fusion through LiDAR Data Subscription**: The node subscribes to LiDAR data on the `/scan` topic. The associated callback function processes the range data to detect obstacles, enabling the robot to adjust its trajectory based on the proximity of obstacles.
- **Watchdog Mechanism for Safety**: A watchdog timer monitors the reception of MQTT messages. If no message is received within a predefined timeout period, the watchdog triggers a stop command to prevent unintended robot movement.
- **LED Status Indicators**: LED indicators provide visual feedback of the state of the robot: blue when idle, green when the follow-line mode is active and red when obstacles are detected or a stop is triggered.

The entire system is compiled and launched using the following steps:

1. **Compile the Workspace**: Navigate to the workspace directory and build the package:

```
cd ~/yahboomcar_ros2_ws/yahboomcar_ws
colcon build --packages-select pkg_autonomous_follow_road
source install/setup.bash
```

2. **Launch the ROS2 Node**: Run the launch file to start the node:

```
ros2 launch pkg_autonomous_follow_road
autonomous_follow_road_launch.py
```

The integration provided by the Rosmaster library ensures effective communication between the hardware and software components, while the sensor fusion and safety mechanisms contribute to reliable autonomous operation.

Chapter 6

Testing and Results

After developing the code, the rover behaviour was initially tested using the follow-line mode before integrating road signals with the follow-road functionality.

Once the container is launched by executing the script `run_docker.sh`, shown in Figure 6.1, the ROS2 workspace can be navigated and the script initiated, assuming that the navigation package has been compiled in ROS2 as outlined in Section 5.4.4.

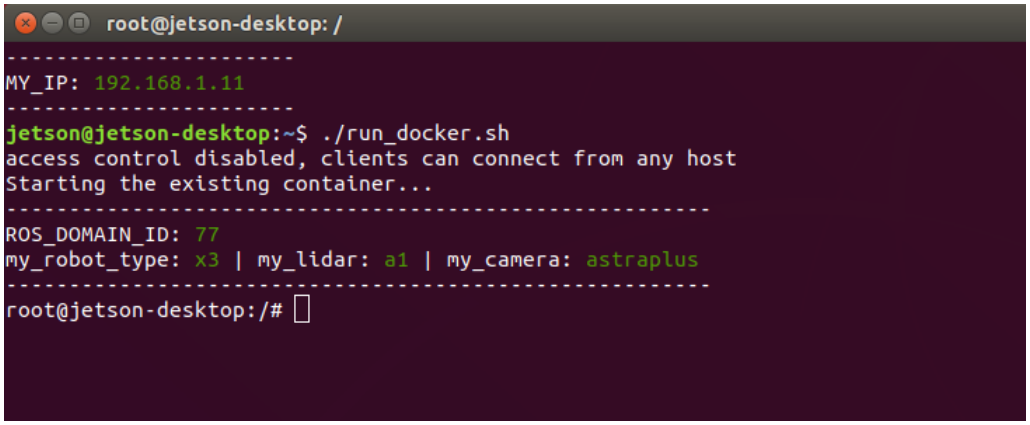


Figure 6.1: Access to the container

Subsequently, the ROS script is executed using the following command:

```
ros2 launch
pkg_autonomous_follow_road autonomous_follow_road_launch.py
```

This command initiates several ROS2 nodes essential for the operation of the autonomous driving mode along a road or line. For example, the `sllidar_node` communicates details about the LiDAR SDK version, firmware and hardware, indicating that it is beginning to send laser scan data correctly (Figure 6.2).

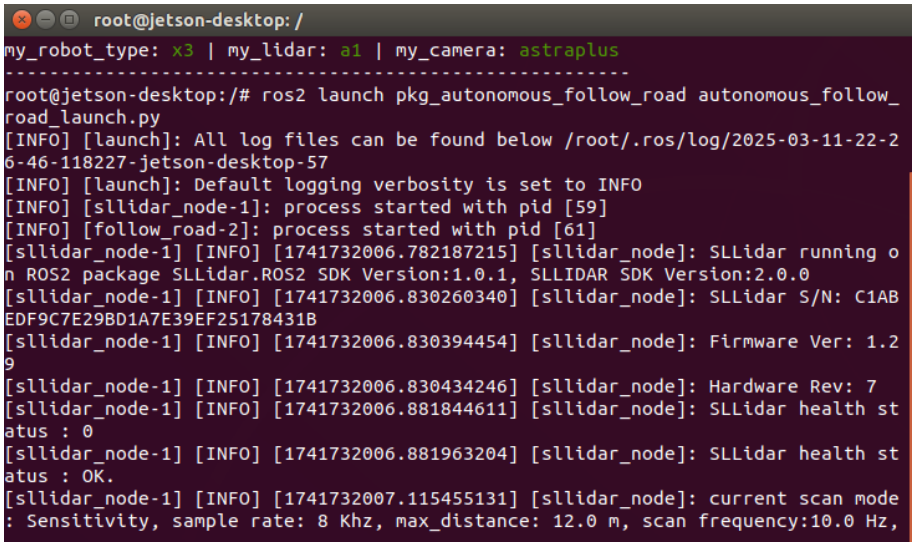


Figure 6.2: ROS2 node

In a separate terminal, the Python script is started with:

```
python3 object_detection_module_new.py
```

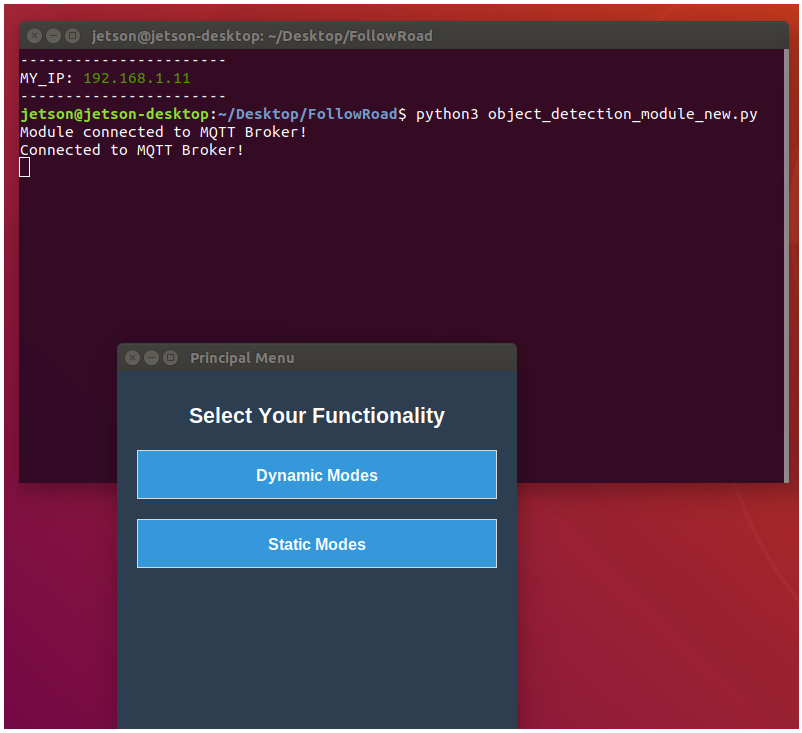


Figure 6.3: Principal Menu

This action launches an interactive menu, as illustrated in Figure 6.3. The system features two operational modes for handling different types of situations. The robot functions in Static Mode when stationary for processing tasks which do not need movement. The Dynamic Modes option directs users to a secondary menu, from which they can access the Reach Target and Follow Me functions developed in a previous thesis, as well as the Follow Line and Follow Road options created in this thesis (Figure 6.4). The Reach Target mode of the robot system detects targets through the field of view and tracks them while navigating towards the person designated as the target through continuous detection methods. The Follow Me mode renders real-time object tracking capability that lets the robot follow chosen individuals or objects as it moves within the field of view.

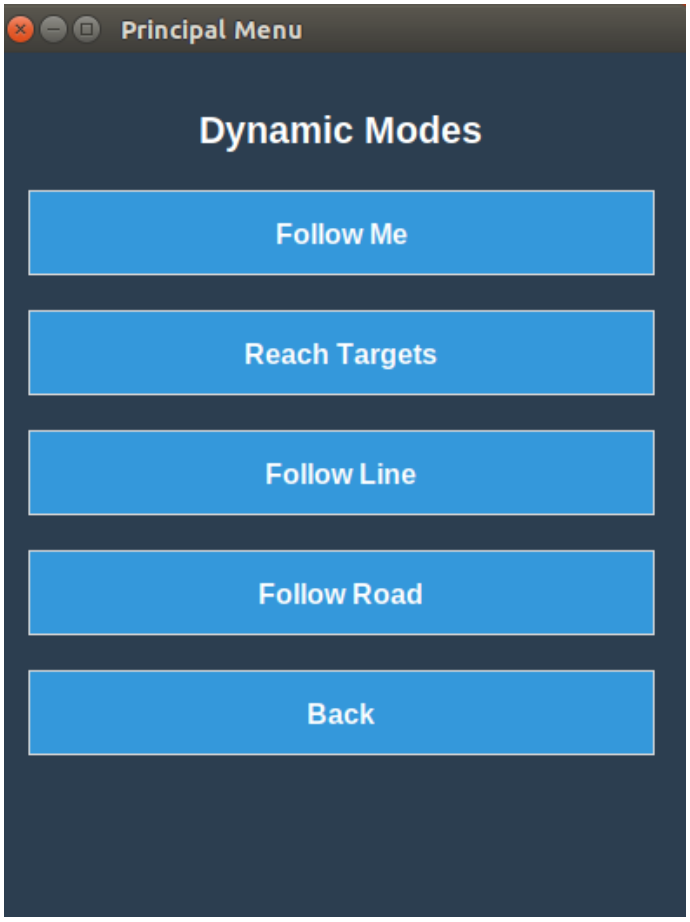


Figure 6.4: Dynamic Modes Menu

When the user selects the "Follow Line" option, a colour calibration window featuring adjustable sliders is displayed before initiating the line-tracking process, as illustrated in Figure 6.5. Additionally, two supplementary windows are opened: one displaying the live video frame and the other showing the corresponding binary

mask (Figure 6.6).

At this stage, the user is provided with the opportunity to fine-tune the colour parameters of the line using the HSV (Hue, Saturation, and Value) colour space. This calibration ensures optimal detection of the line under varying lighting conditions and surface characteristics. Once the calibration is completed, the script can proceed by launching the "Follow Line Core" module, which operates using the selected parameters. If the user does not manually adjust the sliders, the system will default to pre-configured HSV values, which are optimised for detecting a green-coloured line.

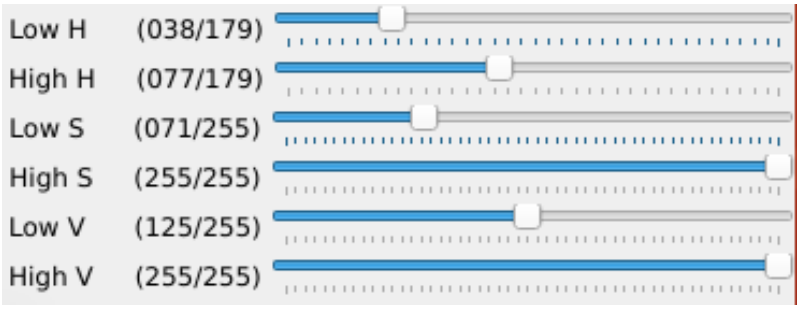


Figure 6.5: Colour calibration interface

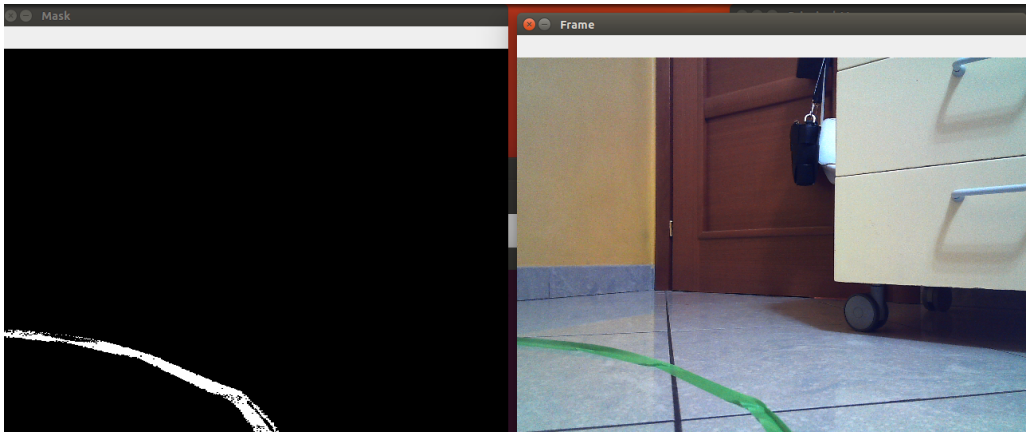


Figure 6.6: Mask frame

At this point, Figure 6.7 illustrates the interaction between the follow-line node and the commands received via MQTT. With each command (such as "left" or "activate"), the robot adjusts its linear and angular speeds to remain on course or to turn in the required direction. In addition, the system monitors for obstacles and checks the battery voltage.

```
root@jetson-desktop: /
[follow_road-2] Follow-Line activated
[follow_road-2] Received MQTT message: {"command": "left", "angle": 6}
[follow_road-2] Translating left with angular speed: 0.16000000000000003, linear
 speed: 0.36000000000000004
[follow_road-2] Received MQTT message: {"command": "activate"}
[follow_road-2] Follow-Line activated
[follow_road-2] Received MQTT message: {"command": "left", "angle": 6}
[follow_road-2] Translating left with angular speed: 0.16000000000000003, linear
 speed: 0.36000000000000004
[follow_road-2] Received MQTT message: {"command": "activate"}
[follow_road-2] Follow-Line activated
[follow_road-2] Received MQTT message: {"command": "left", "angle": 21}
[follow_road-2] Turning left with angular speed: 0.1, linear speed: 0.3200000000
0000006
[follow_road-2] Battery Voltage: 11.8V
[follow_road-2] Received MQTT message: {"command": "activate"}
[follow_road-2] No obstacles. Following line.
[follow_road-2] Follow-Line activated
[follow_road-2] Received MQTT message: {"command": "left", "angle": 7}
[follow_road-2] Translating left with angular speed: 0.16000000000000003, linear
 speed: 0.36000000000000004
[follow_road-2] Received MQTT message: {"command": "activate"}
[follow_road-2] Follow-Line activated
```

Figure 6.7: Interaction between the follow line and commands via MQTT

6.1 Testing Scenarios

The script was evaluated using several circuits of varying sizes, created with adhesive tape in different environments under various lighting conditions. This approach enabled the assessment of system performance across a range of scenarios.

Initially, the system was tested without incorporating road sign signals. During this phase, the line tracking performance was generally satisfactory, although some instabilities were observed.

Refer to Figures 6.8, 6.9 and 6.10 for visual examples of the circuit configurations.

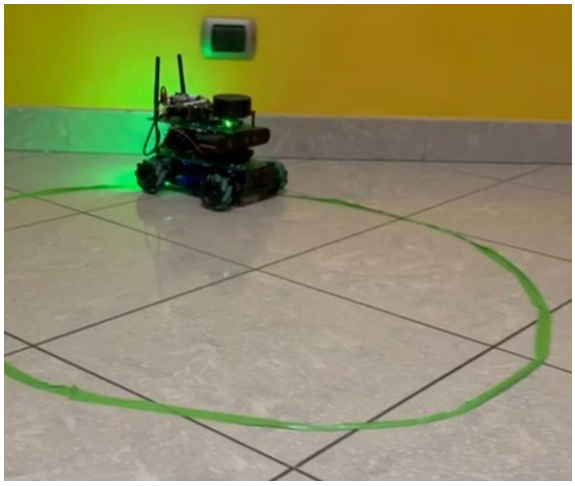


Figure 6.8: Set-up with a small circuit

```
Turning right!
angle: 19.5
Turning right!
angle: 18.1875
Turning right!
angle: 20.0625
Turning right!
angle: 19.125
Turning right!
angle: 18.375
Turning right!
angle: 17.4375
Turning right!
angle: 16.5
Turning right!
angle: 17.25
Turning right!
angle: 14.25
Turning right!
angle: 15.374999999999998
Turning right!
angle: 13.875
Turning right!
```

Figure 6.9: Commands detected on terminal

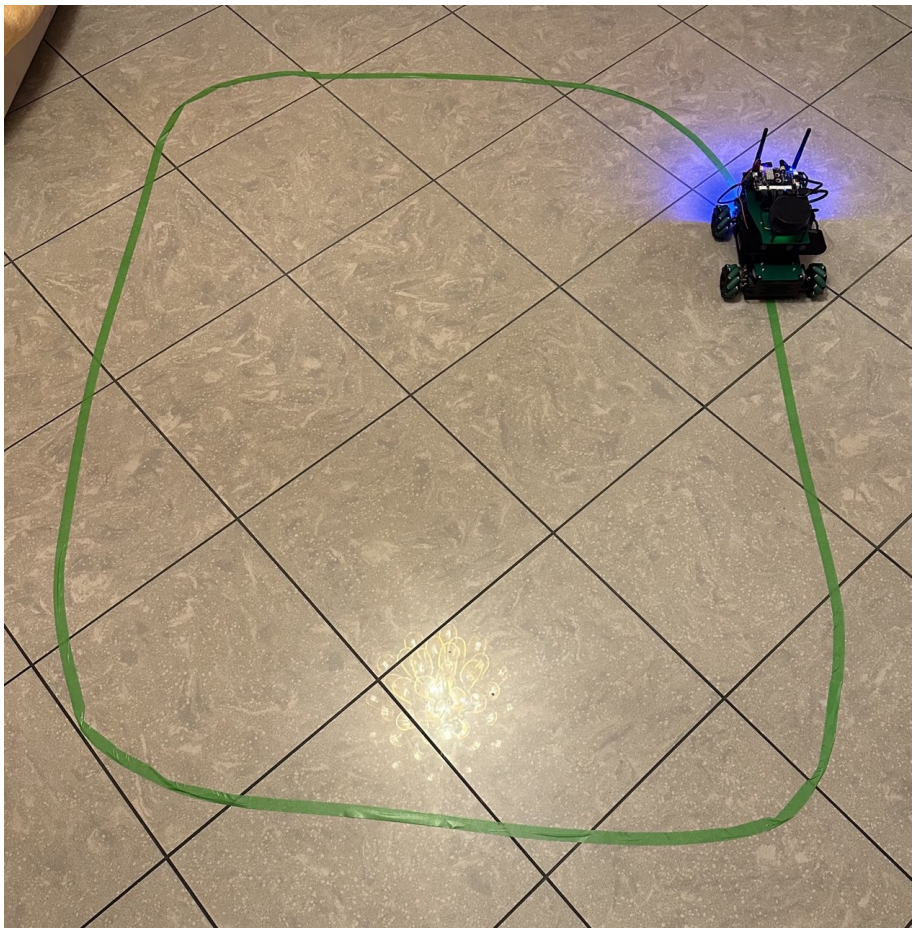


Figure 6.10: Set-up with a larger circuit

Subsequently, the Follow Road function, which integrates road sign signals, was examined. Although the code was designed to recognise four road signs (Go Straight, Turn Left, Turn Right, and Stop), the system experienced significant difficulties in accurately responding to all signals. It frequently detected incorrect signs, leading to confusion, as demonstrated in Figure 6.11.

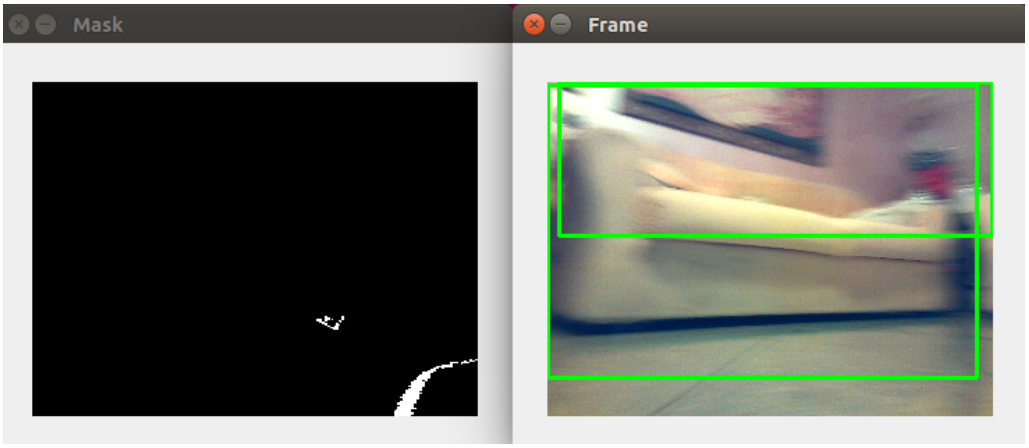


Figure 6.11: Detected a false Go Straight

As a result, the integration was limited to the Stop signal, which consistently produced correct responses (Figures 6.12, 6.13 and 6.14).

Figure 6.12: Stop detected in the MCA company

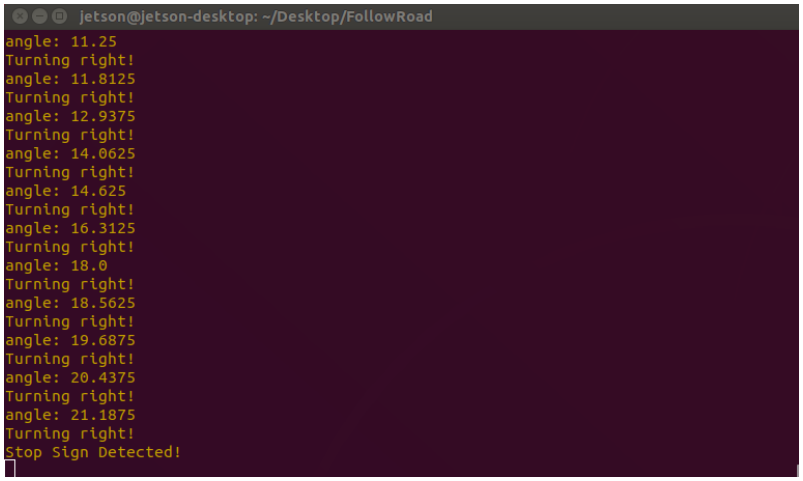


Figure 6.13: Stop detected on terminal

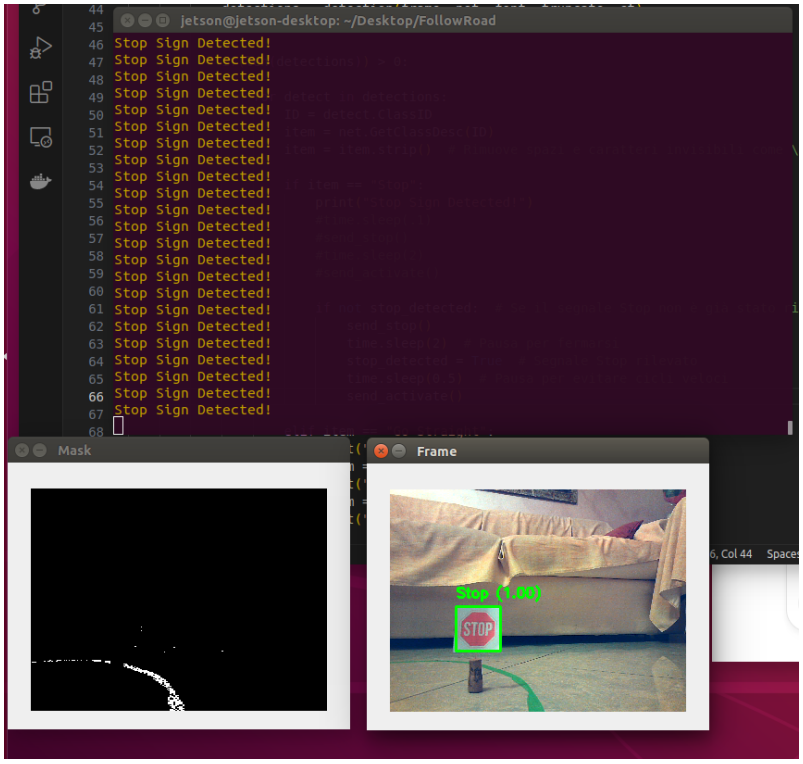


Figure 6.14: Stop detected at home

When the robot is unable to detect the green line in the mask, the display turns completely black, signalling that the line is no longer present. As a result, the robot halts its movement. This mechanism acts as a safety measure to prevent the robot from moving in an undefined direction (Figure 6.15).

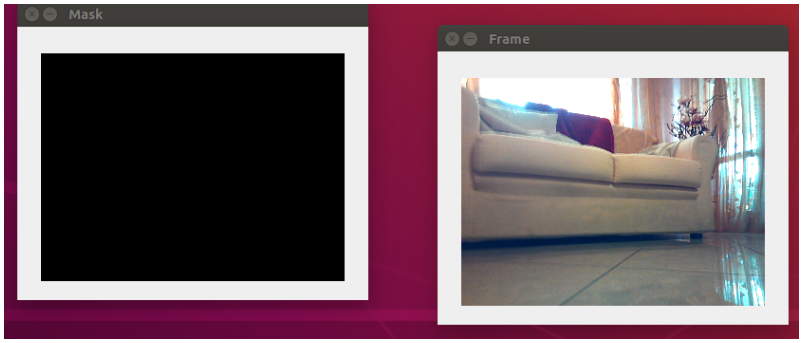


Figure 6.15: No line recognised

6.2 Performance Evaluation of the Rover

In order to complete the experimental segment of this thesis, a comprehensive suite of tests and quantitative analyses was implemented. These tests were designed to highlight and measure the performance of the *follow line* and *follow road* systems. This approach outlines avenues for subsequent optimisation. The following sections elaborate on the methodological framework and the corresponding results obtained, summarised in the Figure 6.16.

```
Standard deviation: 64.440
Percentage of errors within the threshold of 10 pixels: 19.637%
Mean steering angle: 9.751
Mean CPU usage: 77.619 %
Mean Memory usage: 65.450 %
Mean processing time per frame: 0.044 sec
Mean FPS: 28.594
```

Figure 6.16: Performance data

6.2.1 Analysis of Tracking Accuracy

The assessment of tracking accuracy was conducted by recording the horizontal error, which is defined as the pixel distance between the centre of the acquired image and the centre of the detected line. For each frame, this measurement was subjected to statistical analysis using specific metrics, as detailed below:

- **Mean error:** The average error was found to be approximately under 80 pixels. This indicates that, under standard conditions, the centre of the detected line is, on average, displaced by under 80 pixels from the centre of the image. Such a value points to the presence of a systematic deviation, suggesting that further optimisation is required to achieve more precise alignment.

To convert the error from pixels to centimetres, the following relation can be used:

$$\text{Error in cm} = \text{Error in pixels} \times \frac{\text{Width of the scene in cm}}{\text{Width of the image in pixels}}.$$

In this case, the camera operates at 160×120 pixels resolution with the mount height of $H \approx 13\,\text{cm}$ above the ground as it faces forward. Because the camera is not significantly inclined downward, we approximate the effective distance to the ground (at the centre of the view) by considering the geometry of the situation. With a horizontal field of view (FOV) of 60°, the distance to the ground where the central line of sight intersects can be estimated by:

$$D \approx H \times \tan(60^\circ).$$

Substituting $H = 13\,\text{cm}$:

$$D \approx 13\,\text{cm} \times 1.732 \approx 22.5\,\text{cm}.$$

Here, 22.5 cm represents the effective width of the scene at the level of interest, which is used to convert pixel measurements to real-world dimensions.

Given the image width of 160 pixels, the scale is:

$$\frac{22.5\,\text{cm}}{160\,\text{pixels}} \approx 0.14\,\text{cm per pixel}.$$

Therefore, 80 pixels is equivalent to:

$$80 \times 0.14 \approx 11.25\,\text{cm}.$$

Therefore, the observed error corresponds to an average displacement of approximately 11.25 cm from the centre of the image.

- **Standard deviation:** With a standard deviation of approximately 64 pixels, the results reveal a significant variability between frames. While the system demonstrates accuracy in certain frames, other instances exhibit more pronounced deviations, thereby highlighting potential inconsistencies in the tracking performance.
- **Percentage of errors within a 10-pixel threshold:** Analysis shows that only about 20% of the frames have an error of less than 10 pixels. This threshold is considered acceptable for the correct functioning of the system. However, the finding that the majority of frames exceed this optimal range underscores the need for further calibration and refinement of the detection algorithms.

The data were visualised using two graphical representations: a line graph (Figure 6.17) and a box plot (Figure 6.18). These visual tools provide valuable insights into both the temporal evolution of the error and its overall distribution across the dataset. Analysing these patterns is essential for assessing the system stability over time and identifying any inconsistencies in the robot tracking performance.

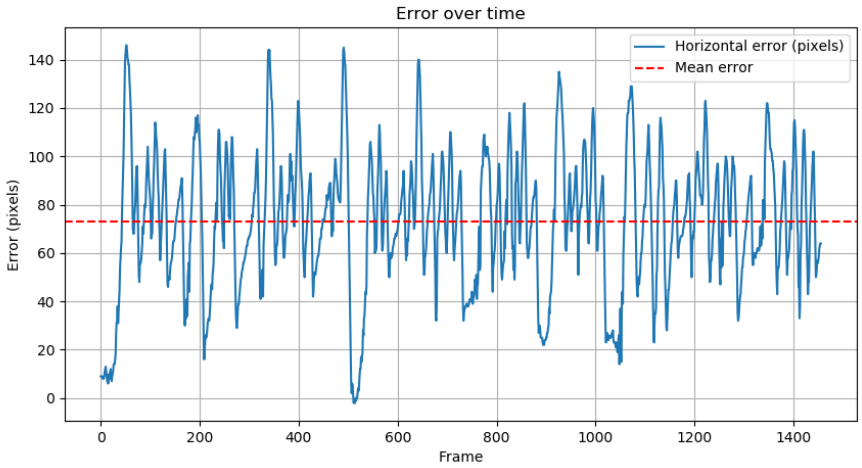


Figure 6.17: Temporal progression of the error

The line graph illustrates how the error fluctuates during operation. Significant variations suggest that the robot frequently deviates from the centre of the line, requiring continuous corrective adjustments. Sharp peaks, whether positive or negative, may indicate moments when the line is not correctly detected or when the robot makes sudden steering corrections. These fluctuations highlight potential weaknesses in the line-following algorithm or sensor reliability, which should be addressed to improve overall tracking accuracy.

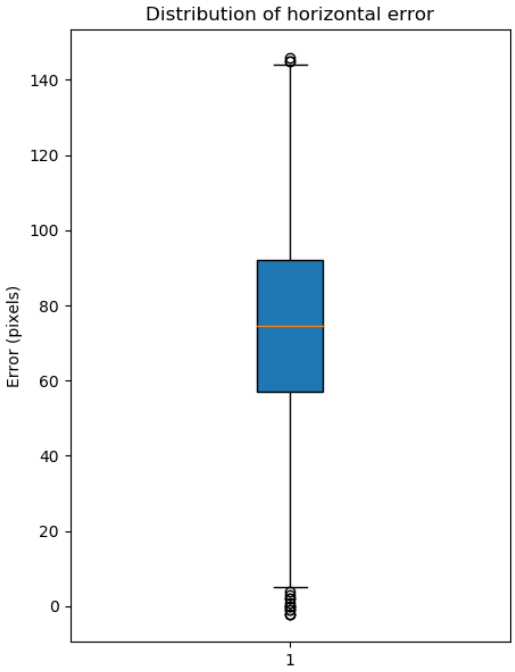


Figure 6.18: Error distribution analysis

The box plot provides a summary of the error distribution over multiple test runs. The **median**, represented by the central line within the box, appears to be around 70-80 pixels. If this value is significantly different from zero, it implies that the rover is, on average, misaligned with the ideal centre of the path by that margin. The **interquartile range (IQR)**, defined by the height of the box, represents the middle 50% of the data (spanning from the 25th to the 75th percentile). A wide IQR suggests considerable variability in the error, meaning the alignment of the rover is inconsistent throughout the test. In an optimal line-following system, both the median and the IQR should be smaller, indicating a more stable trajectory with fewer deviations.

Furthermore, the presence of outliers (extreme data points beyond the whiskers of the box plot) suggests instances where the robot significantly failed to track the line. These extreme errors may result from sudden changes in environmental conditions, suboptimal sensor calibration, or delays in the control response.

6.2.2 Computational Performance Analysis

The performance evaluation of the system was performed by quantifying the processing time for each individual frame. To achieve this, the methodology involved

recording timestamps using functions such as `time.time()` at the initiation and completion of the image acquisition and processing cycle. This approach facilitated the accurate computation of the elapsed time for each frame.

Subsequently, the frame rate was derived by calculating the number of frames processed per second, resulting in an average of approximately 28.594 FPS. This metric is pivotal as it directly reflects the efficiency of the system in handling real-time data.

Moreover, an in-depth analysis was undertaken to assess the latency between the moment of frame acquisition and the subsequent decision-making process. This latency is indicative of the system responsiveness and is a critical parameter in applications where prompt processing is essential. The measured mean processing time per frame was found to be around 0.044 seconds, suggesting a good performance level in terms of speed.

6.2.3 Steering Command Analysis

In this section, the computed steering angle is recorded at each iteration, allowing for a detailed evaluation of the system dynamic response. A graph is subsequently generated, which depicts the evolution of the steering command either as a function of time or in relation to the position error. The mean steering angle observed during the trials is 9.751, reflecting the average magnitude of directional adjustment executed by the system.

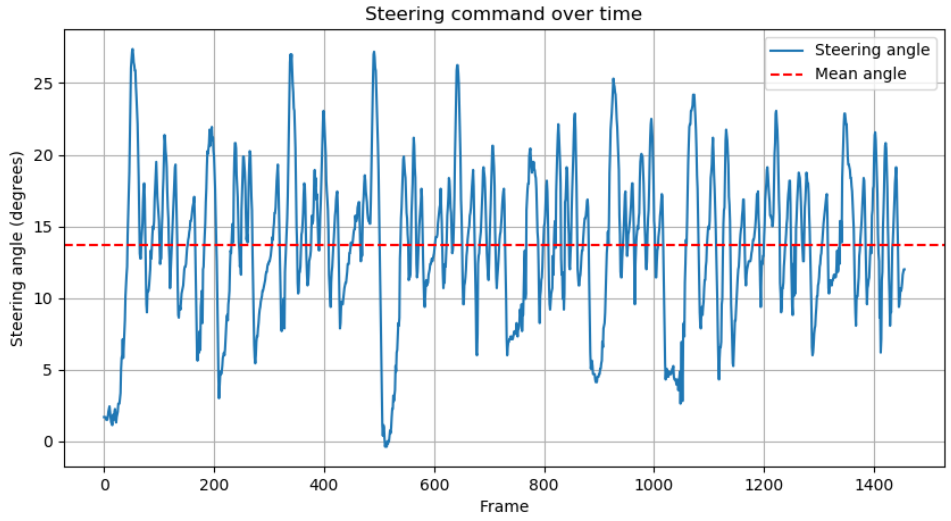


Figure 6.19: Steering angle variations over time

Figure 6.19 provides insight into the system steering performance, illustrating how quickly it responds to deviations and corrects errors.

The blue line represents the steering angle computed by the system at each frame, while the red dashed line denotes the average steering angle across the entire run. The presence of noticeable oscillations, both positive and negative, indicates continuous adjustments made to maintain the desired trajectory.

It is important to note that the chosen trajectory in this experiment is circular. In a perfectly straight path, the steering angle would ideally be zero, since no directional adjustment is needed. However, a circular path requires a constant non-zero steering angle to maintain the curvature. Therefore, the observed mean steering angle, which remains close to 15° confirms that the system is executing a steady turning command consistent with a circular trajectory. The fluctuations around the mean represent the dynamic corrections of the system to counteract disturbances and measurement noise while following the circular path.

6.2.4 Utilisation of System Resources

Key components such as the CPU and memory play a fundamental role in determining the overall responsiveness and stability of the system. Therefore, continuous monitoring of these resources during execution is essential to identify potential inefficiencies, detect bottlenecks, and implement corrective measures.

By tracking CPU and memory consumption over time, it is possible to detect patterns that may indicate excessive resource usage or underutilisation. Such insights can guide optimisation efforts, ensuring that the system operates within an acceptable performance range. The collected data can be visualised using a line graph, which aid in interpreting trends and making informed decisions and provide a clear visualisation of computational load fluctuations over time.

In the present analysis, the average CPU utilisation was recorded at 77.62%, while the average memory usage reached 65.46%. These figures provide a quantitative basis for assessing system performance and can be used as reference points for comparison against predefined benchmarks or for evaluating improvements following optimisation efforts.

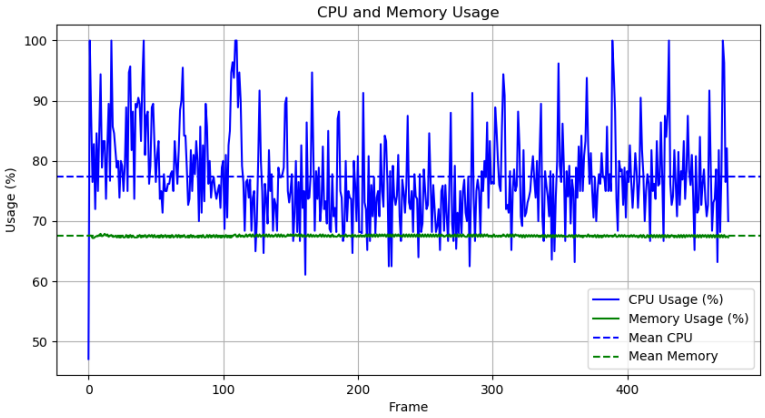


Figure 6.20: Graphical representation of CPU utilisation over time

The graph in Figure 6.20 displays CPU usage (represented by the blue line) and RAM usage (represented by the green line) over a given time period. Several key observations can be drawn from this data:

- **High CPU utilisation:** CPU usage occasionally reaches 100%, which may hinder real-time frame processing, potentially causing delays or performance degradation.
- **Elevated memory consumption:** Memory usage remains consistently high, which may lead to slowdowns, inefficient memory allocation, or, in extreme cases, system crashes due to insufficient available memory.
- **Acceptable performance thresholds:** Maintaining CPU usage below 80% and memory consumption under 70% significantly reduces the risk of hardware-related performance bottlenecks.

Possible solutions include refining image processing algorithms, reducing input frame resolution or employing hardware acceleration to improve efficiency.

Chapter 7

Conclusions and Future Developments

The main aim of this thesis was to design and assess an autonomous navigation system that employs the follow line technique, while also incorporating real-time road sign recognition and management. The use of the ROSMASTER X3, including the Jetson Nano, has allowed for the exploitation of advanced real-time processing capabilities. This integration has produced a system that functions in controlled settings. However, several significant limitations have been identified. This chapter examines the main challenges and constraints experienced during the project and outlines possible avenues for future improvements.

7.1 System Limitations

The implementation of the system was restricted by the computational capacity of the Jetson Nano. These constraints had a particular impact on the frame rate, which needed to be above 30 FPS to support optimal tracking and object detection. During testing, the Jetson Nano frequently generated CPU throttling alerts due to the high computational load, resulting in the need for reduced speeds to maintain accurate tracking and road sign recognition.

In addition, the current system configuration has revealed other structural issues, for example, the position of the camera may not be ideal for detecting road lines.

7.1.1 Sensitivity to Environmental Variations

One of the most critical challenges encountered is the sensitivity of the system to environmental conditions. The performance of the algorithm is markedly affected by variations in ambient lighting, which can alter the perceived contrast of the pathway. In instances where the surface exhibited low contrast, the system struggled to accurately detect the guiding line. This vulnerability not only compromises the

reliability of the line-following mechanism but also affects the overall stability of the navigation process. Future work should consider incorporating adaptive thresholding and dynamic calibration techniques to mitigate these issues.

7.1.2 Limitations in Road Sign Recognition

The recognition of road signs, while successful in static scenarios, presents several challenges during dynamic operation:

- **Proximity-Dependent Recognition:** When the rover is in motion, the system tends to recognise signs only when the vehicle is in close proximity. This delay in detection is critical, as it can lead to the execution of incorrect manoeuvres or delayed responses.
- **Confusion Among Signs:** The system has demonstrated a notably robust recognition capability only for stop signs. Other signs are frequently misclassified, suggesting that the feature extraction and classification components of the algorithm require further refinement. The limited size and diversity of the dataset contribute significantly to this problem, leading to an increased rate of false positives that directly impact the decision-making process of the rover.

7.1.3 Limitations of memory

Limited memory emerged as the most significant issue encountered during the project. The problem was largely due to the very large size of the container, which quickly consumed the available memory. As a result, it often became impossible to update the system or install necessary packages and libraries. This limitation not only affected routine maintenance but also restricted the overall functionality and scalability of the system. Future work should explore more memory-efficient container configurations or optimisation techniques to ensure smoother updates and installations, thereby enhancing the performance and reliability of the system

7.1.4 Processing Speed and Frame Rate Issues

The system has also been observed to operate at a lower frame rate than desired. This limitation has a twofold effect: it reduces the frequency at which visual data can be processed, and it contributes to a lag in the overall system response. The consequence of these low frame rates is a diminished capacity to perform real-time adjustments, which is particularly problematic in dynamic environments where rapid processing is crucial for safe and effective navigation.

7.2 Future Developments

In light of the limitations observed, several promising directions for future research and development have been identified:

- **Algorithm Improvement:** It is essential to improve the efficiency of the line-following and, especially, sign recognition algorithms. This may involve using better image processing methods, such as adaptive thresholding and straightforward machine learning techniques that can deal with changes in environmental conditions.
- **Dataset Expansion and Enhancement:** Addressing the challenges in recognising road signs requires a larger and more varied dataset. The dataset should cover a broader range of environmental conditions and different sign designs, while also reducing false positives. Expanding the road sign database will enable the system to generalise more effectively, resulting in improved precision. Additionally, it is important to refine the handling of signs that have not produced satisfactory results in the current system.
- **Real-Time Processing Improvements:** Improving the frame rate of the system is crucial for achieving better overall performance. This can be achieved through hardware acceleration or by optimising the software to run more efficiently. Techniques such as parallel processing and the use of dedicated hardware for neural network operations could greatly improve real-time processing.
- **Enhanced Performance in Dynamic Conditions:** Future work should focus on developing methods that allow the system to accurately recognise and process road signs while in motion. This may involve creating predictive algorithms that utilise contextual information to detect signs sooner, thereby reducing the dependency on close-range detection.

7.3 Final Reflections

In summary, this project presented a considerable challenge. It involved assembling the robot from individual components and creating a code to work effectively within a complete robotic system. This required a careful examination of the system architecture to ensure smooth communication among all parts. Furthermore, the software needed thorough testing and adjustments to guarantee reliable performance.

The integration process highlighted the importance of a clear and organised system design, as even small miscommunications between components could lead to significant performance issues. The necessity of rigorous testing became apparent, as it allowed the identification and resolution of unexpected problems before they could affect the overall functionality of the robot.

Despite these obstacles, the project achieved its main goal: creating an autonomous robot that follows a line and responds appropriately to road signs. This success not only demonstrates the feasibility of the chosen approach but also shows that careful planning and systematic testing can overcome many technical challenges.

Moreover, the work underscores the potential for further improvements that could lead to a more responsive and reliable autonomous robot.

Bibliography

[1] Fabio Marchisio, "Autonomous Robot Driving using Sensor Fusion," Master's thesis, Politecnico di Torino, 2024. [Online]. Available: https://webthesis.biblio.polito.it/33899/

[2] J. Doe and J. Smith, "Sensor Fusion Techniques for Autonomous Navigation," *MDPI Sensors*, vol. 21, no. 6, p. 2140, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/6/2140

[3] Shenzhen Yahboom Technology Co. Ltd., "Company profile," 2015, last accessed: Feb. 4, 2025. [Online]. Available: http://www.yahboom.net/aboutus

[4] Shenzhen Yahboom Technology Co., "ROSMaster X3 Product Description," 2025, last accessed: Feb. 4, 2025. [Online]. Available: https://category.yahboom.net/products/rosmaster-x3

[5] Ultralytics. (2025) Ai in self-driving cars. Last accessed: Feb. 5, 2025. [Online]. Available: https://www.ultralytics.com/blog/ai-in-self-driving-cars

[6] TechNexion. (2025) Applications and advancements of ai in robotics. [Online]. Available: https://www.technexion.com/resources/applications-and-advancements-of-ai-in-robotics

[7] ROS - Robot Operating System. Last accessed: Feb. 8, 2025. [Online]. Available: https://www.ros.org/

[8] ROS introduction. [Online]. Available: https://wiki.ros.org/ROS/Introduction

[9] ROS Concepts. Last accessed: Feb. 8, 2025. [Online]. Available: https://wiki.ros.org/ROS/Concepts

[10] (2021) Intro to robot operating system (ros) – part 1. Last accessed: Feb. 8, 2025. [Online]. Available: https://circuitcellar.com/research-design-hub/designsolutions/intro-to-robot-operating-system-ros-part-1/

[11] Shenzhen Yahboom Technology Co. Ros2 dds. [Online]. Available: https://github.com/YahboomTechnology/ROSMASTERX3/blob/main/04.X3-ROS2-Tutorials/08.%20ROS2%20Basic%20Tutorial/17.ROS2%20DDS/17.ROS2%20DDS.pdf

[12] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," 10 2016, pp. 1–10.

[13] Lidar. [Online]. Available: https://github.com/YahboomTechnology/ROSMASTERX3/blob/main/03.X3-ROS1-Tutorials/12.Lidar%20course/1.Lidar%20basics-SLAM/1.Lidar%20basics(For%20SLAM%20lidar).pdf

[14] Lidar triangulation. [Online]. Available: https://www.prayogindia.in/product/rp-lidar-a1m8-360-degrees-laser-range-finder-6-meter-range/

[15] Lidar tof. [Online]. Available: https://community.element14.com/learn/learning-center/essentials/w/documents/5037/time-of-flight-sensors

[16] Sedanur Kırcı, "Depth sensing camera technologies," 2023. [Online]. Available: https://medium.com/@kircisedanur2/depth-sensing-camera-technologies-afaf9ddb5a01

[17] Orbbec. Astra series. [Online]. Available: https://www.orbbec.com/products/structured-light-camera/astra-series/

[18] Andrew McWilliams. (2013) How a depth sensor works - in 5 minutes. [Online]. Available: https://jahya.net/blog/how-depth-sensor-works-in-5-minutes/

[19] Zhao, Zhong-Qiu and Zheng, Peng and Xu, Shou-Tao and Wu, Xindong, "Object Detection With Deep Learning: A Review," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, pp. 1–21, 01 2019.

[20] MathWorks. What Is Object Detection? [Online]. Available: https://it.mathworks.com/discovery/object-detection.html

[21] Gaudenz Boesch. (2024) Deep Learning Object Detection: The Definitive 2025 Guide. Last accessed: Feb. 8, 2025. [Online]. Available: https://viso.ai/deep-learning/object-detection/

[22] (2018) Object Detection MobileNet SSD v2. Last accessed: Feb. 8, 2025. [Online]. Available: https://roboflow.com/model/mobilenet-ssd-v2

[23] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, " MobileNetV2: Inverted Residuals and Linear Bottlenecks ," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2018, pp. 4510–4520. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2018.00474

[24] Liu, Wei and Anguelov, Dragomir and Erhan, Dumitru and Szegedy, Christian and Reed, Scott and Fu, Cheng-Yang and Berg, Alexander, "SSD: Single Shot MultiBox Detector," vol. 9905, 10 2016, pp. 21–37.

[25] Eddie Forson. (2017) Understanding SSD MultiBox — Real-Time Object Detection In Deep Learning. Last accessed: Feb. 12, 2025. [Online]. Available: https://medium.com/towards-data-science/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab

[26] Yahboom. ROSMASTER X3 ROS2 Robot with Mecanum Wheel for Jetson NANO 4GB. [Online]. Available: https://category.yahboom.net/products/rosmaster-x3?variant=48500437451068

[27] Yahboom Technology. Jetson nano 4gb b01/sub developer kit. [Online]. Available: https://github.com/YahboomTechnology/Jetson-NANO-4GB

[28] Amazon. Jetson nano b01 4gb development kit official board for ai and robotics yahboom provide ros programming courses. Last accessed: Feb. 24, 2025. [Online]. Available: https://www.amazon.it/Development-Ufficiale-Robotica-Yahboom-Programmazione/dp/

B0BNQDV3FR?language=en_GB
[29] Amazon. Yahboom Multifunctional Programming Development Board for Raspberry Pi Jetson Nano Orin Building ROS1 ROS2 Robot Expansion Board with 9-axis IMU Sensor STM32F103C8T6 Core (ROS Expansion Board). Last accessed: Feb. 26, 2025. [Online]. Available: https://www.amazon.com/Yahboom-Multifunctional-Development-Raspberry-STM32F103C8T6/dp/B0BX539VH1?th=1
[30] Yahboom. About expansion board_v1.0. Last accessed: Feb. 26, 2025. [Online]. Available: http://www.yahboom.net/study/ROSMASTER-X3
[31] Slamtec. Rplidar a1 dimension and weight. Last accessed: Feb. 26, 2025. [Online]. Available: https://www.slamtec.com/en/Lidar/A1Spec
[32] Orbbec3D Astra Pro Plus. Last accessed: Feb. 26, 2025. [Online]. Available: https://www.mybotshop.de/Orbbec3D-Astra-Pro-Plus_2
[33] Yahboom. Kinematic analysis of mecanum wheel. Last accessed: Feb. 26, 2025. [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
[34] LabelImg. Last accessed: Feb. 20, 2025. [Online]. Available: https://github.com/HumanSignal/labelImg
[35] A. Calio'. Deep learning-based real-time detection and object tracking on an autonomous rover with gpu based embedded device. [Online]. Available: https://webthesis.biblio.polito.it/18081/
[36] Docker. [Online]. Available: https://www.docker.com/