POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Integrating Real-Time Object Detection with LiDAR Data for Enhanced Robotic Autonomous Navigation

Supervisor
Prof. Marina Indri

Advisors at LINKS Foundation
Dott. Francesco Aglieco
Dott. Gianluca Prato

Candidate
Irisa Ibrahimi

317958

December 2024

Summary

The adoption of autonomous mobile robots in complex environments, such as warehouses, factories, offices, airports, and metropolitan areas, has been steadily increasing in recent years due to technology advancements in fields such as artificial intelligence, edge low-power computing platforms, and sensor systems. The main focus of this thesis was the integration of YOLO (You Only Look Once), a neural network for object identification, within the navigation system already installed onboard a robotic platform to improve its detection capabilities and handling of dynamic obstacles on the road, such as cars and other vehicles. This thesis, developed within the LINKS Foundation in Turin, originated as an extension of an autonomous navigation project initially geared toward indoor mail delivery by TurtleBot robots. To address the challenges posed by outdoor navigation and in the context of last-mile deliveries, the project was subsequently oriented toward the use of Agilex's Scout 2.0, a mobile robot designed to operate in outdoor contexts. The work adopts ROS 2, the next-generation robotic framework, and Nav2, its navigation platform, which goal is to provide the tools for safe and adaptive navigation in dynamic outdoor environments. The available LiDAR data has been integrated with the results of the AI-based objects detection, obtained through YOLO and performed on the camera data, enhancing the Nav2 Stack's cost map for a more accurate and complete representation of the surrounding environment, which is critical for autonomous outdoor navigation. This data fusion overcame some of the limitations of LiDAR in detecting dynamic obstacles at varying distances, improving the system's ability to make quick and safe decisions when navigating complex urban delivery scenarios.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Thanks to the major technological advancements achieved in the last years in the fields of artificial intelligence, edge computing, and sensor systems, the field of autonomous mobile robotics has received a significant boost in growth. From controlled indoor locations like offices and warehouses to challenging outdoor environments, such as cities and natural environments, this evolution has made it possible to employ autonomous robots in a wider range of more complex and varied conditions. In this context the emerging advanced autonomous navigation capabilities that can manage both static and dynamic obstacles while ensuring safe and effective operation are necessary for the successful integration of these robots into real-world use cases.

1.1 Motivation

Obstacle avoidance is a significant issue in autonomous navigation systems, and especially in dynamic outside environments, where robots must interact with several dynamic objects at the same time. The ability to describe and predict the behavior of obstacles is still an active research area, despite the fact that several solutions combining different algorithms and exploiting a number of different sensors have been released lately. Some of these solutions even aim at understanding the semantic context of identified objects and their expected paths, which is a task that goes beyond obstacle detection.

Autonomous robots must be able to face challenging urban scenarios like intersections, mixed use roads, and pedestrian crossings in real-world settings. The implementation of proper responses to changes in the scenario the robot is navigating is dependent on the robot's capacity to discriminate among various obstacle types (e.g. parked cars, cyclists, pedestrians, autonomous vehicles). More complex navigation techniques are enabled by these contextual awareness capabilities, such as updating path planning in response to detected obstacle movements or stopping when vehicle trajectories suggest a chance of accident.

Modern autonomous navigation systems mainly make use of LiDAR technology to obtain obstacle detection capabilities and create a map of the environment. While LiDAR sensors provide near exact distance measurements and allow for reliable detection of static obstacles, they are limited when dealing with identifying and tracking dynamic obstacles, especially when posed at varying distances and set in complex urban scenarios. This limitation proves to be especially critical in outdoor environments where the robots must interact with a large set of moving objects, including vehicles, pedestrians, and other autonomous systems.

1.2 Technical Approach and Innovation

The main contribution of this thesis work is the successful integration of a state-of-the-art object detection AI algorithms with the sensor systems, in this case camera and LiDAR, already available and installed on a robotic platform. The final developed system improves the conventional navigation approach based on cost maps by providing the results of AI-based object detection and classification, thus improving the robot's environmental perception capabilities. This integration is implemented within the ROS 2 framework and its navigation stack, Nav2.

The creation of a sensor fusion algorithm whose goal is to merge LiDAR readings with camera-based object detection results is the most significant breakthrough of this work. This method allows to overcome the limitations of single-sensor object detection systems and offers a more robust way to deal with moving obstacles in complex outdoor settings without the need of updating the hardware of the robotic platform. Improvements in obstacle recognition and tracking capabilities are achieved through this implementation, solely through the use of onboard cameras and LiDAR.

1.3 Research Objectives

The work evolved from a previous indoor navigation project, developed in collaboration with the LINKS Foundation in Turin, utilizing TurtleBot robots for creating an internal mail delivery to a more sophisticated outdoor navigation system built upon the previous results in obstacle characterization and behavioral algorithms.

This thesis aims to address the challenges described in this section by posing a set of key objectives:

1. Integration of a state-of-the-art computer vision detection algorithms with existing LiDAR-based navigation systems inside the ROS 2 framework

2. Development of a robust methodology to provide labeled obstacle information coming from sensor fusion operations to the existing autonomous navigation solutions
3. Implementation and testing of the developed system in simulated environments and possibly physical testing of the solution on real hardware.

1.4 Thesis Structure

The structure of this thesis reflects the work done on analyzing the use case and technical challenges and solutions, the implementation methodology of the proposed solution and its experimental validation:

- **Chapter 2** presents the state of the art relative to outdoor autonomous navigation challenges and the relative enabling technologies
- **Chapters 3 and 4** list the software and hardware components utilized during the implementation of the system
- **Chapter 5** discusses the system design and implementation
- **Chapter 6** presents the results of the simulation work done to test the system
- **Chapter 7** concludes with lessons learned and possible directions for future research

Chapter 2

Outdoor Autonomous Navigation Challenges and Enabling Technologies

Outdoor autonomous navigation is a rapidly evolving field, driven by the need for robots to operate in diverse and unstructured environments. This chapter's goal is to familiarize the reader with the subject of mobile robot autonomous navigation in outside environments and common service robotics applications.

As previously said, the topic will be addressed with a particular emphasis on outdoor settings, offering a formal description and highlighting all the essential features required to complete the navigation job.

Additionally, a categorization that helps to find the solution suggested in this thesis in the existing landscape is provided, along with some of the most popular algorithms, data sets, and approaches utilized to tackle the navigation planning problem.

2.1 Mobile Robots Basics

The rapid expansion of mobile robotics has transformed numerous application fields, from automated surveillance - from the ground, sea or air - to planetary exploration, without forgetting emergency rescue scenarios and industrial automation applications.

These scenarios can be performed by wheeled robots designed for outdoor environments which serve as a major example of this technological advancement. These autonomous systems are based on a set of principal functional components — perception, localization, cognition, (motion)planning, and locomotion — and use those to create an enabling framework for reliable navigation across diverse and unknown terrain. These robots are in fact able to autonomously decide what to do and move efficiently from one place to another by coordinating their perception and control components.

This is a demonstration of how different subsystems should interact in order to achieve autonomous operation in challenging outdoor environments. A brief introduction to these macro components is presented below.

Perception

Perception provides the information required for understanding the environment and adapting the navigation to it in real time. Several sensors, such as depth cameras, LiDAR, infrared sensors, or ultrasonic sensors, are commonly used to collect information and data about the robot's surroundings. These sensors can be used together, and coupled with algorithms to give a multidimensional picture of the environment the robot is navigating. The robot can in fact use its perception capabilities to recognize landmarks, or in some cases detect obstacles, and even decipher the properties of the ground by understanding the materials.

By recognizing obstacles and being able to understand navigation routes, perception can produce accurate results and help with planning the robot motion, as well as enabling the production of real-time maps with dynamic obstacle detection.

Planning

The robot motion planning capabilities uses the input from the perception components to generate a route avoiding collisions from the robot's starting position to its planned destination. Planning algorithms, (such as: A*, Rapidly-exploring Random Trees (RRT) Dynamic Window Approach (DWA)) [20], can be used to select an optimized and safe path to the destination. These algorithms are designed to consider either static or dynamic obstacles, and refine the path in real time according to the incoming sensor data.

Motion planning is empowered also by cognitive models that can represent using data structures the robot's understanding of its current state and of its surrounding environment (either outdoor on indoor), allowing it to react to changes if they happen during the navigation.

Localization

Robotic systems, especially wheeled platforms which have been designed for outdoor operations, should include integrated localization and navigation capabilities to be able to operate autonomously in a set of different environments as the outdoor scenarios require. While localization and positioning systems like GPS (with open access to the satellite constellations) and SLAM (where the map is not previously known) provide informations to robots about their position, and orientation, these technologies also bring limitations such as - as stated above - poor signal quality in urban areas for satellite based navigation or insufficient precision for certain applications regarding SLAM algorithms.

To overcome these challenges, robots must adopt information acquired from multiple souces (either sensors and algorithms), including state-of-the-art solutions such as visual odometry or by combining perception and motion planning systems to support continuous reliable mobility in scenarios with complex terrain or unknown environments.

Cognition

The cognitive system of the robot have to read and process data from its sensors in order to make sense of its environment, enabling spatial awareness.

The acquired data can in fact be used by the algorithms running in the robot's operative system in several ways, including anticipating future events such as motion prediction and thus responding swiftly to sudden changes in trajectories. The robot can in fact track and understand the type of moving objects, thanks to its cognitive abilities, and then make predictions about their future positions by creating and continuously updating a map, as the robot and the other objects move through the environment. The robot can also track and report its level of confidence about each detected object and thus focus on the most relevant - to the task it is accomplishing - elements in its surroundings, such as barriers or navigational signs.

As the robot continues to move through the environment, the decisions made about its course and motion can be improved and adapted, balancing its goal between accomplishing its planned task and safety requirements which should have the priority.

Locomotion

Finally, locomotion capabilities refer to the physical mechanisms (e.g., wheels or legs) adopted by the robot designers to move across the various terrains which the robot has to go through to reach its navigational goal. Robots with wheels, even more than robots with legs, adopt navigational capabilities to handle the challenges posed by outdoor environments, such as uneven terrain and obstacles like rocks or ditches. In these kinds of robots the locomotion capabilities need to be further supported also by planning capabilities to adapt to and overcome these physical limitations, for example by calculating feasible trajectories that maintain the robot's stability while reaching the desired navigation objectives.

2.2 Challenges in Outdoor Autonomous Navigation

2.2.1 Problem Statement

Autonomous navigation in outdoor settings presents specific challenges for perception capabilities in wheeled robots mainly due to the unpredictability of environments in terms of configurations and by the presence of other actors.

Indoor navigation in fact is less challenging from this perspective, thanks to a more controlled and previously known environment (e.g., controlled lighting, consistent flooring, reduced or absent unplanned obstructions).

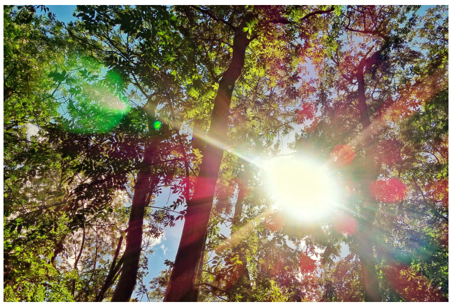
Outdoor conditions on the other hand introduce variability in terrain, weather, and through the presence of unplanned dynamic obstacles. The goal of this thesis is to enhance the robot perception and cognition capabilities by integrating an AI-based object detection algorithm using perception information from camera with LIDAR data, with the broader aim of adding object recognition capabilities and environment understanding to the robot's cognition capabilities enabling robust decision-making in real-world outdoor scenarios.

2.2.2 Perception Challenges

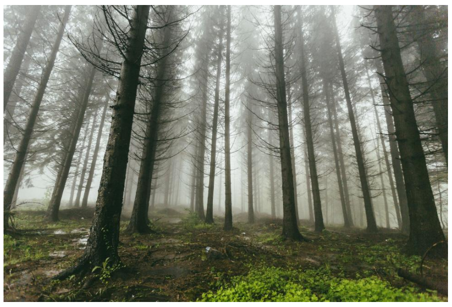
In this section we explore some of the main outdoor perception challenges. These challenges stem from the environmental complexity and sensor integration issues that an outdoor environment poses. Environmental conditions present in fact significant challenges in perception capabilities, as the potential variability in weather, lighting, and terrain types can impact the sensor functionalities, often reducing or even completely disabling the quality of data coming from both LIDAR and camera perception inputs. The limited range and resolution of some of these sensor can introduce additional challenges, particularly relative to the detection and precise classification of distant objects present in the environment. Some specific sensing issues, as analyzed in [10], relative to camera are visualized in Fig. 2.1.

The complexity of sensor fusion operations - required to use data from different sensors at the same time for real time precise scene depth estimation and object classification - introduces further challenges, as combining the data acquired from camera and LIDAR sensors often presents challenges in operations such as timestamp synchronization, FOV calibration, and data matching. The high computational demands of processing large volumes of sensor data in real-time needs to be considered as well, as the appropriate hardware must be installed on board of the robot, balancing the need for computing power and the need to optimize the energy consumption of more powerful hardware. A comprehensive review of perception sensors, compiled in [10] can be found in 2.1, including advantages and

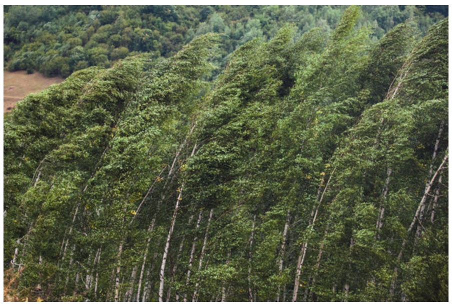
(a) Image homogeneity.



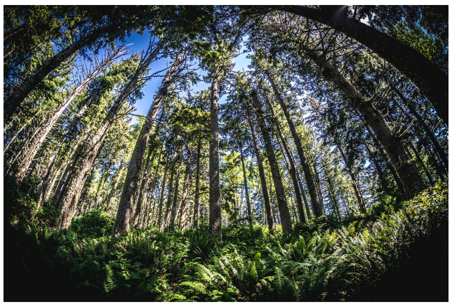
(b) Lens flare and overexposure due to direct sunlight.



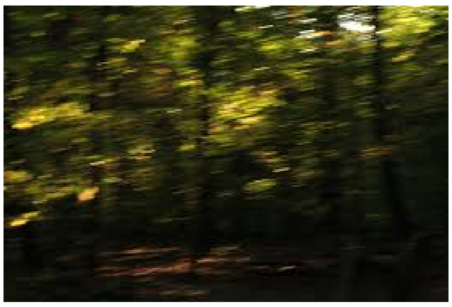
(c) Decreased visibility due to fog or rain.



(d) Unexpected motion effects due to wind.



(e) Effects of perspective changes.



(f) Blurring due to robot motion.

Figure 2.1: Camera sensing challenges in outdoor conditions. [10]

disadvantages for each sensor technology.

The process of object detection and classification itself represents another challenge, as a large number of objects with different shapes, sizes, and appearances (even inside the same class of objects) can be encountered by the robot while navigating in outdoor environments. These objects can either be dynamic, such as pedestrians or vehicles in city environments, or static, like trees or rocks in natural environments, and their detection and classification requires a reliable detection

Sensor Technology	**Sensing Type**	**Advantages**	**Disadvantages**
RGB camera	Imaging sensor	Allows for relatively inexpensive high-resolution imaging	Does not include depth information
RGB-D camera	Imaging and ranging sensor	Relates images to depth values	Generally low-resolution to reduce costs
Thermal camera	Temperature imaging sensor	Temperature readings in image format can improve segmentation and help detect animals	Generally low-resolution and more expensive than normal cameras
Hyperspectral sensor	Imaging sensor with many specialized channels	Allows for better segmentation (e.g., using vegetation indices)	Expensive and heavy-duty when compared to other imaging techniques
Multispectral camera	Imaging sensor with some specialized channels	Allows for better segmentation (e.g., using vegetation indices); inexpensive	Less powerful than its hyperspectral counterpart
Sonar	Sound-based range sensor	Allows for inexpensive obstacle avoidance	Limited detection range and resolution
LiDAR/LaDAR	Laser-based range sensors	Allow for precise 3D sensing	Relatively expensive and difficult to extract information beyond spatial accuracy
Electronic compass	Orientation sensor	Allows for partial pose estimation	May suffer from magnetic interference
Inertial sensors	Motion/vertical orientation sensors	Allow for partial pose estimation	Suffer from measurement drift
GPS/GNSS	Absolute positioning sensors	Allow for localization and pose estimation	Difficult to keep track of satellite signals in remote woodland environments

Table 2.1: Comparison of Sensor Technologies [10]

and classification pipeline that can work across the evolving environmental conditions. Dynamic adaptation is in fact required as navigating a real world outdoor environment requires a continuous evaluation and understanding of the scenario.

2.3 Object Detection Technologies enabling Outdoor Autonomous Navigation

Ranging from simple traditional computer vision algorithms to complex multi-sensor methods, object identification algorithms and perception capabilities in robotics have recently experienced extensive development. Giving an emphasis on real-world applications, this section explores the state of the art in object detection algorithms applied to robotics perception. In this section the possibilities opened by the adoption of LIDAR technology in combination with camera are also considered, as well as the impact of the shift from conventional computer vision techniques to the adoption of deep learning models. Of particular importance when dealing with neural networks are the outdoor datasets which are central for developing reliable detection algorithms that perform well in real-world scenarios.

2.3.1 Traditional and Deep Learning based Computer Vision Algorithms for Object Detection

Modern autonomous mobile robots require sophisticated perception capabilities to navigate complex outdoor environments effectively. The integration of various machine learning and computer vision approaches has revolutionized object detection systems, enabling robust performance across diverse operational conditions. Some fundamental approaches and the relevant implementation libraries are listed below.

1. **Traditional Computer Vision Algorithms** The approach of traditional computer vision to object detection prior to the deep learning revolution relied on manually selected features and mathematical formulas. As an example, the *Viola-Jones framework* [15] introduced in 2001, is considered a milestone in this field since it enabled real-time face (even though it can be adapted to other objects and classes) detection.

   Scale-Invariant Feature Transform (SIFT) [23] should also be mentioned as it has been proved as a traditional method for applications requiring robust object detection capabilities. This algorithm is based on the identification of distinctive points in the image which are invariant to transformation (e.g.

scale, rotation, and illumination variations). Its ability to generate descriptors that capture local image features made it particularly useful and facilitated its adoption in object detection and recognition tasks.

The use of *Histogram of Oriented Gradients* (HOG) descriptors, together with *Support Vector Machines* (SVM), has provided to the community another powerful framework for simple object detection use cases [31]. The approach of using gradient distribution to model the appearance and shape of the object to be detected has proved especially effective for use cases requiring the detection of objects with relatively consistent shapes, such as traffic signs or people. The *Deformable Parts Model* (DPM) has extended the concept o using gradient distribuiton for object detection by modeling the objects as a collection of parts arranged in variable (deformable) configurations, enabling more precise detection of articulated targets such as people.

The traditional approaches listed in this section, even though proven to be computationally efficient, often face limitations in handling the task assigned to them especially in complex scenes presenting challenges such as occlusions and harsh environmental or illumination conditions. Their reliance on manually designed features, meaning that the features corresponding to the object to be detected should be computed each time, in fact means that these algorithms present poor generalization capabilities. Nevertheless, these algorithms are still used today in some contexts and especially in applications where computational resources are constrained or interpretability is a requirement.

2. **Advanced Deep Learning Architectures** Since **Convolutional Neural Networks** (CNNs) and related architectures can directly extract and "learn" complex features from large datasets containing the objects to be detected, the use of Deep Learning (DL) in object detection signifies a move away from mathematics-driven methodologies and toward data-driven ones. Due to their ability to learn hierarchical feature representations from a variety of complicated scenarios, these architectures — of which *R-CNN* and *Mask R-CNN* are some of the most popular examples — achieve optimal object detection and segmentation performances.

   In deep learning further improvements in object detection using DL have been brough by the introduction of **Region-Based Approaches**, thanks to their ability to focus on specific regions of the image which are deemed likely to contain the objects to be detected. The *R-CNN* family and *Mask R-CNN* implement this approach by first generating region proposals, then applying neural networks to classify objects inside the proposed regions and then refine the detected object boundaries within these regions. This approach enables precise object detection and segmentation by concentrating computational resources in relevant areas. Mask R-CNN in particular has brought further improvements thanks to its proposal of - instead of using selective search for proposing regions inside the image - implementing learnable methods for enhanced detection precision. On the other hand Mask R-CNN extends object detection capabilities by being able to perform instance segmentation for a more precise identification of the image region containing the object.

A slimmer deep learning model can be found in the **Single-Shot Detector** architecture which is optimized for real-time object detection scenarios and is able to find a balance between inference speed and detection accuracy thanks to its ability to simultaneously predict bounding boxes and class probabilities. The most notable implementation of this architecture is *YOLO v4/v5*, which utilizes spatial attention mechanisms for real-time detection in a single network pass, *Single Shot Multibox Detector* (SSD), designed to adopt multi-scale feature maps with pre-defined anchor boxes, and *RetinaNet*, created to address a class imbalance in the training dataset.

Transformer-Based Detection represents the latest advancement in the object detection field and takes a leap forward from more traditional DL by leveraging transformers' self-attention mechanisms to capture latent global relationships across different images and thus enhance object detection performance through learned long-range dependencies. This approach is implemented in models such as *DETR*, which optimizes the object detection pipeline through end-to-end training, Swin Transformer, which introduces hierarchical feature learning for flexible scale handling, and *ViT-YOLO*, which can get the best of both worlds from single-shot detectors and transformer-based encoders combining transformer flexibility with YOLO's computational efficiency.

Training on Outdoor Datasets

DL models, to obtain good performance in real world settings, are trained on datasets containing a large number of diverse outdoor scenes, such as the COCO (Common Objects in Context) or KITTI Vision Benchmark. A comprehensive review is provided in [10] and a summary can be found in Table 2.2 The scenarios included in these datasets are selected to provide varied environmental conditions, lighting variations, and diverse object categories, which better prepare models for real-world deployment conditions. For instance:

Table 2.2: Comparison of Key Outdoor Environment Training/Testing Datasets

Name	Type	Sensors	Environment	Frames	Labelled
KITTI	2.5D	RGB-D/LIDAR/IMU	Real/Urban	216k	400
nuScenes	3D	RGB-D/LIDAR/IMU	Real/Urban	1.4M	93k
SEMFIRE	3D	RGB/LIDAR/IMU	Real/Forest	1.7k	1.7k
TartanAir	3D	RGB-D/LIDAR/IMU	Synthetic/Mixed	1M	1M
COCO	2D	RGB	Real/Mixed	330k	330k

- **COCO** provides a large collection of annotated images composed of over 330,000 images which can contain objects selected from 80 object categories. The value of this dataset lies in the fact that the object depicted are set in complex life-like scenarios. The dataset's strength are thus its contextual representations of the object in their real-world settings and with different scales, orientations, or environmental conditions.
- **KITTI** includes on the other hand stereo images, LIDAR point clouds, and IMU data, thus better supporting multi-modal perception and localization tasks. This dataset is particularly suited for training DL models which are integrated by design with LIDAR sensors, as it supports both 2D visual features and 3D spatial information, enhancing the overall scene understanding by the maodel and its object detection capabilities.

The use of these datasets in the training pipeline of the deep learning models mentioned above - like YOLO, which has been trained on COCO - can significantly improve the model's performance, particularly in challenging perception scenarios that involve disturbances such as environmental changes, occlusions, or complex interactions between objects.

2.3.2 Sensor Fusion Basics

LIDAR enhances the overall perception capabilities of robotic systems by providing accurate depth data, which supports 3D mapping applications and spatial awareness beyond what camera systems alone can achieve. The high-resolution point clouds produced by these sensors offer a detailed representation of the robot's environment and enable the detection of obstacles at greater distances than cameras can achieve. For example, while a LIDAR sensor can precisely calculate the distance from an object, a camera may encounter difficulties with this task, especially in dimly lit areas.

For supporting outdoor autonomous navigation, it is important that robots have strong perception abilities in settings. To overcome the limits of individual sensors and produce an accurate representation of the environment, it is possible to integrate data acquired from different sensors, including cameras, LIDAR, and radar. Each sensor has its own advantage: radar performs best in rough weather such as fog or rain conditions, LIDAR can provide accurate depth and distance data while cameras provide color and texture information. This technique improves object detection, obstacle avoidance, and navigation capabilities in outside environments which are often dynamic and unpredictable.

$$F_{fusion} = \phi(F_{visual}, F_{lidar}, F_{radar}) \tag{2.1}$$

In this equation, F_{fusion} represents the output fused feature set, created using the fusion function ϕ, while F_{visual}, F_{lidar} and F_{radar} are the feature sets coming from multiple sensors (camera, LiDAR, radar). In this document are presented two of the main sensor fusion techniques:

- **Early Fusion**: Obtained by combining raw data from the RGB camera (I_{rgb}), LiDAR point cloud (P_{lidar}), and radar signal (S_{radar}) sensors into a single unified input, thus enabling a neural network (CNN) using the sensor data to learn and infer a joint feature representations (F_{early}).

$$F_{early} = \text{CNN}([I_{rgb}||P_{lidar}||S_{radar}]) \tag{2.2}$$

- **Late Fusion**: This technique first processes the data of each sensor (D_{visual} for camera, D_{lidar} for LiDAR and D_{radar} for radar data) individually and then combines their outputs (e.g., detection scores or feature maps) at a later stage (D_{visual}). This approach enables the possibility of weighting each modality through the parameters w_1, w_2, and w_3, thus it is possible to conclude that this approach adjusts for unreliable inputs by giving the opportunity to weight their importance.

$$F_{late} = w_1 D_{visual} + w_2 D_{lidar} + w_3 D_{radar} \tag{2.3}$$

2.4 Addressing perception challenges in common scenarios

Autonomous wheeled robots are increasingly being adopted in various scenarios today. Thanks to the presence of LiDAR and other sensors as shown in Fig. 2.2 and object detection technologies which are based on camera data, robots are in fact able to detect and classify elements present in outdoor scenarios with ever more accuracy in multiple and variegated settings.

In a scenario involving forest or wild path navigation computer vision models trained on datasets including natural elements such as particular trees - as shown in Fig. 2.3 and Fig. 2.4 -, obstacles such as rocks or small vegetation support the robot in identifying obstacles, while LIDAR readings can support the robot in estimating precisely the distance from these obstacles allowing for their avoidance in dense vegetation.

When a robot operates in urban environments, its perception capabilities can be used to recognize elements of the road (e.g. signs, vehicles and road users) using LIDAR readings for executing distance measurements that are used to calculate path and speed adjustments during navigation.

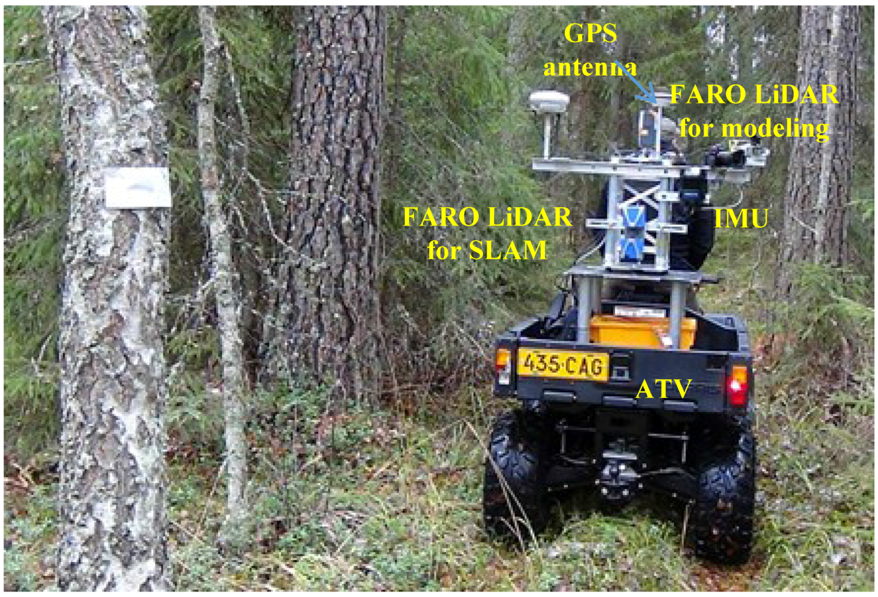


Figure 2.2: Robot equipped with sensors. [30]

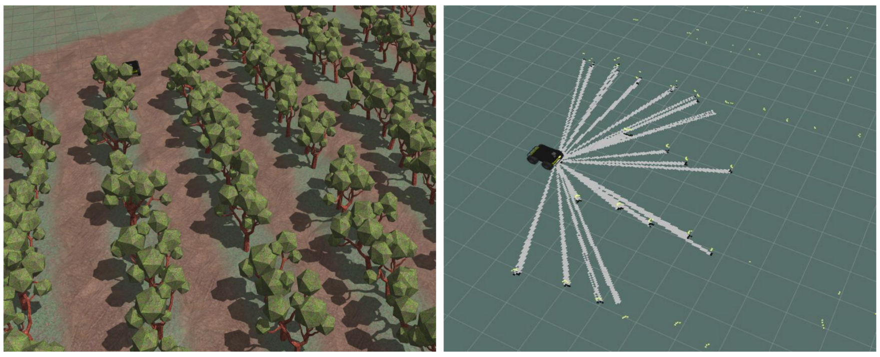


Figure 2.3: Forest map of individual trees captured using LIDAR. [2]

In agricultural applications, another common field of application as described in [16], the robot's perception capabilities enable the automated detection of crop rows or produce (Fig. 2.5) and farming systems using the robot cameras coupled with the appropriately trained detection algorithms while LIDAR technology can assist the robot in terrain mapping operations and provide detailed depth information. Irregular ground can in fact make the robot stuck and incorrect distance estimation from the plant may cause damage during harvesting operations (Fig. 2.6).

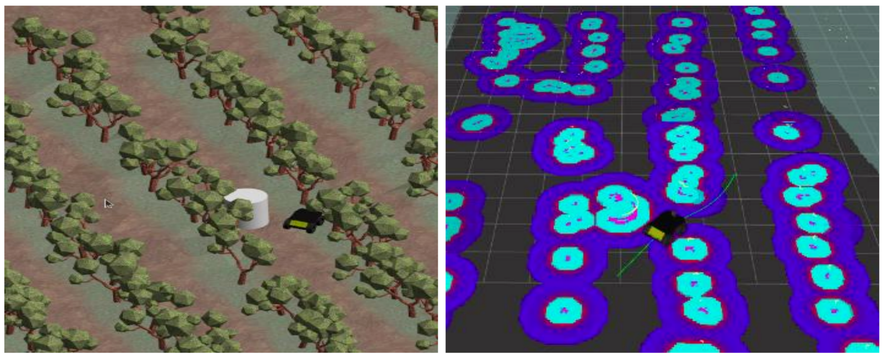


Figure 2.4: Robot moving between two trees. [2]

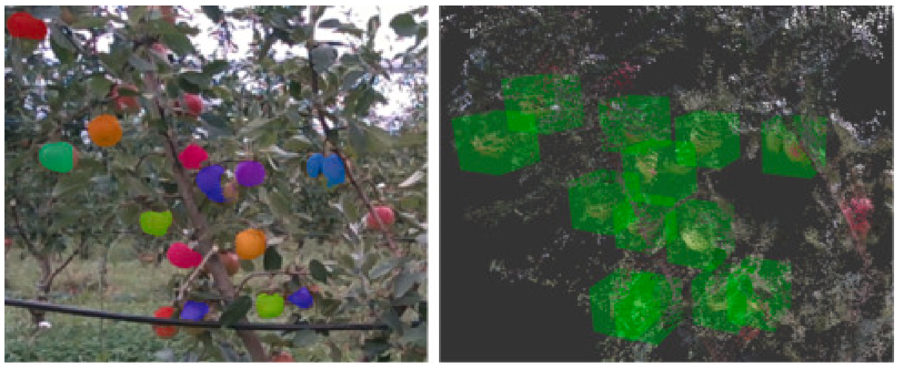


Figure 2.5: Fruit detection and segmentation using camera and LIDAR [16]

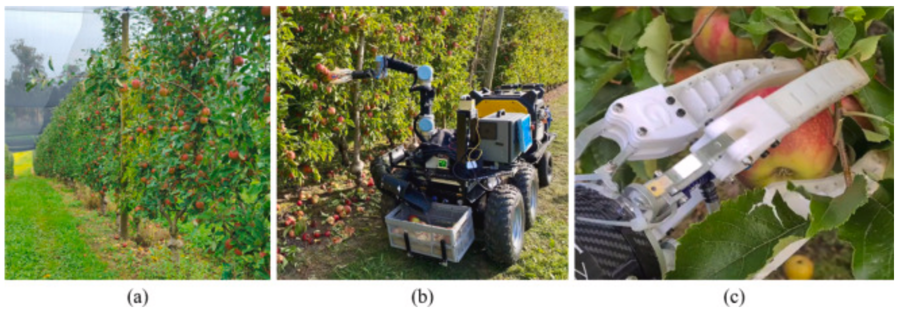


Figure 2.6: Fruit detection (a) and grasping (b, c) [16]

Chapter 3

Software

This chapter first introduces an overview of the YOLO algorithm for object detection (Sec. 3.1). It then describes the core aspects of ROS 2 (Sec. 3.2), the upgraded edition of the Robotic Operating System (ROS), used in this thesis to develop algorithms and deploy them in the simulation used to test the system. This section begins by explaining ROS concepts to understand the principles of the framework and then addresses the improvements that ROS 2 offers compared to ROS 1. It then proceeds to explain how camera and LiDAR sensors are integrated within ROS 2, as well as third party libraries.

Section 3.3 of the document explores the ROS 2 Navigation Stack, an improvement of the ROS Navigation Stack, that enables robots to navigate autonomously.

The final section (Sec. 3.4) illustrates the simulation and visualization tools used in this work.

3.1 YOLO algorithm

The YOLO (You Only Look Once) [14] family was first introduced in 2015 [22] and represent the state of art real-time object detection, since it can detect objects at first glance, performing detection and classification simultaneously, making it especially useful in robotics. To do its job, YOLO employs a technique that divides the image into a grid; each cell in this grid has the goal to predict bounding box coordinates, the probability of an object being present in that area, and the object's class, such as a person, car, or animal (Fig. 3.1).

Among the features that make YOLO the best among real-time object detectors are its speed and accuracy, which allow it to perform well even in complex situations. However, it does face challenges when it comes to detecting small or overlapping objects. In the field of robotics, YOLO is mainly used for the recognition of dynamic obstacles, to improve the autonomous driving ability of mobile robots.

In the context of this thesis, YOLOv8s was used, where “s” indicates the “small” version of the model: this version was chosen because, despite its light weight, it can guarantee good results even on computers with limited resources, such as those without a dedicated GPU as in the case of the NUC used for this project.

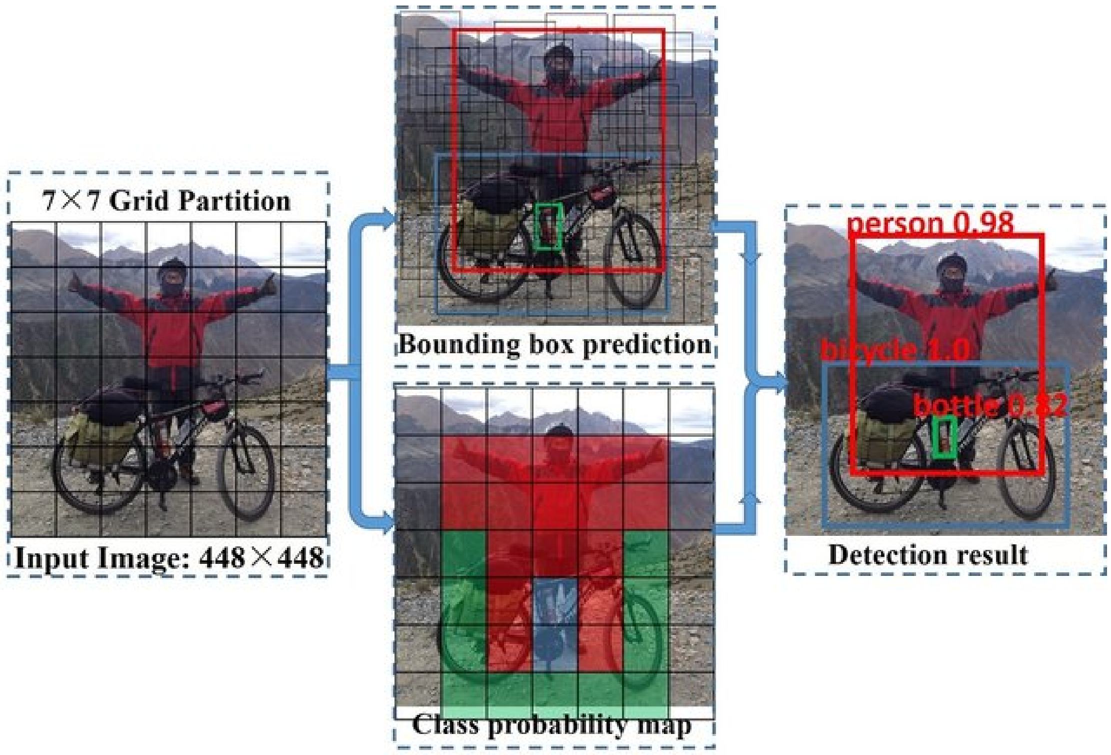


Figure 3.1: YOLO grid and prediction [38].

3.2 ROS2

3.2.1 Architecture of ROS 2 and Advantages over ROS

ROS (Robot Operating System) is an open-source framework, which was originally developed in 2006 at Stanford University [32], that offers libraries and tools to help software developers create robot applications. It is made to work with a variety of software and hardware platforms, such as sensors or actuators, making it easier to integrate various robotic components.

Despite its name, ROS is a framework that runs on an operating system (usually Linux) and not an operating system itself. It serves as a middleware and it allows different parts of robotic systems to communicate with each another. A list of important ROS terms and architecture components is provided below:

- **Nodes**: The fundamental units of ROS. Usually, each node handles a single task (such as processing sensor data or controlling motors). Nodes are created via ROS libraries compatible with C++ and Python, enabling them to transmit and/or receive data from other nodes. In ROS 1 a master node was present, which is deprecated in ROS 2.

- **Messages**: Data packets sent between nodes. Messages can be of various types; for example, they can include strings, numbers (such as integers or floating-ponts), booleans, but also arrays or other more complex structures. It is also possible to define custom messages through textual `.msg` files. Nodes communicate by publishing or subscribing to topics, which carry these messages.

- **Topics**: Channels that nodes use to exchange messages are called topics. A publisher node publishes a message over a topic, and all nodes that are subscribed to that topic get it; these nodes are called subscribers.

- **Services**: Services are another method of communication between nodes. Contrary to topics, services allow for synchronous Request/Response communication between nodes. A service server node only reacts in response to a request from a service client node, which is able to both submit and receive requests. The connection between two nodes is severed once the service's request and answer are finished.

- **Actions**: Like services, action clients send a task to an action server with a specific goal in mind, and they will receive a response. In contrast to services, an action server updates with continuous feedback the client on its progress as the action is being carried out and they are also preemptable.

- **Parameter Management**: Nodes can store and exchange variables, configuration settings, and other runtime parameters thanks to parameter management. The behavior of nodes and the robotic system as a whole can be modified and configured using parameters.

Developers can create their ROS 2 programs as needed thanks to the build mechanism. A key component of ROS 2 is the separation of code into packages, each of which contains a manifest file (`package.xml`) with important information about the package itself, such as its dependencies on other packages. The meta-build tool, which is `colcon`, cannot operate without this manifest.

A ROS workspace consists of various folders. The packages' source code is located in the source space (`src` subfolder). Cache files and other temporary data that refer to the build system are stored in the build space (`build` subdirectory). The installed targets are located in the install space (`install` subdirectory); this folder is essential in ROS 2 since it does not offer a development space, while ROS 1 does not require this space at all because packages can be produced without installing them. The utilities of the obsolete development space are likewise transferred to the install space in ROS 2. Finally, the logging data from console output during building is contained in the log space (`log` subfolder).

Differences between ROS 1 and ROS 2

There are two main versions of ROS, ROS 1 and ROS 2. The first commits to the ROS 2 repository were made in 2015 while the first release traces back to the end of 2017. It addresses some concerns that were left unresolved in ROS 1, such as security, real-time issues, communication between nodes and so on.

The primary distinction is that ROS 1 uses a client-server architecture in which the ROS Master interfaces with every other node, whereas ROS 2 lacks master and slave nodes but instead has a peer-to-peer infrastructure (Fig. 3.2, which makes ROS 2 decentralized and gives the possibility to develop distributed applications. The communication between nodes is based on DDS (Data Distribution Service) [21]. This upgrade allowed ROS 2 to solve one of the major limitations of the first version of the middleware, which could not operate on real-time embedded systems. Through the use of DDS, ROS 2 can ensure scalable, high-performance communications, without which distributed, real-time systems cannot function properly.

Another important feature introduced by ROS 2 are lifecycle nodes [26]. While regular nodes can only be active or inactive, lifecycle nodes are essentially nodes that can take on the following states, which are controlled by a finite state machine: unconfigured, inactive, active, and finalized (Fig. 3.3) . Using this feature allows to resolve real-time issues, since a life cycle ensures that every node has been properly instantiated prior to the application being run. Until any forms of communication are established, a node is unconfigured. After that, it becomes inactive, which means that it's ready to start its work but it's still inoperative. The node that transitions to active state processes data, publishes on topics, generates console output, and more when it's in operation. The last step is the finalized state, before possibly being destroyed.

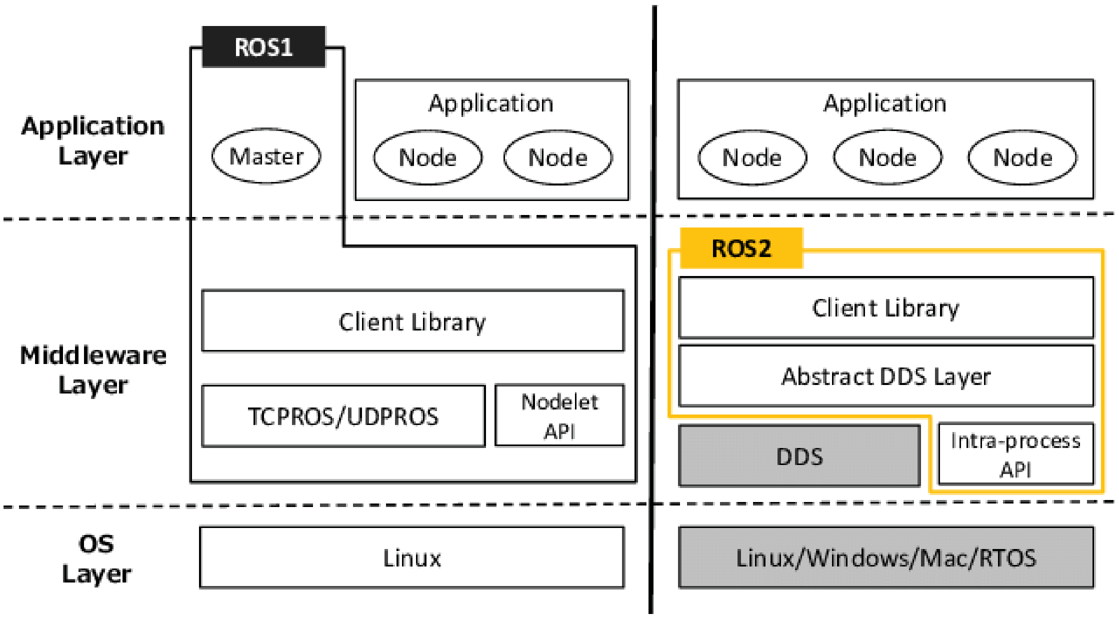


Figure 3.2: Differences between ROS 1 and ROS 2 architectures.

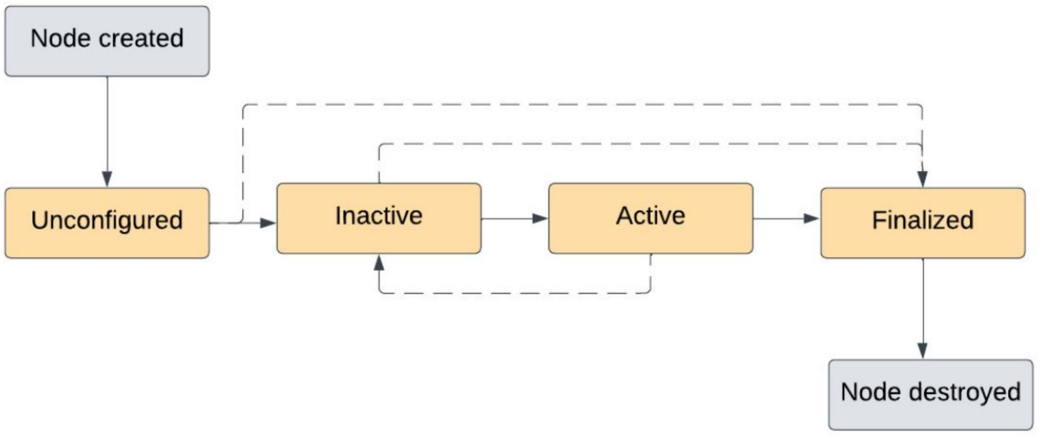


Figure 3.3: Lifecycle nodes state machine. [3]

3.2.2 Interfacing ROS 2 with Sensor Systems: Camera and Lidar

In order to construct reliable navigation systems for autonomous wheeled mobile robots it is necessary to integrate many sensor systems. Cameras and LiDAR are two of the most important sensors for precise navigation and object detection. Interfacing with these sensors in ROS 2 needs a good comprehension of the data structures used for the processing and combination of the sensory data in real time to perform decisions, and in the communication paradigms that have been utilized by the framework.

Overview of Camera and LiDAR Sensors in Robotic Applications

Cameras are sensors that allow to capture static images or video through RGB images and, in some cases, can measure the distance of objects; these cameras are known as depth cameras and are typically more expensive. Thanks to this data, a robot can recognize objects in its path, which is useful for object recognition and tracking. Integrating ROS 2 with object identification techniques, like those based on YOLO (You Only Look Once), improve the robot's perception of its surroundings.

On the other hand, LiDAR (Light Detection and Ranging) sensors use laser pulses to measure the distances of nearby objects within a certain range depending on its capacities and produce high-resolution 2D or 3D maps of the surroundings. LiDAR is an essential tool for path planning, SLAM (Simultaneous Localization and Mapping) and collision avoidance because of its ability in identifying obstacles and mapping out outside terrain.

ROS 2 Sensor Integration Architecture

Camera Integration: Incorporated as nodes, cameras in ROS 2 systems record and publish picture data to designated topics. Other nodes that implement tasks like data fusion, image processing, or visualization can subscribe to these topics.

Typical ROS 2 Camera Topics:

- `/camera/color/image_raw`: publishes the camera sensor's raw image feed.
- `/camera/camera_info`: offers camera calibration information, including inherent parameters like distortion coefficients and focal length.

To interface a camera with ROS 2, a driver package that serves as a link between the hardware and the ROS 2 framework is required. The type of camera and its connectivity determine which open-source programs are available: one of the most useful is `usb_cam` package, that is compatible with USB cameras of any kind.

LiDAR Integration: the typical LiDAR Topics in ROS 2 are based on the LiDAR type:

- **2D LiDAR**: Data is published on topics such as `/scan` using the `sensor_msgs/LaserScan` message.

- **3D LiDAR**: Data is published on topics like `/pointcloud` using `sensor_msgs/PointCloud2`.

To integrate LiDAR with ROS 2, it is necessary to install and configure drivers compatible with the sensor model. Some common packages and drivers for several LiDAR sensor types are shown below:

- `rplidar_ros2` [27]: for low-cost 2D LiDAR sensors, like the RPLIDAR A1 or A2.
- `velodyne_pointcloud` [12]: ideal for high-resolution 3D LiDAR sensors from Velodyne, used in applications requiring detailed 3D mapping.

The creation of data fusion techniques to improve navigation is made possible by integrating ROS 2 with both cameras and LiDAR. One method is to use nodes that subscribe to both image and LiDAR topics and use algorithms to combine data for path planning and obstacle identification. LiDAR and camera data can be aligned using the `sensor_fusion` library or custom nodes by using the `tf2` library [11] for spatial calibration and time synchronization.

3.2.3 Integrating ROS 2 with Third-Party Libraries (e.g., YOLO, OpenCV)

The capability of autonomous robotic systems is improved by the integration of third-party libraries with ROS 2, enabling better perception, control, and data processing capabilities.

ROS 2 nodes can import and utilize OpenCV (Open Computer Vision Library) [5] directly. OpenCV is the world's biggest open-source computer vision library that enables image processing, making it possible to do a variety of picture preparation operations (such as scaling, filtering, and color conversions) prior to transmitting data to YOLO or other processing nodes.

On the other hand, YOLO has to be installed using Python packages like `torch`, or it is possible to use pre-trained YOLO models on various dataset through the Ultralytics library [34].

3.3 Navigation2

Navigation2 stack is a crucial component of the ROS 2 framework and was created to solve the problem of enabling autonomous robots movement in a range of uncertain scenarios. Nav2 is essential to ROS 2's ecosystem and consists of an vast collection of packages, plugins, and libraries that enable real-time movement, sensing, and response by robots. Its important features include path planning, obstacle avoidance, costmap creation, sensor data processing, and failure recovery behaviors. Along with tools for loading and storing maps, Nav2 also provides a localization capability called Adaptive Monte-Carlo Localization (AMCL) [8], which is a more sophisticated version of the Monte-Carlo scan matcher used in some particle-based SLAM algorithms.

3.3.1 Structure of the Nav2 Stack

Nav2's design is modular and consists of a number of key parts that cooperate to allow autonomous navigation. A flexible and scalable system is made possible by the distinct tasks that each component is intended to do. The principal elements are provided below.

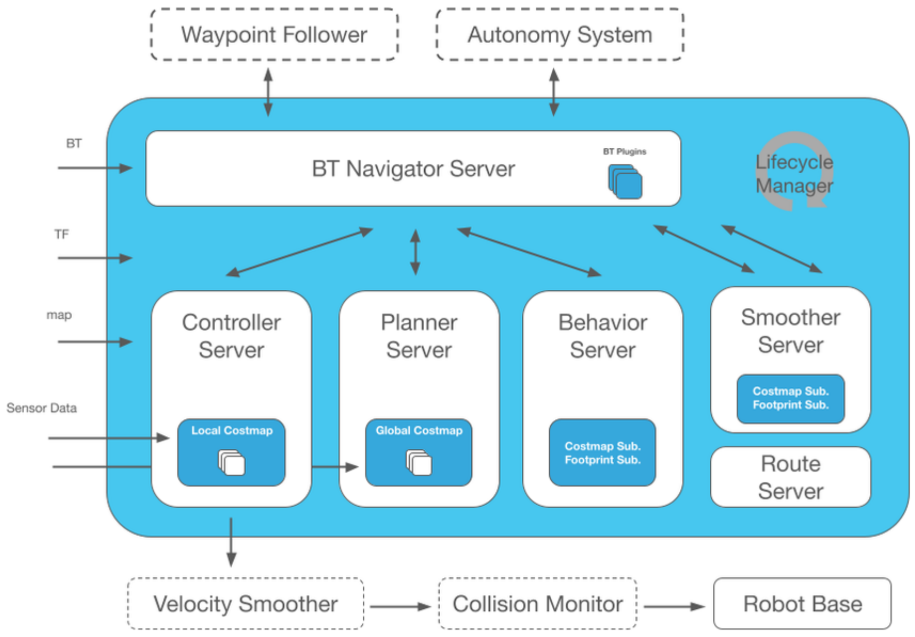


Figure 3.4: Nav2 architecture [9].

Planner Server

The planner server's main job is to compute the best path for directing a robot across a complex environment to a predetermined destination by taking into account the robot's current location and its target position in terms of coordinates. Depending on how the planners are configured, the routes may follow sparse or predetermined routes, the shortest way, or a full coverage path. Enabling safe, effective, and intelligent navigation for robots is the aim of all routes.

Controller Server

The controller server, formerly referred to as local planners in ROS 1, is in charge of making sure that the navigation routes created by the planner server are executed precisely. The controller server converts the high-level navigation paths that are calculated by the planner server into a set of control commands that direct the robot's movements. Its main duty is to oversee the implementation of these plans, modifying the robot's course as it moves through the demanding and dynamic environment.

Behaviour Server

Robot activities can be managed dynamically thanks to the behavior server's integration with the behavior trees [4]. Managing recovery behaviors is one of the behavior server's key functions: robots may run into unknown or malfunctioning conditions while navigating, and the recovery behavior server is made to handle these situations. Dynamic objects, transient obstructions, or things that weren't initially shown on the navigation map could all be considered barriers. The recovery server handles the unexpected difficulties to guarantee that the robot can recover gracefully, while the planners and controllers concentrate on directing the robot through a known and expected environment. This could entail moving from a bad spot into open space, trying a different route, or even backing up or spinning in place. In Fig. 3.5 there is a behavior tree that models the search and grasp plan of a two-armed robot [36].

Smoother Server

A smoother server's primary responsibility is to provide an improved version of the path received from the planner server. By taking into account variables like kinematics and acceleration that could affect the robot's movement, it focuses on enhancing the path's quality. One of the smoother server's most important roles is to address any problems with sudden movements that could occur when navigating. It seeks to reduce these abrupt shifts in speed or direction and to go farther away from costly locations and obstructions.

Waypoint Follower

Waypoint follower is a type of navigation that allows a path to be constructed by giving the robot a set of points to follow. It is usually used in conjunction with

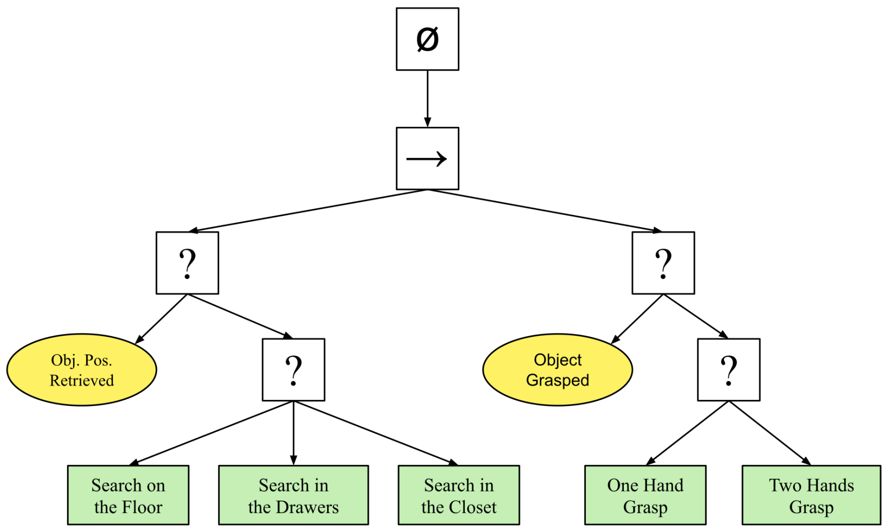


Figure 3.5: Example of a Behaviour Tree [36].

the other planners.

3.3.2 Implementation of Costmap and Obstacle Management

The costmap is a fundamental component in Nav2's navigation system, and it's essential in enabling the robot to perceive and react to its surrounding environment. Costmaps are data structures that represent spatial information about obstacles and the traversability of the robot's surroundings. Nav2 utilizes both global and local costmaps for a better environmental awareness and real-time obstacle management. In Nav2 the implementation of the costmap representation is possible thanks to the `nav2_costmap_2d` package, that subscribes to the sensor data and builds a 2D or 3D space representation of the space in the form of an occupancy grid. The cells can assume an integer value between 0 and 255 depending on the sensor data and resolution parameters. Actually, only three values, free, unknown, and occupied, which are by default set to 0, 255, and 254 respectively, can really be represented by the underlying structure that is being used.

In contrast to traditional monolithic costmaps, which store all the data in a single grid of values, the costmap layers approach tracks a single type of constraint or obstacle and then modifies a master costmap that is used for path planning. This method performs far better than the monolithic costmap approach in dynamic, people-filled situations.

There are essentially three basic layers in Nav2:

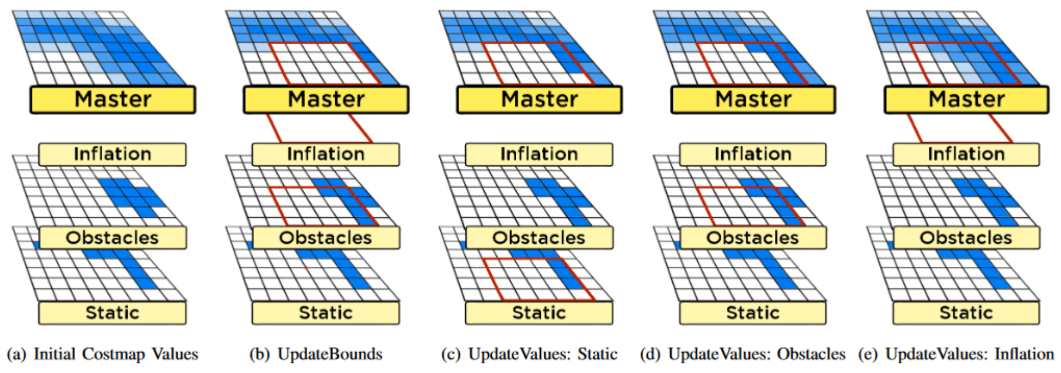

Master
Inflation
Obstacles
Static
(a) Initial Costmap Values
(b) UpdateBounds
(c) UpdateValues: Static
(d) UpdateValues: Obstacles
(e) UpdateValues: Inflation

Figure 3.6: Example of application of costmap layers [18].

- **Static layer**: contains the costs relative to the static map.
- **Obstacle layer**: the cells are continuously marked as free or occupied according to data from sensors.
- **Inflation layer**: provides a safety margin for the robot's navigation by propagating cost values out from occupied cells that decrease with distance.

3.3.3 Limitations of Nav2 in Dynamic Obstacle Handling

Although Nav2 is a quite a strong navigation framework for autonomous robots, it presents limitations when managing dynamic obstacles in complex environments. These restrictions could make it difficult to manage the robot's navigation in certain environments, like busy public areas or industrial settings.

Real-Time Perception and Processing Delays

To update costmaps and identify obstacles, Nav2 uses sensor data from cameras and LiDARs. The latency associated with sensing, processing, and updating costmaps can affect the robot's real-time reaction in situations with quickly changing variables. This may result in suboptimal path planning when barriers move faster than the costmap's update rate or in possible collisions in the event that the robot's local planner is unable to rapidly recalculate a safe path.

Predictive Limitations

Dynamic obstacles cannot be predicted by Nav2 in its basic configuration. The system is able to detect and avoid moving objects, but it is not able to forecast where they will be in the future. This restriction may be an issue when there are fast-moving cars or people who move irregularly, as reactive planning might not be enough to prevent crashes.

3.3.4 Dynamic Obstacle Layer, a NAV2 plug-in

To overcome the issue of dynamic obstacle handling, a previous thesis in collaboration with the LINKS Foundation focused on creating a plugin for Nav2 [7], [28]. This plugin introduced the Dynamic Obstacle Layer (DOL), a new layer used to identify and track moving obstacles. LiDAR data are integrated into the local cost map, that dynamically marks cells as occupied or vacant.

DOL is able to distinguish between static and dynamic obstacles. To do so, it treats the cost map as an image and uses image processing algorithms to determine dynamic obstacles from static ones. The static background is substracted from the cost map image using moving average filters, leaving only the pixels that correspond to moving objects. Next, the detected pixels are grouped into distinct blobs to represent individual dynamic obstacles. To ensure safe navigation, the motion of dynamic obstacles is tracked in real time using a Kalman filter, which estimates the direction and speed of each obstacle. This data is then used to update the local cost map by assigning a 2D Gaussian shape around the obstacles. The size and intensity of the Gaussian are directly proportional to the velocity of the obstacle: obstacles that are faster or approaching the robot receive more inflation, so a higher avoidance priority. This management of LiDAR data allows the local planner to account for both the position and dynamics of obstacles, improving the robot's ability to navigate complex and variable environments.

3.4 Simulation and Visualization Tools

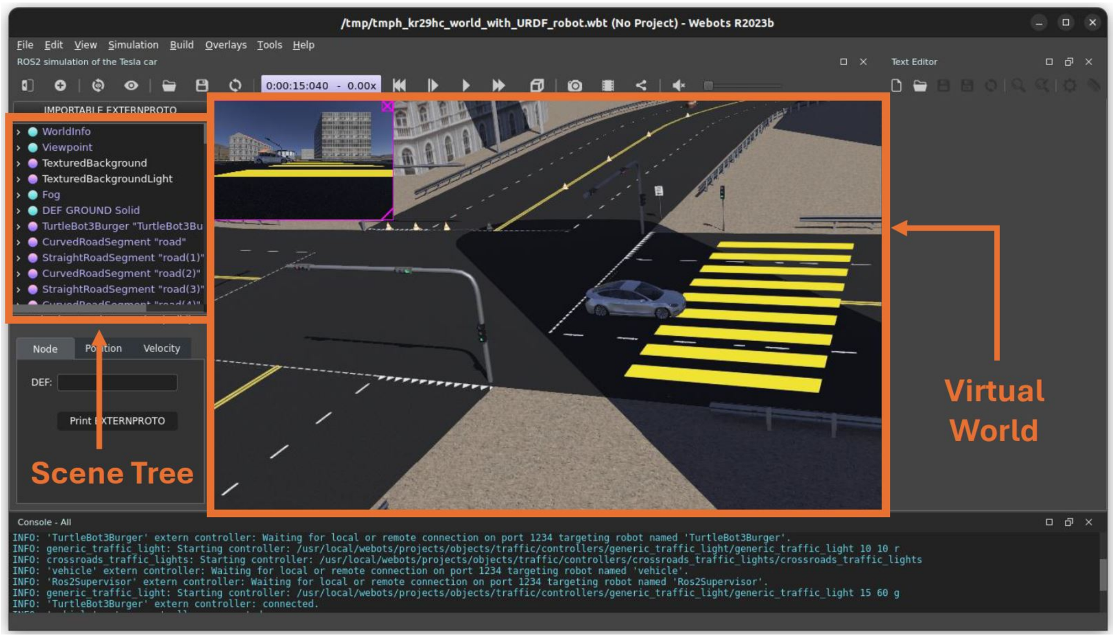
3.4.1 Webots

Simulations play a central role in the development and validation process of complex robotic systems, especially in situations where field tests may be costly, dangerous or logistically difficult. Simulations are an appropriate method for testing system performance: they can be used to quickly and repeatedly run different simulations of system behaviour after changes in design, algorithms or control strategy.

For this thesis work, an extensive use of simulation has been made, in order to develop and validate the sensory fusion algorithm for real-time obstacle classification and tracking. For this purpose, Webots was chosen. It is a robotic simulator developed by Cyberbotics [6], that allows users to model mobile robots or manipulators, simulate and test their devices in 3D virtual worlds (Fig. 3.7).

It is very versatile, since it combines a user friendly interface with physics properties (mass, friction, etc) for simulations. Users can design complete 3D scenarios, including obstacles, robots and objects. Environments can be created from scratch or by using predetermined models from Webots' large library.

To run a simulation, a Webots world file is required. It includes a description of each object, specifying details such as position, orientation, appearance and physical properties. Worlds follow a hierarchical structure, known as Scene Trees, in which objects may contain other objects: for example, a mobile robot may include a number of sensors, such as LiDAR and a camera, which are considered child elements of the robot itself. The robot or its sensors can also be imported from a PROTO file, that is an external file that can be reused in multiple worlds. A controller is used to describe the behaviour of a robot, which can be developed using a choice of programming languages such as C, C++ or Python.


/tmp/tmph_kr29hc_world_with_URDF_robot.wbt (No Project) - Webots R2023b
File Edit View Simulation Build Overlays Tools Help
ROS2 simulation of the Tesla car
Text Editor
0:00:15:040 - 0.00x
IMPORTABLE EXTERNPROTO
WorldInfo
Viewpoint
TexturedBackground
TexturedBackgroundLight
Fog
DEF GROUND Solid
TurtleBot3Burger "TurtleBot3Bu
CurvedRoadSegment "road"
StraightRoadSegment "road(1)"
CurvedRoadSegment "road(2)"
StraightRoadSegment "road(3)"
Node
Position
Velocity
DEF:
Print EXTERNPROTO
Scene Tree
Virtual World
Console - All

```
INFO: 'TurtleBot3Burger' extern controller: Waiting for local or remote connection on port 1234 targeting robot named 'TurtleBot3Burger'.
INFO: generic_traffic_light: Starting controller: /usr/local/webots/projects/objects/traffic/controllers/generic_traffic_light/generic_traffic_light 10 10 r
INFO: crossroads_traffic_lights: Starting controller: /usr/local/webots/projects/objects/traffic/controllers/crossroads_traffic_lights/crossroads_traffic_lights
INFO: 'vehicle' extern controller: Waiting for local or remote connection on port 1234 targeting robot named 'vehicle'.
INFO: 'Ros2Supervisor' extern controller: Waiting for local or remote connection on port 1234 targeting robot named 'Ros2Supervisor'.
INFO: generic_traffic_light: Starting controller: /usr/local/webots/projects/objects/traffic/controllers/generic_traffic_light/generic_traffic_light 15 60 g
INFO: 'TurtleBot3Burger' extern controller: connected.
```

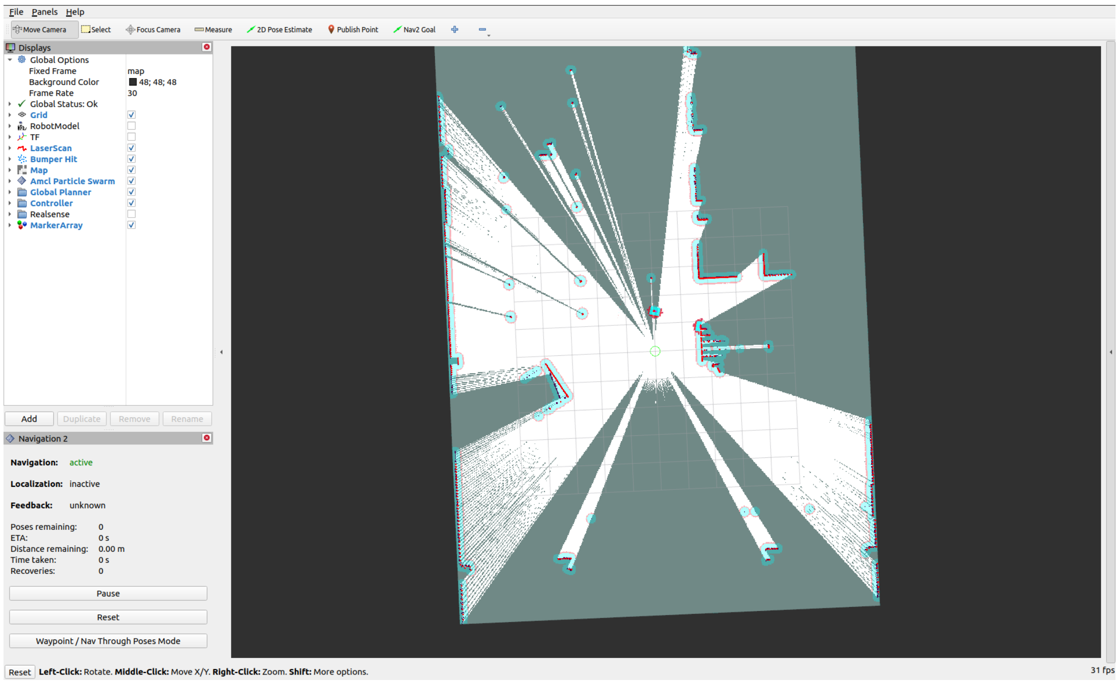
Figure 3.7: Webots simulation environment and its interface

3.4.2 RViz2

RViz2 (Robot Visualization 2) [13], the evolution of RViz, is a 3D visualization tool included in ROS 2, designed to display sensor data, robot status, and navigation information in a three-dimensional environment. RViz2 is essential for developers that work with autonomous robots, as it allows them to visualize data from sensors such as LiDAR and cameras, but also to load the URDF model of the robot to visualize its structure and follow its movement (Fig. 3.8).

RViz2 can be configured to receive data from ROS 2 topics published by the robot nodes, such as location, LiDAR data, or images from a camera.

For all these reasons, it is used to test robotic systems in simulation, using simulators like Gazebo or Webots, but also to monitor robots in the real world.



Global Options
Fixed Frame map
Background Color ■ 48; 48; 48
Frame Rate 30
✓ Global Status: Ok
Grid ✓
RobotModel ☐
TF ☐
LaserScan ✓
Bumper Hit ✓
Map ✓
Amcl Particle Swarm ✓
Global Planner ✓
Controller ✓
Realsense ☐
MarkerArray ✓

Navigation: active
Localization: inactive
Feedback: unknown

Poses remaining: 0
ETA: 0 s
Distance remaining: 0.00 m
Time taken: 0 s
Recoveries: 0

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:** Zoom. **Shift:** More options.

Figure 3.8: Example of a Rviz2 configuration.[33]

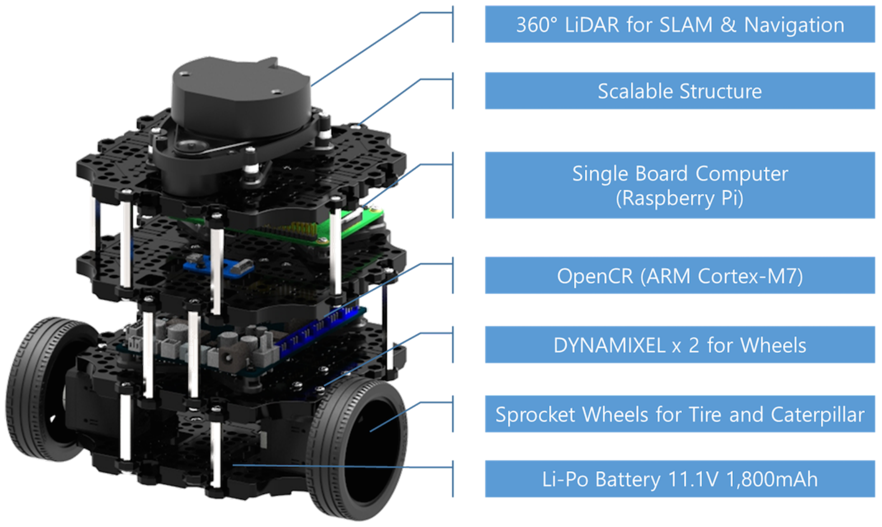
Chapter 4

Hardware

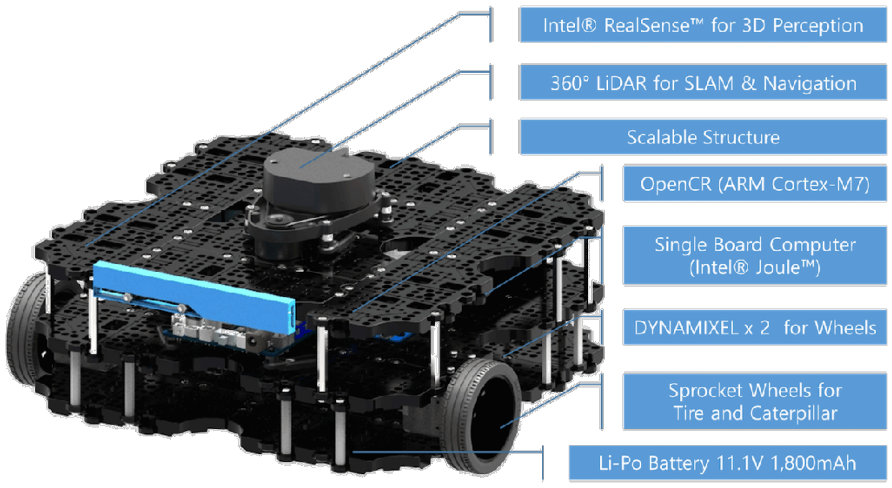
4.1 Turtlebot3

TurtleBot3 is a small, affordable, programmable, ROS-based mobile robot that can be used for product prototyping, research, education, and hobbies. TurtleBot3's objective is to reduce the platform's size and cost without compromising its functionality while still allowing for expansion. Depending on how the mechanical components are rebuilt and optional components like the computer and sensor are used, the TurtleBot3 can be modified in a number of ways. Additionally, TurtleBot3 has developed a compact, affordable SBC (single Board Computer) that works well with 3D printing technology, a 360-degree distance sensor, and a powerful embedded system. The TurtleBot can run SLAM (Simultaneous Localization and Mapping) algorithms to build a map by its own, making it a good choice for home service robots.

There are two different models (Fig. 4.1), Burger and Waffle; for the simulations, the TurtleBot3 Burger model was used.


360° LiDAR for SLAM & Navigation
Scalable Structure
Single Board Computer (Raspberry Pi)
OpenCR (ARM Cortex-M7)
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery 11.1V 1,800mAh

(a) TurtleBot3 Burger.


Intel® RealSense™ for 3D Perception
360° LiDAR for SLAM & Navigation
Scalable Structure
OpenCR (ARM Cortex-M7)
Single Board Computer (Intel® Joule™)
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery 11.1V 1,800mAh

(b) TurtleBot3 Waffle.

Figure 4.1: Models of TurtleBot3 [24].

4.1.1 General Technical Specification

Below, Table 4.1 shows the technical specification of the TurtleBot3 models.

Items	**Burger**	**Waffle**
Maximum translational velocity	0.22 m/s	0.26 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)	1.82 rad/s (104.27 deg/s)
Maximum payload	15kg	30kg
Size (L x W x H)	138mm x 178mm x 192mm	281mm x 306mm x 141mm
Weight (+ SBC + Battery + Sensors)	1kg	1.8kg
Threshold of climbing	10 mm or lower	10 mm or lower
Expected operating time	2h 30m	2h
SBC (Single Board Computers)	Raspberry Pi	Raspberry Pi
MCU	32-bit ARM Cortex®-M7	32-bit ARM Cortex®-M7

Table 4.1: Comparison of Burger and Waffle specifications.

The RPLiDAR A2 was used during simulation; the main specification are reported in Table 4.2.

Property	**Value**
Name	RPlidar A2
Horizontal Resolution	800
Field of View	360°
Maximum Range	12
Minimum Range	0.2
Rotational Speed	10Hz (5Hz-15Hz)

Table 4.2: Specifications of the RPlidar A2.

As for the camera, a generic camera with FOV 110° and 720x480 pixel resolution was used for simulations.

4.2 AgileX Scout 2.0

The AgileX Scout 2.0 [25] is a compact, all-terrain unmanned ground vehicle (UGV) designed for research, development, and industrial applications. It features a four-wheel-drive system with independent suspension, enabling it to navigate various terrains and obstacles up to 10 cm in height. Each wheel is powered by a 400W brushless servo motor, providing robust mobility and a maximum payload capacity of 50 kg.

In addition to supporting various types of sensors, Scout 2.0 is compatible with ROS 2, making it an excellent mobile robot for use in research.

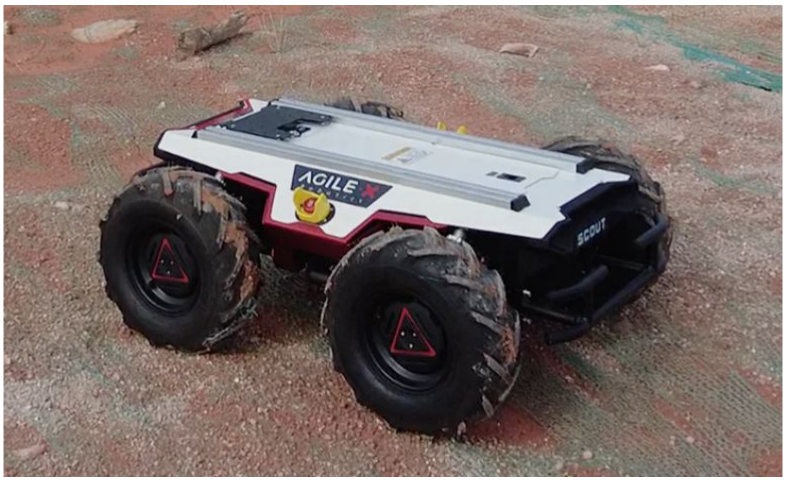


Figure 4.2: AgileX Scout 2.0

4.2.1 General Technical Specification

The list of the main features of the Scout 2.0 si reported in Table 4.3.

Dimensions	930L x 699W x 349H mm
Weight	68kg
Payload Capacity	50kg
Top Speed	1.5km/h
Runtime	Up to 8h
Charging Time	3.5h (30Ah) / 7h (60Ah)
Max Climbing Grade	30° (full payload)
Drive Motor	2500 Lines Magnetic Incremental Encoder
Battery	24V 30Ah (24V 60Ah optional) Lithium
Communication Interface	CAN Bus
Steering	Differential

Table 4.3: Specifications of Agilex Scout 2.0.

4.3 Intel NUC

NUCs (Next Unit of Computing) are mini PCs developed by Intel, compact in size (about 12x12 cm), chosen for this thesis development after the personal PC proved insufficient to handle a dual boot system or a virtual machine needed to run Ubuntu. Table 4.4 shows the specifications of the NUC used, courtesy of the LINKS Foundation.

Feature	**Details**
Version	Intel(R) Core(TM) i7-10710U CPU @ 1.10GHz
Manufacturer	Intel(R) Corporation
Family	Core i7
Type	Central Processor
External Clock	100 MHz
Max Speed	4700 MHz
Cache Handle	L1: 0x0043, L2: 0x0044, L3: 0x0045
RAM Memory	32 GB DDR4 (2 x 16 GB Kingston)
Core/Thread Count	6 Core, 12 Thread
Storage	Samsung SSD 970 EVO Plus 1TB
Graphics Card	Intel Corporation Comet Lake UHD Graphics (integrated GPU)
Operating System	Ubuntu 22.04.5 LTS (Jammy)
Characteristics	64-bit capable, Multi-Core, Hardware Thread, Execute Protection, Enhanced Virtualization, Power/Performance Control

Table 4.4: NUC Technical Specifications.

Chapter 5

Design and Implementation

5.1 Introduction

This work builds upon a pre-existing thesis in LINKS foundation, introduced in Sec. 3.3.4. That thesis used LiDAR for dynamic obstacle detection and tracking and integrated this information within Nav2 using a new costmap layer.

It was decided to not modify that work, but instead expand it using a modular architecture based on ROS 2.

This decision was based on some considerations.

First, the pre-existing system had proved to be robust in handling LiDAR-based dynamic obstacle detection, and modifications could introduce side effects or reduce its stability.

Second, a modular approach enables independent adding of new features without compromising the integrity of the existing work and maintaining a clear separation between components. This also facilitates future extensions without requiring significant rework. This approach allows to add features to the preexisting system, while minimizing changes and allowing for future updates.

The software architecture includes the following functional elements:

- LiDAR detects moving obstacles in the proximity of the robot (Sec. 3.3.4). The obstacles positions are incorporated into the local costmap.

- To classify the obstacles, the YOLO neural network analyzes captured images from the robot's camera. After creating the bounding boxes, the obstacles' angular position, with respect to the robot, is computed (Sec. 5.2.1).

- The detections from LiDAR are transformed into the robot's frame (Sec. 5.2.3).

- A comparison is made between the angular positions given by the camera and those calculated from the LiDAR's detections, ensuring consistency between the two sources of information. Each moving obstacle is then assigned its corresponding label (Sec. 5.2.4).
- The new data is used in the navigation stack and can be used to make safety decisions for obstacle avoidance.

A simplified overview of the design is shown in Fig. 5.1

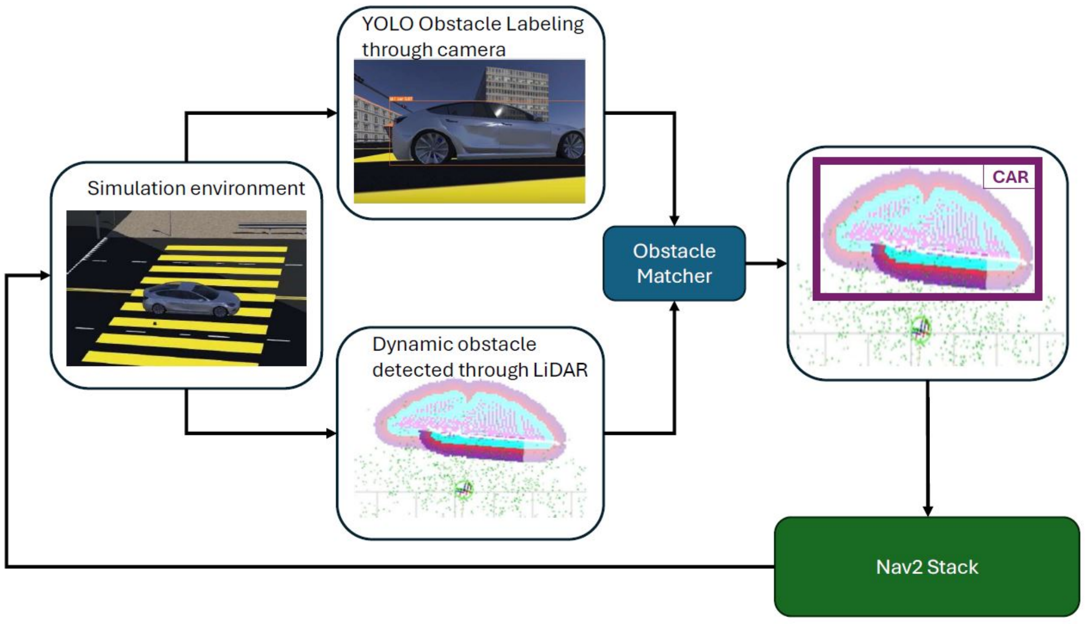

YOLO Obstacle Labeling through camera
Simulation environment
Obstacle Matcher
CAR
Dynamic obstacle detected through LiDAR
Nav2 Stack

Figure 5.1: Overall design of the implemented work

Finally, Sec. 5.3 illustrates the software implementation based on ROS 2 middleware, detailing the code structure in the workspaces, messages exchanged, topics and nodes involved.

5.2 Methodology for LiDAR and Camera Integration in Obstacle Detection

The goal of this work is to classify dynamic obstacles on the roads, already identified using LiDAR and integrated into the local costmap 3.3.4, into specific categories. Since the final application is designed for urban environments, the target classes to be identified include cars, bicycles, pedestrians, and traffic lights, to allow the system to distinguish and respond properly. This will help the moving robot to perform more accurate decisions in real world scenarios: for example, distinguishing whether a car or a pedestrian is approaching could change how the robot will behave.

5.2.1 Calculating the Object Detection Angle with Camera

The camera perceives a portion of space through a 2D image. The area it can observe depends on the type of camera and is referred to as the Field of View (FOV). Using YOLO, identified objects are surrounded by rectangles known as bounding boxes. To determine the position of an obstacle relative to the robot, it is possible to calculate the angle between the center of the bounding box and the center of the image. Angle detection is useful to convert simple visual perception to an interpretation of the surrounding space of the robot.

The angle calculation is based on the pixel coordinates of the obstacle within the image captured by the camera. Knowing the FOV and resolution of the camera (Fig. 5.2), a geometric transformation is performed to convert the coordinates into an angle.

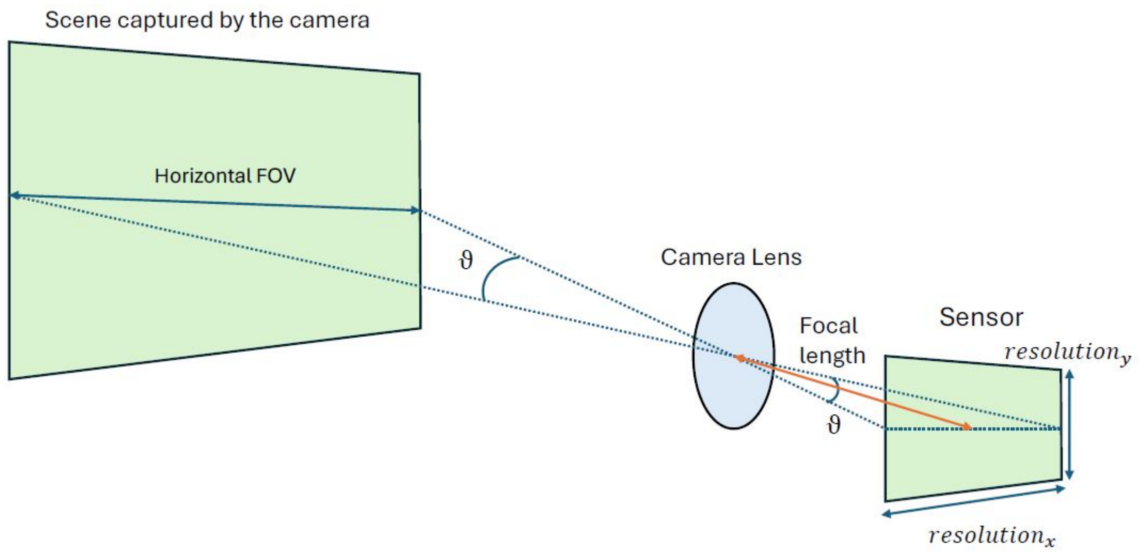


Figure 5.2: Optical Diagram of a Camera: Field of View, Focal Length, and Sensor Resolution

The implemented formula considers only the calculation of the horizontal angle, while the same considerations can be applied for the vertical angle.

The camera is aligned with the direction of movement of the robot and acquires a continuous stream of images. For each detected object, its position is identified by extracting the horizontal coordinate of the bounding box center in pixels. The YOLO model considers the upper left corner of the image as the center of its reference system, with the x-axis directed to the right and the y-axis directed downward. To convert the bounding box coordinates relative to the camera's central point, the following formula was used:

$$x_{\text{center}} = \frac{\text{res}_x}{2} \tag{5.1}$$

with res_x being the image width resolution of the camera and x_{center} the central point of the camera.

The coordinates of the center of the bounding box were then translated:

$$x_{\text{bb\_new}} = x_{\text{bb}} - x_{\text{center}} \tag{5.2}$$

where x_{bb} is the previous x coordinate of the center of the bounding box, while $x_{\text{bb\_new}}$ is the translated one.

Using simple trigonometric formulas, the virtual focal length f of the camera was computed using the horizontal field of view (FOV_h) as:

$$f = \frac{\text{res}_x}{2 \cdot \tan\left(\frac{\text{FOV}_h}{2}\right)} \tag{5.3}$$

allowing the obstacle angle θ_{cam} to be calculated through the final formula that uses only the two camera parameters:

$$\theta_{cam} = \arctan\left(\frac{x_{\text{bb\_new}}}{f}\right) = \arctan\left(\frac{x - \frac{\text{res}_x}{2}}{\frac{\text{res}_x}{2} \cdot \frac{1}{\tan\left(\frac{\text{FOV}_h}{2}\right)}}\right) \tag{5.4}$$

Using the angle as a spatial measure has the advantage that it does not require information about the absolute distance to the obstacle, which can only be obtained from much more expensive depth cameras - that is, cameras that can detect the distance at which an object is located - but it still provides accurate spatial information about the direction along which the obstacle is located.

5.2.2 Frame Analysis

In order to combine the dynamic obstacle information from the LiDAR with the camera identification, it is important to first consider the reference frames.

Obstacle Detection and Tracking with LiDAR

From a preexisting work [28], LiDAR is used not only to calculate the position of the center and the dimensions of dynamic obstacles, but also to track their velocity and maintain tracking even when they are no longer visible by the LiDAR, through the use of the Kalman filter and the Hungarian algorithm. The message that the LiDAR publishes is of type `Obstacle`, which is included in an array of obstacles using the `ObstacleArray` message. The position coordinates of the obstacles are referred to the map frame (Fig. 5.3).

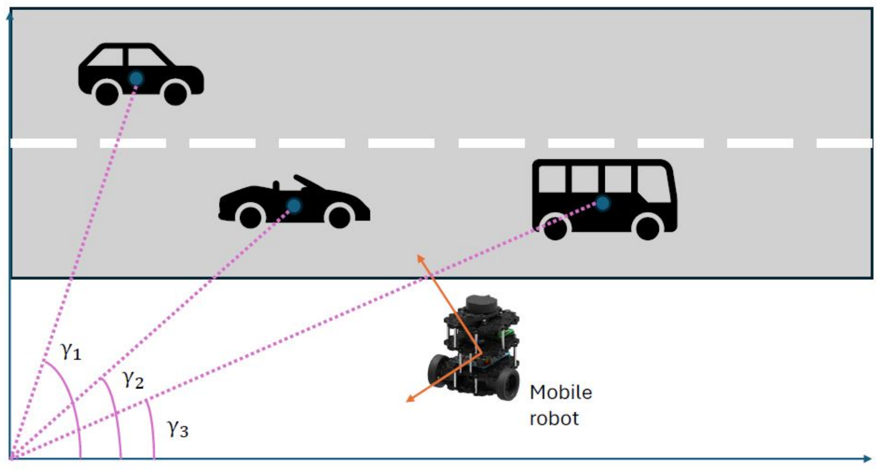


Figure 5.3: Integration of LiDAR detection into the map frame

Obstacle Detection with the Camera

The camera detects obstacles relative to the robot's frame, which moves, while LiDAR provides detections in the fixed map frame. To combine data from both sensors, their detections must be expressed in the same reference frame. This requires transforming the LiDAR's detections into the robot's frame, ensuring accurate matching of obstacles between the two sensors through a direct comparison of detections based on their angular positions.

5.2.3 Frame Transformation and Alignment

The available information for integrating sensor data includes:

1. The obstacle detected by the camera within a bounding box, whose center is identified by an angular position θ_{cam} relative to the robot's frame.

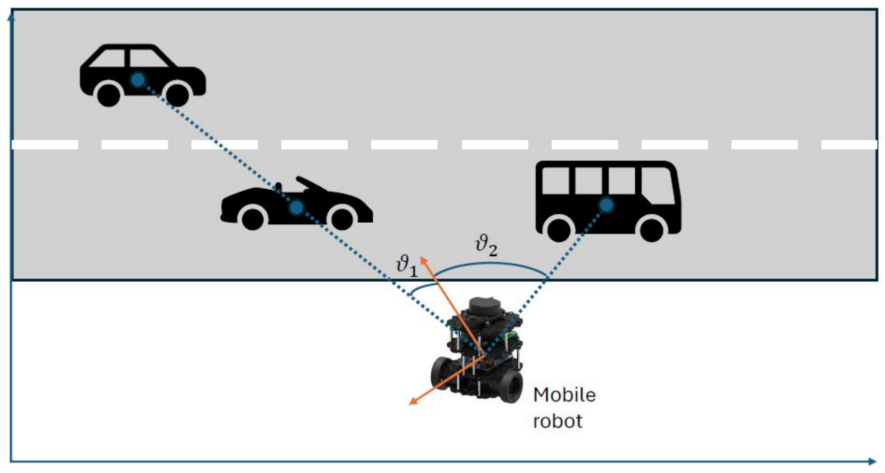


Figure 5.4: Obstacles detected by the camera in the camera frame

2. The coordinates of the obstacle's center identified by the 2D LiDAR with respect to the origin of the map, given as $(x_{\text{obs}}, y_{\text{obs}})$.
3. The robot's position and orientation relative to the map, expressed as a combination of translation $(x_{\text{robot}}, y_{\text{robot}}, z_{\text{robot}})$ and rotation, encoded as a quaternion $q = (q_w, q_x, q_y, q_z)$.

The aim is to use the second and third information to identify the angle between the direction of the robot and the obstacle, as seen by the LiDAR. This angle can be compared with the angle computed by the camera, given in the first point.

The `tf2` library [11] is used to keep track of the mutual position between reference frames (map, robot, and camera). This is a very powerful tool as it can track multiple frame coordinates that change over time and handle distributed systems. `Tf2` uses quaternions [39], extensions of complex numbers that offer a compact and unambiguous way to express the orientation of an object.

In the context of a mobile wheeled robot in urban environment, the only relevant orientation is around the z-axis. To have a more intuitive format, that is also compatible with the camera data, a conversion between quaternions and Roll, Pitch, Yaw (RPY) angles is used.

RPY angles describe how an object is oriented in three-dimensional space by performing three consecutive rotations around the main axes (x, y and z) in a defined order [29]: Roll (ϕ) around the x axis, Pitch (θ) around the y axis, and Yaw (ψ) around the z axis. The global rotation is thus obtained by composition of rotations defined with respect to a fixed frame. These angles are frequently utilized in fields like robotics, aeronautics, and navigation to provide a clear and understandable representation of orientation.

The robot's angular orientation is converted from its quaternion representation, as given by `tf2` library, to degrees, simplifying integration with the camera's angular detections.
The formula 5.5 is used for this conversion:

$$\begin{bmatrix}\phi\\ \theta\\ \psi\end{bmatrix} = \begin{bmatrix}\operatorname{atan2}(2(q_w q_x + q_y q_z), 1 - 2(q_x^2 + q_y^2))\\ -\pi/2 + 2\operatorname{atan2}\left(\sqrt{1 + 2(q_w q_y - q_x q_z)}, \sqrt{1 - 2(q_w q_y - q_x q_z)}\right)\\ \operatorname{atan2}(2(q_w q_z + q_x q_y), 1 - 2(q_y^2 + q_z^2))\end{bmatrix} \tag{5.5}$$

The quaternion is represented as $q = (q_w, q_x, q_y, q_z)$, where q_w is the scalar component, and q_x, q_y, q_z are the vector components. Since, as mentioned earlier, the only angle of interest is the one around the z-axis, only the last equation is used: ψ is the robot's yaw angle, or yaw_{robot} (5.6).

$$\psi = yaw_{robot} = \operatorname{atan2}\left(2(q_w q_z + q_x q_y), 1 - 2(q_y^2 + q_z^2)\right) \tag{5.6}$$

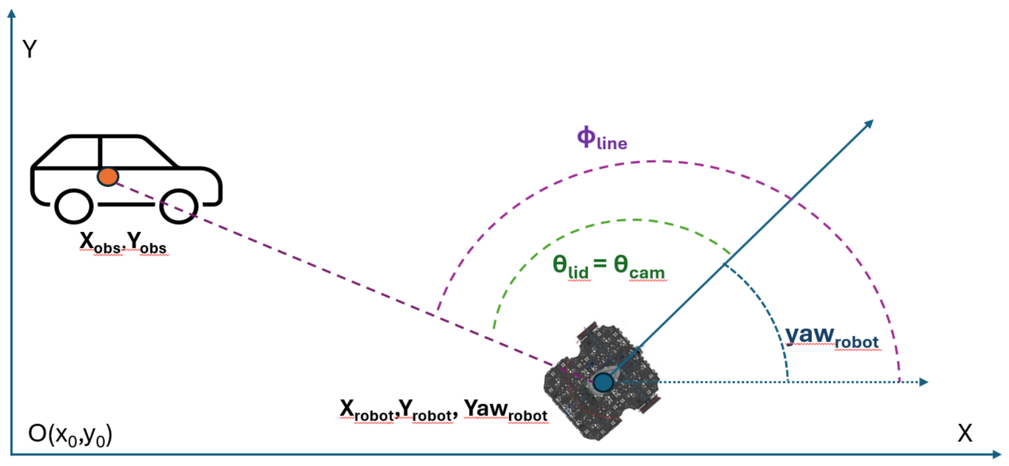

Y
Φ_{line}
X_{obs}, Y_{obs}
$\theta_{lid} = \theta_{cam}$
yaw_{robot}
$X_{robot}, Y_{robot}, Yaw_{robot}$
$O(x_0,y_0)$
X

Figure 5.5: θ_{lid} calculation

As shown in Fig. 5.5, the line connecting the robot and the obstacle is defined, and its angle ϕ_{line} relative to the global map frame is computed using the following equation:

$$\phi_{line} = atan2(y_{\text{obs}} - y_{\text{robot}}, x_{\text{obs}} - x_{\text{robot}}) \tag{5.7}$$

Next, the robot's orientation with respect to the map, yaw_{robot}, is subtracted from ϕ_{line} to determine the angle between the robot's direction and the obstacle. This relationship is expressed as:

$$\theta_{lid} = \phi_{line} - \mathrm{yaw}_{robot} \tag{5.8}$$

combining these steps, a single equation can be obtained:

$$\theta_{\mathrm{lid}} = \mathrm{atan2}(y_{\mathrm{obs}} - y_{\mathrm{robot}}, x_{\mathrm{obs}} - x_{\mathrm{robot}}) - \mathrm{yaw}_{\mathrm{robot}} \tag{5.9}$$

Once obtained the angle θ_{lid}, it is compared with the angle θ_{cam} obtained by the camera.

5.2.4 Obstacle Matching

If the angles θ_{lid} and θ_{cam} match within a small predefined margin, it can be concluded that the obstacle detected by the LiDAR is the same as the one identified by the camera. Consequently, the label assigned by the camera is applied to the dynamic obstacle.

To perform the comparison between detections, it is important to consider that the camera only provides the angular position of an object, while LiDAR also provides the distance. In the case of alignment of two obstacles, it is therefore necessary to consider only the one closest to the camera, because other obstacles in the same direction are assumed to be occluded by the first obstacle and are therefore not visible to the camera. Once the closest obstacle has been identified, it is then labeled with the class provided by YOLO.

5.3 Implementation of the YOLO-LiDAR Matching System

5.3.1 Sensor Fusion Architecture

This section gives an overview of the implementation of the above-mentioned approach within the ROS 2 framework.

The main platform used for this thesis was an Intel NUC computer (model NUC10FNH) running Linux Ubuntu 22.04 and ROS 2 Humble Hawsbill was used as main platform for this thesis. The implementation was divided into two parts: the first part involves using a ROS 2 package to do real time object detection using Ultralytics YOLO, to make object detection integration with different robotic platforms possible; the second part involves creating a ROS 2 node to fuse the dynamic obstacle information obtained from LiDAR and camera and to match the data before passing the information to the costmap.

To achieve the goal, four ROS 2 workspaces were created:

- `nav2_ws`. In this workspace, the `nav2` package was cloned from its official repository and compiled, allowing modifications. The `navigation2_dynamic` package was also included, which implements dynamic obstacle detection and obstacle tracking using LiDAR only.
- `ros2_yolo`. This workspace includes a package that implements the YOLO algorithm for real-time object detection and calculates the position of objects with respect to the camera located on the robot.
- `obstacle_matcher_ws`. This is the workspace where data fusion takes place; it integrates information from both the camera and LiDAR and merges them to label the obstacles.
- `ros2_webots_simulation`. It contains all the packages necessary to run Webots and the simulation of the environment used to verify the correct functioning of the code.

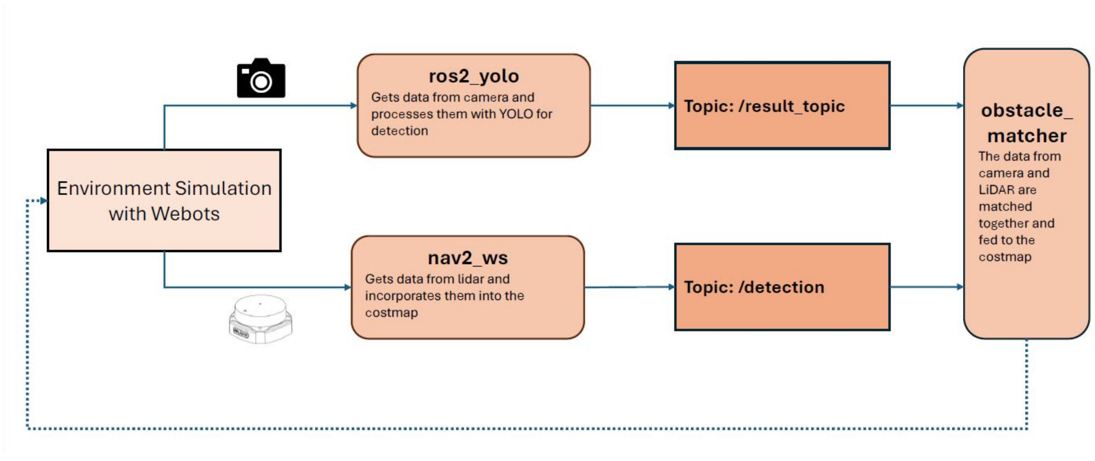

ros2_yolo
Gets data from camera and processes them with YOLO for detection
Topic: /result_topic
obstacle_ matcher
The data from camera and LiDAR are matched together and fed to the costmap
Environment Simulation with Webots
nav2_ws
Gets data from lidar and incorporates them into the costmap
Topic: /detection

Figure 5.6: Block diagram of the functioning of the implemented code

5.3.2 Messages and Topics

In order to effectively communicate the bounding boxes resulting from the YOLO algorithm, two custom messages were created, `Detection.msg` and `DetectionArray.msg`:

- `Detection.msg`

```
std_msgs/Header header
string classname  # Detected object class (e.g.,
                   'person', 'car', etc.)
float32 conf      # Confidence level of detection (0.0
                   to 1.0)
float32 angle     # Detected angle
float32 min_angle # Optional: Minimum angle (set to 0
                   if not used)
float32 max_angle # Optional: Maximum angle (set to 0
                   if not used)
```

- `DetectionArray.msg`

```
    std_msgs/Header header
    Detection[] detections
```

The header contains the timestamped data in a particular coordinate frame, while the Universally Unique Identifier (UUID) [37] is a 128-bit label used to uniquely identify the obstacles.

When the data from the sensors are fused, the communication of the labeled dynamic obstacles information happens through two message types, `LabeledObstacle.msg` and `LabeledObstacleArray.msg`, defined in the `obstacle_matcher_interfaces` package. These messages were taken from the previous work, which used messages of type `Obstacle.msg` and `ObstacleArray.msg`, to which the field containing the object label was simply added.

- `LabeledObstacle.msg`. This message contains information about a single obstacle. It includes:

```
std_msgs/Header header
unique_identifier_msgs/UUID id
geometry_msgs/Point position # center position
geometry_msgs/Vector3 velocity
geometry_msgs/Vector3 size
string classname      # Detected object class (e.g.,
                    'person', 'car', etc.)
```

- `LabeledObstacleArray.msg`: A message that gathers all the `LabeledObstacle` messages into an array. It includes:

```
std_msgs/Header header
LabeledObstacle[] labeled_obstacles
```

Velocity and size contain the estimated velocity vector and the 3D bounding box of the obstacle (even though only two dimensions are actually used), respectively. Finally, a string is used to identify the object class, filled with the information coming from YOLO detection.

5.3.3 Integration of YOLO in ROS 2

The `ros2_yolo` workspace has the following structure:

```
ros2_yolo
├─ launch
│  └─ tracker.launch.xml
├─ msg
│  ├─ Detection.msg
│  ├─ DetectionArray.msg
│  └─ YoloResult.msg
├─ script
│  └─ tracker_node.py
├─ package.xml
└─ CMakeLists.txt
```

In the `ros2_yolo` workspace, a node called `tracker_node` was implemented that receives data from the camera in the form of images, performs object detection using YOLO and then proceeds to calculate the angle of each object found relative to the robot frame.

To make it possible to use YOLO, it is imported directly from the `ultralytics` library. The `cv_bridge` library is also imported, which makes it possible to convert `sensor_msgs/Image` messages into OpenCV objects and vice versa.

The node subscribes to the topic `image_raw`, but through the `input_topic` parameter it can be easily configured; for this work, both the topic `/usb_cam/image_raw` was used using a webcam for testing, and the topic `/camera/image_raw` of the turtlebot3 burger for simulations in a virtual environment.

The main function of this node is `calculate_angle`, which calculates the horizontal angle of an obstacle according to its position in the image frame, taking into account the FOV of the camera and its resolution as explained in Section 5.2.1. This function returns the angle in degrees; the angle is positive on the left and negative on the right with respect to the centre point of the camera.

The results of the detections are published on the topic `yolo_result` via the message `DetectionArray.msg` but also on another topic, `yolo_image`, which returns the images with the bounding boxes, the class name and the confidence of the detection.

In `tracker.launch.xml` file, before launching the node, it is possibile to set various parameters, including the YOLO model, the YOLO classes to detect, the confidence threshold, what is the type of device used and other parameters.

5.3.4 Obstacle Matcher

The obstacle_matcher workspace has the following structure:

```
obstacle_matcher_ws
├─ launch
│   └─ obstacle_matcher.launch.py
├─ obstacle_matcher_pkg
│   ├─ __init__.py
│   └─ obstacle_matcher.py
├─ resource
├─ test
├─ package.xml
├─ setup.cfg
└─ setup.py
```

In this package, the `obstacle_matcher`, the node that performs the actual sensor fusion, is developed. It takes the position of the obstacles detected via LiDAR and the labeled detections made by the camera thanks to YOLO to output a message containing all the information.

Section 5.3.4 shows the implemented pseudo-code to allow a better understanding of the code structure and the developed functions.

Specifically, the node subscribes to the `/yolo_result` topic, published by the previously described node, and the `/detection` topic, where the LiDAR data arrives.

As a first step, it is necessary to synchronize the stream of information: this is done using the `message_filters` library [35], in particular using the `ApproximateTimeSynchronizer` class, which allows incoming messages to be synchronized by their timestamp with a certain tolerance. To match obstacles, it is necessary for them to be on the same frame; the `quaternion_to_rpy` function was therefore used to convert quaternions into angles expressed in degrees as explained in Sec. 5.2.3. The `update_robot_position` function takes advantage of the previous one and uses the `tf2` library to derive the robot's pose with respect to the map (`map` to `base_link`), to obtain the position in `x`, `y`, `z` coordinates and the orientation in degrees, in particular the yaw angle.

To solve the problem of two obstacles aligned with respect to the camera, the function `filter_closest_obstacles` was also implemented, which takes the obstacles detected by the LiDAR, calculates their angle and distance with respect to the robot and then takes only the closest one for each angular sector.

Finally, a message of type `LabeledObstacleArray` is published on the topic `/labeled_detection`, in which all the obstacles with their respective labels are found.

Pseudocode for ObstacleMatcherNode

```


class ObstacleMatcherNode:
    def __init__(self):
        # Node initialization
        create_subscribers()  # YOLO and LiDAR topics
        synchronize_messages()  # Synchronize YOLO and LiDAR
            inputs
        initialize_tf2_buffer()  # Retrieve robot pose
        setup_publishers()  # Publisher for labeled obstacles

    def quaternion_to_rpy(quaternion):
        # Convert quaternion to yaw angle (degrees)
        extract_values(quaternion)
        compute_yaw()
        return yaw_degrees

    def update_robot_position():
        try:
            # Retrieve transform map -> base_link
            x, y, z = extract_translation()
            yaw = quaternion_to_rpy(rotation)
            return {"x": x, "y": y, "z": z, "yaw": yaw}
        except Error:
            log_error()
            return None

    def sync_callback(yolo_msg, obstacle_msg):
        # Sync YOLO and LiDAR data
        robot_pos = update_robot_position()
        closest_obstacles = filter_closest_obstacles(obstacle_msg)
        labeled_obstacles = match_with_camera(yolo_msg,
           closest_obstacles)
        process_matched_obstacles(labeled_obstacles)

    def filter_closest_obstacles(obstacle_msg):
        # Filter closest obstacles for each angle
        for obstacle in obstacle_msg:
            compute_angle_and_distance()
            if is_closer_than_previous():
                update_closest_obstacle()
        return closest_obstacles

    def match_with_camera(yolo_msg, closest_obstacles):
        # Match YOLO detections with closest LiDAR obstacles
        for detection in yolo_msg:
            for angle_key in closest_obstacles:
                if angles_match(detection.angle, angle_key):
                    create_labeled_obstacle(detection, obstacle)
        return labeled_obstacles

    def process_matched_obstacles(labeled_obstacles):
        # Publish labeled obstacles
        if labeled_obstacles:
            publish_to_topic(labeled_obstacles)
```

Listing 5.1: Main pseudo-code structure of the ObstacleMatcherNode

Chapter 6

Simulations and Tests

This chapter provides an overview of the general configuration of the simulation environment used to test, debug, and validate the implementation created for this thesis. The key tools are Rviz2, which is used for debugging and internal output visualization, and Webots 2023a, which is used to create and simulate the outdoor environments and the robot model.

6.1 Function check with USB webcam

Before checking the correctness of the implementation on a simulated environment, a USB webcam was used as an input sensor to check whether the integration between YOLO and ROS 2 was behaving as expected. The main objective of this phase was to ensure that YOLO was able to:

1. Correctly recognise objects
2. Return consistent bounding boxes and classes and correctly calculate the angular position of objects

For these checks, the Logitech C925-E webcam model was used (Fig. 6.1), with a resolution of 640×480 pixels and FOV of 78°.

The parameters for recognition were configured in the `tracker.launch.xml` file. In particular YOLOv8s model was chosen and the input topic was set to receive the image from the webcam. The minimum confidence threshold - where confidence is the numeric value expressing the model's trust into its prediction - to accept a detection as valid and the IoU (Intersection over Union) threshold were defined; the IoU determines how much the bounding boxes must overlap to be considered the same object. Choosing a higher minimum confidence threshold allows for a

(a) Frontal view. (b) Side view.

Figure 6.1: Logitech C925-E webcam [17], used for the first tests of the implementation.

more accurate model, reducing false positives, while a lower IoU threshold makes it possible to detect more objects, even if some can be false positives. This is useful for applications where it is important not to lose relevant objects (e.g., pedestrian tracking). The chosen values provide a balance between precision and recall (ability of the model to correctly identify all positive cases).

Furthermore, since the NUC does not have a GPU, the device parameter is set to cpu (if a GPU can be used, improving performance, this parameter can be replaced with cuda):

```
[...]
  <arg name="yolo_model" default="yolov8s.pt"/>
  <arg name="input_topic" default="/image_raw"/>
  <arg name="result_topic" default="/yolo_result"/>
  <arg name="result_image_topic" default="/yolo_image"/>
  <arg name="conf_thres" default="0.70"/>
  <arg name="iou_thres" default="0.45"/>
  <arg name="max_det" default="300"/>
  <arg name="device" default="cpu"/>
[...]
```

YOLO was trained on the COCO dataset [1], that is a large dataset used in computer visions that includes 80 object categories, so within the parameters we define the classes of interest:

```
[...]
<param name="classes" value="0,1,2,3,5,7,9" value-
sep=","/>-<!-- person, bicycle, car, motorcycle, bus,
truck, traffic light -->
[...]
```

Inside the code, the correct resolution and FOV values of the webcam were included in the function that calculates the angular position of the obstacles.

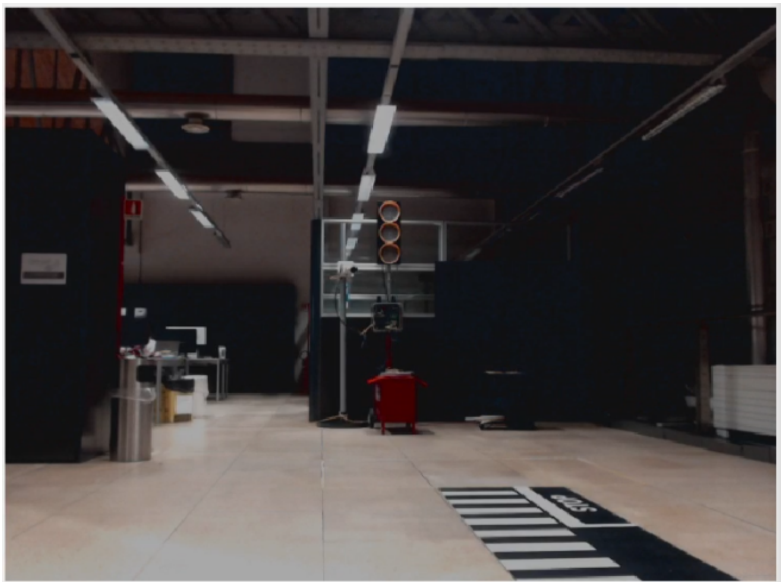
Once the configuration was complete, tests were carried out within the LINKS Foundation offices. Initially, a traffic light for demo purpose was placed approximately in the center of the image to check whether the zero angle was recognized correctly (Fig. 6.2). For the visualization of the results, RViz2 was used, comparing the input topic representing what the camera sees and the output topic showing the bounding boxes with labels and the detection confidence score.

Reading the `yolo_result` topic, on which `detection` messages are published, it can be seen that YOLO not only recognizes the class correctly, but also calculates the angular position of the traffic light, which is approximately zero (Fig. 6.4a).

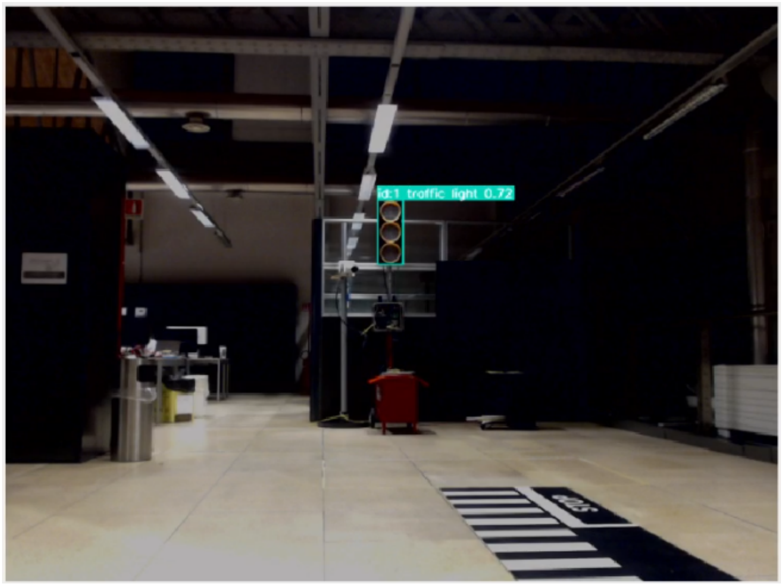
In order to verify that the recognition could also take place in the presence of several objects, the same experiment was repeated introducing two people, one placed at one quarterof the camera frame and the other on the right, close to the edge of the frame, so that an approximate check of the angles could be made (Fig. 6.3).

It can be verified that on `yolo_result` topic there are three identified objects, each associated with its class and angle (Fig. 6.4b).

Once the correct functioning of YOLO and the angular position calculation function were verified, testing proceeded in the simulated environment with Webots.

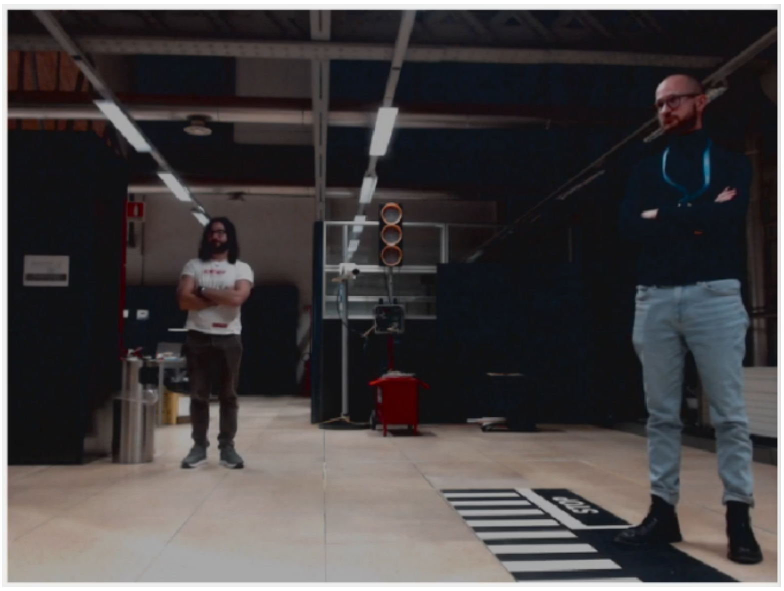


(a) Camera view without YOLO's labeling.

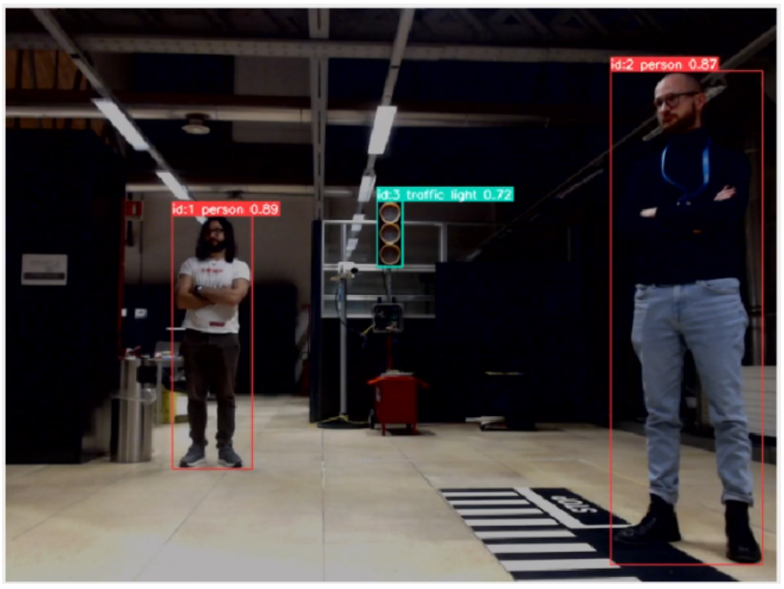


(b) Camera view with YOLO's labeling and bounding box.

Figure 6.2: Validation of YOLO's object recognition of a traffic light using RViz2.

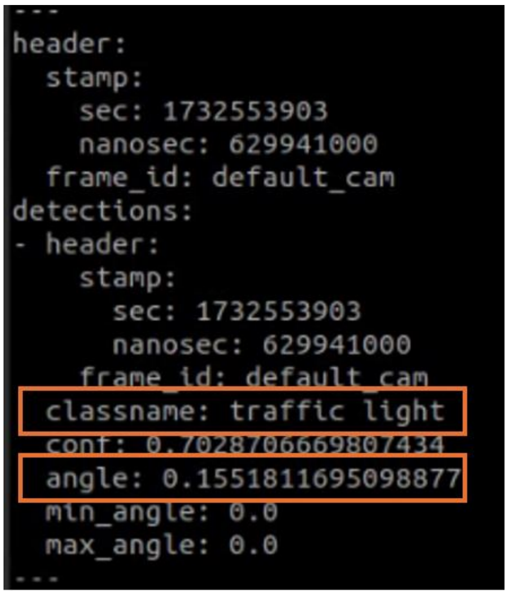


(a) Camera view without YOLO's labeling.



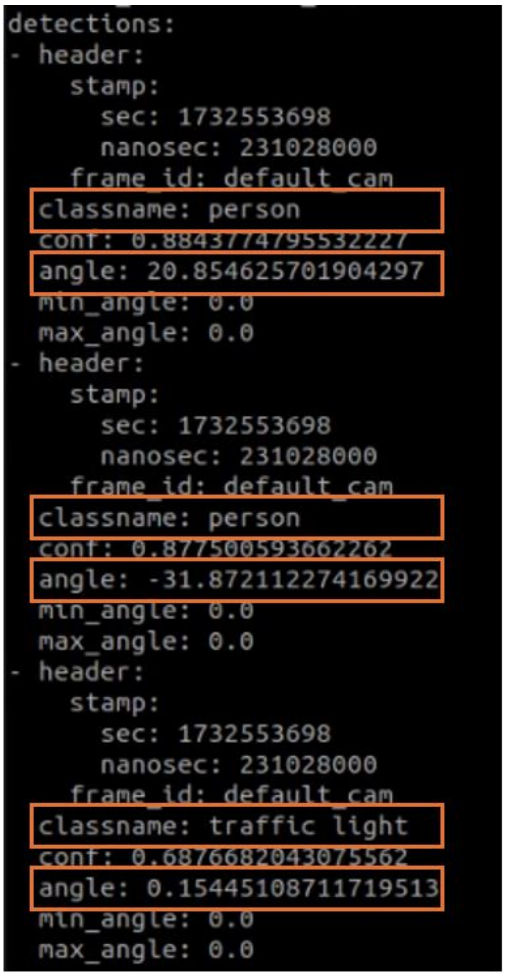
(b) Camera view with YOLO's labeling and bounding box.

Figure 6.3: Validation of YOLO's object recognition for more obstacles using RViz2.



```
---
header:
  stamp:
    sec: 1732553903
    nanosec: 629941000
  frame_id: default_cam
detections:
- header:
    stamp:
      sec: 1732553903
      nanosec: 629941000
    frame_id: default_cam
  classname: traffic light
  conf: 0.7028706669807434
  angle: 0.1551811695098877
  min_angle: 0.0
  max_angle: 0.0
---
```

(a) Detection for a traffic light.



```
detections:
- header:
    stamp:
      sec: 1732553698
      nanosec: 231028000
    frame_id: default_cam
  classname: person
  conf: 0.8843774795532227
  angle: 20.854625701904297
  min_angle: 0.0
  max_angle: 0.0
- header:
    stamp:
      sec: 1732553698
      nanosec: 231028000
    frame_id: default_cam
  classname: person
  conf: 0.8775005593662262
  angle: -31.872112274169922
  min_angle: 0.0
  max_angle: 0.0
- header:
    stamp:
      sec: 1732553698
      nanosec: 231028000
    frame_id: default_cam
  classname: traffic light
  conf: 0.6876682043075562
  angle: 0.15445108711719513
  min_angle: 0.0
  max_angle: 0.0
```

(b) Detection for a traffic light and two people.

Figure 6.4: Published data on `yolo_result` topic, including the obstacles' classes and angles.

6.2 Webots Simulation Environment

In order to test the developed system, a simulation of outdoor environment was created in Webots (see Sec. 3.4.1). This environment was designed in a way that a realistic city was depicted with necessary infrastructure such as streets, traffic signs and operational traffic lights. Also, a car was included which serves the purpose of a dynamic object in the simulation.

The primary goal of this configuration was to create a situation where the system is exposed to operational conditions but within a controlled environment, in order to assess the object recognition functionality. Urban elements like traffic lights enable us to test the YOLO model's capability in recognizing static objects, while the car enables us to evaluate the system's performance for dynamic obstacles scenarios. The setting also allows to verify if the implementation can correctly calculate the obstacles angular position.

After building the simulated world, the model of the TurtleBot3 Burger was included as an external PROTO file (Fig. 6.5). Next, a LiDAR and a camera were added to the robot to equip it with the necessary sensors for the simulation. The camera chosen features a field of view (FOV) of 110° and a resolution of 720x480 pixels, while the RPLidar A2 model was adopted for the LiDAR.

```
Camera {
    translation  0 0 0.085
    fieldOfView 1.92     # 110°
    width 720
    height 480
}

Lidar {
    name "RPlidar_A2"
    [...]
    horizontalResolution 800
    fieldOfView 6.28   # 360°
    numberOfLayers 1
    near 0.05
    minRange 0.2
    maxRange 12
    [...]
 }
```

Before running the code, it is essential to have a map of the simulated environment. This mapping was performed using the Nav2 SLAM (Simultaneous Localization and Mapping) Toolbox [19], an important technique in robotics that

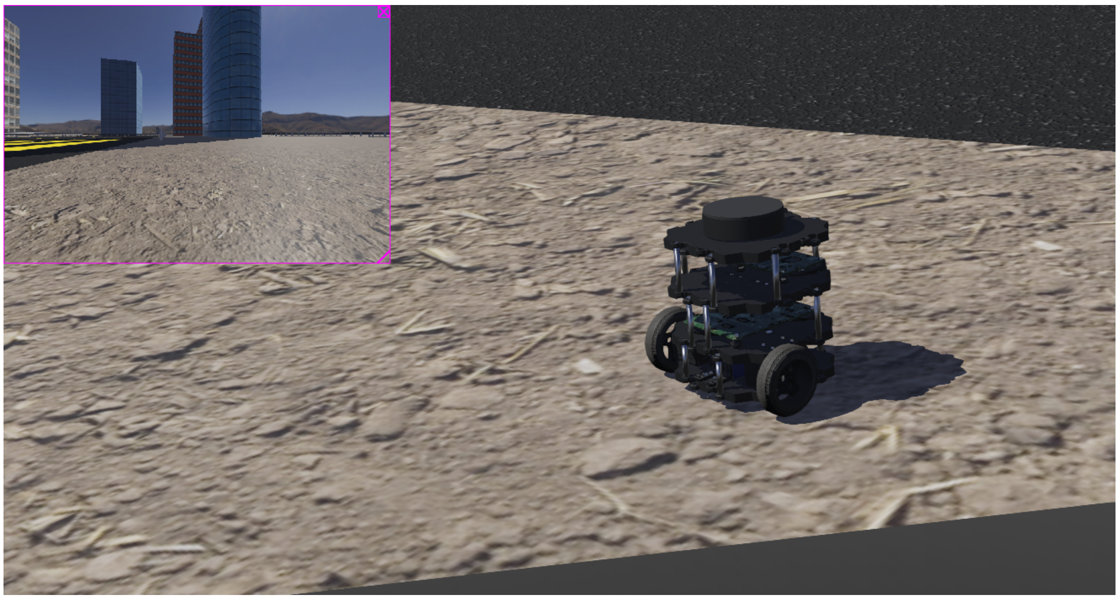


Figure 6.5: TurtleBot3 Burger inside the Webots simulation; in the corner, the camera view of the robot.

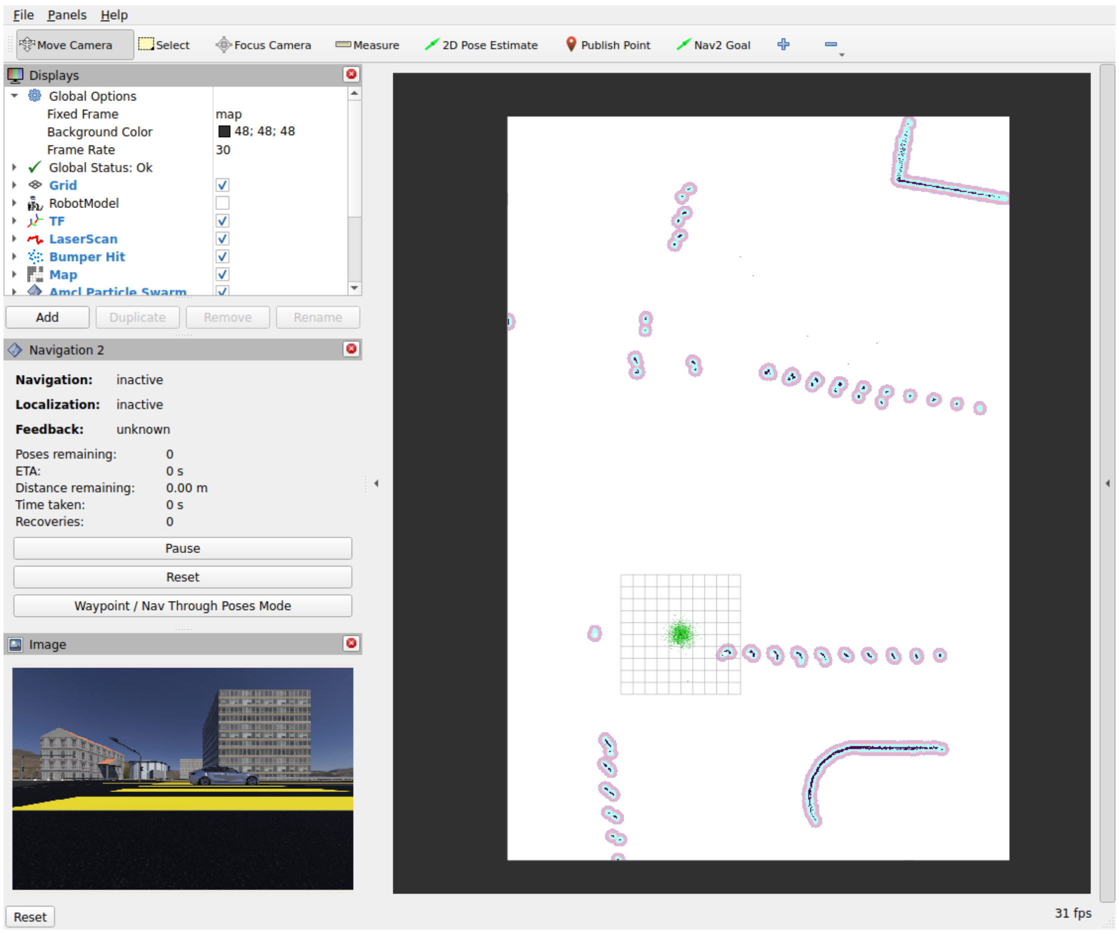
allows a mobile robot to localize itself, i.e. determine its own pose, and map an unknown environment. The TurtleBot was made to move within the simulated environment via the `teleop_twist_keyboard` node, which allows the robot to be controlled from the keyboard to explore different areas. At this stage, the LiDAR acquired information about the existing structures, such as walls and objects, and identified the static obstacles. The result is a two-dimension map, which can be saved and visualized within RViz2 (Fig. 6.6).

6.2.1 Traffic Light Detection and Angle Calculation in Simulation

After setting the right camera parameters and input topic within the code, a traffic light was placed at a known location within the simulated world, in order to provide a fixed reference to test again the system's ability to identify static objects accurately. Knowing the position of the traffic light, it was possible to compare the data provided by the code, such as class detection and angle calculation, with the expected theoretical values.

The traffic light was positioned two meters forward and one meter to the right of the robot (Fig. 6.7), with an expected angle of:

$$\theta = \arctan\left(-\frac{1}{2}\right) \approx -26.57^\circ \tag{6.1}$$



Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
Global Status: Ok	
Grid	✓
RobotModel	☐
TF	✓
LaserScan	✓
Bumper Hit	✓
Map	✓
Amcl Particle Swarm	✓

Navigation: inactive
Localization: inactive
Feedback: unknown

Poses remaining:	0
ETA:	0 s
Distance remaining:	0.00 m
Time taken:	0 s
Recoveries:	0

Figure 6.6: Map obtained through SLAM of the simulated environment, visualized on RViz2. On the lower left corner it can be seen the camera view of the robot.

By running the code, it can be verified on RViz2 that YOLO correctly identified the traffic light as such and was able to calculate its angle, as can be seen in Fig. 6.8 and Fig. 6.9.

There is a small discrepancy between the theoretical angle and the angle calculated by the system, due to the fact that the traffic light was manually positioned within the simulated environment.

6.2.2 Sensor Fusion for Static Obstacles

The next task following the verification of the right functioning of YOLO and angle calculation was to integrate this component with the information coming from the LiDAR. Simulations were conducted to this end, testing the fusion of the visual input data from the camera comparing with LiDAR data. The main objective of

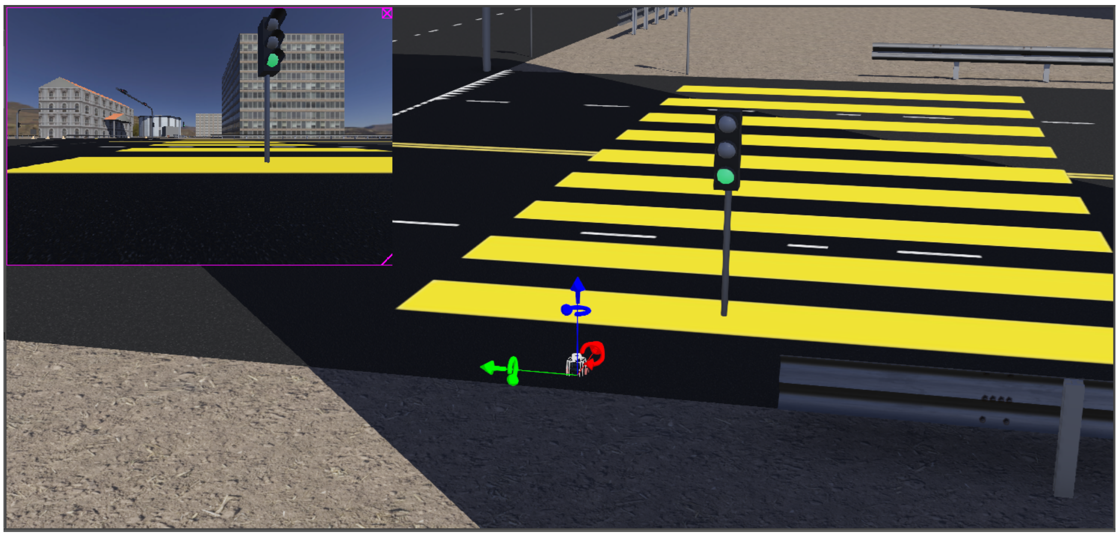


Figure 6.7: Webots simulation with the TurtleBot3 Burger and its frame and the traffic light placed in a known position.

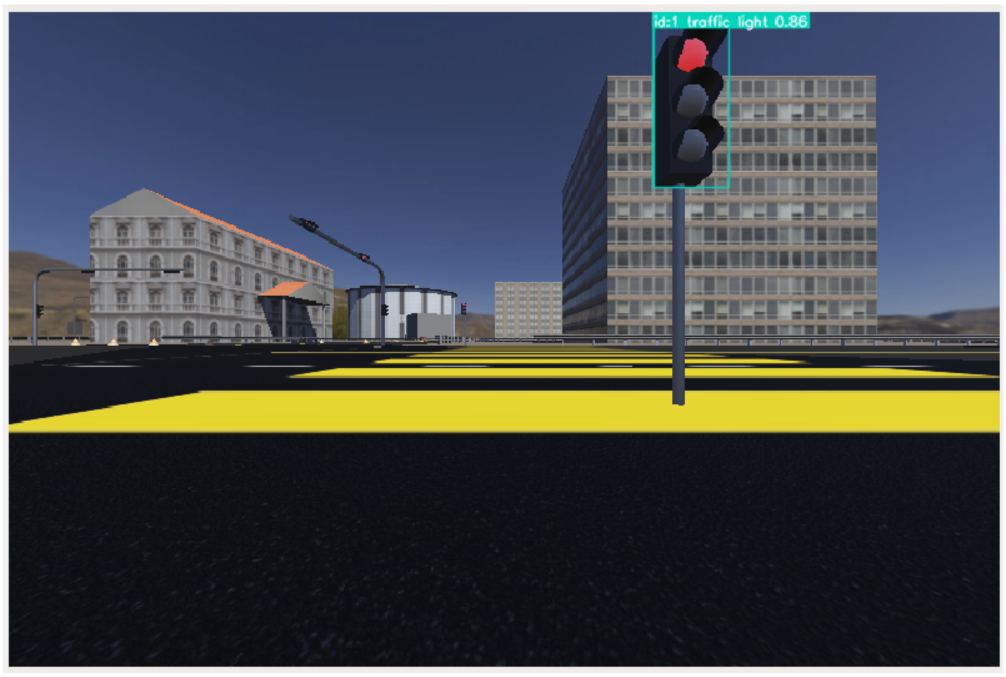


Figure 6.8: RViz2 visualization of the `yolo_image` topic, with the bounding box and label of the traffic light.

this phase was to check if the implemented code was able to match the sensors information for static obstacles.

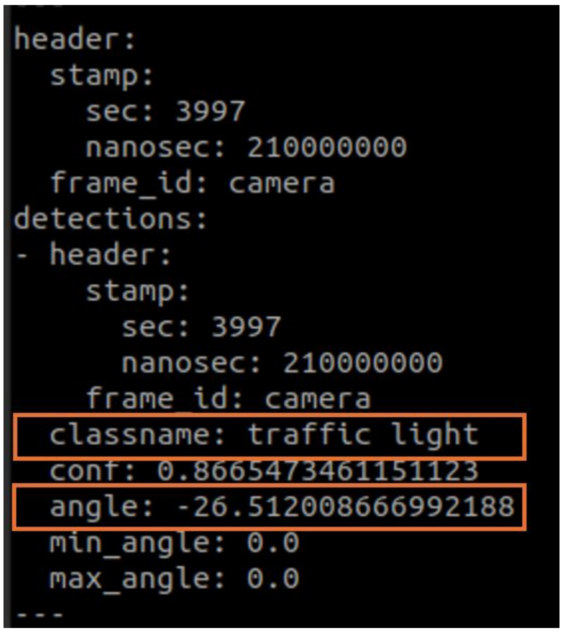

```
header:
  stamp:
    sec: 3997
    nanosec: 210000000
  frame_id: camera
detections:
- header:
    stamp:
      sec: 3997
      nanosec: 210000000
    frame_id: camera
  classname: traffic light
  conf: 0.8665473461151123
  angle: -26.512008666992188
  min_angle: 0.0
  max_angle: 0.0
---
```

Figure 6.9: `yolo_result` topic containig the label and the angle of the traffic light obstacle.

This thesis work, as previously mentioned, is based on an existing project, which includes a plug-in for Nav2 designed to handle only dynamic obstacles. This made impossible to perform tests in presence of static obstacles.

In order to validate the work by initially using simpler conditions, a ROS 2 node in charge of detecting static obstacles was developed.

This node, `detection_publisher`, is used to forcibly publish on the topic `/detection`, simulating the presence of obstacles of known position and using the `obstacles.msg` message already employed in the dynamic obstacles plugin by setting the velocity to zero. This made it possible to carry out preliminary tests to facilitate debugging before verifying the system's behaviour in the presence of dynamic obstacles.

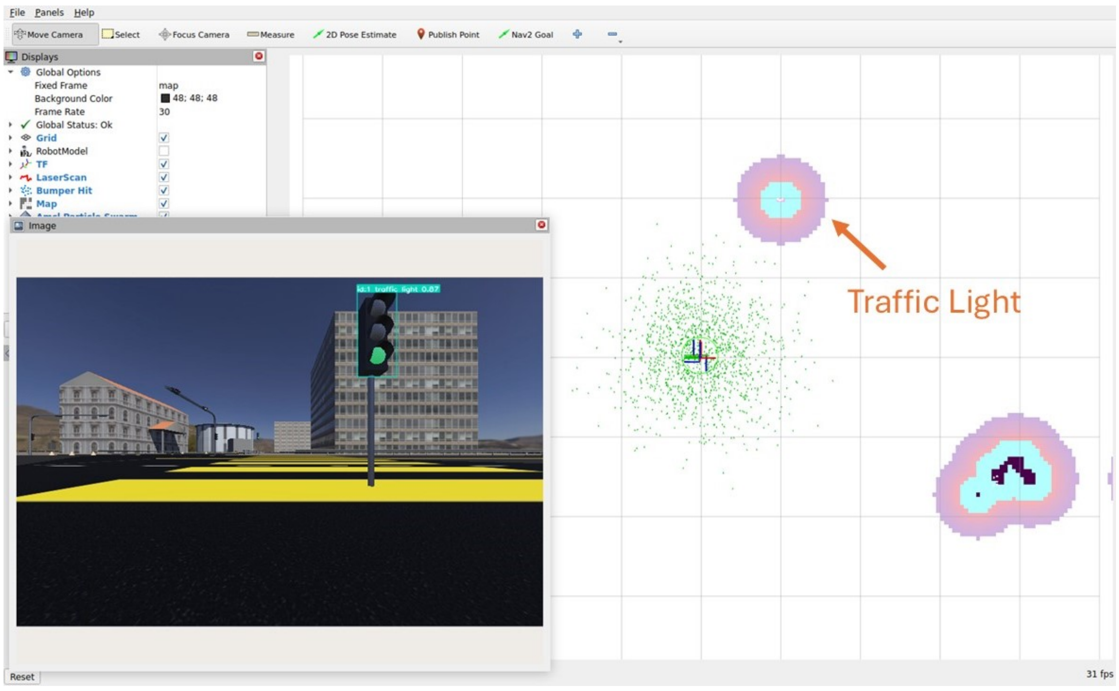
The obstacle chosen for the test was again a traffic light, positioned at the same location used in the previous test. To carry out the simulation, three ROS 2 nodes were launched:

- **tracker_node**: as already seen, this is the node responsible for detection through YOLO, which publishes information about the angles and labels of the obstacles.
- **detection_publisher**: this is the node that forcibly publishes the known

position of the traffic light on the topic `/detection`.

- **obstacle_matcher**: this is the node that implements the fusion of the data, matching the obstacles detected by the LiDAR with the angular position provided by the `tracker_node` calculated using the camera parameters.

RViz2 was once again used to display both the map and the publication of labels and bounding boxes generated by YOLO (Fig. 6.10)

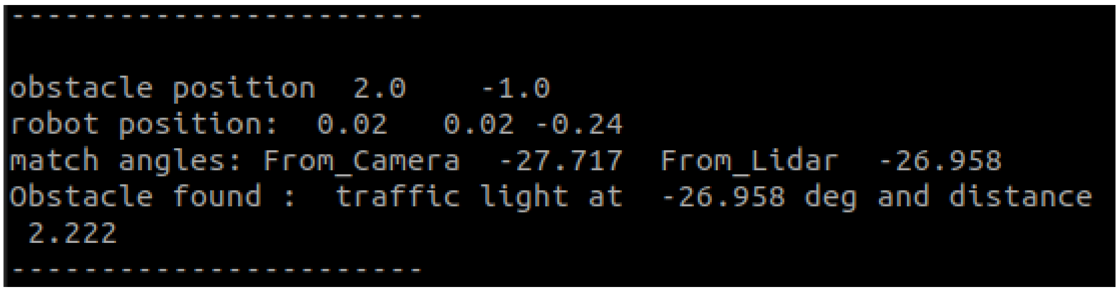


```
Fixed Frame        map
Background Color   48; 48; 48
Frame Rate         30
```

Figure 6.10: RViz2 visualization during the test using static obstacles, with local map and camera view of the robot.

During the tests, a screenshot was taken of the `obstacle_matcher` node terminal, in which screen prints were implemented showing the results of the match between the data from the two sensors (Fig. 6.11).

In particular, it is possible to read the angles calculated for the traffic light with respect to the robot according to the two sources, the camera and LiDAR. The detected angles show a small discrepancy: -27.717° from the camera and -26.958° from LiDAR, confirming that the matching between the readings of the two sensors was performed correctly, as well as the recognition of the obstacle. Since LiDAR offers greater accuracy in calculating angles than the camera, the angle calculated by LiDAR is considered to be the value closest to the actual position of the traffic light.



```
-----------------------

obstacle position  2.0    -1.0
robot position:  0.02   0.02 -0.24
match angles: From_Camera  -27.717  From_Lidar  -26.958
Obstacle found :  traffic light at  -26.958 deg and distance
 2.222
-----------------------
```

Figure 6.11: `obstacle_matcher` node terminal.

6.2.3 Sensor Fusion for Dynamic Obstacles

The last phase of the tests focused on a dynamic environment with a car as a moving obstacle. The car's controller was programmed to move back and forth along the road with a speed initially set at 15m/s. However, it quickly became clear that the system could not function properly at such high speeds. Although the YOLO detection and calculation of angles takes place in real time, the same is not true for the `navigation2_dynamic` package. Although not developed in the context of this thesis, this package is required as it manages the publication of the obstacles detected by the LiDAR, but it is computationally intensive and therefore has a rather low publish frequency. The speed was therefore lowered to 2m/s, to ensure that the module could work as expected.

During the test in the simulation environment, the car was driven back and forth along the road a total of four times. In the first forward passage the car was recognized by the camera, but the obstacle match did not work, probably due to the slow launch of the nodes, which are initialized at different times. On the first backward pass, the match did not happen again, giving a discrepancy between the camera and LiDAR readings of about 11° (Fig. 6.13). This discrepancy is probably due to the delay in publishing the LiDAR data.

From the second pass onward, although with a delay, the recognition always took place and the sensor match was successful (Fig. 6.14, Fig. 6.15). This suggests that once the node processes have stabilized, the system is able to become more reliable, although some latencies in terms of speed are still present probably due to computing power restrictions, as explained below.

It can be concluded that, using a velocity of 2m/s, the obstacle fusion process works, but not optimally.
The algorithm that handles the publication and tracking of obstacles via LiDAR is computationally onerous, which is why it struggles to track obstacles moving

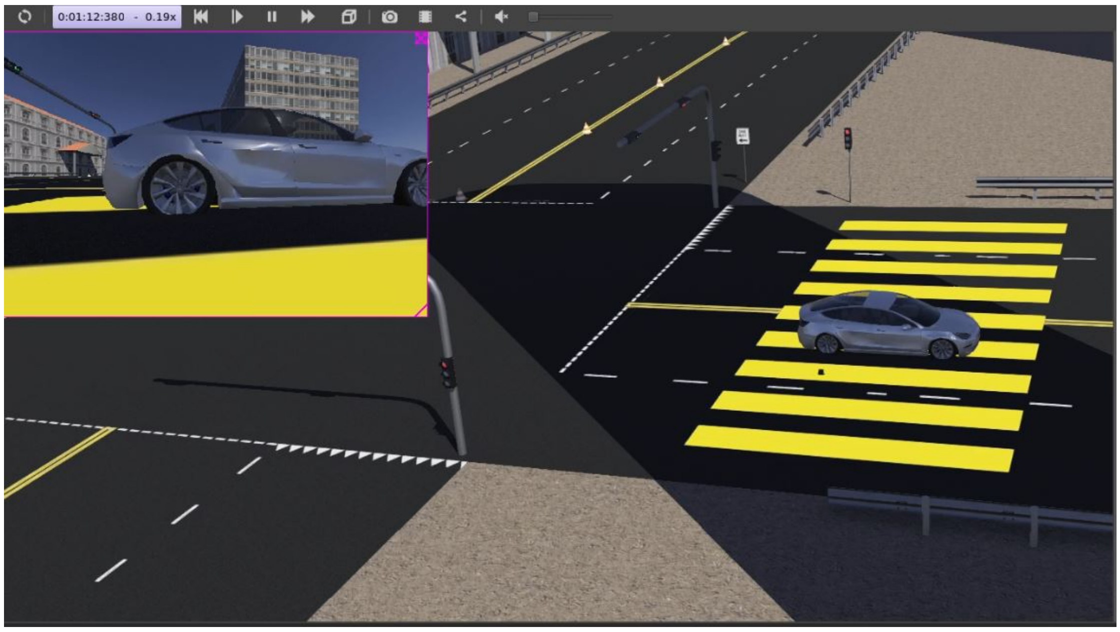
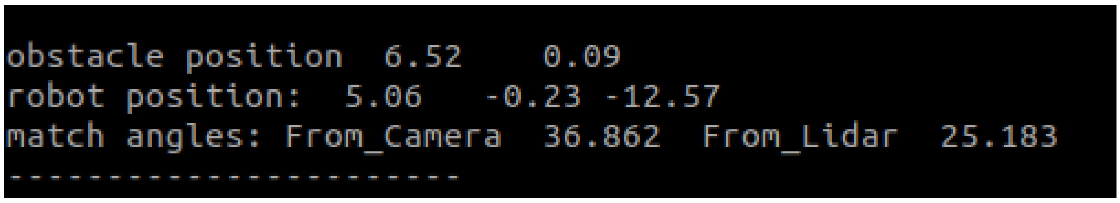


Figure 6.12: Webots simulation with the car. On the upper left corner, it can be seen that the simulation proceeds slowly due to the high coputational load required

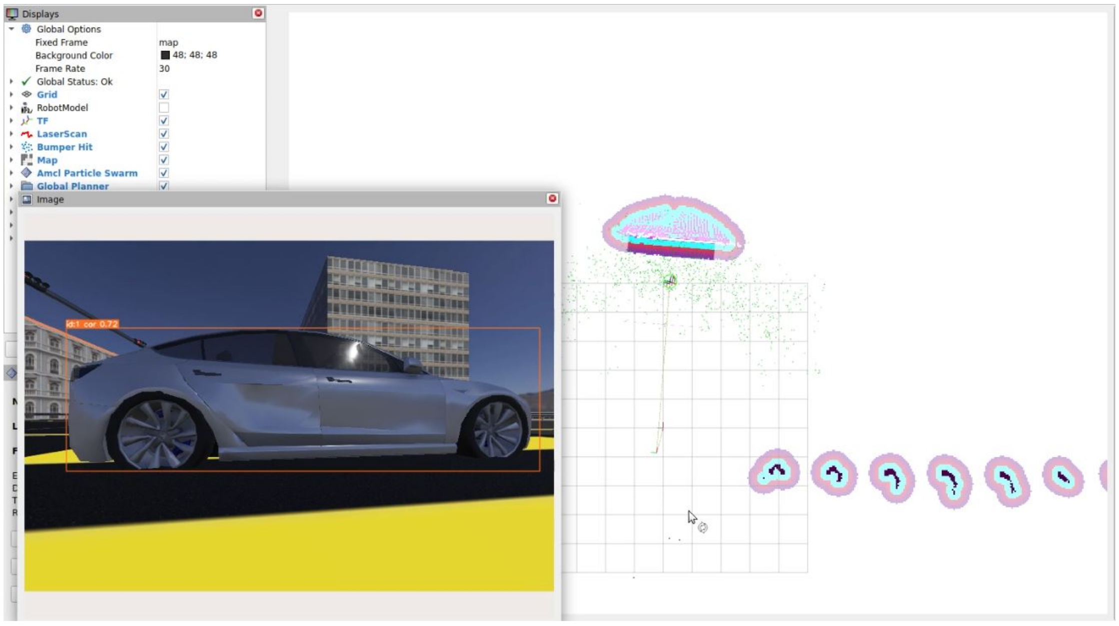


```
obstacle position  6.52    0.09
robot position:  5.06   -0.23 -12.57
match angles: From_Camera  36.862  From_Lidar  25.183
-----------------------
```

Figure 6.13: Terminal output captured during the second pass of the car.

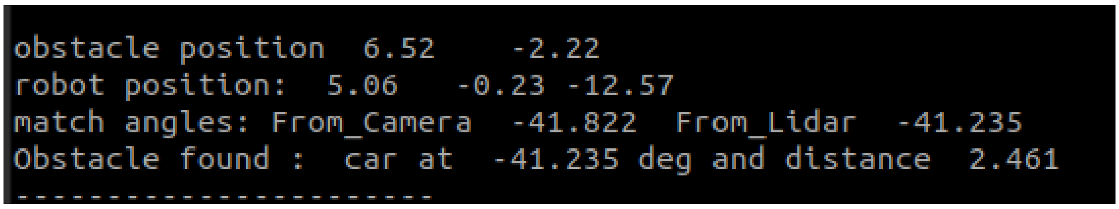
above a certain speed.

Another problem occurs while running the code: when all the packages required for the simulation (the detection, data fusion and obstacle management nodes) are executed at the same time, the simulation on Webots experiences a significant slowdown, dropping from a speed of 0.95x to around 0.25x, reaching as low as 0.15x (Fig. 6.12). This is because the NUC, being a general purpose computer, struggles to manage the high demand for computational resources: the YOLO algorithm needs a lot of computing power for image processing, and publishing and updating obstacles on the map using LiDAR increases the overall load. The use of dedicated hardware, such as NVIDIA Jetson boards with a dedicated GPU, would improve the performance.



```
Global Options
  Fixed Frame          map
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: Ok
Grid                   ✓
RobotModel             ☐
TF                     ✓
LaserScan              ✓
Bumper Hit             ✓
Map                    ✓
Amcl Particle Swarm    ✓
Global Planner         ✓
```

Figure 6.14: RViz2 visualization of the car obstacle detected by YOLO and LiDAR.



```
obstacle position  6.52    -2.22
robot position:  5.06   -0.23 -12.57
match angles: From_Camera  -41.822  From_Lidar  -41.235
Obstacle found :  car at  -41.235 deg and distance  2.461
-----------------------
```

Figure 6.15: Terminal output captured during the third pass of the car, highlighting the correct match between the obstacles.

Chapter 7

Conclusions and Future Works

This thesis work has made advances through the creation and verification of an improved perception system that combines object recognition capabilities with conventional navigation frameworks. Thanks to the work carried out in collaboration with the LINKS Foundation, this thesis has contributed to design a solution that enhances prior methodologies and proposing a more robust and flexible system for autonomous navigation in dynamic environments.

The development phase returned as a result a modular and extendable solution that makes use of the ROS2 Humble ecosystem and can combine LiDAR sensor data collected onboard a robotic platform with YOLO-based object identification findings using sensor fusion techniques. The constraints of robots with single-sensor navigation systems — camera or LiDAR — are addressed by this integration, especially in dynamic outside conditions where conventional methods have been proven to be ineffective. In fact, it has been found that the implemented system allows for improved obstacle characterization by utilizing multi-modal sensing capabilities. Additionally, the system has been designed to integrate with the current Nav2 stack frameworks.

The work aims not just to achieve sensor fusion but to develop a navigation system that is adaptive and more flexible in dynamic environments. The semantic information provided by the object recognition system — specifically the ability to label obstacles—opens up new possibilities for improving the robot's behavior. While recognizing a car allows for a predictive model accounting for higher speeds, identifying a person could invoke a model with slower but less predictable movements. This labeling capability can therefore improve dynamic obstacles avoidance, contributing to more adaptive navigation strategies, such as improved trajectory prediction skills based on a deeper semantic comprehension of identified objects.

The validation phase, which was conducted in the Webots simulation environment, showed promising results, especially in situations that involved a set of predefined dynamic obstacles intended to be realistic. The system performance evaluation revealed increased system responsiveness to environmental changes, robust real-time processing capabilities, and good item detection accuracy. The validation phase of the thesis work did, however, also highlight some limitations in the existing implementation. Due to its current implementation and its moderate to high processing power requirements, the system needs specific hardware to perform well in real time. Furthermore, even while simulation results are promising, they reflect scenarios that do not yet accurately reflect the complexity of the real world.

This thesis work is part of a larger autonomous last-mile delivery initiative being carried out in Turin by the LINKS Foundation. Future advancements in autonomous delivery systems developed in this project and set in challenging urban scenarios will be built upon the described system and methodology.

While progress has been made also thanks to this thesis work, research will continue to achieve robust, real-world autonomous navigation capabilities. The path forward should have as a first step a comprehensive real-world testing operation of the developed system using physical robotic platforms, particularly the AgileX Scout 2.0 used in the LINKS Foundation project, in outdoor uncontrolled urban environments.

In this project, sensor fusion is currently implemented using a late fusion strategy. This means that sensor data is processed separately to extract features or predictions, which are then combined at a higher level. Future improvements could look into early fusion strategies, where raw data from various sensors is merged into a single representation before any additional processing or analysis takes place. These two methods can be used at the same time, so that the advantages of both can be exploited: for example, early fusion can be used to get an initial representation, which can then be refined later with late fusion, so as to have more accurate data. To get a better understanding of the robot's surroundings, the information from the LiDAR and RGB camera can be supplemented with a depth camera, which provides three-dimensional information about the distance of objects from the sensor. This information can create more accurate maps and give more robustness to the detections, especially in urban environments.

Finally, matching between the detections can be done by other methods, such as by comparing bounding boxes generated by YOLO with the space occupied by obstacles according to lidar, but also by creating algorithms in which tracking occurs based on speed and trajectory, comparing the motion vectors estimated by both sensors.

Bibliography

[1] Coco dataset. URL https://cocodataset.org/#home. Accessed: 2024-11-12.

[2] Fida Ben Abdallah, Anis Bouali, and Pierre-Jean Meausoone. Autonomous navigation of a forestry robot equipped with a scanning laser. *AgriEngineering*, 5(1):1–11, 2023.

[3] Foxglove Blog. How to use ros 2 lifecycle nodes, 2024. URL https://foxglove.dev/blog/how-to-use-ros2-lifecycle-nodes. Accessed: 2024-11-07.

[4] Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, 1st edition, 2018. doi: 10.1201/9780429489105.

[5] Ivan Culjak, David Abram, Tomislav Pribanic, Hrvoje Dzapo, and Mario Cifrek. A brief introduction to opencv. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 1725–1730, 2012.

[6] Cyberbotics. The robot simulator company. https://cyberbotics.com/. Accessed: 2024-11-12.

[7] Pangcheng David Cen Cheng, Marina Indri, Fiorella Sibona, Matteo De Rose, and Gianluca Prato. Dynamic path planning of a mobile robot adopting a costmap layer approach in ros2. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2022. doi: 10.1109/ETFA52439.2022.9921458.

[8] F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte carlo localization for mobile robots. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, volume 2, pages 1322–1328 vol.2, 1999. doi: 10.1109/ROBOT.1999.772544.

[9] Nav2 Documentation. Nav2 documentation: Navigation2 framework for ros2, 2024. URL https://docs.nav2.org/. Accessed: 2024-11-19.

[10] João Filipe Ferreira, David Portugal, Maria Eduarda Andrada, Pedro Machado, Rui P Rocha, and Paulo Peixoto. Sensing and artificial perception for robots in precision forestry: A survey. *Robotics*, 12(5):139, 2023.

[11] Tully Foote. tf: The transform library. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6, 2013. doi: 10.1109/TePRA.2013.6556373.

[12] Open Source Robotics Foundation. velodyne_driver - ros index, 2024. URL https://index.ros.org/p/velodyne_driver/. Accessed: 2024-11-25.

[13] ROS Index. Rviz2 - ros index, 2024. URL https://index.ros.org/p/rviz2/. Accessed: 2024-11-27.

[14] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. Ultralytics yolov8, 2023. URL https://github.com/ultralytics/ultralytics.

[15] Ajay Jose, Harish Thodupunoori, and Binoy B Nair. A novel traffic sign recognition system combining viola–jones framework and deep learning. In *Soft Computing and Signal Processing: Proceedings of ICSCSP 2018, Volume 1*, pages 507–517. Springer, 2019.

[16] Hanwen Kang, Xing Wang, and Chao Chen. Accurate fruit localisation using high resolution lidar-camera fusion and instance segmentation. *Computers and Electronics in Agriculture*, 203:107450, 2022.

[17] Logitech. C925e business webcam. https://www.logitech.com/it-it/products/webcams/c925e-business-webcam.html.

[18] David V. Lu, Dave Hershberger, and William D. Smart. Layered costmaps for context-sensitive navigation. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 709–715, 2014. doi: 10.1109/IROS.2014.6942636.

[19] Steve Macenski. Slam toolbox. https://github.com/SteveMacenski/slam_toolbox, 2024.

[20] Steve Macenski, Tom Moore, David V. Lu, Alexey Merzlyakov, and Michael Ferguson. From the desks of ros maintainers: A survey of modern capable mobile robotics algorithms in the robot operating system 2. *Robotics and Autonomous Systems*, 168:104493, 2023. ISSN 0921-8890. doi: https://doi.org/10.1016/j.robot.2023.104493. URL https://www.sciencedirect.com/science/article/pii/S092188902300132X.

[21] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2016. doi: 10.1145/2968478.2968502.

[22] Ruta Mulajkar and Sanjay Yede. Yolo version v1 to v8 comprehensive review. In *2024 International Conference on Inventive Computation Technologies (ICICT)*, pages 472–478, 2024. doi: 10.1109/ICICT60155.2024.10544452.

[23] Thao Nguyen, Eun-Ae Park, Jiho Han, Dong-Chul Park, and Soo-Young Min. Object detection using scale invariant feature transform. In *Genetic and Evolutionary Computing: Proceedings of the Seventh International Conference on Genetic and Evolutionary Computing, ICGEC 2013, August 25-27, 2013-Prague, Czech Republic*, pages 65–72. Springer, 2014.

[24] Robo-Dyne. Turtlebot3 burger pi - robo-dyne, 2024. URL https://www.robo-dyne.com/turtlebot-3-burger-pi/?lang=it.

[25] AgileX Robotics. Scout 2.0 - agilex robotics, 2024. URL https://global.agilex.ai/products/scout-2-0.

[26] Open Robotics. Node lifecycle in ros 2, n.d. URL https://design.ros2.org/articles/node_lifecycle.html. Accessed: 2024-11-20.

[27] ROS Community. rplidar_ros package, 2024. URL https://index.ros.org/p/rplidar_ros/. Accessed: 2024-11-25.

[28] Matteo De Rose. Lidar-based dynamic path planning of a mobile robot adopting a costmap layer approach in ros2. Dicembre 2021. URL http://webthesis.biblio.polito.it/21253/.

[29] Bruno Siciliano. *Robotica – Modellistica, Pianificazione e Controllo*. McGraw-Hill, 2008.

[30] Jian Tang, Yuwei Chen, Antero Kukko, Harri Kaartinen, Anttoni Jaakkola, Ehsan Khoramshahi, Teemu Hakala, Juha Hyyppä, Markus Holopainen, and Hannu Hyyppä. Slam-aided stem mapping for forest inventory with small-footprint mobile lidar. *Forests*, 6(12):4588–4606, 2015.

[31] Suaibia Tasnim and Wang Qi. Progress in object detection: An in-depth analysis of methods and use cases. *European Journal of Electrical Engineering and Computer Science*, 7(4):39–45, 2023.

[32] Ricardo Tellez. A history of ros (robot operating system), 2019. URL https://www.theconstruct.ai/history-ros/. Accessed: 2024-11-10.

[33] TurtleBot. Turtlebot4 user manual: Rviz, 2024. URL https://turtlebot.github.io/turtlebot4-user-manual/software/rviz.html. Accessed: 2024-11-27.

[34] Ultralytics. Ultralytics repository, 2024. URL https://github.com/ultralytics/ultralytics. Accessed: 2024-11-27.

[35] ROS Wiki. Message filters. http://wiki.ros.org/message_filters, Accessed 2024.

[36] Wikipedia contributors. Behavior tree (artificial intelligence, robotics and control) — Wikipedia, the free encyclopedia, 2024. URL https://en.wikipedia.org/w/index.php?title=Behavior_tree_(artificial_intelligence,_robotics_and_control)&oldid=1214363898. [Online; accessed 25-November-2024].

[37] Wikipedia contributors. Universally unique identifier — Wikipedia, the free encyclopedia, 2024. URL https://en.wikipedia.org/w/index.php?title=Universally_unique_identifier&oldid=1256107477. [Online; accessed 10-November-2024].

[38] Qingtian Wu and Yimin Zhou. Real-time object detection based on unmanned aerial vehicle. pages 574–579, 05 2019. doi: 10.1109/DDCLS.2019.8908984.

[39] Fuzhen Zhang. Quaternions and matrices of quaternions. *Linear Algebra and its Applications*, 251:21–57, 1997. ISSN 0024-3795. doi: https://doi.org/10.1016/0024-3795(95)00543-9. URL https://www.sciencedirect.com/science/article/pii/0024379595005439.