HOCHSCHULE SCHMALKALDEN

UNIVERSITY OF APPLIED SCIENCES

Optimizing Outdoor Navigation: A Reinforcement Learning Approach to Tactical Planning in Mobile Robots.

Master Thesis

Dhavalkumar Vijaybhai Lad
Matriculation number: 313824

March, 2025

Supervisor: Prof. Dr. Frank Schrödel
Co-supervisor: M.Eng. Yekaterina Strigina

M.Eng. Mechatronics and Robotics, Schmalkalden University of Applied Sciences

* * *

"This thesis is dedicated to my beloved family, whose love and support have never wavered despite the distance, reminding me every day that I carry a part of home with me wherever I go."

* * *

Declaration of Independence

This is to certify that the thesis titled: Optimizing Outdoor Navigation: A Reinforcement Learning Approach to Tactical Planning in Mobile Robots submitted by Dhavalkumar Vijaybhai Lad (313824), to the Schmalkalden University of Applied Sciences for the award of the degree Master of Engineering in the field of Mechatronics and Robotics. All thesis work was done by Dhavalkumar Vijaybhai Lad under the guidance of the university supervisors.

The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Place, date Signature

Acknowledgments

The journey of this thesis has been filled with challenges, discoveries, and countless learning experiences. Along the way, I have been fortunate to receive support, guidance, and encouragement from many incredible individuals, to whom I am deeply thankful.

First and foremost, I am grateful to the Lord Almighty for His guidance, strength, and support, which have enabled me to complete this work.

I extend my sincere thanks to my supervisor, **Prof. Dr.-Ing. Frank Schrödel**, whose insights and expertise have been invaluable throughout this research. His unique way of analyzing problems encouraged me to explore solutions beyond my initial scope, pushing me to tackle challenges I never thought I could handle. His constructive feedback and constant support have played a key role in shaping both this thesis and my growth as a researcher.

I would also like to express my gratitude to my co-supervisor, **M.Eng. Yekaterina Strigina**, for her timely guidance and thoughtful direction whenever I faced obstacles. Her ability to provide a fresh perspective helped me navigate complex problems and find solutions that would have otherwise gone unnoticed. Her support at crucial moments made a significant difference in overcoming key challenges during this work.

This research would not have been the same without the help of my friends and colleagues. Harsh Somani for his assistance with testing, implementation, and data collection, Mayank Khandelwal and the Field Robot Event team 2024 for designing and building the GrassHopper robot, and Jovan Oliveira and Swaraj Tendulkar for their guidance in perception-related aspects. Their contributions and collaboration have been invaluable in bringing this research to life.

Finally, my deepest thanks go to my family and friends, whose unwavering support and belief in me have been my greatest source of motivation. Their encouragement pushed me beyond my own self-doubts, reminding me that I was capable of more than I often believed. Their faith and constant support have been the backbone of this journey, and for that, I am truly grateful.

This thesis stands as a reflection of all the support, patience, and guidance I have received, and I am thankful to each and every person who has been a part of it.

Abstract

The increasing deployment of autonomous mobile robots in outdoor environments requires the development of intelligent decision-making frameworks capable of navigating complex and dynamic environment. Traditional navigation techniques, including rule-based and classical motion-planning approaches, lack adaptability and require extensive parameter tuning to handle real-world uncertainties. To address these limitations, this research presents a framework based on Reinforcement Learning (RL) for tactical decision-making, enabling a mobile robot to autonomously navigate toward a predefined goal while avoiding static and dynamic obstacles in real time.

This study leverages the Proximal Policy Optimization (PPO) algorithm, an RL approach well suited for continuous learning and stable policy optimization. The proposed framework is implemented on a four-wheel-drive mobile robot, GrassHopper, which integrates key perception and localization technologies, including Light Detection and Ranging (LiDAR), a Depth Camera, a Global Navigation Satellite System (GNSS), and an Inertial Measurement Unit (IMU). These sensors facilitate high-precision obstacle detection, path optimization, and real-time maneuvering. To create an efficient and structured learning environment, the system incorporates multiple advanced software frameworks, including Robot Operating System 2 (ROS2), Gazebo, OpenAI Gym, and Stable Baselines3, enabling scalable training and simulation-based validation.

The research follows a two-stage development pipeline comprising Software-in-the-Loop (SIL) training and Hardware-in-the-Loop (HIL) validation. In the SIL phase, a custom RL training environment is designed using OpenAI Gym within the Gazebo simulator. The RL agent undergoes iterative training, refining its policy through interaction with a simulated environment, while optimizing its decision-making based on sensor data, perception algorithms, localization estimates, and trajectory planning strategies. The RL agent learns to select from three maneuver options: Global Path Follow, Local Path Follow, and Full Stop based on real-time environmental conditions, ensuring adaptability to both static and dynamic obstacles. A task-specific reward function is formulated to encourage safe and efficient navigation while dynamically adapting to the robot's interactions with its surroundings. However, further refinements are necessary to improve generalization and robustness.

Following simulation-based training, individual system components undergo HIL testing to validate their functionality under real-world conditions. Key modules, including GNSS-based localization using an Extended Kalman Filter (EKF), dynamic occupancy grid generation, object classification with depth estimation, and trajectory execution via a Pure Pursuit Controller, are systematically evaluated on the physical GrassHopper platform. Although these subsystems perform as expected in controlled testing environ-

ments, constraints related to real-time implementation prevent full-scale deployment of the RL model in HIL, making hardware-based reinforcement learning deployment a subject for future exploration.

By integrating RL into the tactical decision-making pipeline, this research advances autonomous navigation by enabling data-driven adaptive control mechanisms without relying on predefined rule sets. The findings contribute to the ongoing development of real-world robotic navigation, providing a foundation for future research on optimizing reward functions, improving real-world deployment strategies, and incorporating additional safety constraints for improved obstacle avoidance and maneuverability.

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Definition
2D	2 Dimension
3D	3 Dimension
4-WD	4 Wheel Drive
A2C	Advantage Actor Critic
A3C	Asynchronous Advantage Actor-Critic
AD	Autonomous Driving
AI	Artificial Intelligence
ANavS®	Advanced Navigation Solutions
API	Application Programming Interface
Avs	Autonomous Vehicles
CAGR	Compound Annual Growth Rate
DARPA	Defense Advanced Research Projects Agency
DC	Direct Current
DQN	Deep Q-Network
EKF	Extended Kalman Filter
FoV	Field of View
FSM	Finite State Machines
GLONASS	Global Navigation Satellite System
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HIL	Hardware-In-Loop
IL	Imitation Learning
IMU	Inertial Measurement Unit
KL	Kullback-Leibler
LiDAR	Light Detection and Ranging
LiPo	Lithium Polymer
MDP	Markov Decision Process
ML	Machine Learning
PC	Personal Computer
PDB	Power Distribution Board
POMDP	Partially Observable Markov Decision Process
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
ROS	Robot Operating System
RRT	Rapidly-exploring Random Trees
RTK	Real-Time Kinematic

Abbreviation	Definition
RViz	ROS Visualization
SB3	Stable Baselines 3
SIL	Software-In-Loop
SLAM	Simultaneous Localization and Mapping
ToF	Time of Flight
TRPO	Trust Region Policy Optimization
URDF	Unified Robotics Description Format
USB	Universal Serial Bus
VESC	Vedder Electronic Speed Controller
XML	Extensible Markup Language
YOLO	You Only Look Once

Chapter 1

Introduction

Mobile robotics is currently one of the fastest-growing fields of scientific research, driven by the remarkable capabilities of mobile robots to substitute humans in numerous applications. These applications range from surveillance, planetary exploration, patrolling, and emergency rescue operations to reconnaissance, petrochemical applications, industrial automation, construction, entertainment, museum guidance, personal services, interventions in extreme environments, transportation, and medical care.[43] Many of these applications are already commercially available, reflecting the rapid advancements in this field.

A robot is autonomous when it can determine the actions required to perform a task using a perception system to understand its surroundings. It also needs a cognition unit or control system to coordinate all the subsystems that comprise the robot.[43] The core aspects of mobile robotics include locomotion, perception, cognition, and navigation. Locomotion challenges are addressed through the study of mechanics, kinematics, dynamics, and control theory. Perception involves signal analysis and specialized fields like computer vision and sensor technologies. Cognition is responsible for analyzing sensor input and making decisions to achieve the robot's objectives, acting as the control system's brain. Navigation encompasses planning algorithms, information theory, and artificial intelligence to enable the robot to move efficiently and effectively in its environment.[43]

One prominent application of mobile robotics is in "last mile" delivery services, where autonomous robots deliver goods to their final destinations. These robots can operate independently but may require remote human intervention in situations they cannot resolve alone, such as when encountering obstacles. Last mile delivery robots are employed in various contexts, including food delivery, package delivery, hospital logistics, and room service.[46]

The market for autonomous last mile delivery is becoming popular, with three main types of products: autonomous delivery vans, delivery robots, and delivery drones.[27] Last mile delivery is typically the most expensive part of the delivery process. However, autonomous solutions have the potential to significantly reduce costs and improve efficiency in an environmentally friendly manner. Estimates suggest that adopting autonomous last mile delivery technologies could reduce current costs by 55% in the short term and over 80% in the long term.[27]

The global market for delivery robots was valued at $300 million in 2021, and it is projected to grow at a compound annual growth rate (CAGR) of 30.3% through 2030.[9] As this market expands, it is anticipated that robots delivering packages and transporting goods will become a common sight in urban areas.

Beyond cost savings, autonomous delivery robots also offer environmental benefits. It is estimated that last mile delivery accounts for more than 20% of urban pollution[5], a figure that could be significantly reduced with the implementation of efficient autonomous electric robots. Researchers also highlight that these robots can lower transportation costs, with last mile costs currently constituting 40% of the total delivery expense.[5] Autonomous delivery vehicles represent a substantial reallocation of carrier costs, making services more economical and efficient than traditional delivery methods.

1.1 Thesis Motivation

The motivation behind this thesis arises from the increasing need for efficient and reliable autonomous navigation systems in mobile robots, particularly in outdoor environments where surroundings are unknown and dynamic. Traditional navigation methods often struggle with dynamic and unstructured settings, making it challenging for robots to make informed decisions about movement and tactical planning. Effective navigation in such scenarios requires not only the ability to traverse from one point to another but also the capacity to plan and adapt strategies based on situational demands. Reinforcement learning (RL), a branch of machine learning, offers a potential approach to enhance these tactical planning capabilities by enabling robots to learn from real-time environmental feedback. While RL provides a way for robots to improve their navigation strategies over time, its effectiveness depends on various factors, such as the complexity of the environment and the quality of training data.

The challenges of autonomous outdoor navigation extend beyond mobile robots in general and are particularly evident in last-mile delivery, where efficient route planning and adaptability are crucial. The final segment of the supply chain, which involves transporting goods from distribution hubs to end-users, is often inefficient and costly due to unpredictable traffic conditions and urban constraints. Autonomous delivery robots offer a promising solution by operating during low-traffic periods, reducing transportation expenses, and minimizing urban pollution. Reinforcement learning can play a role in improving their ability to navigate complex and dynamic environments, helping them adapt to obstacles and optimize their routes. While this approach has the potential to enhance the overall effectiveness of autonomous delivery systems, its impact depends on various factors, including the quality of training data and real-world implementation challenges.

1.2 Thesis Goal

The primary goal of this thesis is to develop a reinforcement learning-based framework for optimizing outdoor navigation in mobile robots. To achieve this, the thesis aims to:

- Design and implement a reinforcement learning algorithm for the tactical planning of mobile robots in outdoor navigation.
- Integrate this algorithm into a mobile robot's control system to enhance its autonomous decision-making capabilities.
- Conduct extensive training and experiments to evaluate the performance of the proposed system in various outdoor scenarios using Gazebo simulation.
- Demonstrate the practical applicability of the system in real-world tasks, such as autonomous delivery services.

By accomplishing these objectives, the thesis seeks to advance the field of mobile robotics, providing a robust solution for autonomous outdoor navigation and addressing critical challenges in current applications.

1.3 Thesis Outline

Following the introduction, the thesis is structured into the following chapters:

- **Chapter 2** explores outdoor mobile robots, focusing on essential hardware components such as LiDAR, depth cameras, GNSS, and IMU, which support autonomous navigation. It also reviews decision-making methods, emphasizing reinforcement learning as a key approach for optimizing tactical planning in dynamic environments.
- **Chapter 3** defines the problem setting and introduces the conceptual framework for reinforcement learning-based tactical planning. It outlines the hardware and software architectures, perception and decision-making layers, and trajectory planning, emphasizing how RL can enhance mobile robot navigation in dynamic outdoor environments.
- **Chapter 4** details the implementation of the proposed approach, beginning with software-in-the-loop (SIL) simulations in Gazebo to develop and test the RL agent. It then discusses the reinforcement learning training process using a custom Gym environment and the transition to hardware-in-the-loop (HIL) testing for real-world validation.
- **Chapter 5** evaluates the trained RL model in Software-in-the-Loop (SIL) simulation, assessing its tactical decision-making, adaptability, and navigation efficiency. Performance is measured through success and failure rates, demonstrating the model's reliability. The chapter also discusses Hardware-in-the-Loop (HIL) feasibility, highlighting challenges preventing real-world deployment.
- **Chapter 6** concludes the thesis by summarizing key findings and discussing limitations. It also provides insights into possible future research directions and improvements in autonomous mobile robot navigation.

Chapter 2

Fundamentals

Chapter 2 opens by examining the essential technologies that power autonomous outdoor navigation, including sensors like Light Detection and Ranging (LiDAR), depth cameras, Global Navigation Satellite System (GNSS) and Inertial Measurement Unit (IMU), which help robots to sense surroundings and localize in it. The chapter then transitions into a comprehensive analysis of decision-making frameworks, ranging from classical rule-based approaches to modern machine learning techniques, with a focus on reinforcement learning. The aim is to establish a clear understanding of how these systems and strategies integrate to optimize tactical planning for mobile robots in complex, real-world scenarios.

2.1 Outdoor Mobile Robots

Over the last few years, the development of mobile robots has significantly advanced, driven by the need for efficient and autonomous solutions for outdoor navigation. These advancements are propelled by various applications, ranging from agriculture and environmental monitoring to urban delivery services. The goal is to create mobile robots capable of navigating diverse and dynamic outdoor environments autonomously, reliably, and safely.

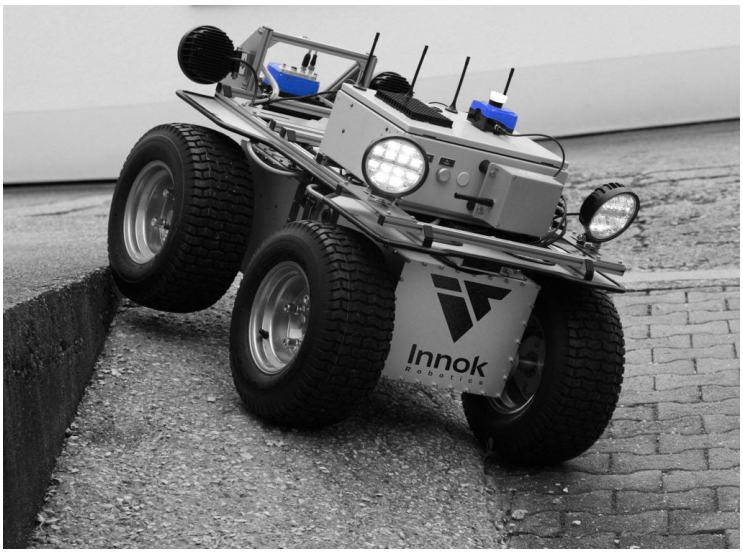

Figure 2.1: HEROS robot platform. [21]

The integration of mobile robots into outdoor tasks promises numerous benefits. For instance, in the region of last-mile delivery, these robots can transport packages from local distribution centers directly to consumers, optimizing the delivery process and reducing operational costs. In agriculture, mobile robots can automate tasks such as planting, harvesting, and monitoring crop health, thereby increasing productivity and precision. Similarly, in environmental monitoring, these robots can collect data from hard-to-reach areas, providing valuable insights for research.

However, outdoor mobile robots are designed to operate in complex and unpredictable environments. Unlike indoor robots, which navigate in relatively controlled and stable environments, outdoor robots face numerous challenges. These include variable terrain, obstacles, dynamic elements such as pedestrians, and fluctuating weather conditions.

To achieve autonomous navigation, these robots are equipped with a range of sensors and technologies, including GNSS, LiDAR, depth cameras, and IMU. These components work together to provide the robot with an understanding of its surrounding environment, enabling it to make well-informed decisions and navigate effectively through unknown and uncertain outdoor environment.

2.1.1 Hardware Components: Sensors

For mobile robot systems to achieve true autonomy, they must be equipped with sensor components, information processing capabilities, and actuation mechanisms. This section explores some of the most commonly used sensors in mobile robots for autonomous navigation and explains how they function.

2.1.1.1 Light Detection and Ranging

LiDAR is a cutting-edge technology in the field of autonomous navigation. In LiDAR, laser beams are emitted from a source (transmitter) and reflected back to the receiver after hitting objects in the environment. The system detects the reflected light and calculates the Time of Flight (ToF) to determine the distance of each object, creating a detailed distance map of the scene.[50] This process is illustrated in Figure 2.2, which depicts the basic working principle of a LiDAR sensor.

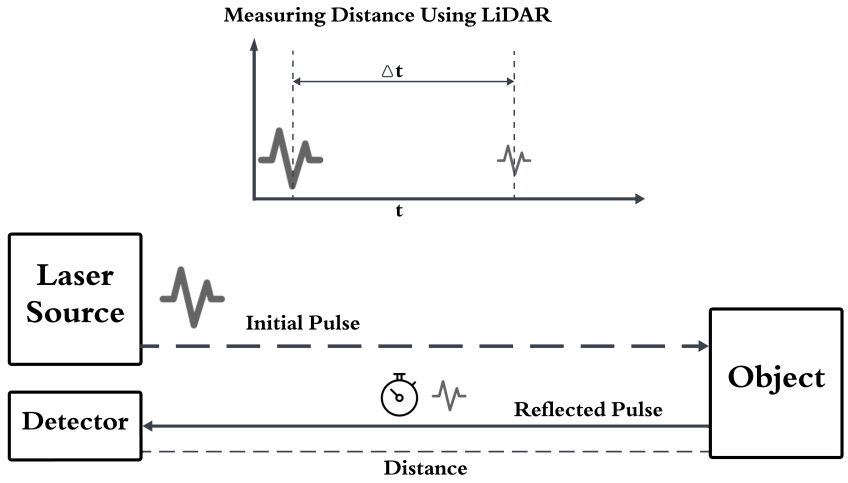

Figure 2.2: LiDAR Working Principle. [50]

2D LiDAR sensors are important in generating accurate two-dimensional representation of the environment. They use near-infrared light to measure distances, emitting thou-

sands of laser pulses per second in a sweeping motion across a 360-degree Field of View (FoV). These pulses create a 'point cloud', which forms a comprehensive, measurable map of the surroundings in a single plane.

Despite being limited to a single plane, 2D LiDAR is highly effective for many applications. These sensors can detect obstacles, track movement, and map environments with a high degree of precision. LiDAR sensors can range up to $100+$ *meters* [29], depending on the models and environmental conditions. By processing the point cloud data through algorithms and AI, mobile robots can create dynamic and static 2D maps of their surroundings. The static map is generated by capturing the fixed features of the environment, such as walls, buildings, and other permanent structures. In contrast, the dynamic map accounts for temporary objects or obstacles that move, like pedestrians or vehicles, allowing the robot to track real-time changes in the environment. These maps enable the robot to plan paths and navigate safely, adjusting in real time based on both static and dynamic elements detected by the LiDAR system.

The features of 2D LiDAR make it suitable for a variety of outdoor navigation tasks. Its high resolution and ability to provide real-time data are crucial for applications that require quick and accurate spatial awareness. In summary, 2D LiDAR sensors are a fundamental component of autonomous outdoor navigation systems.

2.1.1.2 Depth Camera

Cameras are one of the oldest and most advanced sensor technologies used in autonomous navigation. Some autonomous vehicle manufacturers, such as Tesla, prefer to rely primarily on cameras for data acquisition instead of LiDAR or radar systems.[39] This preference is due to the natural human-level understandability of camera data. Unlike LiDAR, which generates point clouds that can be difficult for neural networks to interpret, cameras capture natural reflected light, producing images that humans can easily understand and label.

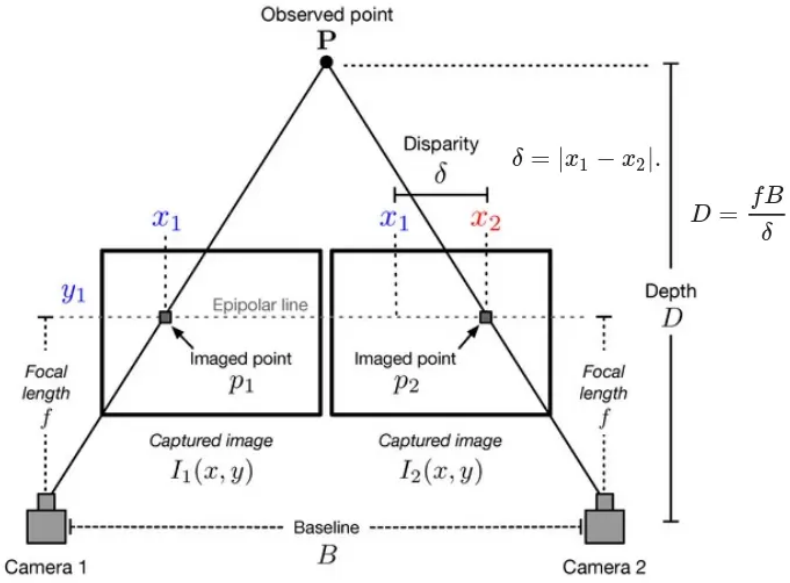

Figure 2.3: Depth Estimation in Stereo Vision. [44]

Depth cameras, in particular, combine color imaging with depth sensing. Stereo vision, the most common technique used in these cameras, works similarly to human binocular vision,as shown in Figure 2.3.[22] By comparing two images captured from slightly

different viewpoints, the system detects the disparity between them, which is used to estimate the distance to objects in the scene. This depth calculation is achieved through specialized algorithms that process the disparity and generate a depth map, enabling the camera to create a 3D representation of the environment. Sufficient texture and detail in the images are necessary for effective depth estimation, making stereo vision cameras especially well-suited for outdoor applications with large fields of view.[22]

Key features of depth cameras include high resolution and accuracy, real-time data processing, enhanced object detection and recognition, and versatility in various lighting conditions. These capabilities make depth cameras essential for tasks like obstacle avoidance, mapping, and path planning, significantly enhancing the robot's ability to navigate complex environments autonomously.

2.1.1.3 Global Navigation Satellite System

GNSS receivers are devices used to determine precise geographic locations on Earth's surface. They rely on signals from a network of satellites orbiting the planet, which allow the receiver to calculate its exact three-dimensional position based on signals from multiple satellites. To obtain accurate location data, the GNSS receiver must maintain contact with at least four satellites at any given time.[15] This requirement ensures high accuracy, as GNSS systems, which include the United States' Global Positioning System (GPS) and the Russian Federation's GLObal NAvigation Satellite System (GLONASS), and Europe's European Satellite Navigation System (GALILEO),[18] are designed to ensure that sufficient satellites are visible from any point on Earth at all times.

GNSS operates using a technique called trilateration [56], as shown in Figure 2.4, which is essential for calculating location, velocity, and elevation. Trilateration works by using distance measurements from multiple satellites without relying on the measurement of angles. Data from a single satellite provides a general location within a large spherical area. Adding data from a second satellite narrows this to the region where the two spheres overlap. Data from a third satellite allows for a precise position on the Earth's surface.[56]

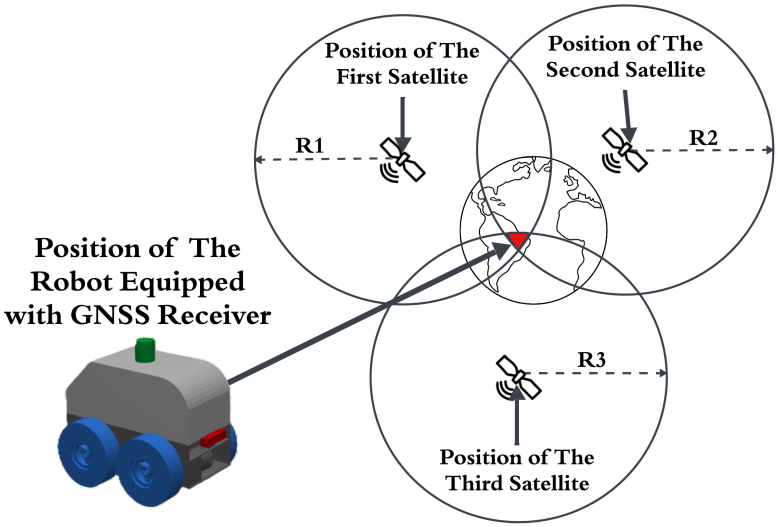

Figure 2.4: Localization using Trilateration technique with GNSS. [55]

GNSS is essential for mobile robot outdoor navigation, providing highly accurate positioning to determine precise locations on Earth's surface. It supports efficient path

planning by supplying real-time location data, enabling robots to avoid obstacles and optimize routes. When combined with other sensors, GNSS facilitates the creation of detailed maps of outdoor environments. These capabilities empower mobile robots to autonomously navigate complex settings and perform tasks such as delivery and surveillance with efficiency.[14]

2.1.1.4 Inertial Measurement Unit

An Inertial Measurement Unit (IMU) is an advanced sensing device critical for measuring and tracking an object's orientation, velocity, and movement in three-dimensional space. Typically, an IMU comprises a 3-axis accelerometer and a 3-axis gyroscope, making it a 6-axis system. Some IMUs also include a 3-axis magnetometer, turning them into 9-axis systems[8], further enhancing their capability to provide precise heading and orientation data. This comprehensive data is invaluable for navigation and motion tracking, especially in environments where GNSS signals are unreliable, such as tunnels or areas with significant electronic interference.

The IMU operates through its core components, each playing a unique role:

1. **Accelerometer:** Measures linear acceleration by detecting the rate of change in velocity. Inside, a proof mass moves relative to its frame when acceleration occurs. This movement is resisted by etched springs, and the distance the mass moves is proportional to the acceleration experienced.[47] The displacement is measured through changes in capacitance, which is then processed into acceleration data. This process is illustrated in Figure 2.5.

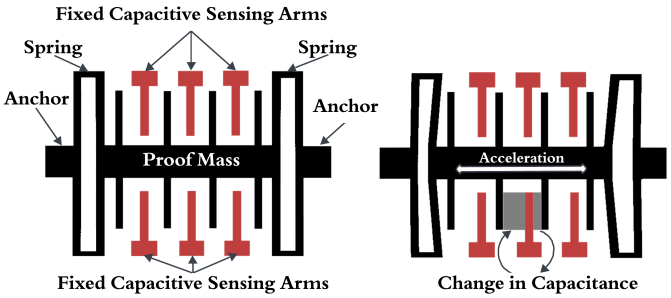

Figure 2.5: MEMS Accelerometer Sensor. [47]

2. **Gyroscope:** Measures angular velocity to determine how fast and in which direction an object rotates. It uses the Coriolis effect, where a resonating mass inside the gyroscope experiences a perpendicular force during rotation.[47] This force is countered by springs and measured through changes in capacitance, which are processed into rotational velocity data. This process is illustrated in Figure 2.6.

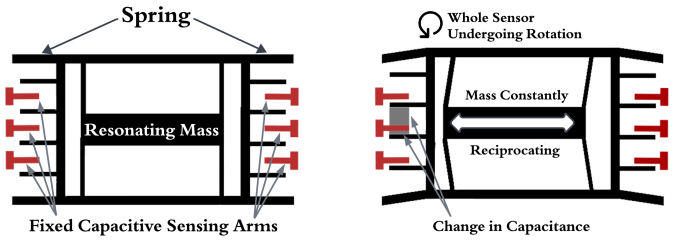

Figure 2.6: MEMS Gyroscope Sensor. [47]

3. **Magnetometer:** Measures magnetic field strength, acting as a digital compass. It uses the Hall Effect, where moving electrons in a conductor are deflected by an external magnetic field, creating a measurable voltage proportional to the field's strength.[47] This data provides heading and orientation relative to Earth's magnetic north. This process is illustrated in Figure 2.7.

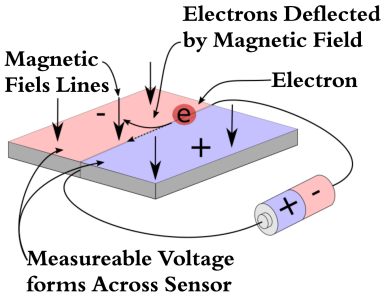

Figure 2.7: MEMS Magnetometer Sensor. [47]

IMUs are essential in robotics, providing real-time data on acceleration, angular velocity, and orientation to help robots understand their position and movement. They enhance navigation by filling gaps in GNSS coverage, enabling dead reckoning and stabilization. IMUs support accurate path planning, trajectory adjustments, and balance, allowing robots to navigate autonomously, follow paths precisely, and perform tasks like delivery and object handling efficiently.

2.1.2 Functional Architecture for Autonomous Driving and Mobile Robot Navigation

A mobile robot, functioning as an intelligent system, must sense its surroundings, perceive its working environment, plan a trajectory, and execute appropriate reactions using the collected information. Robotic control architectures define how these capabilities are integrated to enable autonomous navigation. These control architectures can be categorized into three primary types: deliberative (centralized) navigation, reactive (behavior-based) navigation, and hybrid (deliberative-reactive) navigation.[36]

- **Deliberative (Centralized) Navigation:** Deliberative planning involves searching for the optimal path and generating a plan to reach a goal.[36] The execution system then performs actions based on a static model of the environment and the planning system to accomplish a given task.
- **Reactive (Behaviour-Based) Navigation:** Reactive architectures generate control commands based on the currently perceived environment, eliminating the need for a complete model of the environment.[36] Sensed data is directly coupled to the robot's actuators through specific transfer functions known as task-achieving modules or behaviours.
- **Hybrid (Deliberative-Reactive) Navigation:** Hybrid architectures combine the advantages of deliberative and reactive approaches, offering the benefits of planning in deliberative architectures and the quick response capabilities of reactive architectures in dynamic or unknown environments. Common hybrid control architectures consist of three layers:[36]

    1. **Deliberative Layer:** Responsible for high-level strategic mission planning, including mission adaptation. This layer deals with sensor fusion, map building, and overall planning.[36]

2. **Control Execution Layer:** Coordinates the interaction between the deliberative and reactive layers, ensuring that the appropriate tactics are executed using context-dependent rules.[36] The term tactic, in this context, refers to a predefined, ordered, and structured set of actions.[17] This layer acts as a behaviour coordinator.
3. **Reactive (Behaviour-Based) Layer:** Operates with metric information and numerical control, generating the robot's actions based on immediate sensory input.[36]

Choosing the most appropriate functional architecture for AD (Autonomous Driving) usually depends on the application and the corresponding requirements. These architectures were the breakthrough for developing functional hybrid architectures for AD and are particularly characterized by their hierarchical planning. Furthermore, the well-known DARPA (Defense Advanced Research Projects Agency) challenge gave rise to various hybrid architectures, where each architecture was specialized for solving the course defined by the challenge.[53] Urban scenarios, however, were only partially considered. Later architectures focused more on urban environments. Further research in functional architectures for AD finally led to a three-layered decision-making approach consisting of a strategic, a tactical, and an operational layer.[53]

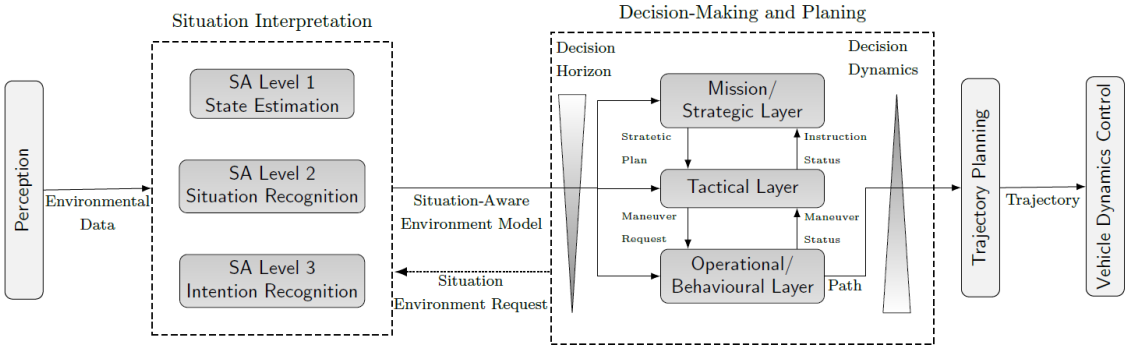

Figure 2.8: Functional Architecture for AD-System. [53]

In this context, the reactive functional architecture that emphasizes real-time decision-making and adaptability. This architecture is particularly suited for dynamic and unpredictable outdoor environments, where quick responses to changes in the surroundings are critical. Integrating RL enhances the robot's ability to navigate efficiently and autonomously while continuously improving its performance through learning and adaptation. Figure 2.8 provides an overview of our functional architecture. The key modules involved in the functional architecture of operation of an autonomous mobile robot:

1. **Perception:** This module collects and processes environmental data through sensors such as cameras, LiDAR, radar, and ultrasonic sensors, generating an understanding of the robot's surroundings.[53]
2. **Situation Interpretation:** This module is composed of three levels:
   - **State Estimation:** Estimation and representation of the current environment and entities.[53]
   - **Situation Recognition:** Comprehension, classifying and combining with mission-related goal to the current situation.[53]
   - **Intention Recognition:** Prediction and intention detection of future situations and status.[53]

3. **Decision-Making and Planning:**
    - **Mission/Strategic Layer:** Handles high-level planning and decision-making, determining the overall route and mission objectives.[53]
    - **Tactical Layer:** Focuses on mid-term planning and maneuvering, translating strategic plans into specific maneuvers such as obstacle avoidance, follow the path and speed adjustments.[53]
    - **Operational/Behavioural Layer:** Manages short-term, immediate actions, providing precise control commands for smooth and safe robot operation.[53]
4. **Trajectory Planning:** Translates geometric paths with velocity annotations into trajectories for the robot to follow, considering vehicle dynamics, road conditions, and safety constraints.[53]
5. **Vehicle Dynamics Control:** Translates the planned trajectory into control commands for the mobile robot's actuators (linear velocity, angular velocity, brakes), ensuring accurate path following.[53]

With a well-defined functional architecture for autonomous driving and mobile robot navigation in place, the next crucial aspect to consider is how these systems make informed decisions in dynamic environments. In the following section, we will explore various decision-making solutions relevant to autonomous driving, analysing their strengths and applicability to mobile robot navigation.

2.2 Analyse Decision-Making Relevant Solutions for Autonomous Driving

Autonomous driving technology has rapidly advanced over the past decade, driven by the need for safer, more efficient, and convenient transportation systems. Decision-making is a critical component in Autonomous Vehicles (AVs), ensuring that these systems can navigate complex environments, respond to dynamic situations, and make safe and effective choices in real-time. The tactical decision-making approaches for autonomous vehicles roughly fall into three main directions: classical approaches, utility/reward-based approaches, and machine learning approaches as shown in Figure 2.9.[31]

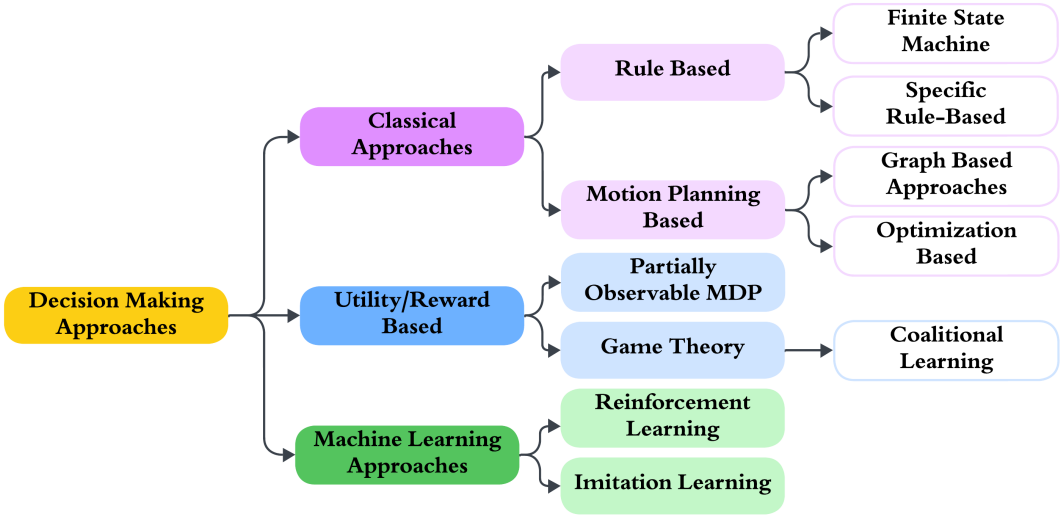

Figure 2.9: Categorization of Decision-Making Approaches for Autonomous Vehicles. [31]

2.2.1 Classical Approaches

Classical approaches to autonomous driving rely on established algorithms and deterministic methods for decision-making and control. They emphasize rule-based logic and mathematical models to navigate structured environments, ensuring predictable and reliable behavior. These methods are computationally efficient and interpretable but struggle with complex or unpredictable scenarios requiring adaptability and dynamic learning. Below are two key categories within classical approaches.

2.2.1.1 Rule-Based Approaches

A Rule-Based system in autonomous driving uses a set of predefined rules to make decisions, where knowledge is typically encoded in the form of if-then-else statements. These rules capture expert knowledge within a specific domain and enable the system to make decisions based on this embedded logic. The rule-based approach is often represented through frameworks like Finite State Machines (FSM) or specific-rule-based methods.[31] For example, mission tasks can be broken down into high-level behaviors such as ”drive down road,” and ”handle intersection,” with simpler sub-behaviors aimed at completing mission planning tasksas a part of Mission/Strategic Layer explained in Section 2.1.2. However, they are limited by their lack of flexibility and adaptability, particularly in complex, dynamic environments. Due to their deterministic nature, rule-based systems struggle to handle the unpredictability of real-world driving scenarios, where rapid, context-sensitive decision-making is required.[31]

2.2.1.2 Motion Planning Based

Motion planning-based systems in autonomous driving involve generating a feasible path for the vehicle by considering factors such as obstacles, vehicle dynamics, and traffic rules. Common algorithms used in this approach include A*, Dijkstra’s algorithm, and Rapidly-exploring Random Trees (RRT), which are particularly effective in structured environments where the vehicle’s movements are predictable.[31] Additionally, a prediction model is often used to anticipate the motion and intentions of surrounding vehicles, allowing the vehicle to plan its behavior reactively. This approach aligns well with the Tactical Layer, as explained in Section 2.1.2, which focuses on mid-term planning and maneuvering. By translating strategic plans into specific maneuvers such as obstacle avoidance, path-following, and speed adjustments, motion planning ensures that vehicles can navigate their environments safely and efficiently.

While classical approaches have laid the groundwork for autonomous vehicle development, their inherent limitations make them unsuitable for dynamic and unstructured environments. The lack of adaptability and context-aware decision-making restricts their performance in real-world driving scenarios, emphasizing the need for more advanced and learning-based systems to address these challenges.

2.2.2 Utility/Reward Based

Utility or reward-based approaches in autonomous driving focus on optimizing decision-making by evaluating the long-term outcomes of actions. These methods seek to maximize cumulative rewards while navigating dynamic environments, often using models that account for uncertainties and interactions between agents.

2.2.2.1 Partially Observable Markov Decision Process

The POMDP extends the traditional Markov Decision Process (MDP) to handle situations where an autonomous vehicle has incomplete or noisy information about its environment.[31] In such scenarios, the intentions of other agents and the true state of the environment are encoded in hidden variables, which cannot be directly observed. POMDPs allow the agent to make decisions based on estimates of the true state, considering uncertainty in both the environment and the agent's observations. This approach is particularly useful in automated driving, where uncertainty and unpredictability are inherent. The agent evaluates action sequences by considering their long-term reward outcomes, aiming to optimize its decision-making over time. While POMDPs are powerful for modeling uncertainty, they suffer from significant computational challenges.[31] Their complexity increases exponentially with the number of states and the planning horizon, leading to what is known as the 'curse of history' and the 'curse of dimensionality'. These limitations make real-time implementation of POMDPs difficult, as solving them requires vast computational resources.[31]

2.2.2.2 Game Theory

Game-Theory approaches consider the interactions between multiple agents, such as other vehicles and pedestrians, as strategic games. This method is valuable for predicting the actions of other road users and making optimal decisions in multi-agent environments.[31] By modeling these interactions, AVs can anticipate the behavior of others and adjust their strategies accordingly, enhancing safety and efficiency. However, the complexity and need for simplifying assumptions can limit the practical application of game theory. These models often require extensive computational resources and can become intractable in highly dynamic environments with numerous interacting agents.[31] Despite these challenges, game-theoretic approaches offer significant advantages in understanding and navigating the strategic behaviors of multiple agents, making them a valuable tool in the development of autonomous driving systems.[31]

While utility/reward-based methods offer significant advantages in handling uncertainties and strategic agent interactions, they are often computationally demanding and challenging to implement in real-time. Their reliance on high computational resources and difficulty in maintaining efficiency under rapidly changing conditions make them less suitable for dynamic and complex driving environments, where quick, context-aware decision-making is essential.

2.2.3 Machine Learning Approaches

Machine learning (ML) has emerged as a transformative enabler in autonomous driving and mobile robot navigation, offering data-driven techniques to improve perception, decision-making, and control. By leveraging large datasets and powerful algorithms, ML allows systems to learn, adapt, and optimize their behavior in complex and dynamic environments. This section explores key machine learning approaches, including Imitation Learning (IL) and Reinforcement Learning (RL), followed by an introduction to core machine learning concepts driving the development of autonomous systems.

2.2.3.1 Imitation Learning

IL is a supervised learning approach that involves training models to mimic expert behavior by using datasets of human demonstrations. This method is particularly effective in autonomous driving scenarios where large-scale datasets of expert driving are readily available, allowing systems to learn from these demonstrations. [31] While IL

can bring performance close to human levels, it faces several limitations. These include the high data requirements and the challenge of acquiring sufficient supervised data in all scenarios. Additionally, IL reliance on the quality of expert policies means that it cannot outperform the suboptimal expert it mimics.[31] Computational constraints also pose a challenge, particularly for real-time applications requiring onboard processing. Traditional IL methods struggle with learning hierarchical policies, unobserved states, highlighting the need for more robust and adaptable approaches.

2.2.3.2 Reinforcement Learning

RL is a machine learning paradigm where agents learn to make decisions by receiving rewards or penalties based on their actions.[31] This learning process allows the agent to improve its performance over time by continuously interacting with the environment. In autonomous driving and mobile robot navigation, RL is especially powerful as it enables the agent to develop optimal policies for navigating complex and dynamic environments. By aiming to maximize the expected cumulative reward, RL agents can learn to make precise and efficient decisions that enhance their autonomous capabilities. This iterative learning process helps the agent adapt to new situations and refine its strategies, leading to more robust and intelligent autonomous systems.[31] The detailed information on RL will be discussed further in Section 2.3.

2.2.3.3 Understanding Machine Learning Algorithms

Machine learning (ML) is a subset of Artificial Intelligence (AI) that enables machines to learn from data and make informed decisions without explicit programming.[19] This learning process allows machines to go beyond their initial training data, leveraging algorithms to form knowledge structures that can generalize to new, unseen situations.[19] In the context of autonomous driving, ML plays a vital role by allowing vehicles to navigate complex environments with advanced decision-making capabilities.

At the core of ML are algorithms—structured processes that enable AI systems to discover insights, predict outcomes, and identify patterns.[6] These algorithms are computational models designed to understand patterns and forecast results based on data.[30] Training them on large datasets enhances their accuracy and reliability, leading to better performance in tasks such as navigation, obstacle avoidance, and decision-making. By learning from extensive data, ML algorithms enable mobile robots to understand their environments, adapt to new situations, and optimize their movements for efficient and safe operation. These algorithms reinforce modern AI, enabling computers to make decisions based on learned experiences rather than static programming.

Types of Machine Learning Algorithms:

ML algorithms can be broadly categorized into the following types:

- **Supervised Learning:** In supervised learning, models are trained on labelled data, meaning the desired outputs are known.[30] The algorithm learns to map inputs to outputs by analysing examples with known outcomes.
- **Unsupervised Learning:** Unsupervised learning deals with unlabelled data. The algorithm seeks to find hidden patterns or intrinsic structures within the data, identifying relationships without prior knowledge of the outputs.[30]
- **Semi-supervised Learning:** This approach combines both labelled and unlabelled data.[54] The labelled data provides guidance, while the unlabelled data helps the algorithm generalize better. This method can improve learning efficiency and accuracy.

- **Reinforcement Learning:** Reinforcement learning involves training agents to make a sequence of decisions by rewarding positive actions and penalizing negative ones.[30] The algorithm explores different strategies through trial and error, learning from past experiences to optimize its decisions.

Recent advancements in deep learning have significantly enhanced the capabilities of machine learning in autonomous driving. Unlike traditional methods that rely on handcrafted rules, ML algorithms scale effectively with data, enhancing performance as more data is used for training.[17] This scalability allows for managing a wide array of driving scenarios, making ML a powerful tool for the development of autonomous vehicles. Numerous ML approaches have been researched and implemented, contributing to higher levels of autonomy in driving systems.[17]

Among the various ML approaches, RL stands out for its ability to enable autonomous agents to learn optimal decision-making strategies through direct interaction with the environment. The flexibility and adaptability of RL make it a promising avenue for handling complex and dynamic navigation challenges, setting the stage for a deeper exploration in the next section.

2.3 Reinforcement Learning

This section provides a comprehensive overview of Reinforcement Learning (RL), covering its core concepts, essential functions, and problem-solving approaches. Key elements such as value functions, policy optimization, and reward design are explored to build a strong understanding of RL frameworks. The section concludes with a focused introduction to Proximal Policy Optimization (PPO), a central technique and the primary focus of this thesis.

2.3.1 Key Concepts of Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning that focuses on training agents to make a sequence of decisions through trial and error.[7] Unlike supervised learning, where agents learn from a fixed set of labelled data, RL involves learning directly from interactions with the environment. This learning framework is based on the principle of rewarding desirable behaviors and penalizing undesirable ones, thereby guiding the agent towards optimal behavior over time.[30] Through this iterative process, agents are able to refine their policies, improving their performance as they gain more experience.

One of the most compelling demonstrations of RL's potential is its success in training bots to stand out at complex games such as Dota, Go, and various Atari games.[32] These RL-trained agents have not only managed to outperform skilled human players but have also set new performance benchmarks in these games.[32] The ability of RL to drive such advancements highlights its adaptability and effectiveness, extending beyond gaming into practical applications in robotics and autonomous systems. By leveraging RL techniques, robots can learn to perform intricate tasks, navigate complex environments, and adapt to new challenges, thus pushing the boundaries of what autonomous systems can achieve.

Reinforcement Learning (RL) revolves around the interaction between two primary components: the **agent** and the **environment**, as shown in Figure 2.10. The agent is the learner or decision-maker, whereas the environment consists of everything the agent interacts with.[1] The agent observes the state of the environment, either partially or fully, and makes decisions by selecting actions from a predefined set. These actions cause

changes in the state of the environment, either directly due to the agent's actions or autonomously due to inherent dynamics of the environment.[1] Based on these actions and their outcomes, the agent receives rewards, which serve as feedback on the quality of its actions.

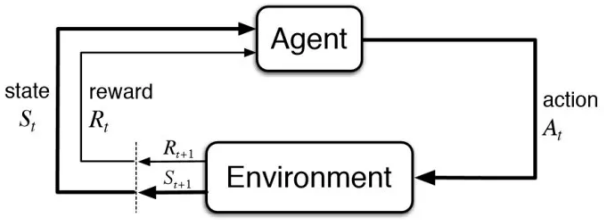

Figure 2.10: Agent-Environment Interaction Loop. [20]

The ultimate objective in RL is to develop a policy or a strategy that defines the action the agent should take in each state to maximize the cumulative reward over time. RL algorithms are designed to help the agent learn this optimal policy through continuous interaction with the environment, allowing the agent to improve its performance based on past experiences.[32]

To better apprehend the components involved in RL, it is important to familiarize oneself with the key terminology of RL.

- **State** S**:** The state S is a complete description of the world where the states are fully observable.[20]
- **Observation** O**:** An observation O is a partial description of a state, which may omit information.[1] It omit all the data that is not useful for the agent to complete the task.[32]
- **Action** A_t**:** An action A_t is a specific move or decision taken by the agent from a set of possible actions in order to reach a target. The agent performs actions in the environment to transition from one state to next state.[20]
- **Action Space:** The action space is the set of all possible actions that an agent can take in a given environment.[20] It defines the range of decisions available to the agent at any point in time. There are two main types of action spaces:
  1. **Discrete Action Space:** This consists of a finite number of actions.[20] For example, in a navigation task, the actions might be limited to turning left, turning right, moving forward, or stopping. Each action in a discrete action space is distinct and countable.
  2. **Continuous Action Space:** This consists of an infinite number of possible actions.[20] For instance, instead of discrete turns, an agent might continuously adjust the steering angle, allowing for precise movements. Continuous action spaces provide more precise control but are harder to manage.
- **Policies:** A policy is a rule used by an agent to decide what actions to take in a given state.[1] Essentially, the policy serves as the agent's decision-making mechanism, mapping states to actions. There are two main types of policies: deterministic and stochastic.[20]
  1. **Deterministic Policy** μ**:** A deterministic policy outputs a specific action with probability one for a given state. This means that, given the same state, the agent will always take the same action.[20] In a car driving scenario with

three actions (turn left, go straight, and turn right), an RL agent with a deterministic policy would consistently choose one of these actions for any given state without considering uncertainties. Deterministic policies are typically denoted by μ:[1]

$$A_t = \mu(S_t)$$

2. **Stochastic Policy π:** A stochastic policy, on the other hand, outputs a probability distribution over the possible actions for a given state.[20] This means that, given the same state, the agent can choose different actions according to the assigned probabilities. In the same car driving scenario, a stochastic policy might output probabilities like 20% for turning left, 50% for going straight, and 30% for turning right.[20] This type of policy is useful in non-deterministic environments where uncertainty and variability are inherent. Stochastic policies are usually denoted by π:[1]

$$A_t \sim \pi(\cdot \mid S_t)$$

- **Episode (Trajectory) τ:** An episode/trajectory τ refers to a sequence of state S_t and action A_t taken by the agent over a period of time.[32] It represents a complete cycle of interaction between the agent and environment, from the starting state to terminal state. An episode can be represented as:[32]

$$\tau = ((S_0, A_0), (S_1, A_1), (S_2, A_2), \ldots)$$

The end of an episode can occur under three main conditions:[32]

  - **Success:** The agent successfully completes the task.
  - **Truncated:** The agent does not finish the task within a predefined number of time-steps, causing the episode to be truncated.
  - **Failure:** The agent reaches a terminal state without accomplishing the task.

The state at the beginning of an episode is sampled from the initial-state distribution ρ_0.[1]

$$S_0 \sim \rho_0(\cdot)$$

The environment changes its state according to its natural laws and the last action performed by the agent. The transitions between states, from S_t to S_{t+1}, can be either deterministic or stochastic:[1]

  - **Deterministic:** The next state S_{t+1} is determined by a function of the current state S_t and the action A_t:[1]

$$S_{t+1} = f(S_t, A_t)$$

  - **Stochastic:** The next state S_{t+1} is determined by a probability distribution conditioned on the current state S_t and the action A_t:[1]

$$S_{t+1} \sim P(\cdot \mid S_t, A_t)$$

- **Step:** A step refers to a single transition within an episode, where the agent takes one action and the environment transitions to the next state. In each step, the agent observes the current state, selects an action, and receives feedback in the form of a new state and a reward. The sequence of steps throughout an episode allows the agent to learn and adapt its behavior over time.

- **Reward r:** The reward r is a signal that indicates how good or bad an action taken by the agent is in a given state.[20] The reward is a crucial component of RL as it directly influences the learning process of the agent. At a specific time-step t, the reward is often represented as:[20]

$$r_t = R(S_t, A_t, S_{t+1})$$

  This formula implies that reward r_t is the reward at time-step t depends on the current state S_t, the action A_t, and the next state S_{t+1}. In some cases, the reward might only depend on the current state S_t, simplified as $r_t = R(S_t)$, or on the state-action pair, $r_t = R(S_t, A_t)$.[1]

- **Reward Function R:** The reward function R is a function that provides a scaler feedback signal based on the state and action (and potentially the next state).[48] The design of the reward function is critical and can significantly impact the agent's learning performance and behavior.[1] The detailed information on reward function for RL will be discussed further in Section 2.3.3.

- **Return $R(\tau)$:** The return $R(\tau)$ is the cumulative reward that an agent aims to maximize over a trajectory τ. There are two types of return used in RL algorithms:[1]

  1. **Finite-Horizon Undiscounted Return:** This is the sum of rewards obtained over a fixed number of time-steps T:[1]

$$R(\tau) = \sum_{t=0}^{T} r_t$$

  2. **Infinite-Horizon Discounted Return:** This is the sum of all rewards ever obtained by the agent, but discounted by a Discounting Factor $\gamma \in (0, 1)$ that reduces the value of rewards received further in the future:[1]

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

- **Discounting Factor γ:** the discount factor γ determines how future rewards are weighted relative to immediate rewards.[20] A discount factor of $\gamma = 0$ means the agent only cares about immediate rewards, while $\gamma = 1$ implies that future rewards are just as important as immediate ones.[20]

2.3.2 Reinforcement Learning Problem

Reinforcement Learning (RL) revolves around the goal of learning a policy that maximizes the expected return. This goal remains consistent regardless of the choice of return measure, whether it be infinite-horizon discounted return or finite-horizon undiscounted return, and irrespective of the policy choice.

To comprehend expected return, it is crucial to understand probability distributions over trajectories. In environments where both state transitions and policies are stochastic, the probability of a T-step trajectory is expressed as follows:[1]

$$P(\tau \mid \pi) = \rho_0(S_0) \prod_{t=0}^{T-1} P(S_{t+1} \mid S_t, A_t) \pi(A_t \mid S_t) \tag{2.1}$$

where:

- $\rho_0(S_0)$ is the initial state distribution.
- $P(S_{t+1} \mid S_t, A_t)$ represents the probability of transitioning to state S_{t+1} from state S_t after taking action A_t.
- $\pi(A_t \mid S_t)$ is the policy, providing the probability of taking action A_t given state S_t.

The expected return $J(\pi)$ is then defined as the expected value of the return over all possible trajectories under policy π:[1]

$$J(\pi) = \int_{\tau} P(\tau \mid \pi) R(\tau) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)] \tag{2.2}$$

Where $R(\tau)$ is the return (total accumulated reward) for trajectory τ.

The central optimization problem in RL is to find the policy π^* that maximizes this expected return:[1]

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{2.3}$$

In scenarios where the policy is parameterized, say by θ, the optimization problem can be re-framed to find the optimal parameters θ^*:[32]

$$\theta^* = \arg\max_{\theta} J(\pi_\theta) \tag{2.4}$$

Here, the structure of the policy is given (such as the number of neurons, synapses, and layers in the case of a neural network), and the problem reduces to finding the parameters θ^* that maximize the expected return.

2.3.2.1 Value Functions

Value functions are fundamental concepts in reinforcement learning (RL), central to the evaluation and improvement of policies. A value function quantifies the expected return from a state or a state-action pair under a particular policy, serving as a critical tool for decision-making in RL algorithms.[1] The four most commonly used functions are:

- **State-Value Function/On-Policy Value Function:** It is denoted as $V^{\pi}(S)$, represents the expected return (cumulative reward) starting from a state S and following policy π. It provides a measure of how good it is for an agent to be in a given state S when it acts according to the policy π. Formally, it is defined as :[48]

$$V^{\pi}(S) = E_{\pi}\{R_t \mid S_t = S\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid S_t = S\right\} \tag{2.5}$$

- **Optimal value function:** It is denoted as $V^*(S)$, represents the highest possible expected return from a state S, considering all possible policies. It is defined as:[49]

$$V^*(S) = \max_{\pi} V^{\pi}(S) \tag{2.6}$$

- **Action-Value Function / On-Policy Action-Value Function:** It is denoted as $Q^{\pi}(S, A)$, represents the expected return for an agent starting from state S, taking an action A, and thereafter following a policy π.[48] It provides a measures of

how good it is for an agent to take a specific action A in a given state S when it acts according to the policy π. Formally, it is defined as :[48]

$$Q^{\pi}(S,A) = E_{\pi}\{R_t \mid \S_t = S, A_t = A\} = E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1} \mid S_t = S, A_t = A\right\} \quad (2.7)$$

- **Optimal Action-Value Function:** It is denoted as $Q^*(S,A)$, represents the highest possible expected return for an agent starting from state S, taking action A, and then forever acting according to the optimal policy. It is defined as:[49]

$$Q^*(S,A) = \max_{\pi} Q^{\pi}(S,A) \quad (2.8)$$

2.3.2.2 Policy Optimization

Policy optimization is focusing on improving the policy by directly adjusting its parameters to maximize the expected return.[32] This section delves into the mathematical foundation of policy optimization algorithms, which include methods such as proximal policy optimization.

The primary objective in policy optimization is to find the optimal set of parameters θ^* for stochastic parameterized policy π_θ that maximizes the expected return $J(\pi_\theta)$. The expected return $J(\pi_\theta)$ is defined as follow:[3]

$$\theta^* = \arg\max_{\theta} J(\pi_\theta), J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\theta}[R(\tau)] \quad (2.9)$$

Different algorithms may maximize different objective functions, but the goal remains to optimize the parameters to achieve the highest possible return.

To optimize the parameters θ, gradient ascent is employed. Gradient ascent iteratively updates the parameter in the direction that increases the expected return:[3]

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta) \mid \theta_k \quad (2.10)$$

The element α is called learning rate, controlling the step size of the parameter update. The learning rate α is crucial; if it is too large, the steps may overshoot the optimal region, while if is too small, convergence can be slow or the algorithm may get stuck in suboptimal areas.[32]

The element $\nabla_\theta J(\pi_\theta)$ is called policy gradient, determines the direction for parameter updates. It is defined as:[3]

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(A_t \mid S_t) R(\tau)\right] \quad (2.11)$$

This expectation can be estimated by sampling a set of episodes $D = \{\tau_i\}_{i=1,\ldots,N}$ where the agent follows the policy π_θ. The estimator for the policy gradient $\hat{g}$ is computed as:[3]

$$\hat{g} = \frac{q}{|D|}\sum_{\tau\in D}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(A_t \mid S_t) R(\tau) \quad (2.12)$$

To perform an update on the current policy, the following steps are typically followed for policy optimization:[32]

1. Collect Episodes: Gather a set of trajectories $D = \{\tau_i\}_{i=1,\dots,N}$ using the current policy π_θ.
2. Compute Policy Gradient Estimator: Calculate the estimator $\hat{g}$ for the policy gradient using the collected episodes by using equation (2.12).
3. Update Policy Parameters: Adjust the policy parameter according to the gradient ascent.

2.3.2.3 Advantage Function

In reinforcement learning, the concept of relative advantage is often employed to evaluate the benefit of taking specific actions compared to other possible actions in a given state. This evaluation is encapsulated in the advantage function, which provides a measure of how much better an action A is than the average action, assuming that the agent follows the policy π thereafter.[1]

The advantage function $A^\pi(S, A)$ for a policy π quantifies the preference for action A in state S over sampling an action from the policy π. Mathematically, Advantage Function is defined as:[1]

$$A^\pi(S, A) = Q^\pi(S, A) - V^\pi(S) \tag{2.13}$$

where:

- $Q^\pi(S, A)$ is the action-value function, representing the expected return of taking action A in state S and then following policy π.
- $V^\pi(S)$ is the state-value function, representing the expected return of being in state S and following policy π thereafter.

2.3.3 Reward Function

In Reinforcement Learning, the reward function is crucial, as it defines the learning objective and drives the agent's learning process by providing feedback on its actions.[41] Properly crafting this function ensures that the agent learns the desired behavior. Typically, RL methods treat reward functions as black boxes.[26] This section explores the types of reward function and the basic procedure for building one.

2.3.3.1 Types of Reward Function

Depending on the task requirements, different types of reward functions are employed, each offering unique advantages and challenges. Below is a summary of the key types, their definitions, and practical use cases.

Name	Definition	Use Case
Sparse Reward[23]	Given only when the agent achieves a specific goal. Challenging but effective for goal-oriented tasks.	Navigation tasks, such as a robot finding its way through a maze.
Dense Reward[23]	Provides feedback at every step, aiding gradual learning and behavior refinement.	Training a robot arm to reach a target with step-by-step rewards.

Name	Definition	Use Case
Shaped Reward[23]	Incorporates additional hints about desired behavior, leading to faster learning.	Video games where agents collect items while reaching goals.
Inverse Reward[23]	Discourages certain behaviors, essential for safety or cost reduction.	Autonomous driving penalizing dangerous actions like sharp turns.
Composite Reward[23]	Combines multiple reward signals for balancing complex tasks.	Drone delivery maximizing speed, minimizing energy use, and avoiding obstacles.
Extrinsic Reward[16]	Rewards provided by the environment based on the robot's actions and state.	Robot receiving points for navigating to target locations.
Intrinsic Reward[16]	Rewards generated by the robot itself based on internal motivations like curiosity or exploration.	Robot exploring new areas for novelty and learning.

Table 2.1: Types of Reward Functions in Reinforcement Learning

2.3.3.2 Designing a Reward Function

Designing a reward function can be straightforward if we have a clear understanding of the problem. However, in many cases, it can be a challenging task, as there is no one correct way to do it.[28] We start with an initial reward function based on your knowledge, observe the agent's behavior, and then adjust the function to improve performance. [28] A well-crafted reward function can guide the agent in solving the problem efficiently, while a poorly designed one can lead to suboptimal or undesirable behavior.[41]

Designing an effective reward function is more art than science.[23] Here are the key steps for designing a reward function:[41]

1. **Define the Goal:** Clearly define the agent's objective. What should the agent achieve, and what behaviors should it learn? For example, in a game, the goal might be to win or to reach a destination.

2. **Identify Rewards:** Determine which actions lead to positive rewards (desired behavior) and negative rewards (undesirable behavior). For example, reward a robot for moving closer to the goal and penalize it for mistakes, such as collisions.

3. **Ensure Consistency:** Make sure reward values are consistent and aligned with the goal. Avoid too large or small rewards that could distract the agent from the main objective.

4. **Balance Immediate and Long-Term Rewards:** Design the reward function to balance immediate and long-term rewards. Immediate rewards encourage quick, beneficial actions, while long-term rewards help the agent plan for future success.

5. **Prevent Reward Hacking:** Design the reward function to avoid the agent exploiting loopholes for high rewards without achieving the intended goal. For example, prevent a vacuum robot from cleaning the same spot repeatedly.

2.3.4 Reinforcement Learning Algorithms

RL algorithms are computational methods designed to enable an agent to learn optimal behaviors through interactions with its environment. This section will explore and compare various RL algorithms, including model-based versus model-free approaches and policy optimization versus Q-learning methods. We will conclude with an in-depth look at Proximal Policy Optimization (PPO) in Section 2.3.5, as it is the main focus of this thesis. A taxonomy of RL algorithms is shown in Figure 2.11.

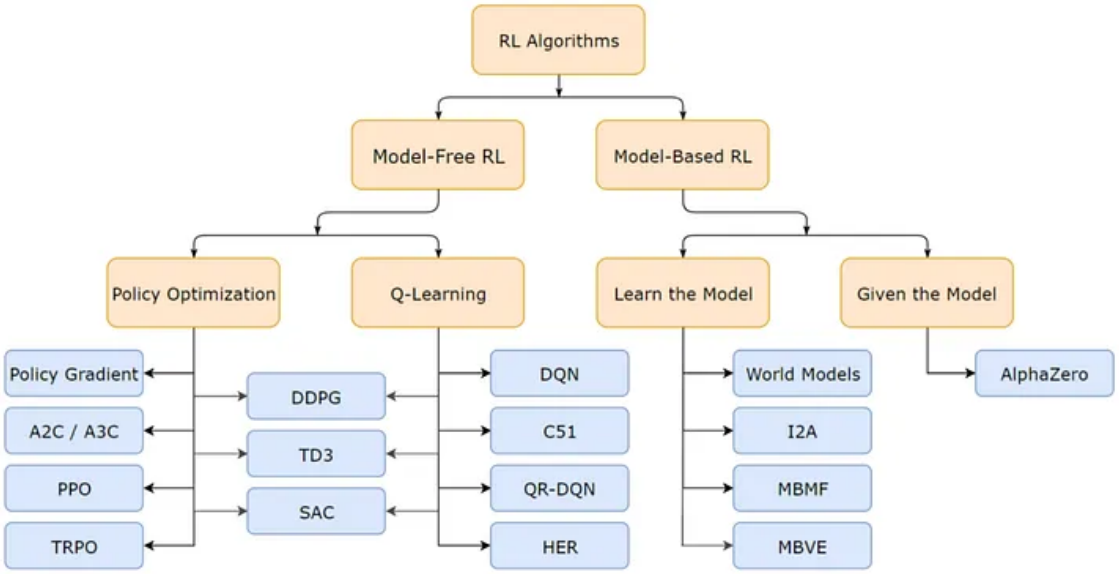

Figure 2.11: Taxonomy of RL Algorithms. [2]

2.3.4.1 Model-Free vs Model-Based RL

One of the most important branching points in an RL algorithm is whether the agent has access to (or learns) a model of the environment. By a model of the environment, we mean a function that predicts state transitions and rewards.[2] Algorithms that use a model are called model-based methods, and those that do not are called model-free.

In model-based reinforcement learning (RL), an agent uses a model to create additional experiences.[10] This approach has several advantages: It does not need a lot of samples, can save time, and offers a safe environment for testing and exploration. However, its performance heavily relies on the accuracy of the model, and it can be quite complex to implement.[10] Model-based algorithms are best suited for environments where we have complete knowledge (environments that are fixed or static in nature) and the outcomes are predictable, allowing the agent to plan ahead.[52]

On the other hand, model-free RL algorithms don't rely on a model. Instead, the agent learns directly through interactions with the environment.[10] This type of algorithm doesn't depend on a model's accuracy and is less computationally complex, making it more suitable for real-life situations. However, it requires extensive exploration, which can be time-consuming and potentially dangerous as it relies on real-life interactions.[10]

Model-free algorithms are particularly advantageous in dynamic environments where complete knowledge is not available.[10] For example, in autonomous driving, where traffic routes and conditions constantly change, model-free RL allows the agent to adapt through continuous learning from the environment. This makes model-free methods

more suitable for the outdoor navigation of mobile robots, as these environments are often unpredictable and require the agent to adapt to new situations rapidly.[52]

In summary, while model-based RL offers advantages in static and predictable environments, model-free RL is better suited for dynamic, real-life applications like outdoor navigation, where adaptability and direct learning from the environment are crucial. Given the dynamic nature of outdoor navigation, model-free approaches generally provide a more effective solution for the tactical planning of mobile robots. To further understand the capabilities of model-free RL, the next section explores key learning paradigms such as policy optimization and Q-learning, which play a fundamental role in decision-making for autonomous navigation.

2.3.4.2 Learning Objectives in Model-Free RL

There are two main approaches to representing and training agents with model-free RL:

1. **Policy Optimization:** Policy optimization methods explicitly represent a policy $\pi_\theta(A|S)$ and optimize parameter θ either directly by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly by maximizing local approximations of $J(\pi_\theta)$. This optimization is almost always performed on-policy, meaning that each update only uses data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximator $V_\varphi(S)$ for on-policy value function $V^\pi(S)$, which helps determine how to update the policy.[2]

   Example of policy optimization methods include:

   - Asynchronous Advantage Actor-Critic(A3C)/ Advantage Actor Critic(A2C): These methods perform gradient ascent to directly maximize performance.[2]
   - Proximal Policy Optimization(PPO): This method maximizes performance indirectly by maximizing a surrogate objective function, which provides a conservative estimate for how much $J(\pi_\theta)$ will change as a result of the update.[2]

   Policy optimization methods are principled, meaning they directly optimize for the desired objective, making them stable and reliable. However, they are typically less sample-efficient because they require on-policy data collection, meaning they cannot reuse past experiences as effectively.[2]

2. **Q-Learning:** Q-learning methods learn an approximator $Q_\theta(S, A)$ for the optimal action-value function $Q^*(S, A)$. They typically use an objective function based on the Bellman equation and perform optimization off-policy, meaning that each update can use data collected at any point during training, regardless of how the agent was exploring the environment at the time.[2] The corresponding policy is obtained via the connection between Q^* and π^*: the actions taken by the Q-learning agent are given by:[2]

$$A(S) = \arg\max_A Q_\theta(S, A)$$

   Examples of Q-learning methods include:

   - Deep Q-Network(DQN): A classic algorithm that significantly advanced the field of deep RL.[2]
   - Categorical DQN(C51): A variant that learns a distribution over returns, with the expectation being Q^*.[2]

Q-learning methods are often more sample-efficient because they can reuse data more effectively than policy optimization techniques. However, they tend to be less stable and reliable because they only indirectly optimize for agent performance by training Q_θ to satisfy a self-consistency equation.[2]

2.3.4.3 Selecting the Appropriate RL Algorithm

Choosing the right reinforcement learning (RL) algorithm depends on factors like sample efficiency, versatility, computational simplicity, and whether the agent can learn in the real world or needs to rely on simulations. Figure 2.12 shows a rough guideline for selecting the RL algorithm.

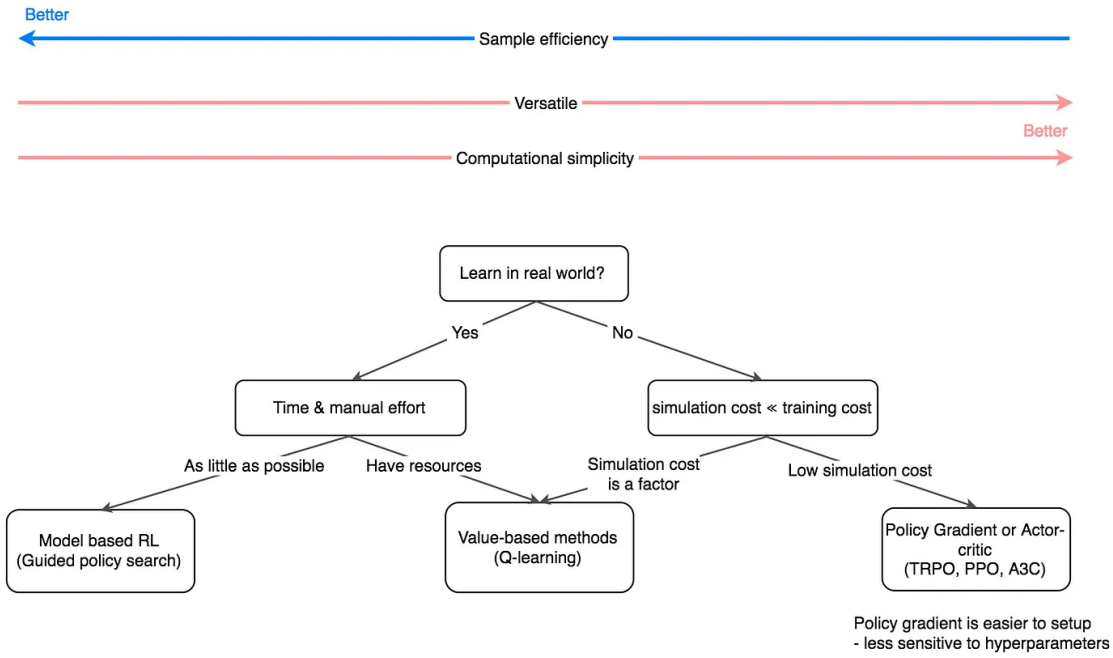

Figure 2.12: Guideline for selecting the RL Algorithm. [25]

Sample efficiency refers to how effectively an algorithm uses data from interactions with the environment to train the model. While it is not necessarily an ultimate goal[25], it is crucial in many RL applications. Model-based RL methods are more sample efficient because they generate additional experiences using an internal model, reducing the need for extensive real-world data. On the other hand, model-free methods, such as policy gradient and Q-learning, are simpler to implement and require fewer computational resources.[25]

When deciding which algorithm to use for outdoor mobile robot navigation, the flowchart provides a clear guide. Training a mobile robot in the real world is impractical due to safety concerns, difficulty in resetting the robot's position, and the long training times required. Additionally, real-world training faces challenges like robot battery life, weather conditions, and other logistical issues. Therefore, simulating the mobile robot in an outdoor environment is a better option. It is safer, cost-effective, and allows for automated processes like resetting the simulation environment in case of success, failure, or truncation during an episode.

As illustrated in the Figure 2.12, algorithms like Trust Region Policy Optimization(TRPO), Proximal Policy Optimization (PPO), and Asynchronous Advantage Actor-Critic (A3C)

are particularly suitable for this purpose. They offer a good balance of ease of setup and effective performance in simulated environments, making them ideal for developing robust navigation policies for mobile robots.

All three algorithms (TRPO, PPO, A3C) work with both discrete and continuous action spaces.[34] A3C operates by asynchronously updating the agent's policy and value functions, which allows for faster learning.[33] TRPO attempts to take the largest possible improvement step without causing performance collapse by using a complex second-order method to constrain policy updates.[4] While effective, TRPO can struggle in some scenarios[51] and is more complex to implement.

PPO, on the other hand, shares the same motivation as TRPO in aiming to improve policy stability and performance without risking collapse.[4] However, PPO simplifies this process by using first-order methods with additional tricks to keep new policies close to old ones. This makes PPO significantly easier to implement and, empirically, it performs at least as well as TRPO.[4]

To compare A3C and PPO algorithms, an experiment was conducted on the CartPole and Lunar Lander environments to evaluate their performance in terms of convergence speed and stability.[12] Results indicate that A3C typically achieves quicker training times but exhibits greater instability in reward values. Conversely, PPO demonstrates a more stable training process at the expense of longer execution times. This evaluation highlights that the choice of algorithm should balance between training time and stability based on specific application needs. For applications requiring rapid training, A3C may be ideal, while PPO is better suited for those prioritizing training stability.[12]

Considering the requirements of tactical planning for mobile robots in outdoor environments, PPO emerges as the best fit among the three algorithms. Its balance of stable training, ease of implementation, and robust performance in both discrete and continuous action spaces makes it particularly suitable for developing reliable navigation policies in simulation environments. Thus, for the purposes of this thesis, PPO is the preferred algorithm due to its stability and practical implementation advantages.

2.3.5 Proximal Policy Optimization

Proximal Policy Optimization (PPO) was developed by John Schulman in 2017[13] as a reinforcement learning algorithm aimed at training AI agents to make decisions in complex, dynamic environments.[24] PPO strikes a balance between performance and comprehension, offering three main advantages over other algorithms: simplicity, stability, and sample efficiency.[45, 13, 24] PPO is an on-policy algorithm.[4, 45]

A key challenge in policy gradient methods is determining the extent of policy updates based on new information.

- Small updates result in slow learning and sample inefficiency.
- Large updates risk drastic changes to the policy, potentially leading to poor performance and instability in training.

PPO employs small policy updates (step sizes) to ensure the agent can reliably reach the optimal solution. A too-large step might misdirect the policy irrecoverably, while a too-small step reduces overall efficiency.[13] The central idea of PPO is to avoid large policy updates, ensuring the policy doesn't change excessively in a single update epoch.[35, 45] This approach effectively balances exploration (trying new things) and exploitation (sticking with what works).[24]

There are two primary variants of PPO: PPO-Penalty and PPO-Clip.[4, 45] This section provides a brief overview of PPO-Penalty and a detailed explanation of PPO-Clip, the focus of this thesis.

2.3.5.1 PPO-Penalty: KL Divergence Monitoring

Kullback-Leibler (KL) divergence measures the difference between two probability distributions. [24] In PPO, it quantifies the difference between the old policy (used for data collection) and the new, updated policy. KL divergence acts as a "distance" measure with specific properties: it is not symmetric, always non-negative, and sensitive to small differences.[24]

In the context of policy updates:

- A small KL divergence indicates a conservative update, meaning the new policy is similar to the old one.[24]
- A large KL divergence suggests a more drastic update, indicating significant changes in the new policy.[24]

PPO uses KL divergence as a diagnostic tool to monitor the magnitude of policy updates. [24] The basic idea involves:

- Setting a threshold for acceptable KL divergence.
- During the policy update process, computing the KL divergence between the old policy and the new policy.
- If the KL divergence exceeds the set threshold, stopping the optimization process for that iteration.

KL divergence early stopping provides a global constraint on how much the overall policy can change, helping maintain stability in the training process.[24]

2.3.5.2 PPO-Clip: Clipped Surrogate Objective

Proximal Policy Optimization (PPO) stands out for its clipped surrogate objective, a feature designed to prevent large, destabilizing policy updates. By clipping the ratio of action probabilities under the new and old policies, PPO ensures that updates remain within a safe range, maintaining stability in the training process.[13, 45] PPO is categorized as a policy gradient method used to train an agent's policy network for optimized decision-making.[13]

Standard Policy Gradient Objective:

To understand PPO's surrogate objective, we first need to grasp the standard policy gradient objective and its terminology. Policy gradient methods operate iteratively: starting with an initial policy, collecting data, using that data to compute an update, and applying the update to form a new policy. This new policy then becomes the old policy for the next iteration, and the cycle continues.[24]

A fundamental component in this process is the probability ratio, $r_t(\theta)$. This ratio compares the probability of an action under the current policy with its probability under the previous policy. [35, 45] Essentially, $r_t(\theta)$ reveals how much more or less likely the new policy is to take a specific action in a given state compared to the old policy. Mathematically, the probability ratio is defined as:[24, 45]

$$r_t(\theta) = \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta old}(A_t|S_t)} \tag{2.14}$$

- If $r_t(\theta) > 1$ the action is more probable in the current policy than in the old policy.[35]
- if $r_t(\theta)$ is between 0 and 1, the action is less probable in the current policy than in the old one.[35]

The policy gradient objective can be expressed as:[24]

$$L_{CPI}(\theta) = \widehat{\mathbb{E}}_t \left[r_t(\theta) \cdot \widehat{A_t} \right] \tag{2.15}$$

Where:

- $L_{CPI}(\theta)$ is the objective function we aim to maximize in reinforcement learning.
- $r_t(\theta)$ is the probability ratio.
- $\widehat{A_t}$ is the advantage estimate, indicating how much better or worse the action A_t was compared to expectations.

By weighting action outcomes with this ratio, the policy is updated to increase the likelihood of beneficial actions and decrease the likelihood of detrimental ones.[24]

The Clipped Surrogate Objective:

However, without constraints, large probability ratios could lead to excessive policy updates, destabilizing the learning process.[35] PPO addresses this by introducing a clipped surrogate objective, which penalizes large deviations from a ratio of 1, ensuring that new policies remain similar to their predecessors.[35, 45]

The clipped surrogate objective for PPO is defined as:[24, 45]

$$L^{CLIP}(\theta) = \widehat{\mathbb{E}}_t \left[\min \left(r_t(\theta) \cdot \widehat{A_t}, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon \right) \cdot \widehat{A_t} \right] \tag{2.16}$$

Where:

- ε is the clipping parameter.
- $\widehat{A_t}$ is estimated advantage at time step t.

The first term inside the minimum is the original policy gradient objective $L_{CPI}(\theta)$. The second term clips the probability ratio, discouraging it from straying outside the interval $[1-\varepsilon, 1+\varepsilon]$. The final objective is the minimum of the clipped and unclipped objectives, providing a lower bound on the unclipped objective.[24]

Understanding the Clipping Mechanism:

The clipping mechanism promotes small, incremental policy improvements, stabilizing training and preventing drastic performance drops. There are two primary case to consider, as shown in Figure 2.13:

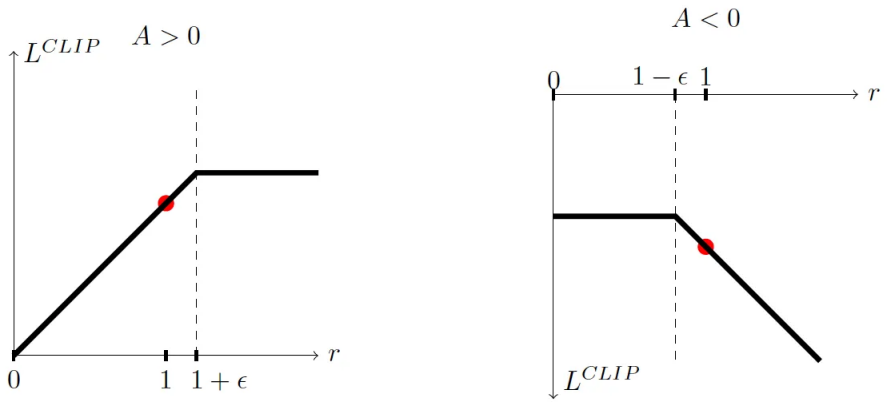

Figure 2.13: Effect of Positive Advantage and Negative Advantage. [45]

Case 1: Positive Advantage $\widehat{A_t}$ (the action was better than expected):[24]

- The objective increases if $r_t(\theta)$ rises, but only up to $1 + \varepsilon$.
- Beyond $1 + \varepsilon$, increasing $r_t(\theta)$ does not further increase the objective.

Case 2: Negative Advantage $\widehat{A_t}$ (the action was worse than expected):[24]

- The objective increases if $r_t(\theta)$ decreases, but only down to $1 - \varepsilon$.
- Beyond $1 - \varepsilon$, decreasing $r_t(\theta)$ does not further increase the objective.

This clipping mechanism ensures stable and efficient learning by limiting the extent of policy updates, thereby balancing exploration (trying new actions) and exploitation (relying on known successful actions).[24]

In summary, PPO-Clip enhances the policy gradient approach by enforcing controlled, small updates through a clipped objective function, making it a robust and efficient choice for reinforcement learning tasks.

Chapter 3

Concept

This chapter explores the core concepts behind reinforcement learning-based tactical planning for mobile robot navigation. It outlines the problem setting, key constraints, and the range of maneuvers available to the RL agent. The chapter also introduces the hardware and software architectures that support perception, decision-making, and trajectory planning. Lastly, it highlights key challenges in implementation and the solutions used to overcome them.

3.1 Problem Setting

This section defines the scenario for the tactical planning problem addressed in this study. The primary objective is to develop a reinforcement learning (RL) framework for tactical planning in mobile robot navigation. The goal is to enable the robot to autonomously reach final goal destination by making optimal decisions about which action or maneuver to execute in different situations.

Such an agent can serve as a tactical planner, making decisions in real-time for navigation tasks where multiple maneuvers are available for mobile robot to reach the goal. The agent's effectiveness in handling unexpected obstacles and conditions that were not part of the planning phase depends entirely on the quality of its training.

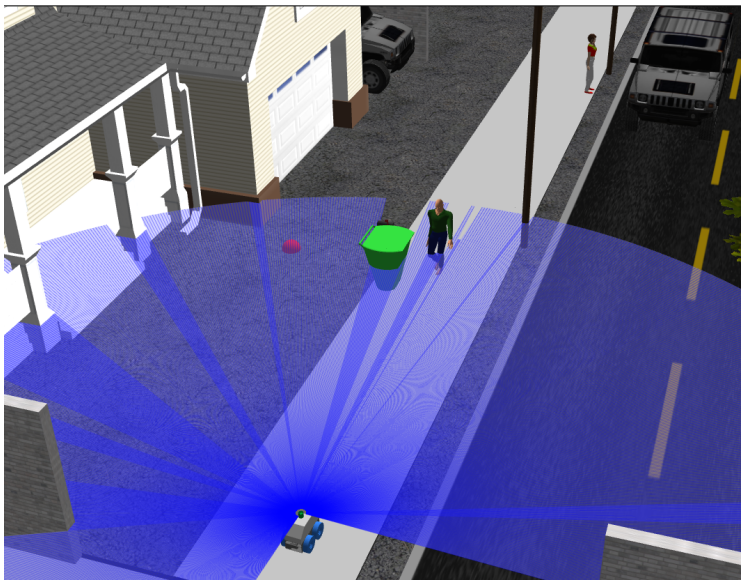

Figure 3.1: Overview of the 3D Environment used in the study.

The different maneuvers that the robot or agent can perform are explained in the Section 3.1.2. A properly trained RL agent can navigate efficiently, even when encountering unforeseen obstacles or dynamic changes in the environment. The final model selects the appropriate maneuver, which is then translated into feasible control commands (e.g., linear and angular velocity) for the mobile robot. These commands enable the robot to follow a given sequence of waypoints, ensuring smooth and efficient navigation.

Figure 3.1 presents the 3D environment used in this thesis to train the RL agent. The custom-built 4-wheel-drive robot, known as GrassHopper, developed at the Schmalkalden University of Applied Sciences, is placed on a sidewalk in an outdoor environment. This environment includes various obstacles such as dustbins and stationary people, simulating real-world scenarios. Additionally, there are two people walking on the sidewalk, further enhancing the realism of the training environment.

The robot's hardware architecture will be described in the Section 3.2. A complete top view of the simulation space can be observed in Appendix A.6, Figure A.1, providing a comprehensive overview of the environment used for training the RL agent. The GrassHopper URDF model utilized in the Gazebo simulation with ROS2-Humble is detailed in Appendix A.1, while the custom-built simulation environment developed for this thesis is provided in Appendix A.2.

3.1.1 Key Specifications and Constraints

The trained RL agent is required to navigate the sidewalk in an outdoor environment and reach long-range goals without colliding with obstacles, all within a reasonable amount of time. This thesis focuses solely on navigation along the sidewalk, with no instances of crossing roads, which is considered future work.

The following are the key specifications and constraints for the RL agent:

- **Movement Constraints:** The angular speed is limited to the range $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$ rad/s, and its linear speed is limited to the range $[0, 1]$ m/s to ensure stability and smooth navigation. Restricting angular speed prevents sudden turns, reducing the risk of instability, while limiting linear speed allows better control in dynamic environments. Backward motion is not allowed because there are no sensors available in the back and the LiDAR provides only partial rear coverage, making reverse navigation unsafe.

- **Laser Sensor Position:** The laser sensor is mounted at the top center of the robot at a height of 0.48 *meters*. This placement is chosen to maximize the field of view (FoV) of the LiDAR, which provides a 270° scanning range. Positioning the sensor at the top ensures an unobstructed view of the surroundings, minimizing interference from the robot's own body. Additionally, since the LiDAR is the primary method for perceiving the environment, obstacles must be taller than 0.48 meters to be detected effectively.

- **Collision Distances:** Due to its rectangular shape, the robot has three distinct collision distances relative to the laser position: 0.45 *meters* at the front, 0.40 *meters* at the corners, and 0.35 *meters* at the sides. These values are determined based on the robot's geometry and the necessary clearance for safe navigation. The front clearance (0.45 *meters*) is slightly extended to accommodate sensor mounts, such as cameras and GNSS modules, which may extend beyond main body of the robot. The 0.40 *meters* clearance at the corners allows for turning, while the 0.35 *meters*

side clearance enables movement through tight spaces. These front, corner, and side regions are better illustrated in the Figure 3.2.

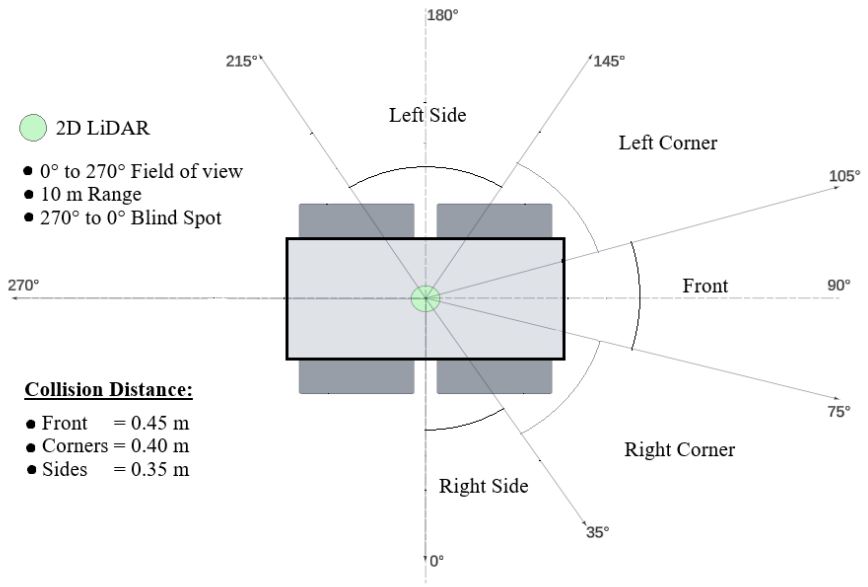

Figure 3.2: LiDAR Field of View and Collision Distance Schematic.

- **Goal Definition:** The navigation goal is defined as a list of waypoints in the (x, y) space, from the start to the end position, such as $[[x_1, y_1], [x_2, y_2], \ldots, [x_n, y_n]]$. The task is considered successfully completed when the distance from the center of the robot to the last waypoint in the path is below 0.55 meters. This distance accounts for the position of the laser relative to the center of the robot. If the target is near a wall or obstacle, the robot is expected to approach it without triggering a collision. To ensure safe navigation, a final threshold distance is determined by adding 0.45 meters (the front collision distance) and a 0.1 meter safety margin, allowing the robot to safely reach the target before detecting any potential collision.

3.1.2 Maneuver Selection

In mobile robot navigation, a wide array of maneuvers can be explored, such as turning left or right, accelerating, decelerating, lane changes, and more. These maneuvers are often crucial for dynamic and complex environments, such as roads with vehicle traffic or multi-lane intersections. However, the scope of this thesis is exclusively focused on navigation along sidewalks, which simplifies the maneuver requirements significantly. As a result, maneuvers involving intricate road scenarios become irrelevant.

To strike a balance between practicality and efficient learning, this thesis narrows the focus to three essential maneuvers: Global Path Following, Local Path Following with Obstacle Avoidance, and Full Stop. These maneuvers were carefully selected to simplify the decision-making process for the reinforcement learning agent, enabling it to focus on essential actions required for effective sidewalk navigation without the complexity of additional, less relevant maneuvers. Below is a detailed explanation of each maneuver:

1. **Global Path Following:** The robot follows a predefined path provided by the user using a pure pursuit controller. This approach ensures that the robot adheres to the planned trajectory accurately.

2. **Local Path Following/Obstacle Avoidance:** A separate function continuously generates a local path from the robot's current position to the next waypoint in the path. This is achieved using a dynamic occupancy grid generated from laser scan data and the A* algorithm, which is explained in Section 3.3.2.1, 3.3.4.2. The robot then follows this locally generated path using the pure pursuit controller, allowing it to effectively navigate around obstacles.

3. **Full Stop:** This maneuver is used to halt the robot in scenarios where its path is blocked by unknown obstacles or pedestrians. The robot should choose Full Stop when there is an object detected directly in front of it and within a critically close distance, or when a pedestrian is detected within a predefined safety margin. The robot stops completely until the path is clear or further instructions are provided.

These maneuvers result in the publication of linear and angular velocity commands, enabling smooth and controlled navigation. In critical situations, such as unexpected robot behavior or unforeseen obstacles, an emergency stop feature is available both on the robot and through a joystick, allowing immediate user intervention for safety. By focusing on these carefully chosen maneuvers, the robot can navigate efficiently while ensuring safe operation in sidewalk environments.

3.2 Hardware Architecture

The GrassHopper, a 4WD robot developed at Schmalkalden University of Applied Sciences, serves as the experimental platform for this study. This section details its hardware architecture, which is divided into two main categories: GrassHopper hardware and external sensors.

GrassHopper Hardware:

- **Power Supply:** The robot is powered by two $24V$ DC Lithium Polymer (LiPo) batteries connected in parallel, offering 5-6 hours of operational time under minimal load and without sensors.
- **Motor Controller:** Each of the four wheels is driven by its own motor, controlled by Vedder Electronic Speed Controllers (VESCs). These motor controllers receive $24V$ DC power from the Power Distribution Board (PDB).
- **Power Distribution Board (PDB):** There are two types of PDBs in the GrassHopper robot. The $24V$ PDB is crucial for distributing high voltage power to components such as the VESCs and the 2D-LiDAR. The $12V$ PDB handles lower voltage applications, providing power for the on-board PC, contractor, and auxiliary systems. These PDBs ensure a stable power supply to all essential parts of the robot.
- **Contractors and Safety Measure:** Safety is paramount in the GrassHopper's design. Contractors act as safety switches, controlling the power supply to the motor controllers (VESCs) and between the DC-DC converter and the $12V$ PDB. These safety measures help prevent overvoltage or short circuits, ensuring safe operation.
- **DC-DC Converter:** This converter steps down the voltage from $24V$ DC to $12V$ DC, providing the necessary power for the on-board computer and other $12V$ components.
- **On-board Computer:** The Intel NUC 12 Pro serves as the brain of the GrassHopper. Powered by the $12V$ supply, it handles all processing tasks and is connected to the sensors and VESCs via USB cables, enabling seamless data communication.

- **Drive Mechanism:** Each motor is linked to a gearbox with a gear ratio of 10 : 1, optimizing torque and speed for efficient movement.

External Sensors:

- **Sick Tim 551 2D-LiDAR:** This advanced sensor offers a 270° Field of View (FoV) with a scanning frequency of 15Hz and a detection range from 0.05m to 10m. With an angular resolution of 1°, the Sick Tim 551 is ideal for precise and efficient obstacle detection, making it an excellent choice for outdoor navigation.
- **Intel RealSense D435i:** The Intel RealSense D435i depth stereo camera significantly enhances the robot's perception capabilities. Acting as the "eye" of the robot, it provides critical environmental data, helping to create a detailed environmental model. Its efficiency in outdoor environments and compatibility with ROS2 middleware make it a perfect fit for this project.
- **SparkFun Ublox ZED-F9P-00B Global Navigation Satellite System (GNSS):** The Ublox ZED-F9P-02B GNSS module provides high-precision positioning with centimeter level accuracy, ideal for dynamic applications. With fast convergence times and high update rates, it ensures accurate location updates. Its compatibility with ROS2 makes it easy to integrate into the robotic system, supporting precise outdoor navigation and path planning.

The schemetic of the detailed hardware setup of GrassHopper robot is avilable in Appendix A.6, Figure A.2. It shows the connections between the power supply, motor controllers, on-board computer, and sensors, highlighting the roles of the PDBs, contractors, and safety measures. This comprehensive layout ensures efficient power distribution and robust operation, facilitating the successful deployment of the robot in dynamic outdoor environments.

3.3 Software Architecture

Designing an intelligent and autonomous navigation system for mobile robots demands a well-structured and adaptable software architecture. This architecture builds upon the Functional Architecture of Autonomous Driving outlined in Section 2.1.2, which comprises five essential layers: Perception, Situation Interpretation, Decision-Making and Planning, Trajectory Planning, and Vehicle Dynamics Control. These layers work in harmony to enable the robot to interpret its environment, plan its path, and execute safe and reliable movements.

A crucial part of this architecture is selecting the right tools and frameworks that facilitate seamless integration and efficient training of reinforcement learning models. After careful evaluation, ROS 2 Humble, Gazebo, OpenAI Gym, and Stable Baselines 3 (SB3) were chosen for their unique capabilities in supporting different aspects of development, simulation, and decision-making. Section 3.3.1 explores these tools in detail, shedding light on their roles and the rationale behind their selection for this thesis.

3.3.1 Tools and Frameworks

To develop and implement the Reinforcement Learning (RL)-based tactical planner for mobile robot outdoor navigation, this thesis leverages a combination of advanced tools and frameworks. Each tool plays a vital role in creating a robust and scalable solution, from simulation and middleware communication to algorithm development and training. Below is a detailed description of each component, highlighting the rationale behind their selection and their contributions to the overall system architecture.

The primary tools and frameworks used are described below:

ROS 2 Humble:

The Robotic Operating System (ROS) 2 Humble serves as a middleware framework enabling seamless communication between robot components. It was chosen for this research due to its modular architecture, real-time capabilities, and extensive support for integrating sensors, actuators, and control systems.[42] Unlike traditional frameworks that often have rigid structures, ROS 2's node-based design allows for scalable and adaptable development while ensuring efficient interaction between the simulated robot and various software components. One of its key advantages is the decoupling of the robot's operating system from the user program, making software updates and hardware transitions much easier.[42] Although ROS has a steep learning curve, its standardized communication protocols, strong open-source community, and well-structured learning resources make it a widely adopted and continuously evolving platform for robotics research and development.[42] Over time, ROS has undergone significant improvements, leading to the development of ROS 2, which enhances performance, security, and real-time communication, making it well-suited for both academic and industrial applications.

Gazebo:

Gazebo is an open-source 3D robotics simulator designed to simulate realistic environments, physical interactions, and integrate various sensors and actuators. It was chosen for this thesis due to its ability to model complex outdoor environments and its extensive support for ROS integration, making it ideal for testing RL algorithms. While tools like Unity and V-Rep offer benefits, Gazebo excels in key areas for robotic applications.[11] Unity, though strong in graphics, is more game-oriented and lacks the variety of robot models and physics engines needed for robotics projects. It also requires complex ROS integration.[11] V-Rep, while feature-rich, has a steeper learning curve and is limited by its educational license.[11] In contrast, Gazebo is open-source, cost-effective, and offers a vast library of pre-built models, sensors, and scenes.[11] Its support for headless operation and flexible physics engine options, along with extensive documentation, make it the best fit for simulating and testing robotics in this research.

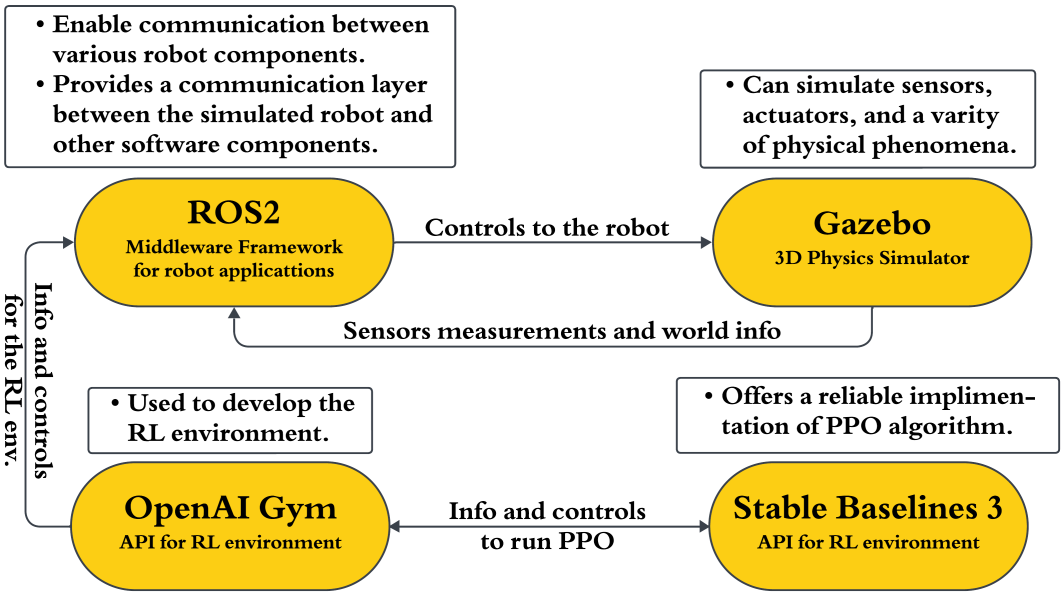

Figure 3.3: Scheme of the overall Architecture. [32]

OpenAI Gym:

OpenAI Gym, on the other hand, is a reinforcement learning (RL) framework that provides a standardized Application Programming Interface (API) for training and benchmarking RL algorithms.[38] Unlike Gazebo, which is focused on realistic robotics simulation, OpenAI Gym offers simplified, task-specific environments that abstract away complex physics and hardware details. It is designed to facilitate RL research by providing predefined environments where an agent can interact, receive rewards, and learn optimal policies. In this thesis, OpenAI Gym is used to define the RL learning framework, allowing the agent to learn decision-making strategies before being tested in the more realistic Gazebo environment.

Stable Baselines 3:

Stable Baselines 3 is a library that provides reliable implementations of various RL algorithms.[40] Built on top of OpenAI Gym, it facilitates the training, evaluation, and deployment of RL models. This thesis used Stable Baselines 3 for implementing the Proximal Policy Optimization (PPO) algorithm, ensuring a robust and efficient training process for the tactical planning model.

By integrating these tools and frameworks, this thesis establishes a comprehensive RL-based tactical planning solution for mobile robot outdoor navigation. Each component plays a vital role, from simulation and middleware communication to algorithm development and training, ensuring scalable performance in dynamic environments.

In the following sections, we delve into each layer of the architecture to understand how the mobile robot leverages reinforcement learning for tactical decision-making. This examination highlights the processes that enable the robot to interpret its environment, plan efficient trajectories, and control its movements. The challenges encountered during the development of the software architecture and the solutions adopted to ensure reliable autonomous navigation are also discussed, providing a comprehensive overview of the system's design.

3.3.2 Perception Layer

The Perception Layer collects and processes environmental data through sensors such as Intel RealSense D435i camera, Sick Tim 551 2D-LiDAR, and Ublox ZED-F9P-02B GNSS module to generate an understanding of the robot's surroundings. This layer is crucial for providing the necessary information about the environment, enabling the robot to navigate safely and interact effectively with its surroundings.

3.3.2.1 Dynamic Occupancy Grid / Local Costmap

The dynamic occupancy grid is a crucial component for autonomous navigation, providing a real-time map of the robot's surroundings. An occupancy grid represents the environment as a grid where each cell indicates whether it is occupied, free, or unknown. This grid is essential for path finding and obstacle avoidance, enabling the robot to navigate safely.

To achieve this, a separate Python node is implemented to dynamically publish the occupancy grid, which moves with the robot and covers an area of 2.5 *meters* in width and 5 *meters* in length. Out of this 2.5 *meters* width, 0.5 *meters* is on the right side of the robot and 2 *meters* is on the left side, as shown in Figure 3.4. This asymmetrical distribution enables the path planner to find a path that avoids obstacles from the left.

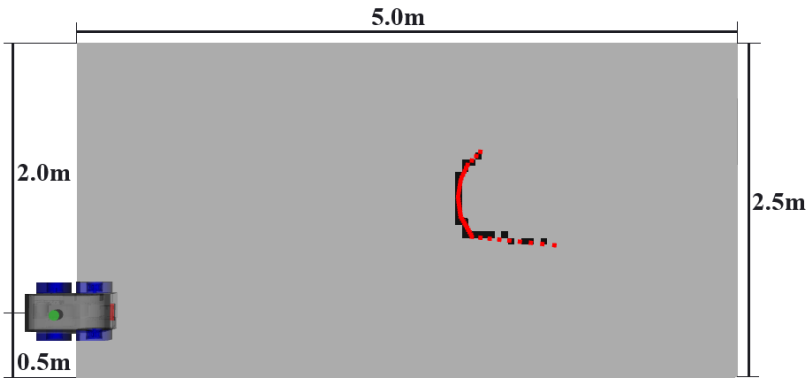

Figure 3.4: Dynamic occupancy grid (2.5 $m \times 5.0\ m$) shows laser scan data in red, with gray representing free space and black indicating occupied cells. The black cells are later inflated for safer local path planning and obstacle avoidance.

The node subscribes to LiDAR and odometry data, using this information to update the grid in real-time. LiDAR data provides distance measurements to nearby obstacles, which are then converted into grid coordinates and marked as occupied cells. Simultaneously, the robot's odometry data is used to update its position within the grid, ensuring the grid moves with the robot and accurately represents the local environment. Every time the update cycle runs, the grid is first emptied and then populated with new data from the sensors, ensuring that it always reflects the current state of the surroundings. The updated occupancy grid is then published at a fixed rate, allowing other system components to utilize this real-time map for path planning and navigation tasks.

The `dynamic_map` node is responsible for managing the dynamic occupancy grid, subscribing to the `/odom` topic for odometry data and the `/scan` topic for LiDAR sensor readings. It processes this data as described above and generates an occupancy grid, which is then published to the `/costmap` topic, making it available for other components such as the path planner. To ensure real-time performance, the occupancy grid is updated at a frequency of 15Hz, keeping the costmap accurate and up-to-date with the robot's surroundings. The overall data flow is illustrated in Figure 3.5. The complete ROS2 package for the Dynamic Map Node can be found in Appendix A.3.

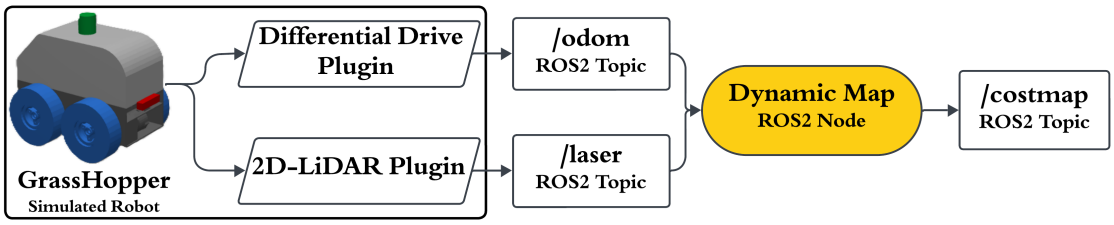

Figure 3.5: Scheme of Dynamic Map Node responsible for generating dynamic occupancy grid.

3.3.2.2 Object Classification, Distance, and Angle Estimation

Accurately determining whether an object is dynamic or static based on its class helps the robot make informed decisions for navigation and interaction, such as avoiding collisions and planning paths. In this section, data from an Intel RealSense D435i camera and a Sick Tim 551 2D-LiDAR sensor are combined to detect, classify and compute distance and angle of objects relative to the robot using YOLOv5 models for object classification.

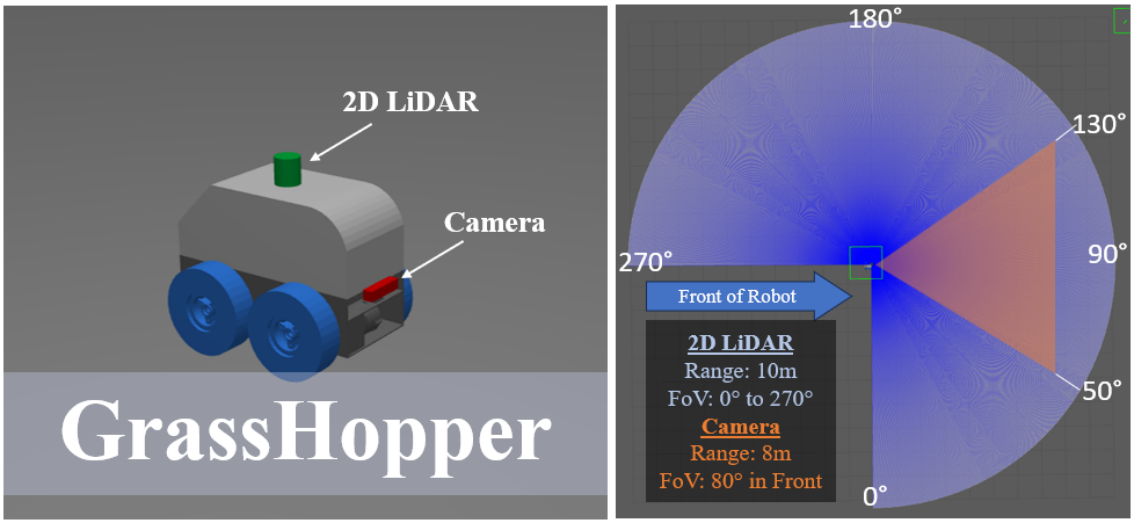

Figure 3.6: GrassHopper, 4-Wheel Drive Robot with 2D-LiDAR and Camera.

The process begins by mapping the LiDAR's field of view (FoV) onto the camera's FoV, which can be visualized in Figure 3.6. The Intel RealSense D435i camera is used to classify objects and determine the horizontal pixel coordinates of the bounding box centers. These pixel coordinates, which range from 0 to 640, represent the horizontal position of the detected objects within the camera's view.

The LiDAR sensor covers a total 80° horizontal FoV, spanning from 50° to 130° relative to the robot. This 80° range corresponds to 80 laser readings, with the laser readings between the 50^{th} and 130^{th} indices covering this angular range. The LiDAR readings from 50° to 130° are used for object depth and angle detection. Each laser reading corresponds to a small fraction of the 80° FoV, allowing precise angular measurement. These 80 readings are mapped to the camera's 640 horizontal pixels.

Each pixel in the camera's image corresponds to a fraction of a degree in the LiDAR's 80° FoV. By mapping these pixels to the LiDAR's laser readings, we can identify which LiDAR index corresponds to each detected object. The mapping between the camera's pixel space and the LiDAR's angular space is given by the equation:

$$\texttt{LiDAR\_Index} = -x_{center} \times \left(\frac{80}{640}\right) \tag{3.1}$$

Where:

- x_{center} is the horizontal pixel value of the object detected in the camera image (ranging from 0 to 640).
- 80 is the number of laser readings provided by the LiDAR in its 80° FoV (spanning from 50° to 130°).
- 640 is the total number of horizontal pixels in the camera's FoV.

This equation allows us to find the corresponding `LiDAR_Index` for center of any detected object. Once the `LiDAR_Index` is identified, the corresponding distance measurement is retrieved from the LiDAR data.

To calculate the object's angle, we need to adjust for the LiDAR's FoV. The LiDAR sensor's 80° FoV spans from 50° to 130° relative to the robot, corresponding to the 50^{th} to 130^{th} laser readings. The negative `LiDAR_Index` from the camera's pixel mapping is adjusted by adding 80 (to account for the shift to negative values) and 50 (to account

for the 50° FoV on the right side of the robot). The angle is then calculated using the following equation:

$$\texttt{Object Angle} = (80 + \texttt{LiDAR\_Index}) + 50 \tag{3.2}$$

Where:

- `LiDAR_Index` is the index calculated using the mapping Equation 3.1.
- 80 shifts the `LiDAR_Index` into a positive range.
- 50 accounts for the 50° FoV on the right side of the robot.

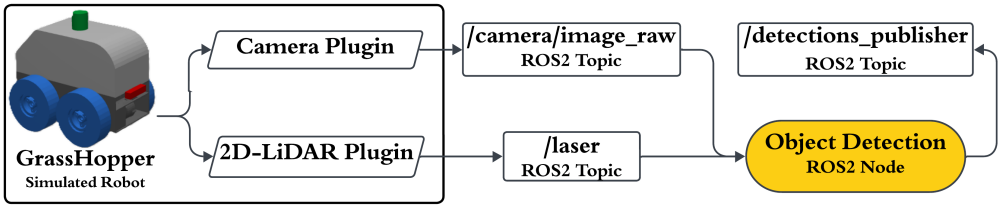

Figure 3.7: Scheme of Object Detection Node responsible for object classification, distance and angle estimation.

These measurements (distance and angle), along with the object class, are packaged into custom ROS messages (`ObjectDetection`) and published on a specific topic `/detections_publisher`. The overall data flow is illustrated in Figure 3.7. Detected objects are labeled and highlighted on the image for visualization, showing bounding boxes and distances. All detected objects, along with their respective distances and angles, are published as a list (`ObjectDetectionList`), enabling other system components to use this data for path planning, obstacle avoidance, and other decision-making tasks requiring object's class and it's precise distance and angle. The complete ROS2 package for the object classification and depth estimation is outlined in the code provided in the Appendix A.4.

3.3.2.3 GNSS-Based Localization

For precise localization in an outdoor environment, I utilize the Ublox ZED-F9P-02B GNSS module. This high-precision module provides accurate positioning data, essential for navigating large, open areas where other sensors may struggle with accuracy. By leveraging satellite signals, the ZED-F9P-02B can deliver centimeter-level precision, making it ideal for tasks that require exact positioning.

To enhance the accuracy and reliability of the GNSS data, the Extended Kalman Filter (EKF) and a dual EKF filter were implemented during the development of GrassHopper. These filters fuse data from the robot's encoders, an Inertial Measurement Unit (IMU), and the GNSS module. The encoders provide information on the robot's wheel rotations, the IMU supplies orientation and acceleration data, and the GNSS offers global position coordinates. By combining these diverse data sources, the EKF corrects for individual sensor inaccuracies and noise, resulting in a robust and precise estimation of the robot's position and movement.

This integrated approach to localization is crucial for outdoor navigation tasks. The fusion of GNSS, IMU, and encoder data ensures that the robot can accurately determine its location and orientation, even in challenging environments. This precise localization capability allows for effective path planning, obstacle avoidance, and overall autonomous operation, enabling the robot to navigate complex outdoor terrains with confidence and reliability.

3.3.3 Situation Interpretation Layer

The Situation Interpretation Layer serves as a crucial bridge between raw sensory data and meaningful situational awareness, enabling the robot to understand and respond to dynamic environments. This layer processes inputs from a variety of sensors to estimate the robot's state, identify obstacles, and interpret environmental features essential for making tactical navigation decisions.

To develop and refine this layer, a comprehensive simulation environment was created using Gazebo. The custom-built virtual outdoor environment replicates the complexity of real-world scenarios, as shown in Appendix A.6, Figure A.1, allowing safe and controlled testing of the robot's situational awareness capabilities. Gazebo offers a variety of pre-built robot models that can be simulated to behave like their real-world counterparts. Additionally, it offers the flexibility to build and simulate custom robots, enabling tailored simulations that accurately meet specific requirements, such as mimicking the unique features and configurations of the GrassHopper mobile robot. Its support for simulating advanced sensors like LiDAR, cameras, and IMUs ensures realistic data streams that closely replicate physical-world sensory inputs.

The simulated data is seamlessly published to ROS2 topics, mimicking real-world sensor outputs. This setup enables efficient processing by ROS2 nodes responsible for perception and state estimation. By integrating Gazebo and ROS2-Humble, perception algorithms can run on the simulated sensor data to detect obstacles, estimate distances, and recognize key features within the environment. This information is then interpreted by the Situation Interpretation Layer to inform the robot's decision-making process, aligning with mission-related objectives.

A critical aspect of this layer is its iterative learning process, facilitated by Gazebo's reset functionalities. Whenever the RL agent collides with obstacles or fails to complete its task within a defined timeframe, the environment and robot state can be reset, enabling incremental improvements through repeated training cycles. This iterative process ensures that the robot develops robust situational awareness and the ability to handle dynamic changes in its environment.

By leveraging Gazebo's advanced simulation capabilities and ROS2-Humble's powerful data handling framework, the Situation Interpretation Layer becomes a vital component for enabling accurate state estimation and comprehensive environment understanding. This foundation empowers the robot to navigate safely, make informed tactical decisions, and adapt to complex and dynamic outdoor environments.

3.3.4 Decision-Making and Planning Layer

As discussed in Section 2.2, reinforcement learning (RL) emerges as a powerful approach for decision-making, enabling autonomous agents to learn optimal strategies through direct interaction with the environment. Its adaptability makes it particularly suited for handling complex and dynamic navigation challenges. Building on this foundation, the Decision-Making Layer serves as the core of the autonomous navigation system, integrating RL algorithms with supporting tools such as ROS2, Gazebo, OpenAI Gym, and Stable Baselines3, as detailed in Section 3.3.1. This section explores the tactical planning mechanisms designed for the GrassHopper mobile robot, outlining the structured decision-making framework that drives efficient and intelligent autonomous navigation.

The Decision-Making Layer is implemented entirely in Python and comprises two main classes:

1. **Gym Environment:** Reinforcement learning relies on a structured framework to define agent-environment interactions, as outlined in Section 2.3.1 This class integrates key RL functionalities, including the observation space, action space, step function, reward function, and reset function, providing a foundation for the robot to learn and optimize its decision-making strategies through continuous interaction and feedback.
2. **Robot Controller:** This class is responsible for managing the communication between different ROS2 nodes, sensors, and actuators. It handles various publishers, subscribers, services and different functionalities to ensure seamless data exchange within the system.

In this section, we will explore these components in detail, illustrating how they integrate to form a robust decision-making layer for autonomous navigation. We will discuss their roles within the overall system, how they interact with each other, and how they contribute to efficient and adaptive decision-making in dynamic environments.

3.3.4.1 Gym Environment

In the development of a reinforcement learning (RL) framework, a critical component is the creation of a custom environment within OpenAI Gym. This custom environment serves as the foundation upon which the RL agent interacts with the world, learns from its experiences, and refines its tactical planning capabilities. The necessity of defining a well-structured environment cannot be overstated, as it encapsulates the parameters and mechanisms through which the agent perceives the world, makes decisions, and receives feedback. The key elements that must be meticulously specified include the observation space, action space, reward function, step method, and reset method. Each of these components plays an integral role in shaping the agent's learning process and overall performance. Additionally, integrating a dynamic occupancy grid, or local map callback function, is crucial. This involves subscribing to the `/costmap` topic of type `OccupancyGrid`, as discussed in Section 3.3.2.1, to find the local path and avoid obstacles.

Local Map Subscriber:

The local map subscriber is responsible for subscribing to the dynamic occupancy grid around the Grasshopper robot. Processes the data being published on `/costmap` topic of type `OccupancyGrid` to find the local path from the robot's current location to the target location using the functionalities available in the `RobotController` class. Since this custom environment inherits from the `RobotController`, it can utilize all the functionality provided by it.

The callback function extracts data from the received message and stores it in variables. It then transforms the robot location and the target location from global coordinates to local coordinates using the `transform_to_local()` method from the `RobotController`. After this, the `inflate_obstacle()` method is used to process the local map data, inflating obstacles in the map. Then, if the robot's location is inside the inflated map, the `find_nearest_free_cell()` method is used to determine the starting point for the A* algorithm, which is then employed to find the path from this starting point to the target point. The A* algorithm returns the path in local coordinates, which is subsequently transformed back to global coordinates using the `transform_to_global()` method. This transformed path is used later for obstacle avoidance actions.

The primary reason for calling this subscriber within the custom environment is that the target location is not accessible in the `RobotController` class. Access to the target location is essential for processing the data and finding the local path. The calculated path is published on the `/local_path` topic of type Path to visualize it in Robot Visualizer 2 (RViz2).

Environment Initialization:

First of all, the customized environment is created as a `Python` class that inherits from the `RobotController` class. This inheritance allows the environment to leverage all the methods developed to send actions to the robot, receive sensor data, and set the new agent state within the RL environment.

1. **Observation Space:** The observation space has been meticulously defined to provide comprehensive data for the RL agent to make informed decisions. It is structured as a dictionary that contains various elements:

    - **Laser Reads:** A continuous array with 270 laser readings normalized between 0 and 1, providing detailed information about the surrounding environment.
    - **Laser Front:** A subset of the laser readings with 30 samples, also normalized between 0 and 1, focused on the area directly in front of the robot.
    - **Laser Corners:** An array with 70 laser readings covering the corners of the robot's field of view, normalized between 0 and 1.
    - **Laser Sides:** A broader set of 120 laser readings covering the sides, again normalized between 0 and 1.
    - **Agent Polar Coordinates:** This observation stores the robot's relative position to the target location in terms of distance and orientation, providing essential spatial awareness for navigation and path planning.
    - **Person:** A discrete value indicating the presence (1) or absence (0) of a person.
    - **Dustbin:** A discrete value indicating the presence (1) or absence (0) of a dustbin.
    - **Person Distance:** A normalized distance value indicating how far a detected person is from the robot.
    - **Dustbin Distance:** A normalized distance value for the proximity of a detected dustbin.
    - **Person Angle:** A normalized angle value indicating the direction of a detected person relative to the robot.
    - **Dustbin Angle:** A normalized angle value showing the direction of a detected dustbin.

    All elements of the observation space are normalized between 0 and 1 to ensure better results during training and faster convergence to the optimal policy.

2. **Action Space:** The action space in this environment is defined as a discrete space with three possible actions, as specified in the line of code `"self.action_space = Discrete(n=3)"`. These actions represent different maneuvers the robot can perform:

    - **0: Full Stop:** The robot halts all movement, which can be used in scenarios requiring immediate stopping due to an unforeseen obstacle or hazard.

- **1: Global Path Following:** The robot follows a predefined global path to navigate towards its goal.
- **2: Local Path Following/Obstacle Avoidance:** The robot dynamically adjusts its path to avoid obstacles and follows a locally computed path as explained in the Local Map Subscriber.

This simplification into discrete actions allows the RL agent to choose from a set of predefined maneuvers, streamlining the decision-making process and facilitating effective learning.

In summary, the careful design of the observation and action spaces is crucial for the RL agent's ability to learn and perform optimally. By providing a rich set of normalized sensory inputs and a simplified set of actions, the environment ensures that the agent can effectively navigate and interact with its surroundings.

Step Method:

The Step Method is a critical part of the reinforcement learning environment, governing the interactions between the agent and its environment at each time step. This section details the sequence of operations within the step method, demonstrating how it handles action execution, processes sensor data, and computes rewards to enable the robot's effective learning and navigation. The following steps provide an in-depth look at how each operation unfolds, driving the robot's learning process and decision-making.

- **Step Increment:** The first operation in the step method is incrementing the step counter (`self._num_steps`). This keeps track of the number of steps taken during the current episode.
- **Discrete Actions:** The `self.discrete_action` variable stores the action selected by the RL algorithm. Based on this value, the step function checks conditions and performs one of the three maneuvers mentioned in Section 3.1.2.
- **Spinning the Node:** To handle callbacks for new messages on subscribed topics, the ROS2 node must be spinning. Instead of using the blocking spin function, a customized `spin()` method is employed. This method repeatedly calls `spin_once()` until all the information are updated, ensuring real-time responsiveness without blocking the main thread.
- **Coordinate Transformation:** The robot's odometry data, provided in Cartesian coordinates, is transformed into polar coordinates relative to the target. This involves computing the distance (d) and angle (θ) to the target, which are used for navigation and state representation.
- **Observation and Information Retrieval:** The `_get_obs()` method retrieves and normalizes the current observations, including laser readings, the robot's location, and object class, distance, and angle, as specified in the Observation Space. The `_get_info()` method gathers additional state information before normalization, which is crucial for computing rewards and evaluating the robot's performance.
- **Reward Calculation:** The reward for each step is computed based on the information gathered. The specifics of the reward function used in this study is explained below in Reward Function.
- **Episode Termination:** The episode can end under three conditions:
  1. **Target Reached:** If the distance to the target is less than 0.55 meters. As explained in Goal Definition.

2. **Collision Detected:** If a collision is detected by the LiDAR. The collision distances vary for different regions of the robot due to its rectangular shape, as detailed in Collision Distance and illustrated in Figure 3.2.
3. **Truncation:** If the agent exceeds a predefined number of steps without reaching the goal, the episode is truncated, preventing infinite runs.

- **Return Values:** The method returns a tuple containing the observation, reward, done flag (indicating if the episode has ended), a placeholder for compatibility, and additional information.

By incrementing the step counter, handling actions, spinning the node, transforming coordinates, updating observations and rewards, and checking for episode termination, the step method encapsulates the essential dynamics of the environment, facilitating effective learning and navigation for the robot.

Reward Function:

In Reinforcement Learning (RL), the reward function is a fundamental component that defines the learning objective and influences the agent's behavior. It serves as feedback, guiding the agent toward optimal decision-making by assigning numerical values to actions based on their desirability. In mobile robot navigation, an effective reward function ensures safe, efficient, and goal-oriented movement while avoiding obstacles and optimizing path planning.

Designing an effective reward function requires a structured approach, ensuring the agent learns the desired behavior efficiently while avoiding pitfalls like reward hacking or local optima. As explained in Section 2.3.3.2, the following steps were followed to design the reward function used in this thesis:

1. **Define the Goal:** The goal of this RL agent is to reach the final target destination by selecting the most appropriate action among three available maneuvers: **Global Path Follow, Local Path Follow**, and **Full Stop**. Additionally, the agent must avoid static and dynamic obstacles encountered along the path. The reward function incentivizes safe and efficient navigation while discouraging unsafe actions like unnecessary stops or obstacle collisions.
2. **Identify Rewards:** Based on the simulation environment created for this thesis, positive rewards are given for progressing along the path, stopping appropriately near obstacles, and maintaining alignment with the optimal path. Negative rewards are assigned for collisions (-10), unnecessary stops, and significant deviations from the path, ensuring the agent prioritizes efficiency and safety.
3. **Ensure Consistency:** Reward values are scaled appropriately—collision penalties (-10) outweigh minor penalties to enforce safety, and global path rewards are higher in open spaces to encourage efficiency. The path completion reward is normalized to maintain consistency across different navigation scenarios.
4. **Balance Immediate and Long-Term Reward:** The function rewards step-wise actions to guide incremental learning while also encouraging long-term success by considering path completion percentage. It ensures a balance between short-term movements and overall navigation efficiency, preventing short-sighted decisions.
5. **Prevent Reward Hacking:** Measures are taken to avoid reward exploitation, such as penalizing unnecessary stops, and incorporating multiple checks (distance, angle, obstacle proximity) to prevent undesired shortcuts.

By following this structured design approach, the reward function ensures that the mobile robot efficiently, safely, and optimally navigates toward its goal. The next section

will further analyze the specific reward function details and the rationale behind the assigned values. Based on this design approach for reward function, the reward at time step 't' for this thesis is developed as follows:

$$r_t = \begin{cases} \frac{path\_completion\_percentage - normalized\_steps}{10}, & \text{if waypoint reached} \\ -10, & \text{if collision detected} \\ reward\_global + 1.5 \times reward\_local + 1.25 \times reward\_stop, & \text{otherwise} \end{cases}$$

Where:

- **Path Completion Reward:** This component provides a continuous incentive for the agent to progress toward the target destination. Based on the percentage of the path completed, a reward is calculated, encouraging the agent to consistently work towards its goal. This aligns with the concept of a **shaped reward** (Table 2.1), where additional hints about desired behaviour (progress along the path) guide learning.
- **Collision Penalty:** A substantial negative reward (-10) is applied upon collision detection. This **inverse reward** (Table 2.1) discourages unsafe behaviours and emphasizes collision avoidance, prioritizing safe navigation.
- **Action-Based Rewards:** Beyond merely reaching the goal, the agent must navigate with precision and adaptability. A structured system of action-based rewards helps refine decision-making, balancing efficiency with situational awareness. By integrating **dense, inverse, composite**, and **extrinsic rewards** (Table 2.1), the agent is guided toward optimal maneuvers. Specifically, `'reward_global'`, `'reward_local'`, and `'reward_stop'` reinforce key aspects of skillful navigation. A detailed breakdown of these action-based rewards is provided below.

  1. **Global Path Follow Reward:** The Global Path Follow Reward encourages the agent to stay aligned with the optimal path while avoiding obstacles. Initially, the agent receives a base reward of 1 for selecting the global path follow action. If the agent maintains good alignment with the path (within $\pm\pi/12$ *radians*) and the path is clear, it receives a bonus reward of 1.2. However, if obstacles are detected within 2 *meters* in front, the reward is negated. This 2 *meters* distance is critical, as it accounts for a 0.8 *meters* safety distance for a person[37], a 0.55 *meters* target reach distance, and an additional 0.6 *meters* for obstacle inflation during local path maneuvers to ensure safe avoidance. If the agent is too close to obstacles at the corners or sides (less than 1 *meter* or 0.8 *meters*, respectively), the reward is reduced by 0.8 to encourage caution.

     If the agent is misaligned with the path, it faces a penalty (e.g., 1.2 times the base reward for obstacles), and in tight spaces, a minimal reward $(1e^{-3})$ is assigned to acknowledge the effort despite inefficiency. The reward function promotes alignment with the path, penalizes dangerous proximity to obstacles, and rewards clear path navigation while ensuring safe obstacle avoidance in complex environments.

  2. **Local Path Follow Reward:** The Local Path Follow Reward encourages the agent to safely navigate around obstacles by adjusting its path in real-time. Initially, the agent receives a base reward of 1 for selecting the local path follow action. If the agent is well-aligned with the path (within $\pm\pi/12$ *radians*), the reward is adjusted based on the distance to nearby obstacles and the presence of dynamic objects. If there is an obstacle within 2 *meters* in front of the

agent, the reward is further modified based on the proximity of detected objects, such as people or dustbins. For example, if both a person and a dustbin are detected, the reward is scaled inversely by the sum of their depths, ensuring the agent navigates cautiously around both. If only a person or a dustbin is detected in certain angular ranges, the reward is increased based on their proximity, emphasizing the importance of adjusting the path to avoid these dynamic obstacles.

If the agent is in a situation where obstacles are detected at the corners or sides (less than 1 *meter* or 0.8 *meters*, respectively), the reward remains unchanged to reflect the need for careful navigation. However, if the agent encounters an obstacle within 2 *meters* and isn't properly maneuvering (e.g., not aligning well with the global path), the reward is penalized. When the agent is misaligned with the path, a small reward $(1e^{-3})$ is given to encourage further attempts, even in challenging environments, such as cluttered spaces.

As explained in the Global Path Follow Reward, the 2 *meters* distance limit ensures that the agent has sufficient space to perform safe obstacle avoidance maneuvers. This consistency across both global and local path rewards supports the agent's overall goal of navigating efficiently while ensuring safety in dynamic environments.

3. **Full Stop Reward:** The Full Stop Reward incentivizes the agent to stop only when necessary, such as near obstacles or dynamic objects. Initially, the reward is set to 0 for the full stop action, discouraging unnecessary stops. If the agent detects no obstacles within 0.8 *meters* in front or 0.6 *meters* at the corners, it is penalized with a negative reward, encouraging movement rather than stopping in open spaces. This ensures the agent does not stop unnecessarily, promoting efficiency.

   However, if dynamic objects like people or dustbins are detected, the reward is adjusted based on their proximity. For instance, if both a person and a dustbin are detected, the reward is scaled by the sum of their depths, encouraging the agent to stop safely. If only a person or dustbin is detected within specific angular ranges (e.g., 75° to 105°), the reward is adjusted based on the proximity of the object. If no dynamic obstacles are detected, the reward is based on the laser distance to the nearest obstacle, ensuring the agent stops only when required for safety. This system promotes efficient, safe stopping behavior in the presence of obstacles while avoiding unnecessary halts.

The weight assignment reflects the strategic importance of each action. Global path follow is given a base reward weight of 1.0, as adhering to a planned path is generally optimal for efficient navigation. Local path follow, requiring dynamic adjustments and obstacle avoidance crucial for safe navigation, is assigned a $1.5\times$ weight. Full stop, while sometimes necessary, should be used sparingly to avoid inefficiencies, and thus receives a $1.25\times$ weight. These weights were determined by analyzing the average reward received for each action over the preceding 500 steps, allowing the reward function to dynamically adapt to the agent's performance and the demands of the environment.

The developed reward function strikes a crucial balance between **goal-oriented progression** and **safety-driven** decision-making, ensuring the agent reaches its destination efficiently while navigating responsibly. It encourages steady progress toward the goal while enforcing strict penalties for unsafe behavior, such as collisions. Additionally, action-based rewards refine decision-making, guiding the agent to make intelligent, context-aware maneuvers. By integrating both safety constraints and incentives for opti-

mal navigation, this reward function enables the agent to operate effectively in dynamic environments, minimizing risks while maximizing efficiency.

Reset Method:

The Reset Method is a vital function within the reinforcement learning environment, activated at the start of each new episode to reinitialize the robot's state and the environment. This ensures that the learning process remains unbiased and independent of any specific starting conditions, allowing the agent to learn generalizable behaviors. The reset process effectively clears previous states and prepares the system for a fresh round of interaction, enabling the robot to start each episode with a clean slate.

The reset method begins by incrementing the episode counter to keep track of the number of episodes completed. Next, it computes the initial pose of the robot using the `randomize_robot_location()` method, which generates a random starting location within the environment.

The method initializes the `_done_set_rob_state` variable to False, acting as a flag to ensure the robot's position is correctly set before moving forward. The `/set_entity_state` service from the `RobotController` class is called with the new pose to update the robot's position in the simulation. A loop spins the ROS node until the service responds, confirming that the robot's state has been set accurately.

After setting the robot's position and updating the sensor data, the method randomly selects a new path for the robot from the predefined path `(self.waypoints_locations)`. The path index is reset for both global and local paths using the `find_nearest_point_index()` method, and the paths are updated accordingly.

The first target location is extracted from the full path using the `randomize_target_location()` method. For visualization, if the `_visualize_target` flag is True, the new target position is updated in the simulation using the `call_set_target_state_service()` method, ensuring the target location is reset in Gazebo for visualization purposes.

The method then computes the initial observation and state information using `spin()`, `transform_coordinates()`, `_get_obs()`, and `_get_info()`. Finally, the step counter is reset to zero, and the method returns the initial observation and information, setting the stage for the new episode to begin.

This comprehensive reset process ensures that each episode starts from a clean slate, promoting robust and unbiased learning.

3.3.4.2 Robot Controller

The Robot Controller is a crucial component of the GrassHopper mobile robot's decision-making layer, managing communication and control for autonomous navigation. It sends velocity commands, processes LiDAR readings, handles data published by the object detection node (which consists of object class, distance, and respective angle), and determines the robot's position and orientation to compute distances from the goal. A dedicated ROS2 node implements these functions, featuring a publisher for commands and subscribers for LiDAR, position, and object detection data, along with a client for resetting the robot position in Gazebo simulation environment. The Robot Controller also includes functionalities like A*, Pure Pursuit, obstacle inflation, and coordinate transformations, which we will explore in detail later.

Action Publisher:

The Action Publisher is a key component of the Robot Controller, responsible for sending velocity commands to both the simulated and real GrassHopper mobile robot. A plugin (`libgazebo ros diff drive`) in the Unified Robotics Description Format (URDF) file facilitates this in simulation, creating a ROS2 topic `/cmd_vel` for linear and angular velocity commands.

The Robot Controller node, a publisher sends these commands to the `/cmd_vel` topic. The velocity values published to this topic are determined by the maneuver selection process within the decision-making layer, ensuring the robot's movements are precise and responsive to its environment. This setup ensures seamless operation in both the simulation environment and with the physical robot.

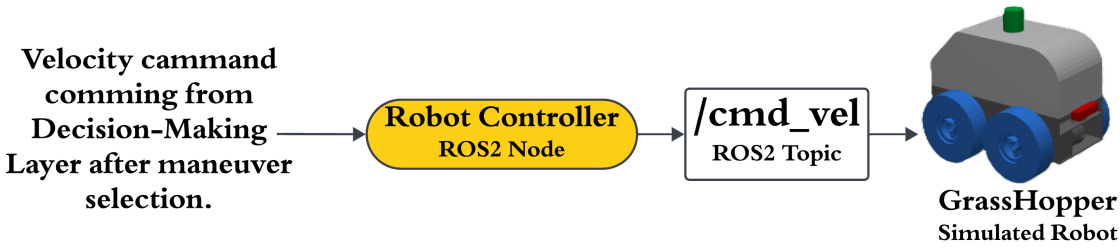

Figure 3.8: Scheme of command velocity flow.

Figure 3.8 illustrates the data flow from the Robot Controller to the GrassHopper robot via the `/cmd_vel` topic, showing how movement commands are published for accurate navigation in both simulation and real-world scenarios.

Subscriber:

The Robot Controller node includes three key subscribers: `/odom`, `/laser`, and `/detections_publisher`. Each subscriber handles specific types of data crucial for the autonomous operation of the GrassHopper mobile robot, helping to observe the surrounding environment and improve decision-making processes.

1. **Odometry Subscriber (`/odom`):** The odometry data provides essential information about the robot's position and orientation. This data is fetched using the `libgazebo ros diff drive` plugin, integrated with the `libgazebo ros imu` sensor plugin, both of which are added to the robot's URDF file. The odometry information, published to the `/odom` topic at $50Hz$, allows the Robot Controller node to retrieve accurate positional data. This data is vital for navigation and path planning, ensuring the robot knows its exact location relative to its goal.
2. **LiDAR Subscriber (`/laser`):** LiDAR readings are crucial for environment mapping. The `libgazebo ros ray sensor` plugin, added to the robot's URDF file, publishes LiDAR data to the `/laser` topic at $15Hz$. The Robot Controller node subscribes to this topic to receive laser data, which it processes by dividing it into smaller regions. This helps in accessing specific data for tasks such as collision detection and measuring distances to objects, enhancing the robot's ability to navigate safely and efficiently.
3. **Object Detection Subscriber (`/detections_publisher`):** The camera data, simulated using the `libgazebo ros camera` plugin, is published to the `/camera/image_raw` topic and processed by the Object Detection node. This node combines camera and LiDAR data, publishing object class, distance, and angle relative to the robot on the `/detections_publisher` topic. The `libgazebo ros`

camera plugin is also added to the robot's URDF file. The Robot Controller node subscribes to this topic to receive processed object detection data, which it uses to identify and avoid obstacles, improving the robot's decision-making capabilities.

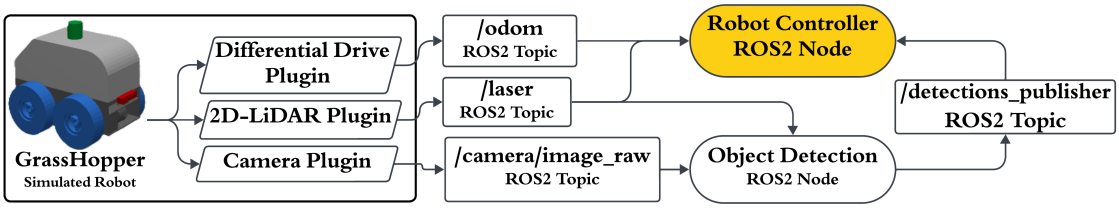

Figure 3.9: Scheme of sensors reading flow.

Figure 3.9 illustrate the data flow from the GrassHopper and sensors simulation in Gazebo to the Robot Controller Node through different topics.

Set Entity State Service:

For each new episode, it is essential to move the GrassHopper robot to a new initial position based on the initial-state distribution $\rho_0(\cdot)$. In Gazebo simulation, this can be efficiently accomplished using the `libgazebo_ros_state.so` plugin. By adding this plugin to the world file, it enables changing the state of all models within the simulation by specifying their new poses and initial velocities. Below is the XML format to add this service in the world file:

```
<world name="world">
  <plugin name="gazebo_ros_state"
     filename="libgazebo_ros_state.so">
    <ros>
      <namespace>/demo</namespace>
      <remapping>model_states:=model_states_demo</remapping>
      <remapping>link_states:=link_states_demo</remapping>
    </ros>
    <update_rate>1.0</update_rate>
  </plugin>
  <!--Rest of the world-->
</world>
```

This plugin creates a ROS2 service (`/demo/set_entity_state`) that processes incoming client requests to update the state of a model. Within the Robot Controller node, a client can be built to send requests to this service, specifying the new state of the robot whenever a new episode begins.

To call this service, the Robot Controller node constructs a request containing the desired position, orientation (pose), and initial velocity (twist) for the robot. The client then sends this request to the `/demo/set_entity_state` service. The service processes the request and updates the robot's state accordingly. This capability is crucial for resetting the simulation environment and ensuring that the robot starts each episode from a known initial state, facilitating consistent and controlled training conditions.

Additional Functionalities in the Robot Controller Node:

Beyond the primary functionalities (publisher, subscribers, service) of the Robot Controller node, several supplementary functions play a crucial role in enhancing its overall

operation. These functions manage tasks such as obstacle inflation, coordinate transformations, and path planning. They provide essential support for both the pre-requisites and post-requirements of the robot's reinforcement learning, decision-making, and navigation capabilities. Below is a detailed description of each supporting function:

Function Name	Input(s)	Output(s)	Description
inflate_obstacles	localmap, inflation radius	inflated costmap	Expands detected obstacles on the local map for robot and pedestrian safety.
transform_to_local	robot location (global coordinates), origin of local map, yaw of the robot	robot location (local coordinates)	Converts global coordinates to local map coordinates.
find_nearest_free_cell	inflated costmap, robot location (local coordinates)	new robot starting position (local coordinates)	Finds the nearest free cell when the robot is too close to obstacles, ensuring a safe start position.
astar	inflated costmap, robot location (local coordinates), goal location (local coordinates)	array of local path coordinates	Generates a local path to the target, avoiding obstacles.
transform_to_global	robot location (local coordinates)	robot location (global coordinates)	Converts local map coordinates to global coordinates.
find_nearest_point_index	path to follow, robot location (global coordinates)	index of the nearest point	Finds the nearest point on the path relative to the robot's location.
pure_pursuit	robot location (global coordinates), yaw, path, index, lookahead distance	linear velocity, angular velocity, index	Calculates desired linear and angular velocities for the robot to follow the path.

Table 3.1: Supporting Functions in Robot Controller Node

3.3.5 Trajectory Planning Layer

The Trajectory Planning Layer is responsible for converting geometric paths into feasible trajectories that the robot can follow while considering vehicle dynamics, environmental constraints, and safety. This layer ensures that the robot does not just reach a destination but follows a smooth and dynamically feasible trajectory while adapting to real-time conditions.

In this system, the RL agent has three primary actions: **Global Path Follow, Local Path Follow,** and **Full Stop**. Since Full Stop can be executed by setting both linear and angular velocities to zero, trajectory planning is primarily required for Global Path Follow and Local Path Follow actions. This layer works alongside the Decision-Making Layer, with both requiring continuous adjustments based on real-time conditions.

For Global Path Follow, a predefined path is manually provided as a sequence of waypoints in $(x,\ y)$ coordinates, structured as $[[x_1, y_1], [x_2, y_2], \ldots, [x_n, y_n]]$, spanning from the start to the goal position. While no additional path planning is performed at this stage, the Pure Pursuit controller dynamically calculates the necessary linear and angular velocity commands to smoothly track the given waypoints, ensuring stability and efficiency throughout the motion.

For Local Path Follow, real-time path adjustments are required to handle obstacles and environmental variations. The `dynamic_map` node continuously updates the robot's perception of its surroundings by maintaining a dynamic occupancy grid, as explained in Section 3.3.2.1. This updated map is processed by the RL Environment Node, which utilizes the A^* algorithm to generate a local path from the robot's current position to the next waypoint on the global path. However, a raw set of waypoints alone is not sufficient—the Pure Pursuit controller further refines this local path into a smooth and executable trajectory by dynamically calculating linear and angular velocity commands.

Regardless of whether the robot follows a manually defined global path or an adaptively generated local path, Pure Pursuit control ensures real-time trajectory generation by continuously computing linear and angular velocity inputs to maintain smooth and efficient motion. Once the trajectory is determined, it is forwarded to the Vehicle Dynamics Control Layer for execution, ensuring that the robot navigates effectively while adapting to real-world conditions.

3.3.6 Vehicle Dynamics Control

The Vehicle Dynamics Control Layer is responsible for executing the velocity commands generated by the Trajectory Planning Layer, ensuring precise motion control of the GrassHopper robot. While the Trajectory Planning Layer determines the required linear and angular velocities to follow a planned trajectory, this layer translates those velocity commands into actuator signals, allowing the robot to follow the designated path while maintaining stability and responsiveness.

For both Global Path Follow and Local Path Follow, the Pure Pursuit controller, implemented within the Robot Controller node (as mentioned in Table 3.1), computes velocity commands based on the given path (as detailed in Section 3.3.5). Once these velocity values are determined, the Vehicle Dynamics Control Layer ensures their seamless transmission to the robot's actuators, enabling real-time execution of planned motions.

Whenever the Decision-Making Layer selects a new action, the Vehicle Dynamics Control Layer applies the corresponding control logic. If the selected action is Full Stop, both linear and angular velocities are set to 0 m/s, bringing the robot to a complete halt. However, for Global Path Follow and Local Path Follow, the `pure_pursuit()` function from the `RobotController` class is invoked, using the planned path as input. This function continuously adjusts the velocity commands based on the robot's real-time position and orientation as a part of the Trajectory Planning Layer. The Vehicle Dynamics Control Layer ensures these dynamically updated commands are precisely forwarded to the actuators, enabling smooth and stable motion execution.

Since this layer is tightly integrated with both the Trajectory Planning Layer and Decision-Making Layer, it must dynamically respond to real-time updates in environmental conditions. As soon as the Trajectory Planning Layer adapts to changes and recalculates the required velocity commands, the Vehicle Dynamics Control Layer takes these updated values and ensures their timely transmission to the actuators. This guarantees stable and accurate motion execution, enabling seamless navigation in both structured and dynamic environments.

3.4 Overcoming Software Development Challenges

During the development of the reinforcement learning-based navigation system for the GrassHopper mobile robot, several challenges were encountered, particularly in the **Perception Layer** of the system architecture. Two primary challenges were encountered: **Dynamic Occupancy Grid Generation** and **Object Detection With Depth Estimation**. The ability to accurately perceive obstacles and free space in real-time is essential for local path planning and collision avoidance. Additionally, identifying the object class and estimating its depth enables the RL agent to make informed decisions based on the type and proximity of obstacles, providing a broader understanding of the surroundings. This section discusses these challenges in detail and the approaches taken to overcome them.

3.4.1 Dynamic Occupancy Grid Generation

Accurately mapping the environment is a crucial aspect of autonomous navigation, especially in dynamic environments where obstacles such as humans, vehicles, or other moving objects frequently change positions. To generate an occupancy grid, two popular SLAM (Simultaneous Localization and Mapping) approaches were explored: Turtlebot3 Cartographer and SLAM Gmapping. Both methods were tested for their ability to generate a dynamic occupancy grid, which is essential for real-time navigation and path planning. However, despite parameter tuning, both approaches struggled with handling dynamic obstacles, leading to incorrect path planning decisions.

3.4.1.1 Turtlebot3 Cartographer Approach

Cartographer is a real-time SLAM method that builds a 2D occupancy grid by fusing LiDAR data, odometry, and IMU readings. It uses scan matching and pose graph optimization to improve localization and mapping accuracy. Key parameters were adjusted to improve the detection and removal of dynamic obstacles:

- `TRAJECTORY_BUILDER_2D.min_range` and `TRAJECTORY_BUILDER_2D.max_range`: Defines the minimum and maximum sensor range for detecting obstacles.
- `TRAJECTORY_BUILDER_2D.motion_filter.max_angle_radians`: Filters small sensor noise by ignoring minor rotational changes.
- `TRAJECTORY_BUILDER_2D.use_online_correlative_scan_matching`: Enables real-time scan matching to improve mapping accuracy.
- `POSE_GRAPH.constraint_builder.min_score`: Determines the minimum confidence score required to consider a scan match valid.

Despite adjusting these parameters, Turtlebot3 Cartographer failed to update the occupancy grid dynamically. When a moving obstacle, such as a person, passed through an area, its occupied cells remained marked in the occupancy grid, even after the object had

moved away. This resulted in "ghost obstacles" that were never cleared, which caused the robot to consider certain paths as blocked, even when they were actually free.

3.4.1.2 SLAM Gmapping Approach

SLAM Gmapping is a widely used SLAM method that creates an occupancy grid based on LiDAR and odometry data. It continuously updates the map as the robot explores the environment. Some key parameters affecting its performance include:

- `lineatUpdat`: Defines the distance the robot must move before updating the map.
- `angularUpdate`: Specifies how much the robot must rotate before incorporating new sensor data.
- `temporalUpdate`: Controls the frequency of map updates.
- `map_updat_interval`: Determines how often the occupancy grid is refreshed with new sensor data.

Even after parameter tuning, SLAM Gmapping also retained past obstacle data, failing to clear occupied cells when a dynamic obstacle moved. As a result, when a person walked through the robot's path, SLAM Gmapping continued to mark that region as occupied, creating phantom obstacles. This often led to incorrect or infeasible local paths, causing the robot to either reroute inefficiently or be unable to find a valid path to the target.

3.4.1.3 Problem Summary and Solution

Problem Summary:

Both Turtlebot3 Cartographer and SLAM Gmapping were effective at generating static maps but struggled with dynamic environments. Their inability to remove outdated obstacle data resulted in persistent false obstacles, leading to navigation failures. This challenge highlights the need for a more dynamic mapping solution that can continuously update the occupancy grid in real time, ensuring accurate path planning.

Solution:

To overcome the limitations of Turtlebot3 Cartographer and SLAM Gmapping in handling dynamic obstacles, a separate custom Python node was developed to generate a real-time dynamic occupancy grid using LiDAR and odometry data. Unlike traditional SLAM-based approaches that retain historical obstacle data, this node ensures that only the latest sensor readings contribute to the occupancy grid.

The node operates in a continuous loop, where:

1. **Clearing the Grid:** At the start of each cycle, the entire occupancy grid is reset, removing outdated obstacle information.
2. **Updating with Sensor Data:** The latest LiDAR scan is processed, and occupied cells are marked based on the detected obstacles.
3. **Incorporating Odometry Data:** The robot's position and movement are considered to ensure consistent grid updates.

This dynamic approach prevents "ghost obstacles" from persisting in the map, allowing the RL-based navigation system to make more informed path-planning decisions. The full process of this method is explained in Section 3.3.2.1.

3.4.2 Object Detection with Depth Estimation

Reliable object detection and depth estimation are vital for reinforcement learning-based navigation, enabling the robot to identify obstacles and assess their distances. This information forms the basis for effective path planning, obstacle avoidance, and meaningful interaction with the environment.

3.4.2.1 Camera-Based Object Detection and Depth Estimation

The initial approach for object detection involved identifying the class of an object and determining its bounding box within the camera frame. Depth estimation was then performed by extracting the depth value from the pixel coordinates at the center of the bounding box. This straightforward method worked seamlessly in real-world experiments using the Intel RealSense D435i camera, which provided accurate depth values corresponding to detected objects.

The combined use of the camera's object detection capability and its integrated depth sensing allowed the robot to accurately estimate distances to obstacles and navigate the environment effectively. However, this method encountered significant challenges when applied to simulation environments using a depth camera plugin.

3.4.2.2 Problem Summary and Solution

Problem Summary:

In the real-world setup, depth estimation was straightforward, as the depth camera provided accurate depth values corresponding to the detected object's pixel coordinates. However, when transitioning to a simulated environment, the depth camera plugin failed to provide reliable depth measurements. Instead of returning correct depth values based on object distance, the plugin randomly assigned a maximum depth value $(10.0m)$, even when objects were much closer.

This inconsistency made it impossible to use the same depth extraction method in simulation as in real-world testing. Without accurate depth information, the reinforcement learning agent lacked the ability to properly assess object distances, leading to incorrect decision-making in navigation tasks. Despite attempts to modify plugin settings and sensor parameters, the issue persisted, making it necessary to explore an alternative approach to depth estimation in simulation.

Solution:

To overcome the limitations of the depth camera plugin in simulation, LiDAR data was integrated with the camera to estimate object distances. Instead of relying solely on the depth camera, the system used the camera for object classification and the LiDAR for distance measurement.

The approach involved mapping the FoV of camera onto the FoV LiDAR to correlate detected objects with LiDAR distance data. This was achieved by:

1. Identifying the bounding box center of the detected object in the camera frame.
2. Mapping the horizontal pixel coordinate of the bounding box center to the corresponding LiDAR index.
3. Retrieving the depth value from the LiDAR data at the mapped index.

By fusing camera and LiDAR data, the system achieved reliable depth estimation, allowing the reinforcement learning agent to make accurate navigation decisions. The full process of object detection and depth estimation is explained in Section 3.3.2.2.

Chapter 4

Implementation

The implementation phase is a key part of this research, where the focus shifts from theoretical concept to practical application. In reinforcement learning (RL), the process of training an agent involves iterative interactions with its environment to gradually learn effective policies. This requires significant time and computational effort, making a simulation environment an ideal starting point. Simulations offer a controlled, efficient, and risk-free platform to test, refine, and test the agent's behavior before moving to real-world deployment.

The primary objectives of this chapter are:

1. **Gazebo Simulation:** To train and evaluate the RL agent, the Gazebo simulation platform was utilized, which provides a realistic 3D environment. Gazebo simulates the robot model, sensors, and environmental dynamics, enabling efficient testing of algorithms such as perception, dynamic occupancy grid mapping, and RL training.

2. **Prerequisite Implementation:** Implement and validate essential components necessary for the agent's successful operation. These include object detection and depth recognition, which allow the robot to perceive and interpret its surroundings, and dynamic occupancy grid creation, which provides a real-time map of the environment. This map is used to identify obstacles and plan safe paths, ensuring the robot can make informed and effective navigation decisions.

3. **Training the RL Agent:** Detail the training process of the agent within the simulated environment. This includes fine-tuning hyperparameters to optimize learning performance and executing the training process in a structured manner. The goal is to develop a functional RL policy capable of navigating a simple environment efficiently.

4. **Transition to Real-World Implementation:** After achieving satisfactory results in simulation, the trained RL agent is deployed onto a real robot. This step bridges the gap between theoretical simulation and practical application, tackling challenges such as sensor noise, hardware calibration, and environmental unpredictability.

This chapter will systematically present the implementation process, starting with the simulation-based development and training of the RL agent. Then further discuss the challenges and strategies for transferring the trained model to a real robot for real-world testing. This structured approach aim to demonstrate a practical and reliable method for enhancing outdoor navigation using reinforcement learning.

4.1 Software-in-the-Loop Simulation

Software-in-the-Loop (SIL) implementation plays a crucial role in system development by enabling early detection of software errors, logical issues, and performance bottlenecks. SIL allows software to be tested in a simulated environment that mirrors real-world hardware, reducing reliance on physical robot and sensors. This approach not only lowers costs and shortens development time but also ensures the software meets essential safety, reliability, and performance requirements before deployment in critical applications, such as autonomous navigation systems for mobile robots.

4.1.1 Gazebo Simulation Setup

Gazebo is a simulation tool used to replicate physical robot and 3D environments in a virtual environment. It enables the development, testing, and validation of subsystems and prerequisites in a controlled environment. In this work, Gazebo is utilized to create a digital twin of the GrassHopper robot, complete with sensors and actuators, as well as a custom outdoor environment designed to mimic real-world conditions. This setup is crucial for training the RL agent and evaluating its performance before transitioning to real-world deployment.

4.1.1.1 Grasshopper Description and URDF

To replicate the real-world robot GrassHopper, developed at Schmalkalden University of Applied Sciences, a simulated version was created for this thesis using the Unified Robot Description Format (URDF). The simulated GrassHopper robot closely mirrors the physical robot in both design and functionality, providing a reliable digital twin for testing and training purposes. Figure 4.1 illustrates the simulated GrassHopper robot used in this thesis.

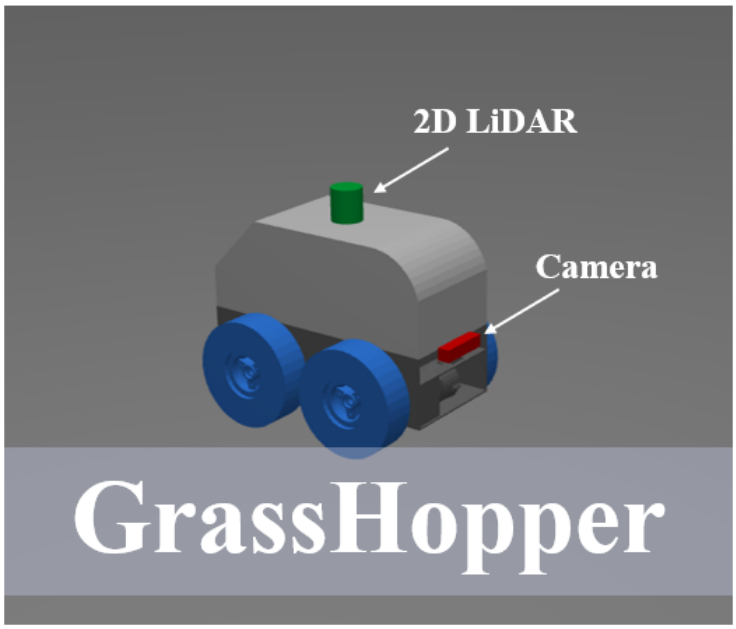

Figure 4.1: 3D Simulation Model of GrassHopper.

URDF is an XML-based format used to describe the complete physical structure of a robot, including its base, wheels, sensors, and actuators. It consists of various links representing the different components of the robot, with key attributes such as inertial

properties, geometry, and visual appearance for each link. Additionally, it includes joints that define the connections between these links. The GrassHopper URDF model used in this thesis incorporates all the essential elements required to create a realistic simulation model of the robot.

Key Features of the GrassHopper URDF Model:

1. **Physical Structure:**
   - The URDF specifies all links and joints, including the base, wheels, LiDAR and camera sensor.
   - Each link includes an inertial module to simulate realistic physics, ensuring accurate representation of mass and inertia for each component.
2. **Drive System:**
   - The robot's motion is controlled using the `libgazebo_ros_diff_drive.so` plugin, which simulates a differential drive system.
   - This plugin allows precise control over wheel velocities and robot movement in the simulated environment.
3. **Sensors:**
   - The GrassHopper is equipped with a 2D LiDAR mounted on the top and a camera on the front, both accurately modeled in the simulation.
   - The `libgazebo_ros_ray_sensor.so` plugin is used for simulating the 2D LiDAR, providing range measurements for obstacle detection and mapping.
   - The `libgazebo_ros_camera.so` plugin simulates the camera, enabling vision-based tasks such as object detection.

The simulated GrassHopper features a 2D LiDAR, a camera, and a differential drive system, equipping it with the necessary sensing and motion capabilities for autonomous navigation tasks. By utilizing the URDF model and Gazebo simulation tools, this setup achieves a high level of realism and precision, effectively bridging the gap between simulation and real-world application. The complete GrassHopper URDF model utilized in the Gazebo simulation with ROS2-Humble is detailed in Appendix A.1.

4.1.1.2 3D Simulation Environment

To train and test the algorithms developed in this thesis, a custom outdoor 3D simulation environment was created in Gazebo. This environment replicates real-world outdoor conditions, providing a platform for evaluating the performance of various algorithms under realistic scenarios. A representation of the complete 3D simulation environment is available in Appendix A.6, in Figure A.1.

The simulation environment includes a 2.5 *meter* wide sidewalk, designed to mimic common urban navigation scenarios. For simplicity, the environment does not feature crosswalks or complex road-crossing scenarios, allowing the focus to remain on obstacle avoidance and navigation along a predefined path. The sidewalk is populated with static obstacles such as dustbins, fire hydrants, post boxes, and stationary individuals, ensuring a variety of real-world conditions for testing perception and navigation algorithms.

Dynamic elements, such as moving pedestrians, are also integrated into the environment to simulate real-world dynamic obstacle scenarios. These moving entities are

programmed to move within the simulation with realistic speed and behavior, creating challenges for the robot's perception and decision-making algorithms.

The environment incorporates realistic physical parameters, including:

- **Wind:** Simulates external forces acting on the robot.
- **Light:** Models varying lighting conditions such as daytime brightness or shadows.
- **Friction:** Ensures accurate interaction between the robot and the ground surface.
- **Gravity:** Provides a natural force that influences the robot's movement and stability.

The custom environment was built using pre-existing Gazebo models, which were carefully selected, modified, and arranged to create a practical and immersive simulation world. By combining both static and dynamic elements with realistic physical properties, this environment closely replicates the challenges the robot is likely to face in real-world conditions, making it an ideal platform for testing and validating the developed algorithms. The custom-built simulation environment developed for this thesis is provided in Appendix A.2.

With the simulation environment fully set up and the GrassHopper robot model ready, the next step involves testing the system prerequisites. This critical phase ensures that the sensor plugins function correctly and that the perception algorithms process data as expected. This step helps catch and resolve any potential software or logical issues before proceeding to the training of the RL agent.

4.1.2 Prerequisite Implementation

Before training the RL agent, it is essential to implement and verify the foundational perception algorithms to ensure they function as intended. This section covers the development and integration of two key subsystems: Object Detection with Depth Estimation and Dynamic Occupancy Grid creation. These modules equip the robot with the ability to perceive and interpret its environment, detect and localize obstacles, and generate a real-time understanding of its surroundings. By effectively implementing these components in the simulation, the groundwork is established for the RL agent to navigate autonomously and make intelligent decisions. The subsections that follow provide insights into their integration with the simulation and the outcomes of these essential systems.

4.1.2.1 Dynamic Occupancy Grid

The `dynamic_map` node is responsible for generating a dynamic occupancy grid around the robot based with the help of 2D LiDAR data and odometry information. This dynamic occupancy grid/local map moves continuously with the robot, providing essential environmental awareness for navigation. The local map is highly beneficial for outdoor environments, where real-time obstacle detection and mapping are crucial for safe and effective decision-making and path planning.

Node Description:

The `dynamic_map` node plays a vital role in the system by processing laser scan data from the `/scan` topic and odometry data from the `/odom` topic. It combines this information to produce a continuously updated dynamic occupancy grid, enabling obstacle detection and costmap generation. The complete process for generating the dynamic occupancy grid was explained in Section 3.3.2.1. The generated local map is published on the

`/local_map` topic, while a detailed costmap is published on the `/costmap` topic for later use in path planning around obstacles.

The occupancy grid represents the robot's immediate environment, dynamically adjusting as the robot moves and new sensor data becomes available. This real-time mapping allows the system to maintain an accurate perception of its surroundings, which is crucial for navigating dynamic environments. The high update frequency of the node ensures smooth and precise responses to environmental changes, with updates occurring at $15Hz$. Figure 3.5 illustrates the data flow for this mapping process.

Output Visualization and Analysis:

The `dynamic_map` node generates two critical outputs: the local map and the costmap, both essential for real-time navigation. These outputs can be visualized in RViz2 by setting the map as the global frame, providing valuable insight into the robot's perception of its environment. Figure 4.2 illustrates the visualization setup, with the Gazebo simulation environment on the left and the corresponding map outputs in RViz2 on the right. This side-by-side comparison allows for a thorough analysis of the simulated environment and its dynamically generated map.

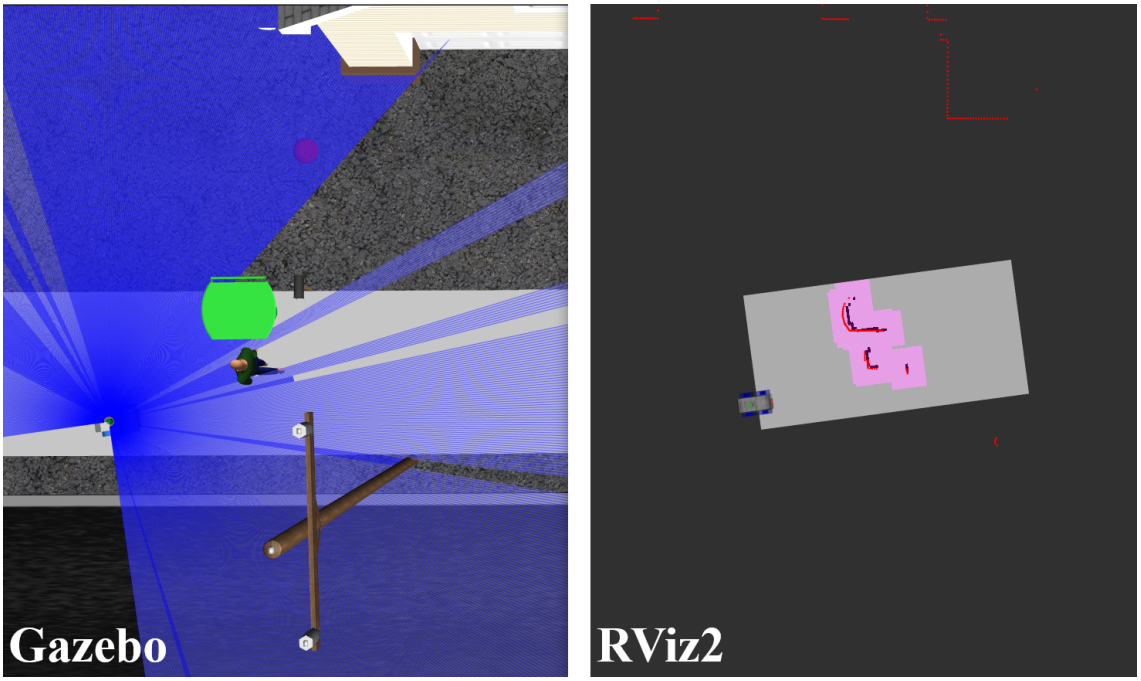

Figure 4.2: Local Map Visualization in RViz2 with corresponding Gazebo environment.

The local map spans a width of 2.5 *meter* and a length of 5 *meter*, with a resolution of 0.05 *meter per cell*, offering precise spatial representation. As shown in Figure 3.4, the asymmetrical map distribution allows the path planner to efficiently plan routes that steer clear of obstacles on the left side. This detailed and continuously updated map enables the robot to perceive its surroundings accurately and plan safe, adaptive paths in real-time.

Summary:

The `dynamic_map` node plays a key role in the robot's navigation system by generating real-time maps from LiDAR and odometry data, facilitating obstacle avoidance and path planning. Its integration with RViz2 supports effective monitoring and debugging. The complete ROS2 package for this node is documented in Appendix A.3.

4.1.2.2 Object Classification, Distance, and Angle Estimation

The `object_detection` node plays a critical role in detecting and classifying objects in the robot's environment while estimating their distances and angles relative to the robot. By processing sensor data from both camera and LiDAR inputs, this node enables the robot to effectively perceive and interpret its surroundings. The detailed classification, depth, and angle information it provides empowers the RL agent to make informed and context-aware decisions, enhancing the robot's situational awareness and navigation capabilities.

Node Description:

The `object_detection` node is responsible for detecting and classifying objects in the robot's environment while estimating their distances and angles relative to the robot. It subscribes to the `/camera/image_raw` topic to receive image data from the camera and the `/scan` topic for LiDAR data. By fusing these data sources, the node accurately associates detected objects with their respective depth and angular positions. The complete process for object classification and distance estimation is outlined in Section 3.3.2.2.

The output from the node consists of three key data points: object class, distance, and angle. For simplicity, only two object classes—**"Person"** and **"Dustbin"**—are considered in this implementation. These data points are packaged into a custom ROS message (`ObjectDetection`) and published on the `/detections_publisher` topic in a list format, such as $[['person', 2.17, 71.33], ['dustbin', 3.03, 108.67]]$, where each entry includes the object class, distance (in meters), and angle (in degrees) relative to the robot.

To improve decision-making for the RL agent, the node prioritizes the closest object of each class, ensuring that the RL agent receives only the most relevant data for obstacle avoidance and navigation. This prevents unnecessary confusion in the decision-making process. The node updates at a frequency of 10 *Hz*, ensuring timely data flow. Figure 3.7 illustrates the data flow for this object detection node.

Output Visualization and Analysis:

The `object_detection` node generates three key outputs: object class, depth, and angle, which are crucial for the robot to understand its environment. To visualize these outputs, `cv2.imshow()` is used to display a separate window where the detected objects are highlighted with bounding boxes. Each bounding box is labeled with the object class, distance (depth), and angle relative to the robot. This visualization provides an immediate and clear view of how the robot perceives and identifies objects in its surroundings, offering valuable insights for system monitoring and debugging.

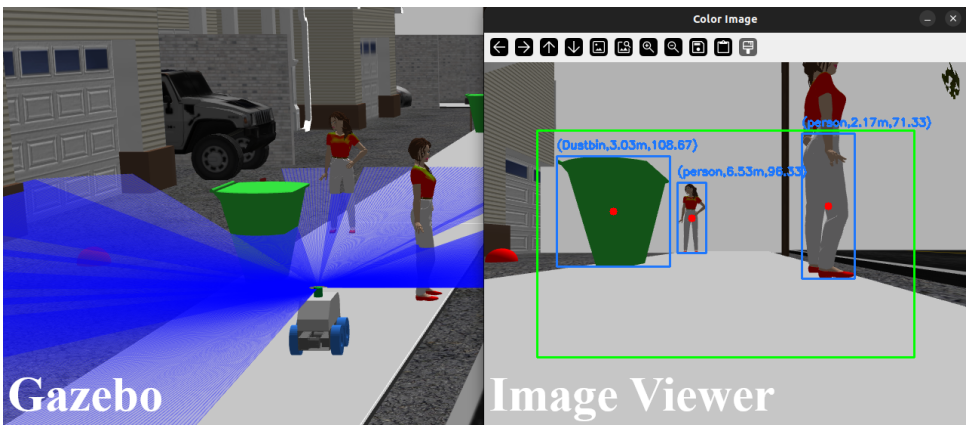

Figure 4.3: Object Detection Visualization with corresponding Gazebo environment.

Figure 4.3 illustrates the setup, with the Gazebo simulation environment shown on the left, and the output from the `cv2.imshow()` window on the right. The bounding boxes in the window are labeled with the respective object class, distance, and angle, making it easy to track the robot's perception of the environment. This visualization setup helps to ensure that the object detection and depth estimation processes are functioning as expected, allowing for easier validation and fine-tuning of the system.

Summary:

The `object_detection` node detects and classifies objects, estimating their distance and angle relative to the robot using data from the camera and LiDAR. It publishes the object class, depth, and angle in a custom message, which is used for decision-making in navigation. The node's real-time output can be visualized through bounding boxes in a separate window, helping to monitor and debug the system. The complete ROS2 package for object classification and depth estimation is provided in Appendix A.4.

In both the `dynamic_map` and `object_detection` nodes, the odometry data required for processing is obtained directly from the `/odom` topic generated by the differential drive plugin in Gazebo. This plugin simulates the robot's wheel movements and provides accurate odometry data, making it a reliable source for position and orientation information during the simulation.

The GrassHopper robot in the simulation is also equipped with GNSS and IMU plugins, allowing the implementation of a dual-EKF node to combine odometry from GNSS, IMU, and differential drive sources. This setup mirrors the capabilities of the real-world robot, where sensor fusion is crucial for accurate navigation. However, to reduce computational overhead in the simulation environment and since the differential drive plugin already provides precise position data, the dual-EKF node was not utilized during simulation. This decision ensured efficient resource usage without compromising the accuracy required for training and testing the RL agent.

With the simulation environment fully set up and the perception prerequisites seamlessly integrated, the groundwork for autonomous decision-making is complete. The system is now primed for the next crucial phase—training the RL agent to navigate and interact intelligently within the simulated environment.

4.2 Training the RL Agent

In this section, we delve into the critical process of training the reinforcement learning (RL) agent for tactical outdoor navigation. The journey begins by designing a custom OpenAI Gym environment tailored to imitate real-world navigation challenges. This is followed by the development of a robust ROS 2 node to execute, monitor, and debug the agent's logic seamlessly. Key stages include hyperparameter tuning, iterative training and retraining cycles, and comprehensive data collection. The collected data serves a dual purpose: evaluating the agent's learning progress and validating its navigation capabilities. By the end of this training process, the RL agent is expected to evolve into a sophisticated decision-maker capable of handling complex outdoor navigation tasks.

4.2.1 Designing the Custom Gym Environment

The custom environment provides a structured platform for developing and testing reinforcement learning (RL) agents by focusing on decision-making and control logic. The custom environment, named `OutdoorEnv`, inherits from two key classes. First, the `gymnasium.Env` class, which defines the structural framework for the environment,

including observation and action spaces, as well as essential functionality to develop `step()`, `reset()`, and `reward()`. Second, the `RobotController` class, which provides essential supportive functionalities, as detailed in Table 3.1. This class subscribes to various ROS 2 topics to enable seamless communication with the simulated Grasshopper robot, its sensors, and perception nodes. Moreover, it is responsible for executing the robot's movement within the 3D simulation environment based on the actions selected by the RL agent, ensuring smooth and efficient operation. These features are comprehensively explained in Section 3.3.4.2.

A well-designed RL environment requires clearly defined observation and action spaces. The observation space captures crucial information about the robot's internal state and surrounding environment, enabling the agent to make informed decisions. Conversely, the action space outlines the set of possible maneuvers the agent can execute. Careful consideration was given to designing these spaces to promote efficient learning of optimal navigation strategies, as described in Section 3.3.4.1.

To enable efficient training, `OutdoorEnv` implements several essential functions fundamental to RL agent interactions:

1. **Step Function:** This function manages the agent's interaction with the environment by executing an action, updating the robot's state, and returning the new state, reward, and episode completion status.
2. **Reset Function:** Responsible for reinitializing the environment to its initial state, this function is vital for beginning new episodes during training.
3. **Reward Function:** The reward mechanism evaluates the agent's performance, guiding it toward better navigation strategies by assigning positive or negative rewards based on the task outcome.

These functions form the core of the interaction cycle and contribute significantly to the learning process by offering structured feedback for the agent's actions. Additionally, careful design of these functionalities helps the RL agent adapt and generalize better to complex real-world navigation scenarios, improving long-term performance and tactical decision-making capabilities. These functions are the backbone of agent-environment interactions, and elaborated in Section 3.3.4.2.

To support the seamless integration of the custom environment with the broader system architecture, a ROS2 node was developed. This node facilitates communication between the environment and various system components while offering essential debugging and diagnostic capabilities. It plays a crucial role in managing the system's operations and ensuring efficient data flow during training. The development and features of this critical ROS2 node are explored in the following section.

4.2.2 ROS2 Node for Training and Optimization

In the previous section, a custom Gym environment was developed to support the training of the RL agent. To utilize this environment within Gymnasium, it must first be registered and made accessible, a task handled by the `start_training.py` ROS2 node. Beyond environment registration, this node serves as the control hub for managing the agent's training process. It offers four distinct training modes: `random_agent`, `training`, `retraining`, and `hyperparam_tuning`, each catering to different stages and objectives of agent development. This section explores the design and implementation of the `start_training.py` node, detailing its functionality and the role of each training mode in optimizing the performance of the RL agent.

To use the custom `OutDoorEnv` environment within Gymnasium, it must first be registered. This is accomplished using the `gymnasium.envs.registration.register()` function. The registration process involves specifying several key arguments, such as:

- **id:** A unique identifier for the environment (`OutDoorEnv-v0`).
- **entry_point:** The module and class path for the custom environment (`outdoor_robot_spawner.outdoor_env:OutDoorEnv`).
- **reward_threshold:** A target cumulative reward that signifies the task is considered solved.
- **max_episode_steps:** A target cumulative reward that signifies the task is considered solved.

The other arguments, such as `nondeterministic`, `order_enforce`, and `kwargs`, provide additional information about the environment but do not directly affect its behavior. Once registered, the environment can be created using the `gymnasium.make('OutDoorEnv-v0')` function. This step initializes the environment and prepares it for interaction with the RL agent. By registering and creating the environment, the `start_training.py` node ensures that the custom Gym environment becomes seamlessly accessible to Gymnasium, forming a crucial step in the training process of the RL agent.

With the environment successfully registered and accessible, the next step involves exploring the available training modes offered by the `start_training.py` node to facilitate various stages of the RL agent's development.

4.2.2.1 Random Agent Mode

The `random_agent` mode serves as a crucial debugging phase, designed to identify and resolve potential bugs or logical errors within the complete reinforcement learning (RL) algorithm structure before commencing formal agent training. Its primary objective is to run a predefined number of trial episodes, functioning as a dry run to validate the system's overall integrity.

During these trial episodes, the `random_agent` mode meticulously processes key functional logics and communication protocols, such as publisher-subscriber interactions, environment registration, and creation. Furthermore, it thoroughly tests essential RL components, including the step method, reward mechanism, environment reset procedure, and the proper definition and handling of observation and action spaces.

Notably, this mode does not store any data or learning outcomes from the trial runs. Its sole purpose is to ensure that the RL algorithm, along with supporting nodes for sensor data processing and perception algorithms, works harmoniously as an integrated system. By confirming that all components function as expected, the `random_agent` mode lays the groundwork for a seamless and error-free training process in subsequent stages.

4.2.2.2 Training Mode

The `'training'` is responsible for training the RL agent using the Proximal Policy Optimization (PPO) algorithm from the **Stable Baselines3** library, which plays a central role in this thesis for tactical decision-making in mobile robot navigation. The training process involves three main steps: initializing the PPO model, starting the training, and saving the trained model.

Initialize the PPO Model:

The PPO model is initialized from the Stable Baselines3 library using the following piece of code:

```
model=stable_baselines3.PPO("MultiInputPolicy",env,verbose=1,
          tensorboard_log=log_dir, n_steps=2048, gamma=0.99,
          gae_lambda=0.95, ent_coef=0.0, vf_coef=0.5,
          learning_rate=0.0003, clip_range=0.02)
```

The model is configured with several hyperparameters that influence the learning process. While many default values are retained, the following parameters were specifically adjusted for this research:

- `learning_rate` (default: 0.0003): The gradient update step size.
- `n_steps` (default: 2048): The number of steps to run per policy update.
- `gamma` (default: 0.99): Discount factor γ to compute the infinite-horizon discounted return.
- `clip_range` (default: 0.02): Clipping parameter to stabilize policy updates.
- `gae_lambda` (default: 0.95): Factor to regulate the bias-variance trade-off for the generalized advantage estimator.
- `ent_coef` (default: 0.0): Entropy coefficient for the loss calculation.
- `vf_coef` (default: 0.5): Value function coefficient for the loss calculation.

Careful tuning of these parameters was crucial for developing robust navigation strategies. Detailed explanations of the tuning process are provided in Section 4.2.2.4.

Start the Training:

The complete training process begins with the following piece of code:

```
model.learn(total_timesteps=int(800000),reset_num_timesteps =
      False, callback = [eval_callback,reward_logger],
      tb_log_name=f"PPO_test_{timestamp}",progress_bar=False)
```

The `model.learn()` method puts together the entire training process. It iteratively refine decision-making capabilities of RL agent by processing environmental interactions, executing essential RL algorithm functions, updating model parameters, and evaluating performance through callbacks. The process continues until the specified total timesteps are reached, ensuring the agent undergoes comprehensive training.

Save the Trained Model:

Upon completion of training, the trained model is stored at the specified location with a corresponding timestamp to better organize the training data using the following command:

```
model.save(f"{trained_models_dir}/PPO_test_{timestamp}")
```

The `model.save()` function preserves the trained policy, allowing it to be reloaded for future use. This capability enables the deployment of the trained model in simulation environments to validate and evaluate the agent's navigation strategies. The ability to store and reload models supports iterative improvements and rigorous testing in dynamic scenarios. After validation and testing, this saved model can be easily transferred to the real robot for implementation.

4.2.2.3 Retraining Mode

The `'retraining'` addresses the challenge of lengthy training times often associated with RL models. This mode enables the loading of a previously trained model to continue training from its last state. It is particularly useful for incremental training strategies, where the model is initially trained on simpler environments and then further refined on more complex ones. This approach leverages the knowledge gained in simpler environments to enhance performance in challenging scenarios.

The retraining process involves three main steps: loading the trained model, starting the training, and saving the retrained model. While the last two steps are identical to those in the `'training'`, the first step is distinct.

Load the Trained Model:

The trained model is loaded using the Stable Baselines3 library with the following code:

```
model=stable_baselines3.PPO.load(trained_model_path,env=env,
                                 custom_objects=custom_obj)
```

Here, the `'trained_model_path'` specifies the location of the model to be retrained. Once loaded, the model becomes ready for further training in the same or a different environment.

Start Training and Save the Model:

The steps for starting the training and saving the retrained model remain the same as in `'training'`. By enabling model reusability, `'retraining'` provides a flexible and efficient way to build progressively stronger navigation models while saving valuable training time.

4.2.2.4 Hyperparameter Tuning Mode

Hyperparameters in Reinforcement Learning (RL) algorithms are essential configurations that influence how the model learns from interactions with its environment. Unlike model parameters, which are learned during training, hyperparameters are manually set before training and significantly affect the agent's performance and stability. These include settings like learning rate, discount factor, and the number of steps between policy updates.

Proximal Policy Optimization (PPO), the algorithm central to this thesis, offers more than 20 hyperparameters, each affecting the agent's training behavior. However, not all of them are equally impactful for every use case. This thesis focuses on tuning a carefully selected set of hyperparameters to improve the agent's navigation decision-making:

- **Learning Rate** (`learning_rate`)**:** Controls how quickly the agent updates its policy based on new information.
- **Number of Steps** (`n_steps`)**:** Defines how many steps the agent takes before a policy update.
- **Discount Factor** (`gamma`)**:** Balances the importance of immediate versus future rewards.
- **Clip Range** (`clip_range`)**:** Prevents large, destabilizing policy updates by constraining policy changes.

- **Generalized Advantage Estimator** (`gae_lambda`)**:** Regulates the bias-variance trade-off in policy updates.
- **Entropy Coefficient** (`ent_coef`)**:** Encourages exploration by adding a bonus to the loss function, preventing convergence to suboptimal strategies.
- **Value Function Coefficient** (`vf_coef`)**:** Balances the importance of value estimation in optimization, aiding effective action selection.

Carefully tuning these parameters was essential to achieving desire performance in diverse navigation scenarios.

Tuning Process with Optuna:

To streamline hyperparameter tuning, the **Optuna** optimization framework was employed. Optuna automates the search for optimal hyperparameter values by conducting a series of training trials, each testing a different parameter combination.

The tuning process involves three primary steps:

1. **Defining the Search Space:** Optuna requires specifying the range of possible values for each hyperparameter. The following search space was defined to allow systematic exploration of different configurations:
   - **Number of Steps per Update (**`n_steps`**)**: Integer values between 2048 and 14,336.
   - **Discount Factor (**`gamma`**)**: Log-uniformly sampled between 0.96 and 0.9999.
   - **Learning Rate (**`learning_rate`**)**: Log-uniformly sampled between 1e-5 and 9e-4.
   - **Clipping Range (**`clip_range`**)**: Uniformly sampled between 0.15 and 0.37.
   - **Generalized Advantage Estimation (**`gae_lambda`**)**: Uniformly sampled between 0.94 and 0.99.
   - **Entropy Coefficient (**`ent_coef`**)**: Log-uniformly sampled between 1e-8 and 1e-5.
   - **Value Function Coefficient (**`vf_coef`**)**: Uniformly sampled between 0.55 and 0.65.

   The default values for these parameters are as mentioned in Section 4.2.2.2 Training Mode, providing a baseline for comparison during tuning.
2. **Trial Execution:** The trial runs mirror the same process as the training mode, starting with the initialization of the `'model=stable_baselines3.PPO()'` model and training via the `'model.learn()'` method.
3. **Evaluating Performance:** Each trial's performance is evaluated using `'stable_baselines3.common.evaluation.evaluate_policy()'` function, which assesses the agent's decision-making abilities across several episodes.

By analyzing the trial results, the optimal hyperparameter configuration was identified, significantly improving the agent's learning efficiency and policy robustness.

This tuning process proved crucial in enabling the agent to adapt to complex and dynamic environments, ultimately enhancing its tactical decision-making capabilities.

In the following sections, the training process is demonstrated by training the RL agent in a simulation environment to develop a foundational understanding of navigation

strategies. The objective is to enable the agent to navigate efficiently while avoiding obstacles based on learned policies. This step is crucial to validate the effectiveness of the reward function and model setup, ensuring that the agent can successfully complete the designated navigation tasks in a controlled environment.

4.2.3 Training a RL Agent

The reinforcement learning (RL) agent is trained in a structured environment to learn optimal navigation strategies and develop a robust policy. The primary goal of this training process is to enable the agent to navigate efficiently while avoiding obstacles based on learned behaviors. Additionally, the setup is used to evaluate the effectiveness of the reward function and core logic components, ensuring that the agent is guided toward the desired policy. The controlled environment provides a stable setting for refining decision-making capabilities and optimizing key parameters before real-world deployment. The Dell G15 5515, featuring an AMD Ryzen 5 5600H CPU, 16GB RAM, and an NVIDIA GeForce RTX 3050 GPU, is used for this training.

Environment Configuration:

To create the training scenario, specific changes and configurations were made:

- **Observation Spaces:** Out of the various observation spaces mentioned in Section 3.3.4.1, only a few were selected for simplicity: complete 270° Laser Reads, Agent Polar Coordinates, and obstacle class (Person, Dustbin).
- **Initial Position:** The initial position of the GrassHopper robot was fixed at coordinates $(-19.0, 52.5, 0.1)$ in the environment.
- **Global Path:** The global path was shortened to consist of only five waypoints, spanning approximately 20 *meters* in length from the initial position of the robot.
- **Obstacles:** The environment contained the following obstacles: One static person, one dustbin, and one dynamic person walking in the opposite direction of the robot but offset from the path.

The training setup is illustrated in Figure 4.4.

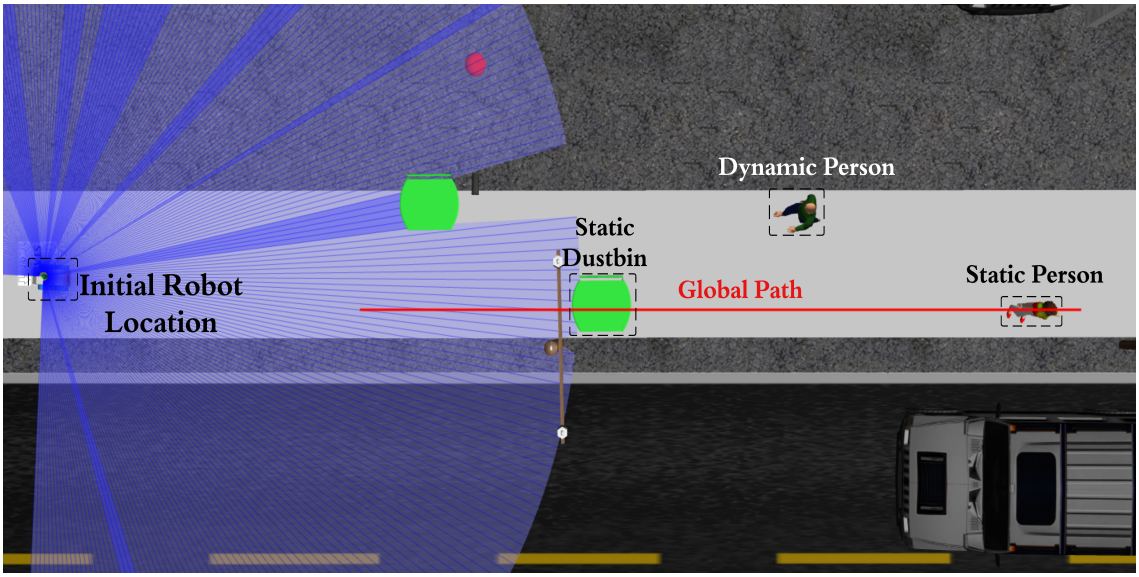

Figure 4.4: Training Environment.

In this scenario, the robot's initial location and global path remained fixed for each episode, providing consistency for training. As mentioned in Section 4.2.2.4, hyperparameter tuning plays a critical role in achieving efficient and desirable results. The hyperparameters chosen using Optuna for optimization are as follows:

1. `learning_rate` : 0.00525469
    - The learning rate (α) determines the step size for updating parameters during training, as shown in Equation 2.3.2.2. A well-chosen learning rate is essential to ensure steady progress toward an optimal policy without stagnation or excessively slow learning.
    - The selected value allows for faster convergence, enabling the agent to adapt quickly to environmental changes, improving training efficiency, especially in complex navigation scenarios.
2. `n_steps` : 13796
    - This parameter defines how many steps the agent collects before performing an update. A higher value helps in stabilizing learning process.
    - The increased value ensures longer trajectory consideration, which is particularly beneficial for navigating along path, allowing the agent to make more informed decisions over the distance.
3. `gamma` : 0.990337159
    - The discount factor (γ) determines how much future rewards influence the current decision. A value closer to 1 favors long-term planning, as explained in Section 2.3.1.
    - The slightly higher value helps the agent prioritize future rewards, improving overall path efficiency and enhancing the ability to avoid obstacles effectively, which is critical in complex environments.
4. `clip_range` : 0.1482
    - PPO uses clipping, as detailed in Section 2.3.5.2, to prevent excessively large policy updates that could destabilize training. A higher clipping range allows for more flexible updates, enabling the agent to explore a wider range of actions while maintaining stability.
    - The selected value facilitates faster adaptation in complex environments, allowing the model to learn more efficiently by updating the policy without risking drastic changes that might hinder progress.
5. `gae_lambda` : 0.940049564
    - The `gae_lambda` parameter controls the trade-off between bias and variance in advantage estimation, as discussed in Section 2.3.2.3 and Section 2.3.5.2. A value closer to 1 uses a longer trajectory, reducing variance but increasing bias, while a value closer to 0 does the opposite, reducing bias but increasing variance.
    - The selected value (0.94) ensures a balance between short-term accuracy and long-term stability, which is crucial for tasks like robotic navigation where both immediate rewards (such as avoiding obstacles) and future rewards (reaching the goal efficiently) are important.

6. `ent_coef` : 3.94106887147e-08
    - The entropy coefficient controls the amount of exploration in the agent's policy. A higher value promotes more exploration by encouraging randomness in action selection, while a lower value favors exploitation of learned policy.
    - The selected near-zero value ensures stability in the learning process, preventing unnecessary deviations from optimal behavior while still allowing the agent to maintain consistency in its decision-making, especially in well-learned tasks like obstacle navigation.
7. `vf_coef` : 0.56014692639
    - The value function coefficient determines the relative importance of the value function in the total loss calculation, which affects the accuracy of state-value estimations, as explained in Section 2.3.2.1.
    - The slightly higher value improves the agent's ability to assess long-term outcomes, enhancing its ability to make informed decisions about future states, which is crucial for tasks like obstacle avoidance and path optimization.

Before starting the training process, the maximum number of steps per episode was set to 1000. Given that the agent can travel a distance of up to 0.2 *meters* per step and initially has a 20 *meters* global path, the robot could theoretically complete the task in a minimum of 100 steps. Therefore, setting the episode limit to 1000 steps provided ample opportunity for successful task completion while allowing room for learning and recovery from mistakes.

The `'total_timesteps'` parameter in `'model.learn()'`, as mentioned in Section 4.2.2.2, defines the total number of timesteps the RL agent uses to learn decision-making. For this task, the value was set to 2,000,000, which took approximately 4 hours to achieve the necessary results, culminating in a 99% success rate in the raining environment.

Training Outcomes:

The training outcomes are illustrated in the following plots:

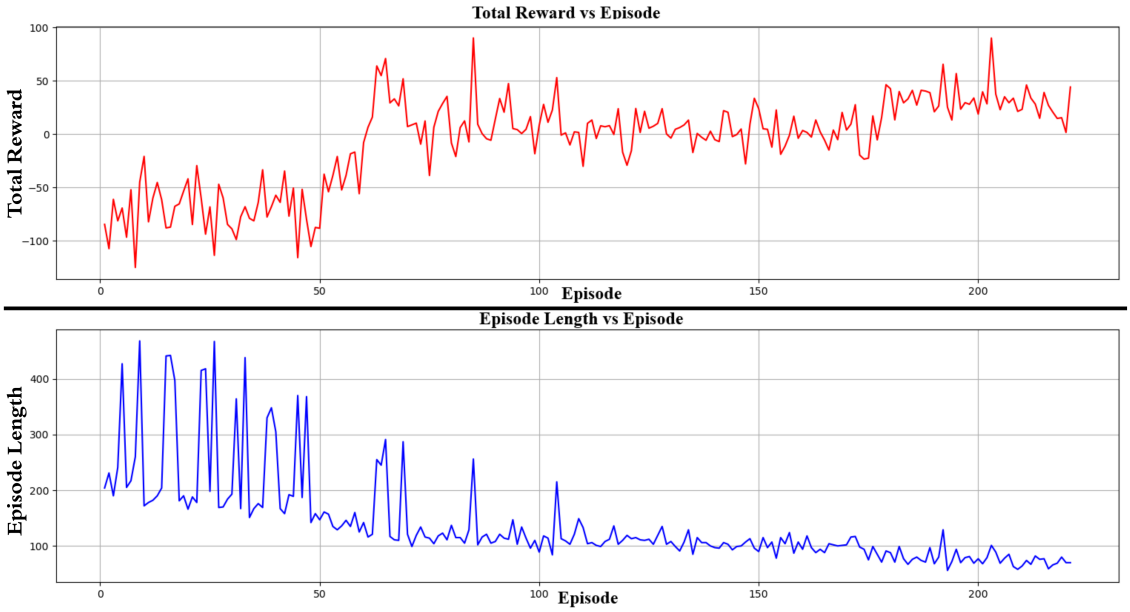

Figure 4.5: Total Reward per Episode (Red Plot), Episode Length vs Episode (Blue Plot), for simplified task.

In the early stages of training (episodes 0–60), the agent exhibits poor performance, as reflected in the negative rewards and long episode lengths. This suggests that the agent is still exploring different actions, often making inefficient and suboptimal decisions, leading to frequent obstacle encounters. The episode length fluctuates significantly, indicating inconsistent behaviour. However, after episode 60, a clear upward trend in total rewards emerges, showing that the agent has started learning effective strategies for navigating obstacles and reaching the goal efficiently. This improvement is accompanied by a steady decrease in episode length, confirming that the agent is taking fewer steps to complete tasks.

Around episode 120, there is a noticeable dip in rewards, but episode length remains relatively stable. This suggests that the agent is still navigating efficiently but may be making more suboptimal decisions when facing challenging obstacle configurations. Such fluctuations are common in RL training, where policy updates sometimes lead to temporary instability before stabilizing again. Despite this drop, the agent quickly recovers, and by episode 170, the reward consistently stays above zero, indicating a significant improvement in decision-making. At this stage, the agent has likely refined its strategy, balancing global path following, local path following, and stopping when necessary to optimize movement.

A sharp spike in reward around episode 210, without a major change in episode length, suggests an instance where the agent encountered favourable conditions or executed an almost perfect navigation strategy. However, since the episode length does not drastically reduce, it implies that the improvement is more in decision-making efficiency rather than speed. By the later episodes, both the total reward and episode length stabilize, showing that the agent has effectively optimized its policy for efficient tactical navigation.

The learning curve suggests that the PPO-based RL training successfully enabled the agent to make tactical navigation decisions. The early struggles, mid-training instability, and final stabilization all indicate a typical RL training trajectory. The agent first explored inefficient and suboptimal decisions, then refined its approach, and finally achieved a balance between decision-making and efficiency. The final state shows that the agent is capable of adaptive navigation, making informed decisions while maintaining a consistent speed and efficiency in reaching its goal.

These improvements validate the effectiveness of the chosen reward function and hyperparameter tuning, establishing a robust foundation for deploying the trained model in real-world scenarios.

After successfully training the reinforcement learning (RL) agent in simulation, the next step is to bridge the gap between the virtual and physical environments by transitioning to Hardware-in-the-Loop (HIL) implementation. While simulation-based training allows for safe, efficient, and iterative policy refinement, real-world deployment introduces new challenges that cannot be fully captured in a simulated environment. The primary focus of HIL is not on training the RL agent but on evaluating its effectiveness in real-world conditions by integrating essential hardware components, perception systems, and supportive algorithms. This phase ensures that all the necessary localization, perception, and other supportive functionalities work as expected in HIL.

4.3 Hardware-in-the-Loop Implementation

In this section, we will be checking and verifying all the necessary localization, perception, and supportive functionalities on the real robot to ensure that when the trained

model is implemented on the real **GrassHopper**, it will work as expected. This involves setting up critical perception and localization mechanisms, including GNSS, IMU, and encoder odometry fusion via a dual Extended Kalman Filter (EKF) node. Additionally, dynamic map creation using a 2D LiDAR sensor, object classification, depth and angle estimation, and a pure-pursuit path follower are implemented to assess path-following accuracy and maneuverability. The hardware setup used in this phase is shown in Figure 4.6 below, featuring the GrassHopper robot equipped with a Sick Tim 551 2D-LiDAR, Intel RealSense D435i camera, and SparkFun Ublox ZED-F9P-00B GNSS.

Figure 4.6: GrassHopper Setup for HIL.

4.3.1 Localization Using Dual EKF Node

Accurate localization is a critical aspect of outdoor navigation, especially in dynamic environments where precise positioning is essential for effective path planning and obstacle avoidance. Unlike controlled indoor settings, outdoor terrains introduce several challenges such as uneven surfaces, varying traction, and GNSS signal fluctuations. A mobile robot operating in such an environment must have a reliable localization system that can continuously track its position and orientation with minimal drift. To achieve this, multiple sensor modalities are integrated to compensate for individual sensor limitations and provide a robust estimation of the robot's state.

In this implementation, the GrassHopper robot utilizes an IMU, wheel encoders, and a GNSS (SparkFun Ublox ZED-F9P-00B) for localization. Each sensor contributes valuable data, but relying on a single source can lead to inaccuracies. For instance, encoders estimate displacement by measuring wheel rotations, but due to the robot's four-wheel drive (4WD) configuration, skidding effects can introduce significant errors in odometry calculations, especially on uneven or slippery surfaces. On the other hand, GNSS provides absolute position data in global coordinates, making it invaluable for outdoor navigation. However, GNSS readings can suffer from momentary signal loss due to environmental obstructions like buildings or trees. To mitigate these drawbacks and ensure a consistent and accurate localization output, a dual Extended Kalman Filter (EKF) node is implemented to fuse data from all these sources.

The dual EKF node effectively combines multiple data streams to generate a refined and noise-reduced odometry estimate. It processes x and y velocity data from wheel

encoders; *roll*, *pitch*, *yaw* velocities, and x-axis acceleration from the IMU; and absolute x, y position data from the GNSS. By continuously integrating these inputs, the dual EKF node corrects for errors, filtering out inconsistencies from individual sensors while maintaining smooth and stable localization data. The final fused odometry output provides an accuracy of ± 0.1 to ± 0.1 *meters* in x and y coordinates and $\pm 1^{\circ}$ in *yaw*, ensuring precise positioning for reliable outdoor navigation. Figure 4.10 verifies these accuracy results of the odometry, demonstrating that the GrassHopper achieves good and precise localization.

4.3.2 Dynamic Occupancy Grid in HIL

In outdoor navigation, a dynamic occupancy grid is essential for ensuring that the mobile robot can effectively perceive and adapt to its surroundings. Unlike static maps, which are pre-built and remain unchanged, a dynamic map continuously updates based on real-time sensor data, making it crucial for navigating in environments where obstacles, terrain, and other factors change over time. This capability is particularly important for local path planning, as it allows the robot to make informed decisions by detecting obstacles and free space, ensuring smooth and collision-free local path planning.

To achieve real-time mapping, the Sick Tim 551 2D LiDAR and the fused odometry from the dual EKF node are utilized as explained in Section 4.1.2.1. These data sources provide the necessary spatial and localization information to construct an up-to-date occupancy grid of the environment. The `'dynamic_map'` node is responsible for processing this information and generating a dynamic occupancy grid. Once the LiDAR node and dual EKF node are fully operational on the GrassHopper robot, and the respective topic names are correctly assigned in the `'dynamic_map'` node, the system begins publishing the occupancy grid output on the `/costmap` topic. This real-time costmap serves as a fundamental input for navigation algorithms, enabling obstacle avoidance and efficient path planning. The real-world implementation of this mapping process is depicted in Figure 4.7, where the left side shows the real-world setup, and the right side displays map visualization on Rviz2.

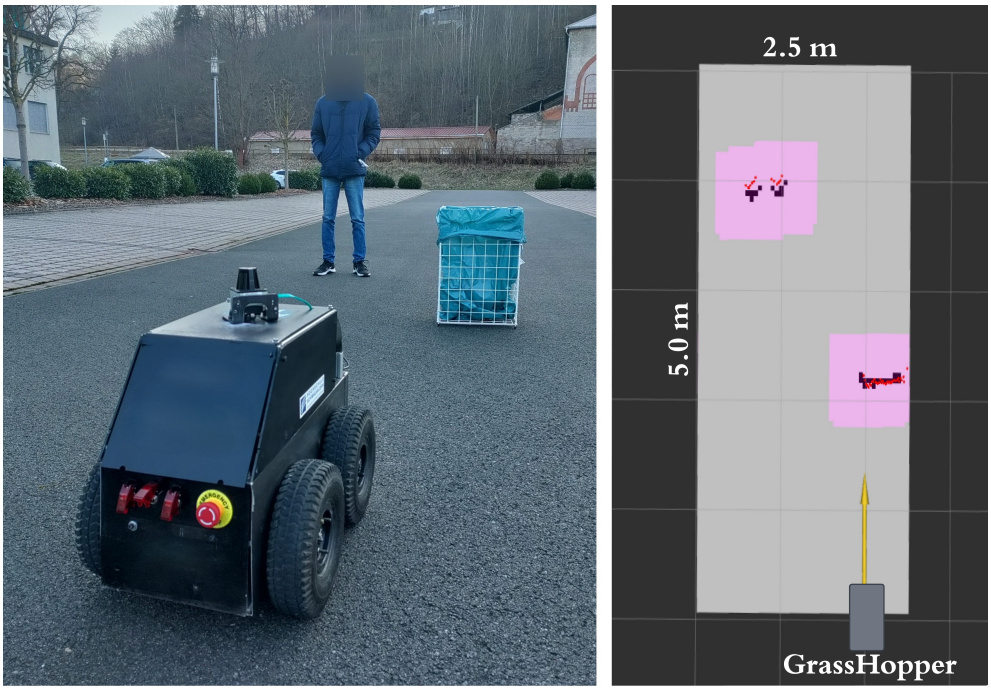

Figure 4.7: Dynamic Occupancy Grid in HIL.

4.3.3 Object Classification, Depth and Angle Estimation in HIL

Accurate object perception is a crucial aspect of outdoor navigation and reinforcement learning (RL)-based decision-making. In real-world environments, the robot must not only detect obstacles but also estimate their depth and angle to navigate safely and efficiently. Object classification plays a key role in understanding the surroundings, while depth and angle estimation provide spatial awareness, enabling better decision-making capabilities. For this task, the **Intel RealSense D435i** depth camera is used in HIL, allowing real-time object detection and distance measurement without relying on external sensor fusion.

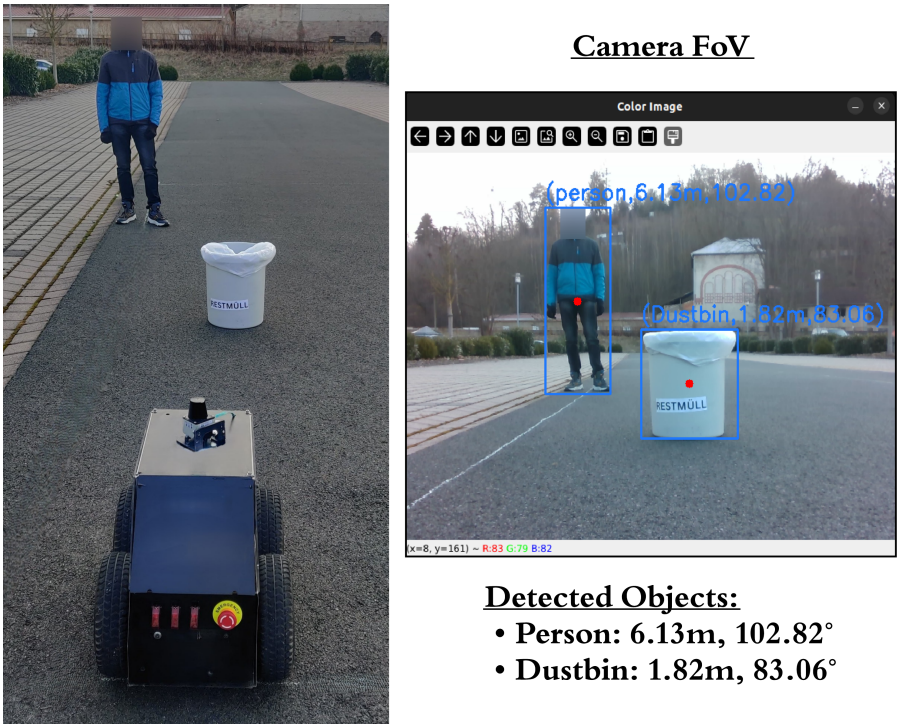

Figure 4.8: Object Classification, Depth and Angel Estimation in HIL.

In the SIL implementation, depth estimation was achieved by fusing data from a 2D LiDAR and a camera, due to limitations in obtaining reliable depth directly from images, as discussed in Section 3.4.2.2. However, in HIL, the Intel RealSense D435i simplifies this process by providing built-in depth sensing, eliminating the need for LiDAR-camera fusion. The `'object_detect'` node is responsible for handling this functionality using `'pyrealsense2.pipeline()'` to interface with the RealSense camera. This node utilizes the **YOLOv5** model for object classification, generating bounding boxes around detected objects in the camera feed.

Once an object is detected, the center pixel coordinates of its bounding box are extracted and used to determine depth using the depth image from the Intel RealSense D435i. The angle calculation is based on the camera's field of view (FOV) and the horizontal pixel distribution, providing accurate spatial positioning of detected objects. All processed information, including object classification, depth, and angle, is published on the ROS2 topic `'/detections_publisher'`, as explained in Section 3.3.2.2. The complete implementation code is available in Appendix A.4, while Figure 4.8 showcases the real-world results of this node running with the HIL camera setup, where the left side shows the real-world setup, and the right side displays the `'cv2.imshow()'` window with object detection.

4.3.4 Pure Pursuit Controller in HIL

In this thesis, the GrassHopper robot follows one of three possible maneuvers: global path follow, local path follow, and full stop. Although the full stop maneuver does not require validation—since setting both linear and angular velocities to zero will inherently halt the robot—the other two maneuvers must be tested to ensure precise and smooth path tracking. To achieve this, the Pure Pursuit Controller is employed for both global and local path following, calculating the appropriate velocity commands needed to follow the desired trajectory. Therefore, in this section, only the Pure Pursuit Controller's functionality is tested and verified in real-world conditions.

Figure 4.9: Complete setup for Pure Pursuit Controller in HIL.

To conduct this validation, a separate Python node is utilized, specifically designed to execute a predefined path using the Pure Pursuit algorithm. This node allows for controlled testing by autonomously guiding the robot along a set trajectory while ensuring smooth and stable motion. Additionally, it facilitates tuning of key controller parameters, such as lookahead distance, linear velocity, and angular velocity, to optimize path tracking. Proper tuning is crucial to avoid abrupt changes in velocity, ensuring that the GrassHopper follows the path smoothly without unnecessary oscillations or deviations. The complete test setup is depicted in Figure 4.9, where the GrassHopper robot is required to complete a rectangular path and return to its initial position.

Figure 4.10 represents the trajectory comparison of the GrassHopper robot while following a predefined path using the Pure Pursuit Controller. The plot consists of three different trajectories: the reference path (red dashed line), the recorded trajectory from the robot's odometry (blue line), and the ground truth measured using the ANavS® Multi-Sensor RTK module (green dots). These trajectories provide insights into the accuracy and precision of the Pure Pursuit Controller.

The reference path represents the ideal trajectory that the robot is intended to follow. The recorded trajectory, derived from the robot's odometry, demonstrates the actual movement of the robot while executing the Pure Pursuit algorithm. Notably, at the corners, the robot tends to stop slightly earlier than expected due to the lookahead distance, which is set to 0.3 m for this test. Additionally, a noticeable notch appears at the corners, which can be attributed to the physical structure of the GrassHopper. Being

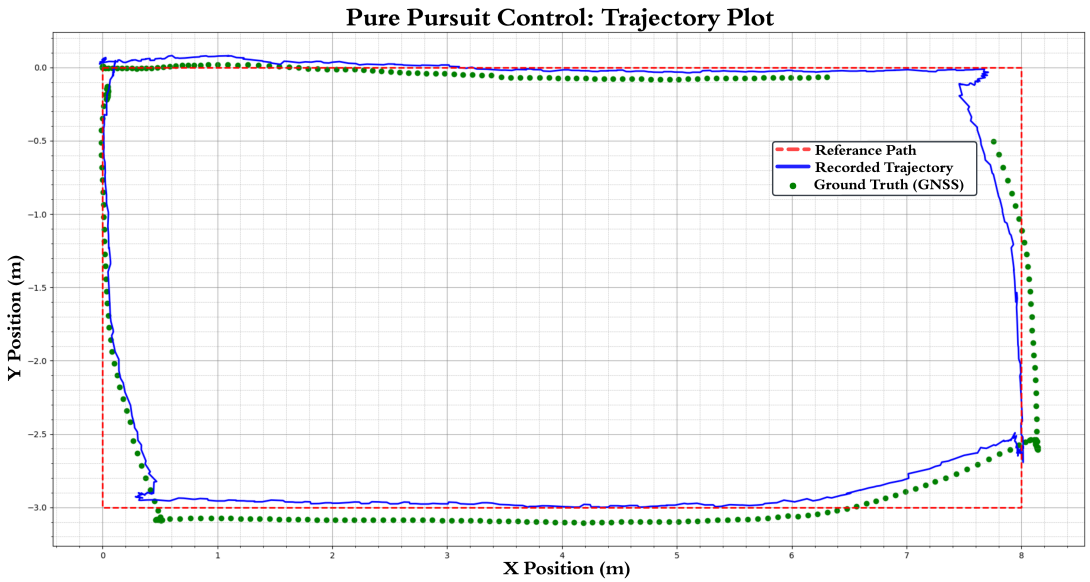

Figure 4.10: Real-World Testing of Pure Pursuit Control: GrassHopper Trajectory Plot.

a rigid 4-wheel drive (4WD) system, the robot experiences skidding while making sharp turns. The friction between the tires and the road surface causes a slight backward motion, which results in a temporary deviation from the path. However, this backward motion does not significantly impact the Pure Pursuit Controller's performance, as the robot successfully corrects its trajectory and ultimately reaches the end target of the predefined path.

The ground truth trajectory, obtained from the ANavS® RTK module, represents the most accurate localization data. At the top-right corner, missing GNSS data points can be observed, which occurred due to a technical issue during data recording. However, the remaining data points confirm two key observations. First, the GrassHopper robot demonstrates precise localization with an accuracy of ± 0.1 m in both x and y coordinates, as verified in Section 4.3.1. Second, the Pure Pursuit Controller follows the reference path with high precision, as indicated by the minimal deviation between the reference and recorded trajectories.

In conclusion, this test validates the effectiveness of the Pure Pursuit Controller in enabling the GrassHopper robot to closely and smoothly follow a predefined trajectory. Despite minor deviations caused by mechanical constraints and skidding at sharp corners, the controller successfully corrects the robot's movement, ensuring accurate path-following. The high localization accuracy and minimal tracking errors further confirm the robustness of the system in real-world conditions.

After successfully testing and verifying all essential components in the HIL setup, including localization, dynamic mapping, path following with the Pure Pursuit Controller, and object classification with depth and angle estimation, the results demonstrate that the system performs reliably in real-world conditions. Each module functioned as expected, providing accurate and stable outputs necessary for the robot's navigation. With these validations in place, it can be concluded that when transferring the trained reinforcement learning model onto the real GrassHopper, the robot will operate as intended, regardless of the specific decisions made by the model. This ensures a seamless transition from simulation to real-world deployment, with all supporting functionalities working effectively to enable smooth and reliable navigation.

Chapter 5

Results

This chapter presents the evaluation results of the trained reinforcement learning model for outdoor navigation in a Software-in-the-Loop (SIL) simulation. The model's performance is assessed based on success rate, failure rate, and truncation instances over multiple test episodes. The experiments provide insights into its ability to navigate efficiently and adapt to varying environmental conditions.

5.1 ROS2 Node for Trained Model Evaluation

Evaluating the trained reinforcement learning model is essential to verify its ability to make effective decisions in dynamic environments. As outlined in Section 4.2.3, the training phase focused on progressively improving the agent's tactical decision-making, allowing it to adapt to different scenarios and choose optimal actions. Now, to assess whether the trained policy enables the agent to make reliable decisions under test conditions, an evaluation is conducted using the `'trained_agent.py'` ROS2 node. This node runs multiple test episodes, monitoring how well the agent selects actions that lead to successful task completion. Instead of relying on a reward system, as in training, evaluation focuses on whether the agent consistently makes the right choices to reach its destination efficiently.

To begin the evaluation, the node registers the custom Gymnasium environment, following a similar process to the one described in Section 4.2.2. However, unlike training, where the model learned by maximizing cumulative rewards, evaluation purely assesses whether the agent executes the correct sequence of decisions to complete the task. Since the goal is to validate decision-making accuracy rather than reward optimization, there is no need to define a reward threshold at this stage. Once the environment is registered, it is initialized using `'env = gymnasium.make('OutDoorEnv-v0')'`, preparing it for structured interaction with the trained model.

One challenge in transitioning from training to evaluation is ensuring that the model operates consistently across different ROS2 distributions and Python versions. Since changes in the software environment can cause issues with deserializing the trained model's parameters, the node addresses this by explicitly defining the action and observation spaces before loading the model:

```
custom_obj = {'action_space': env.action_space,
              'observation_space': env.observation_space}
```

This ensures that the model retains its learned decision-making framework without conflicts, allowing it to operate as intended.

Once the environment is initialized, the trained model is loaded using **Stable Baselines3**, ensuring that the agent applies the exact policy it learned during training. The model is retrieved using:

```
model = stable_baselines3.PPO.load(trained_model_path,
                        env = env, custom_objects = custom_obj)
```

Loading the model in this way ensures that it follows a deterministic approach, executing its learned decision-making strategy without randomness. This consistency is crucial for evaluation, as it allows for an accurate assessment of how well the agent applies its learned policy in different test episodes.

To systematically evaluate the agent's ability to make correct decisions, the node runs multiple test episodes using the `'evaluate_policy'` function from Stable Baselines3:

```
stable_baselines3.common.evaluation.evaluate_policy(model,
                    env=env, n_eval_episodes=n_episodes,
              return_episode_rewards=True, deterministic=True)
```

Throughout the evaluation, the agent continuously makes decisions based on its learned policy while the node monitors the outcomes. The key metrics—success rate, failure rate, and truncated episodes—are recorded to analyze how effectively the agent translates its training into real-time decision-making. At the end of the evaluation, these results are logged in the terminal, providing a summary of the model's performance. This process offers valuable insights into the reliability of the agent's decision-making, ensuring that it can consistently select the best possible actions to reach its objective.

5.2 Performance Evaluation in a Simulation Environment

The `'trained_agent.py'` node described in the previous section is utilized to systematically evaluate the RL model's decision-making capabilities in a controlled simulation environment. The evaluation setup mirrors the training conditions described in Section 4.2.3, featuring a static obstacle (dustbin) and two human agents—one standing still and another moving in the opposite direction of the robot but slightly offset from its intended path. The agent's ability to navigate efficiently while making tactical decisions to avoid obstacles is assessed over 100 test episodes. As outlined earlier, the evaluation prioritizes the model's decision-making quality rather than reward accumulation, measuring performance through success rate, failure rate, and truncation instances. The results of this evaluation are illustrated in Figure 5.1, with the entire testing process taking approximately 5 hours to complete.

The results indicate that the RL agent successfully completed its navigation task in 99 out of 100 episodes, achieving an impressive 99% success rate. Only a single episode resulted in failure (1%), while no episodes were truncated, suggesting that the agent rarely reached a scenario where it could not decide on a viable action within the given constraints. The high success rate highlights the model's robust tactical planning and obstacle avoidance strategies, demonstrating its ability to react effectively to both static and dynamic elements in the environment. Failures, though minimal, likely stemmed from unexpected interactions with the dynamic pedestrian, where the agent may have miscalculated the person's trajectory or taken a poor avoidance maneuver, leading to a collision.

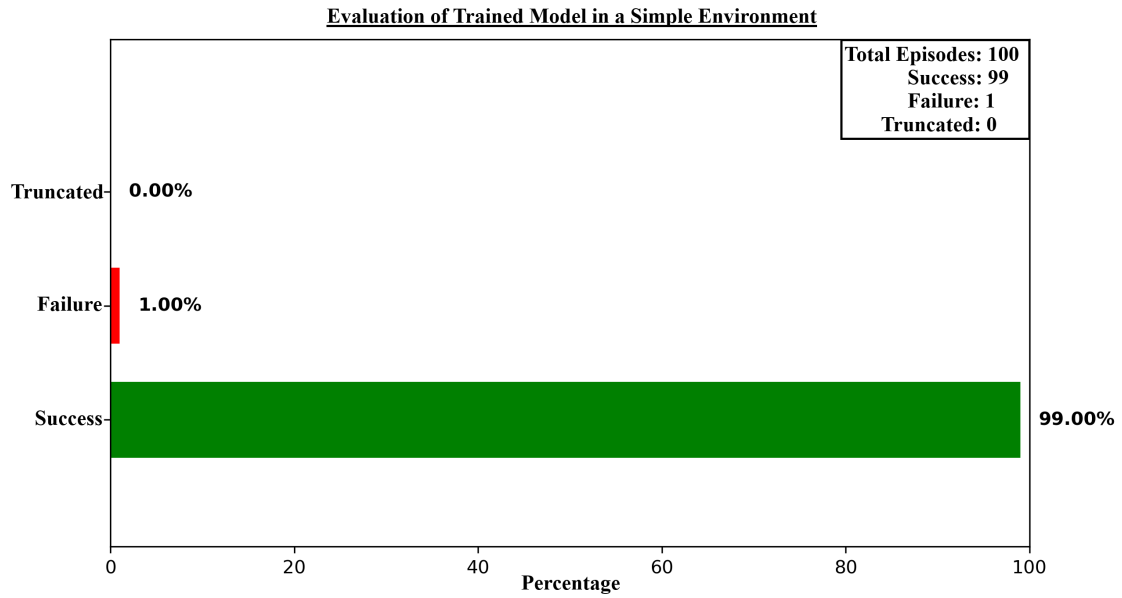

Figure 5.1: Evaluation of Trained Model in a Simple Environment.

Overall, these results validate the model's reliable performance in structured environments, reinforcing its capability to make well-informed decisions under varying conditions. The near-perfect success rate suggests that the learned policy is well-optimized for this specific test scenario, making it a promising candidate for further real-world validation.

5.3 Performance Evaluation on GrassHopper

While the RL model has demonstrated strong decision-making capabilities in the Software-in-the-Loop (SIL) simulation, transitioning it to a Hardware-in-the-Loop (HIL) setup was not feasible within the scope of this thesis. As discussed in Section 4.3, all subsystems of this thesis work are tested, fully operational, and supported by the GrassHopper system. This includes sensor integration, control systems, communication pipelines, and perception, making the robot structurally ready for real-world testing. However, two key factors prevented the direct deployment of the trained RL model onto the physical robot.

The primary challenge was computational power. The onboard Intel NUC12 Pro (described in Section 3.2) lacks a dedicated GPU, making it insufficient for handling the computational load of real-time RL processing. In contrast, as mentioned in Section 4.2.3, the SIL experiments were conducted on a system equipped with an NVIDIA GeForce RTX 3050 GPU, which provided the necessary processing power for smooth execution. The second limiting factor was time constraints. While the robot's odometry system—crucial for real-world localization and navigation—was still under refinement, significant progress was only achieved 15 days prior to the thesis deadline. Given these constraints, full-scale HIL evaluation was not practical. However, since the RL-based tactical planning framework has been validated in simulation with a 99% success rate, as mentioned in Section 5.2, the next logical step is to implement it on the GrassHopper robot once a more powerful onboard computing system with a dedicated GPU is available.

These evaluations confirm that the RL-based tactical planning framework is highly effective in SIL simulations, demonstrating robust decision-making and high success rates. Furthermore, all necessary HIL components, including perception, control, and communication systems, are fully functional, ensuring that real-world implementation is feasible once the computational limitations are addressed. With a dedicated GPU, the GrassHopper robot will be capable of running the RL model in real-time, unlocking its potential for autonomous navigation and decision-making in dynamic outdoor environments.

Chapter 6

Conclusion and Future Scope

This chapter summarizes the key findings of this research, highlighting the effectiveness of the developed reinforcement learning (RL) framework for tactical decision-making in mobile robot navigation. It discusses the strengths and challenges of the approach, highlighting its potential applications in case of RL based tactical planner. Additionally, it explores areas where further improvements can be made, outlining possible directions for future research and real-world implementation.

6.1 Conclusion

The primary objective of this thesis was to develop and evaluate a reinforcement learning-based tactical decision-making framework that enables mobile robots to autonomously navigate outdoor environments. The RL agent was trained to move from a starting position to a defined goal while avoiding obstacles by selecting optimal maneuvers in real time. The research leveraged ROS2, Gazebo, OpenAI Gym, and Stable Baselines3 to create a structured simulation environment where the RL model could be trained and tested effectively.

A key achievement of this work was the successful development and validation of a Proximal Policy Optimization (PPO)-based RL model using Software-in-the-Loop (SIL) simulation. The trained model exhibited a 99% success rate, consistently reaching its target while avoiding obstacles efficiently. This result demonstrates the potential of reinforcement learning as a tactical planner, capable of dynamically adjusting its actions to ensure safe and efficient navigation in various scenarios.

The experimental results highlighted several key advantages of RL in the context of mobile robot navigation:

- **Autonomous Learning and Adaptation:** Unlike traditional navigation approaches that rely on predefined rule sets or heuristics, the RL model continuously improves its decision-making capabilities by interacting with its environment. This allows for better adaptability and robustness in different scenarios.
- **Real-Time Tactical Decision-Making:** The trained RL agent demonstrated the ability to assess its surroundings in real time and make optimal decisions for maneuvering, ensuring smooth and efficient navigation.
- **Beyond Predefined Constraints:** Traditional navigation methods often rely on handcrafted constraints and predefined motion planning algorithms. The RL-based approach allows the agent to explore and optimize its decision-making process beyond human-defined heuristics, leading to better overall performance.

During the Hardware-in-the-Loop (HIL) testing phase, all subsystems, including perception, control, obstacle detection, and localization, were successfully validated and operated as intended. The robot was able to process sensor data, interpret environmental conditions, and execute basic maneuvering commands effectively. However, the deployment of RL-based decision-making in real-world scenarios faced a significant challenge:

- **Hardware Constraints:** The lack of a dedicated GPU on the GrassHopper robot's onboard computer limited the ability to run the RL model in real time. Since reinforcement learning requires substantial computational power for inference, the current hardware setup was insufficient to execute the trained policy efficiently. This highlights the need for upgraded computational resources to enable real-time RL-based decision-making on physical robots.

Overall, this research confirms that reinforcement learning is a viable and effective approach for tactical planning in mobile robot navigation. The results demonstrate that an RL-based agent can successfully learn optimal strategies for navigation and obstacle avoidance. While real-world deployment requires addressing hardware constraints, the developed framework lays the groundwork for future advancements in autonomous robotic systems.

6.2 Future Scope

Although this research demonstrated the feasibility of reinforcement learning (RL) for tactical decision-making in mobile robot navigation, several areas remain open for improvement and expansion. Future work can focus on enhancing the model's complexity, expanding the action space, upgrading sensor perception, optimizing the reward function, and adapting the RL framework for different robotic platforms.

1. **Increasing Complexity in Training and Navigation Scenarios:** The current model was trained on relatively short paths with limited obstacles (dustbin and pedestrian). However, the developed 3D simulation environment (Figure A.1) provides a much larger space for training. Future work could extend the training environment to include longer paths, multiple dynamic obstacles, crossroads, and traffic signal recognition. This would increase the complexity of the decision-making process and make the RL agent more adaptable to real-world navigation challenges. One way to achieve this is by modifying the simulation world to introduce more intersection conditions, dynamic elements like moving vehicles, and varied obstacle types. Additionally, perception algorithms should be improved to handle these new conditions effectively.
2. **Expanding the RL Agent's Action Space and Maneuver Selection:** Currently, the RL agent has only three maneuver options: **Global Path Follow**, **Local Path Follow**, and **Full Stop**. While effective, this limited action space may prevent the robot from executing finer maneuvers in complex situations. Future improvements can involve introducing additional actions such as acceleration, deceleration, lateral shifts, reversing, and adaptive speed control. This expanded maneuver set would allow the agent to navigate more efficiently by selecting from a broader range of actions. Implementing this requires modifications to the RL environment's action space and updating the reward function to accommodate the new maneuvers appropriately.
3. **Sensor Upgrades for Improved Perception and Safety:** The current GrassHopper robot relies on only two sensors: a 2D LiDAR (270° FoV) and a front-facing camera.

This setup leaves a 90° blind spot at the back and limits obstacle detection to the front area. To enhance the robot's situational awareness, future work should incorporate:

- 3D LiDAR to provide full 360° environmental mapping.
- Ultrasonic sensors at the back to detect objects in blind spots.
- Bumper sensors for collision detection, enabling the robot to stop immediately upon impact.

Integrating these additional sensors will significantly improve the robot's perception capabilities, enhancing its ability to navigate safely in cluttered and dynamic environments. This upgrade requires modifying the sensor fusion algorithms to process data from multiple sensors and make more informed decisions.

4. **Improving the Reward Function for More Generalized Learning:** The current reward function is tailored to the existing obstacle configurations and may not perform well when navigation complexity increases. A more generalized and robust reward function is needed to ensure that the RL agent can adapt to a wide range of scenarios while prioritizing both pedestrian safety and goal efficiency. Future work should explore:

   - Adaptive reward scaling, which adjusts penalties and rewards based on obstacle types and proximity.
   - Risk-aware rewards, penalizing unsafe maneuvers that bring the robot too close to obstacles or pedestrians.
   - Hierarchical reward structures, rewarding long-term planning rather than just immediate obstacle avoidance.

   One way to implement this is by training the RL agent in diverse environments with varying difficulty levels while continuously refining the reward function based on real-world constraints.

5. **Expanding the RL Framework to Different Robot Types:** The current RL framework was developed for a wheeled mobile robot navigating outdoor environments. However, RL-based navigation can be extended to various other robotic platforms, including:

   - Legged Robots (Quadrupeds, Humanoids) for traversing rough terrain, climbing stairs, and adapting to uneven surfaces.
   - Aerial Robots (Drones) for 3D navigation, obstacle avoidance in flight, and adaptive route planning.
   - Underwater Robots for autonomous underwater exploration, object retrieval, and marine surveillance.

   Expanding to different platforms would require redefining the state space, action space, and sensor configurations while maintaining the core RL-based decision-making framework.

Enhancing the complexity, action space, perception, reward structure, computational efficiency, and adaptability of the RL framework will significantly improve its real-world viability. By addressing these areas, future research can push the boundaries of autonomous navigation, multi-agent coordination, and real-world deployment of RL-driven robots.

Appendix A

Related Links and Images

A.1 GrassHopper URDF GitHub

The GitHub repository for the GrassHopper URDF model used in the Gazebo simulation with ROS2-Humble in this thesis, is available https://github.com/dhaval-lad/grasshopper_description.git.

A.2 3D Simulation Environment GitHub

The GitHub repository containing the 3D simulation environment, including the `smalltown_final.world` used in this thesis, is available https://github.com/dhaval-lad/grasshopper_gazebo/tree/master/worlds.

A.3 Dynamic Occupancy Grid GitHub

The GitHub repository for creating the dynamic occupancy grid, a key component for the perception layer in this thesis, can be accessed https://github.com/dhaval-lad/Dynamic-Map.git.

A.4 Object Classification and Depth Estimation GitHub

The GitHub repository for object classification and depth estimation, another key component for the preception layer in this thesis, can be accessed https://github.com/dhaval-lad/object_detection.git.

A.5 Thesis Code

The GitHub repository for the complete RL framework developed for this thesis, offering a detailed blueprint of its implementation, can be accessed https://github.com/dhaval-lad/rl_base_tactical_planner.git.

A.6 Important Pictures

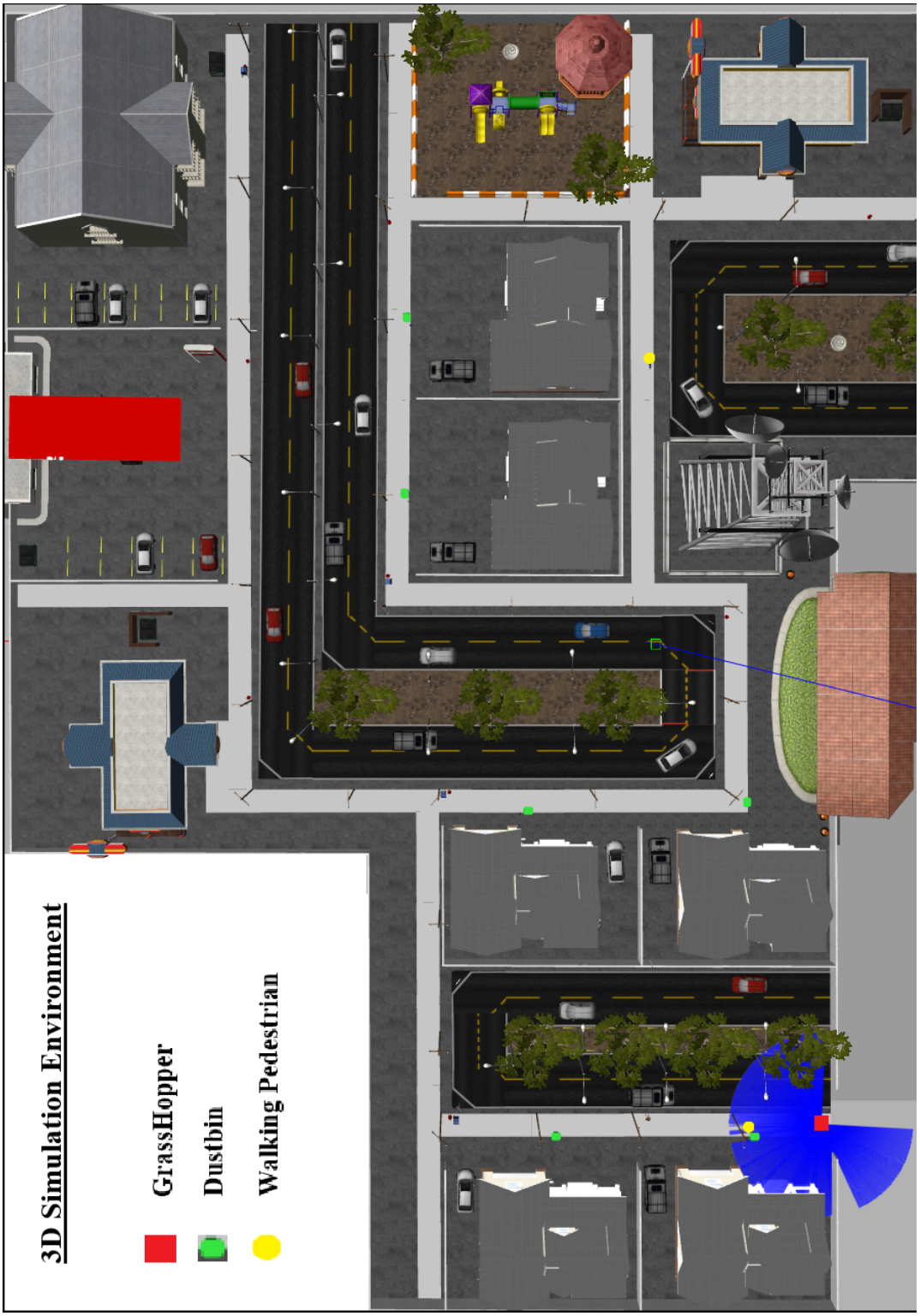

Figure A.1: View of the 3D Environment used in the study.

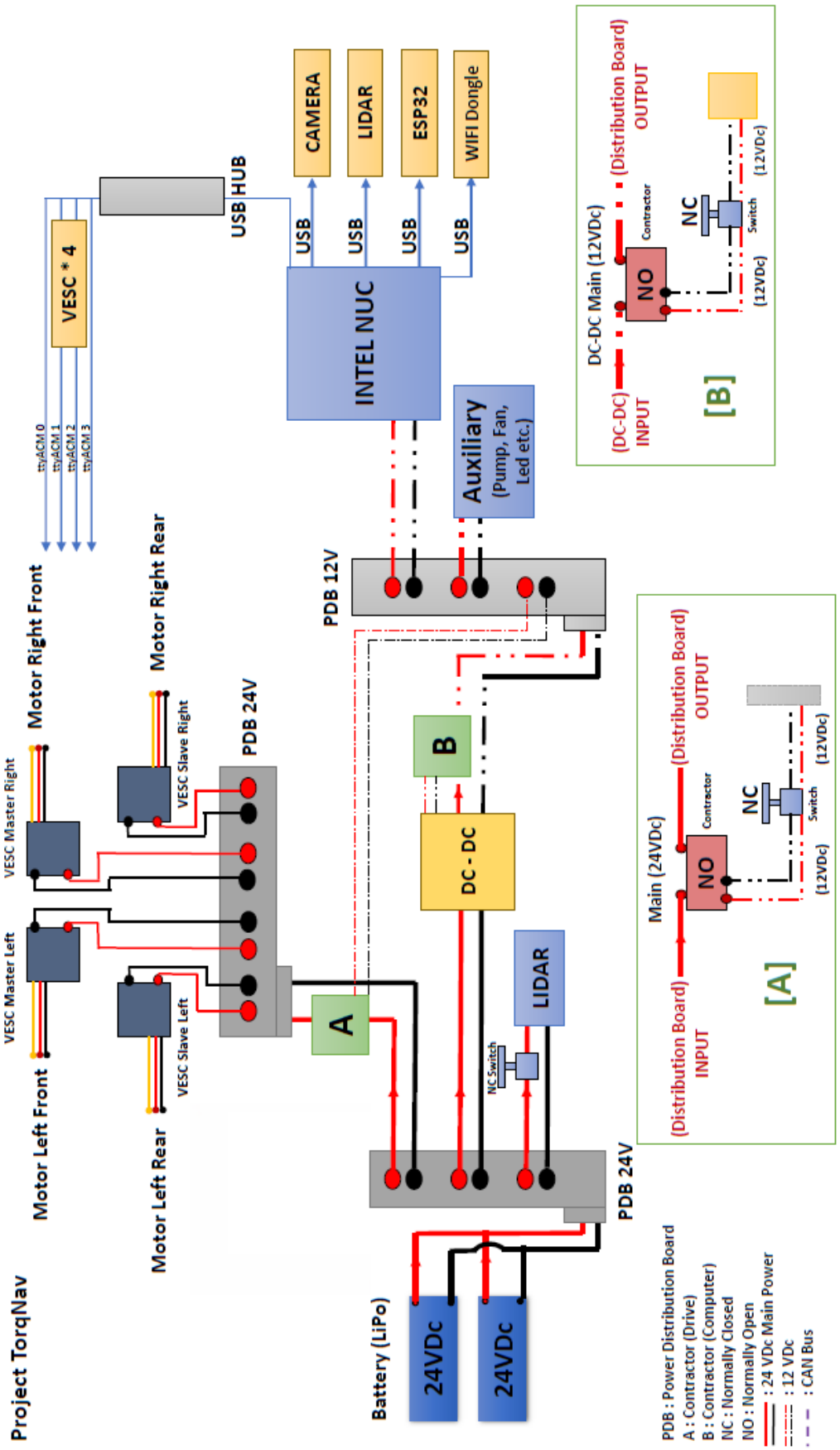

Figure A.2: GrassHopper Electrical Schematic.

Bibliography

[1] Josh Achiam. Part 1: Key Concepts in RL. https://spinningup.openai.com/en/latest/spinningup/rl_intro.html, 2018.

[2] Josh Achiam. Part 2: Kinds of RL Algorithms. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html, 2018.

[3] Josh Achiam. Part 3: Intro to Policy Optimization. https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html, 2018.

[4] Josh Achiam. Proximal Policy Optimization. https://spinningup.openai.com/en/latest/algorithms/ppo.html, 2018.

[5] Lucas G^{a} Alcalde, Nathan Rennolds, and Business Insider España. An autonomous delivery robot can climb stairs, and may be the next big innovation in last-mile delivery. https://www.businessinsider.com/experts-say-an-autonomous-delivery-robot-next-big-thing-2022-2, 2022.

[6] Jafar Alzubi, Anand Nayyar, and Akshi Kumar. Machine learning from theory to algorithms: an overview. In *Journal of physics: conference series*, volume 1142, page 012012. IOP Publishing, 2018.

[7] Shahin Atakishiyev, Mohammad Salameh, Hengshuai Yao, and Randy Goebel. Explainable artificial intelligence for autonomous driving: A comprehensive overview and field guide for future research directions. *IEEE Access*, 2024.

[8] David Česenek. Inertial measurement unit modeling. Master's thesis, Czech Technical Inuvercity in Prague, 2019.

[9] Cesareo Contreras. 10 Delivery Robot Companies to Watch in 2022. The Robotics Applications Conference, 2022.

[10] Raj Dasgupta. Reinforcement Learning: AI Algorithms, Types and Examples. https://www.opit.com/magazine/reinforcement-learning-2/, 2023.

[11] Mirella Santos Pessoa De Melo, José Gomes da Silva Neto, Pedro Jorge Lima Da Silva, João Marcelo Xavier Natario Teixeira, and Veronica Teichrieb. Analysis and comparison of robotics 3d simulators. In *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, pages 242–251. 2019.

[12] Alberto del Rio Ponce, David Jimenez Bermejo, and Javier Serrano Romero. Comparative analysis of a3c and ppo algorithms in reinforcement learning: A survey on general environments. *IEEE Access*, 2024.

[13] DhanushKumar. PPO Algorithm. https://medium.com/@danushidk507/ppo-algorithm, 2024.

[14] Ahmed El-Rabbany. Introduction to gps: the global positioning system. Artech house, 2002.

[15] Arrow Electronics. GPS Receivers. https://www.arrow.com/en/categories/rf-and-microwave/gps/gps-receivers, 2024.

[16] Jonas Eschmann. Reward function design in reinforcement learning. *Reinforcement learning algorithms: Analysis and Applications*, pages 25–33, 2021.

[17] Hajar EL FAKIR. Concept design and implementation of a tactical planning model for mobile logistics solution in unstructured urban environment. Master's thesis, Schmalkalden Univercity of Applied Sciences, 2023.

[18] United Nations Office for Outer Space Affairs. Global Navigation Satellite Systems. https://www.unoosa.org/oosa/de/ourwork/psa/gnss/gnss.html, 2025.

[19] Nitika Garg, Kanakagiri Sujay Ashrith, Gulab Sana Parveen, Kotha Greshwanth Sai, Anish Chintamaneni, and Fatima Hasan. Self-driving car to drive autonomously using image processing and deep learning. *International Journal of Research in Engineering, Science and Management*, 5(1):125–132, 2022.

[20] Majid Ghasemi, Amir Hossein Moosavi, Ibrahim Sorkhoh, Anjali Agrawal, Fadi Alzhouri, and Dariush Ebrahimi. An introduction to reinforcement learning: Fundamental concepts and practical applications. *arXiv preprint arXiv:2408.07712*, 2024.

[21] Innok Robotics Gmbh. HEROS, Modular robot system for custom robots. https://www.innok-robotics.de/en/products/heros, 2024.

[22] Azmi Haider and Hagit Hel-Or. What can we learn from depth camera sensor noise? *Sensors*, 22(14):5448, 2022.

[23] Zhong Hong. Real-World DRL: 5 Essential Reward Functions for Modeling Objectives and Constraints. https://medium.com/@zhonghong9998/real-world-drl-5-essential-reward-functions-for-modeling-objectives-and-constraints, 2024.

[24] Chris Hughes. Understanding PPO: A Game-Changer in AI Decision-Making Explained for RL Newcomers. https://medium.com/@chris.p.hughes10/understanding-ppo-a-game-changer-in-ai-decision-making-explained-for-rl-newcomers, 2024.

[25] Jonathan Hui. RL — Reinforcement Learning Algorithms Comparison. https://jonathan-hui.medium.com/rl-reinforcement-learning-algorithms-comparison, 2021.

[26] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research*, 73:173–208, 2022.

[27] James Jeffs. Mobile Robotics in Logistics, Warehousing and Delivery 2022-2042. https://www.idtechex.com/en/research-report/mobile-robotics-in-logistics-warehousing-and-delivery-2022-2042, 2022.

[28] Adam Daniel Laud. Theory and application of reward shaping in reinforcement learning. University of Illinois at Urbana-Champaign, 2004.

[29] Sentek Solutions Ltd. 2D LiDAR sensors. https://sentekeurope.com/products/2d-lidar-sensors., 2024.

[30] Batta Mahesh et al. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR).*, 9(1):381–386, 2020.

[31] Sumbal Malik, Manzoor Ahmed Khan, Hesham El-Sayed, Jalal Khan, and Obaid Ullah. How do autonomous vehicles decide? 23(1):317, 2022.

[32] Tommaso Van Der Meer. A reinforcement learning approach to path planning for mobile robots. Master's thesis, Univercity of Siena, 2022.

[33] Mohit and Sewak. Actor-critic models and the a3c: The asynchronous advantage actor-critic model. *Deep reinforcement learning: frontiers of artificial intelligence*, pages 141–152, 2019.

[34] Robert Moni. Reinforcement Learning algorithms — an intuitive overview. https://smartlabai.medium.com/reinforcement-learning-algorithms-an-intuitive-overview, 2019.

[35] Faraaz Nadeem. Multi-modal reinforcement learning with videogame audio to learn sonic features. Master's thesis, Massachusetts Institute of Technology, 2020.

[36] D. Nakhaeinia, S. H. Tang, S. B. Mohd Noor, and O. Motlagh. A review of control architectures for autonomous navigation of mobile robots. *International Journal of the Physical Sciences*, 6(2):169–174, 2011.

[37] Margot ME Neggers, Raymond H Cuijpers, Peter AM Ruijten, and Wijnand A IJsselsteijn. The effect of robot speed on comfortable passing distances. *Frontiers in Robotics and AI*, 9:915972, 2022.

[38] Pontus Norman. Training reinforcement learning model with custom openai gym for iiot scenario. *Mid Sweden Univercity*, 2022.

[39] Vong Philavanh. The top sensors used in autonomous delivery robots. https://www.arrow.com/en/research-and-events/articles/the-top-sensors-used-in-autonomous-delivery-robots, 2022.

[40] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[41] rahulsanketpal0431. How to Make a Reward Function in Reinforcement Learning? https://www.geeksforgeeks.org/how-to-make-a-reward-function-in-reinforcement-learning/, 2024.

[42] Atte Rantanen. Robot operating system: overview and case study. Master's thesis, University of Turku, 2024.

[43] Francisco Rubio, Francisco Valero, and Carlos Llopis-Albert. A review of mobile robots: Concepts, methods, theoretical framework, and applications. *International Journal of Advanced Robotic Systems*, 16(2), 2019.

[44] Satya. Depth Estimation From Stereo Images Using Deep Learning. https://medium.com/@satya15july_11937/depth-estimation-from-stereo-images-using-deep-learning, 2023.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[46] Sharan Srinivas, Surya Ramachandiran, and Suchithra Rajendran. Autonomous robot-driven deliveries: A review of recent developments and future directions. *Transportation research part E: logistics and transportation review*, 165:102834, 2022.

[47] Zach Strout. How Does an IMU Work? https://www.sagemotion.com/blog/how-does-an-imu-work, 2022.

[48] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. chapter 3.7 Value Functions. MIT Press Cambridge, 1998.

[49] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. chapter 3.8 Optimal Value Functions. MIT Press Cambridge, 1998.

[50] Synopsys. What is LiDAR? https://www.synopsys.com/glossary/what-is-lidar.html, 2024.

[51] Ujwal Tewari. Which reinforcement learning-rl algorithm to use where, when and in what scenario. https://medium.datadriveninvestor.com/which-reinforcement-learning-rl-algorithm-to-use-where-when-and-in-what-scenario, 2020.

[52] Turing. Reinforcement Learning: What It Is, Algorithms, Types and Examples. https://www.turing.com/kb/reinforcement-learning-algorithms-types-examples, 2024.

[53] Rick Voßwinkel, Maximilian Gerwien, Alexander Jungmann, and Frank Schrödel. Intelligent decision-making and motion planning for automated vehicles. In *Autonomous Driving and Advanced Driver-Assistance Systems (ADAS)*, pages 3–36. CRC Press, 2021.

[54] Katrina Wakefield. A guide to the types of machine learning algorithms and their applications. *SAS Inst. UK*, 2021.

[55] Renbiao Wu, Wenyi Wang, Dan Lu, Lu Wang, and Qiongqiong Jia. Principles of satellite navigation system. In *Adaptive Interference Mitigation in GNSS*, pages 1–29. Springer Singapore, Singapore, 2018.

[56] Fred Zahradnik. What Is Trilateration? How GPS devices use mathematics to determine positioning. https://www.lifewire.com/trilateration-in-gps, 2021.