الجمـــهوريـــة الجـــزائريـــة الديـمقـراطيـــــــة الشـعبيـــــة
THE PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
وزارة التعـلـيـم الـعـالـــي و الـبـحـث الـعـلـمي
THE MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

جـامـعـة عمّـــار ثـلـيـــجـــي بالأغـــواط
AMAR TELIDJI UNIVERSITY OF LAGHOUAT

كـلـيـــــة الـتكنولوجيا
FACULTY OF TECHNOLOGY

قسم الالكترونيك
DEPARTMENT OF ElECTRONIC

Master's dissertation

Domain : Science and Technology
Field : Automatic
Option : and Industrial Automatic Informatic

By :

LARBAOUI ABDELKADER ***BENBEHAZ MOHAMMED***

THEME

Mobile robot control using reinforcement learning

MME. BENKOUIDER	**PR.**	***PRESIDENT***
MME. FEKNOUS SAFIA	***M.C. A***	***EXAMINATOR***
MME . CHOUIREB FATIMA	***PR.***	***SUPERVISOR***

Academic year 2024/2025

Abstract

Mobile robotics is a rapidly evolving field within artificial intelligence, offering promising solutions for autonomous operation in dynamic and unpredictable environments. This thesis investigates the use of deep reinforcement learning techniques to control a differential drive mobile robot, with a particular emphasis on accurate tracking of predefined trajectories. The study combines theoretical foundations in kinematics and localization with advanced learning algorithms such as Q-Learning and Deep Deterministic Policy Gradient (DDPG). The developed control policies are implemented and tested in realistic simulation environments using MATLAB and the Gazebo simulator via the MATLAB-ROS interface. The results demonstrate that reinforcement learning enables efficient robot trajectory tracking, confirming the potential of integrating artificial intelligence with physical modeling to design intelligent and autonomous robotic systems .

الملخص

الروبوتات المتنقلة هي مجال يتطور بسرعة ضمن الذكاء الاصطناعي، وتقدم حلولاً واعدة للتشغيل الذاتي في بيئات ديناميكية وغير متوقعة .تبحث هذه الأطروحة في استخدام تقنيات التعلم العميق المعزز للتحكم في روبوت متنقل بنظام دفع تفاضلي، مع التركيز بشكل خاص على تتبع المسارات المحددة بدقة .تجمع الدراسة بين الأسس النظرية في الحركيات والتحديد المكاني مع خوارزميات التعلم المتقدمة مثل Q-Learning و Deep Deterministic Policy Gradient (DDPG). تم تنفيذ واختبار سياسات التحكم المطورة في بيئات محاكاة واقعية باستخدام MATLAB ومحاكي Gazebo عبر واجهة MATLAB-ROS. تُظهر النتائج أن التعلم المعزز يمكّننا تتبع مسارات الروبوت بكفاءة، مما يؤكد إمكانية دمج الذكاء الاصطناعي مع النمذجة الفيزيائية لتصميم أنظمة روبوتية ذكية ومستقلة.

Résumé

La robotique mobile est un domaine en évolution rapide au sein de l'intelligence artificielle, offrant des solutions prometteuses pour une opération autonome dans des environnements dynamiques et imprévisibles. Cette thèse examine l'utilisation des techniques d'apprentissage par renforcement profond pour contrôler un robot mobile à entraînement différentiel, en mettant particulièrement l'accent sur le suivi précis des trajectoires prédéfinies. L'étude combine des bases théoriques en cinématique et en localisation avec des algorithmes d'apprentissage avancés tels que Q-Learning et Deep Deterministic Policy Gradient (DDPG). Les politiques de contrôle développées sont mises en œuvre et testées dans des environnements de simulation réalistes en utilisant MATLAB et le simulateur Gazebo via l'interface MATLAB-ROS. Les résultats démontrent que l'apprentissage par renforcement permet un suivi efficace des trajectoires des

robots, confirmant le potentiel d'intégrer l'intelligence artificielle avec la modélisation physique pour concevoir des systèmes robotiques intelligents et autonomes.

Acknowledgments

All praise is due to God, by whose grace good deeds are completed. We express our sincere gratitude to Him for granting us health, determination, and patience to successfully complete this project.

We would like to extend our heartfelt thanks and deep appreciation to our esteemed supervisor, *M^{me}*. CHOUIREB Fatima, for her continuous support, valuable advice, and unwavering guidance throughout every stage of this work. Her encouragement and insightful directions were a true source of inspiration. It has been a great honor to work under her supervision.

We are also grateful to all the professors and administrative staff of the Electronics Department at Amar Telidji University of Laghouat for their consistent support and valuable contributions, which provided us with the proper environment to accomplish this work.

Our sincere thanks go as well to the members of the evaluation committee for their efforts in reviewing our project and for the constructive scientific feedback they provided, which helped us improve and refine our work.

Finally, we would like to thank everyone who contributed in any way to the successful completion of this project. May you all be rewarded for your efforts.

Thank you all...

Contents

List of abbreviations

RRT : Probabilistic Roadmap

PRM : Rapidly-exploring random tree

LPA* : Lifelong Planning A star

RHS : Right Hand Side

ROS : Robot Operating System

SLAM : Simultaneous Localization And Mapping

List of figures

List of Tables

General introduction

Mobile robotics has emerged as a highly significant area of research within the broader domains of artificial intelligence (AI) and intelligent systems, driven by its vast potential for developing autonomous agents capable of perceiving, reasoning, and acting in complex, real-world environments. The ability of mobile robots to navigate and adapt to dynamic, unstructured surroundings makes them indispensable for a wide range of applications, including service robotics, autonomous transportation, surveillance, search and rescue, and industrial automation.

This thesis focuses on the application of deep reinforcement learning (DRL) methods for autonomous control of a differential drive mobile robot. Specifically, it investigates how DRL can be leveraged to enable the robot to follow a predefined trajectory with precision. By integrating perception, decision-making, and control into a unified learning framework, the work aims to demonstrate how policy-based learning algorithms can replace traditional rule-based or model-based controllers, offering improved adaptability and performance. Through simulation in MATLAB and MATLAB-ROS interface, the thesis evaluates the effectiveness of DRL algorithms, with attention to training stability, reward shaping, trajectory tracking accuracy, and generalization capabilities.

This work is structured across three main chapters:

Chapter 1 introduces the foundational principles of mobile robotics, focusing on the differential drive robot as the primary model. It covers both continuous and discrete kinematic modeling, along with self-localization techniques such as odometry and LiDAR sensing, which play a crucial role in navigation and obstacle avoidance.

Chapter 2 is dedicated to reinforcement learning algorithms. It begins with Q-Learning as a conceptual entry point, then progresses to Deep Q-Learning using deep neural networks. The chapter further explores the Deep Deterministic Policy Gradient (DDPG) algorithm, which is well-suited for continuous action spaces, supported by simulation examples in MATLAB.

Chapter 3 focuses on the practical implementation, where the DDPG algorithm is applied to autonomously control the robot in both MATLAB and the Gazebo simulator. The system's performance is evaluated through simulations in MATLAB and realistic navigation tasks using the MATLAB-ROS interface, demonstrating its ability to learn, adapt, and effectively track predefined trajectories. This chapter highlights the effectiveness of artificial intelligence,

particularly deep reinforcement learning, in enabling robust and intelligent control of mobile robots, making them capable of operating in unknown and dynamic environments.

.

Chapter *I*

Differential drive mobile robot

1.1 Introduction

This chapter introduces the differential drive mobile robot, one of the most common and fundamental models in mobile robotics. These robots are widely used in academic research, industrial applications, and educational settings due to their simple mechanical structure and ease of control.

The chapter begins with general information about mobile robots, providing a foundational understanding of their types, components, and applications. It then focuses on the kinematic modeling of differential drive robots, covering both continuous and discrete models to describe their motion. Additionally, the chapter discusses essential localization techniques such as odometry, which estimates the robot's position using wheel encoders, and Lidar sensors, which provide environmental perception for navigation and mapping.

By the end of this chapter, readers will have a solid understanding of how differential drive robots move, how to model their behavior mathematically, and how to use sensor data for more accurate motion tracking [1].

1.2 Definition of Mobile Robots

Mobile robots are robotic systems capable of moving through their environment without direct human control. They use technologies such as sensors, localization systems, artificial intelligence, and sometimes pre-programmed maps to navigate. These robots can operate in various environments, including indoor settings like warehouses and factories, or outdoor areas like delivery zones and exploration sites [2].

1.3 Types of mobile rbots

1.3.1 Wheeled robots

Wheeled robots are mobile robots that move using wheels. They are one of the most common types of mobile robots due to their mechanical simplicity, high speed, and energy efficiency on flat surfaces. These robots typically have two or more wheels and can use different wheel configurations such as differential drive, tricycle, or omnidirectional wheels as shown in figure (1.1). Wheeled robots are widely used in indoor navigation, delivery systems, warehouse automation, and research projects.

As an example, a differential-drive robot has two powered wheels and one or more passive caster wheels. It is a classic design used in many mobile robotics platforms [3]. The different moving possibilities of the powerd wheels of this robot are shown in figure 1.2.

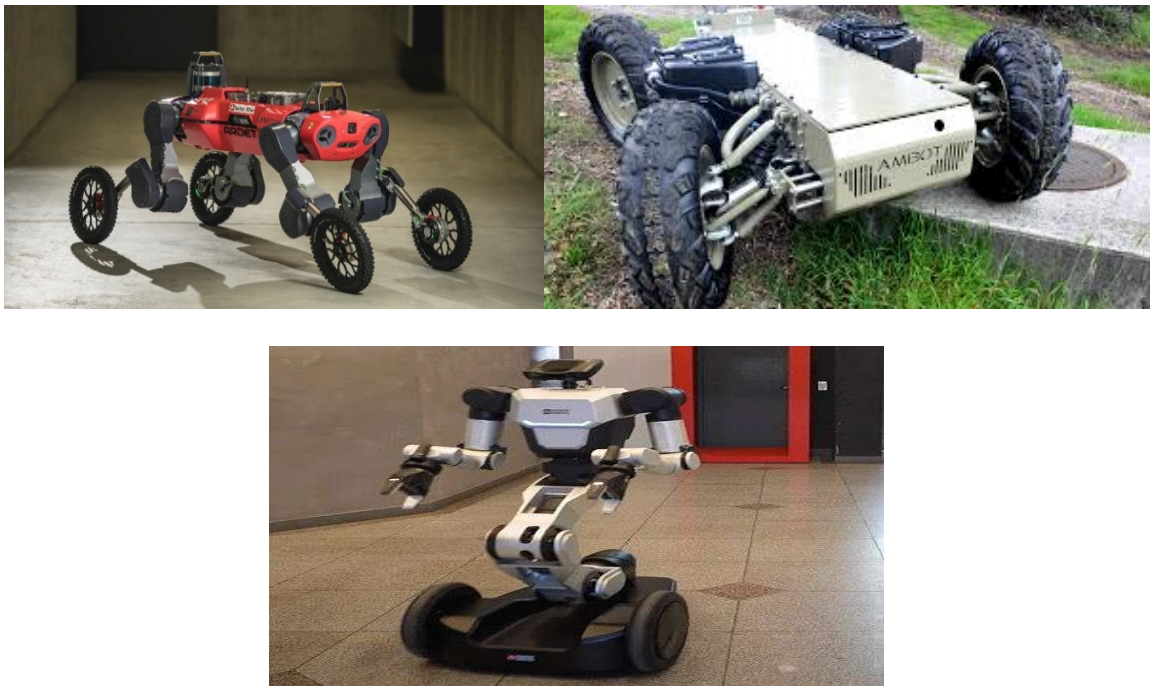

Figure 1.1 Examples of Wheeled robots

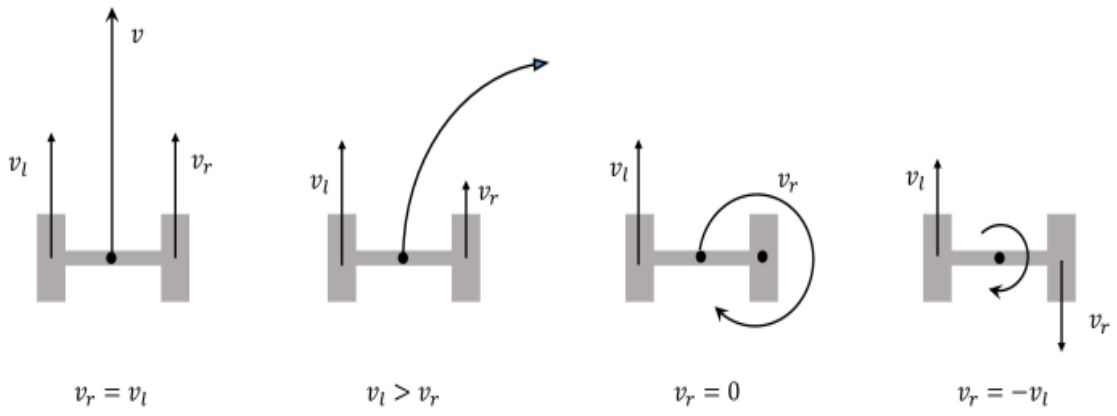

Figure 1.2 The different moving possibilities for unicycle mobile robot [10]

1.3.2 Legged Robots

Legged robots are a type of mobile robots that uses legs instead of wheels for movement (see figure(1.3). They are inspired by biological creatures and are particularly useful in navigating uneven, rough, or cluttered environments where wheeled robots cannot operate efficiently [4]. These robots require complex control systems to maintain balance and coordination, especially when walking or climbing.

Some examples include:

- **Boston Dynamics' Spot:** A highly agile quadruped robot used in construction, inspection, and surveillance.
- **MIT Mini Cheetah:** A small, fast quadruped capable of running, backflipping, and navigating obstacles.
- **c/Honda ASIMO:** A humanoid bipedal robot designed for walking, running, and interacting with humans.

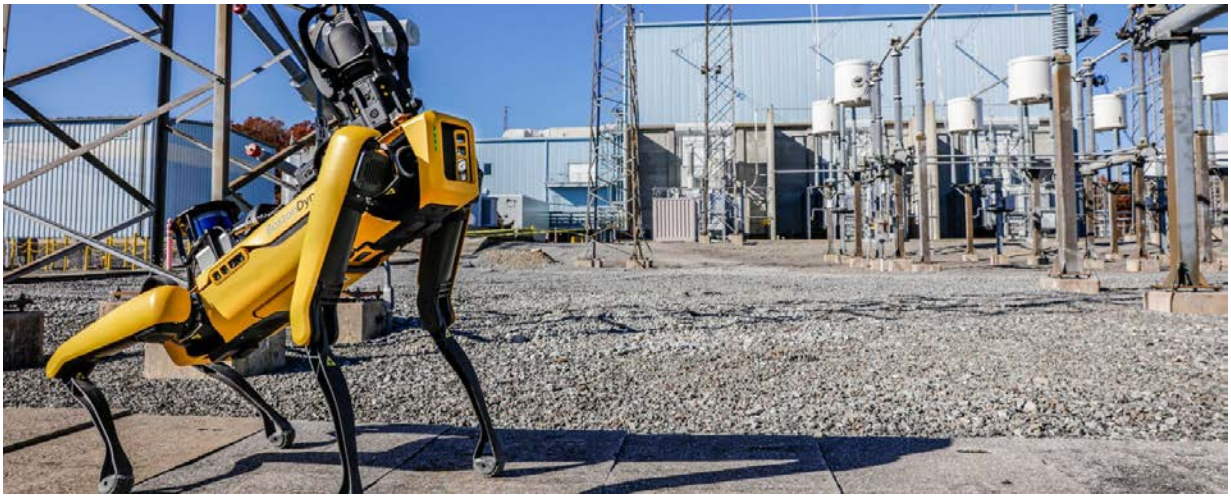

Figure1.3 Examples of legged robots

1.3.3 Snake Robots

Snake robots, also known as serpentine robots, are mobile robots that mimic the movement of biological snakes figure(1.4). They are composed of multiple segments connected by joints, allowing them to bend, twist, and slither through narrow or cluttered spaces [6]. These robots are especially useful in confined or complex environments where traditional robots cannot reach. Some examples include:

- **CMU's Modular Snake Robot (Carnegie Mellon University)：** Designed for search and rescue missions, capable of climbing poles and navigating collapsed structures.
- **HiBot ACM-R5H:** A Japanese-designed snake robot used for inspection in industrial pipes and hazardous areas.
- **Medrobotics Flex® System：** A medical snake-like robot used for minimally invasive surgery in hard-to-reach areas of the body.

Figure 1.1 Example of Snake Robots

1.3.4 Car-type robots

Car-type robots are mobile robots designed with a wheeled structure similar to that of a car as shown in figure (1.5). They typically use two or four wheels for movement and steering, allowing them to navigate environments efficiently [7]. These robots are often equipped with sensors and processors to enable autonomous or remote-controlled operation.

One example of a car-type robot is the Autonomous Delivery Robot developed by companies like Starship Technologies. These robots use a four- or six-wheel design to navigate sidewalks and urban areas to deliver food, packages, or groceries. They are equipped with GPS, cameras, and ultrasonic sensors to detect obstacles and plan routes safely.

Figure 1.2 Example of Car-type robots

1.3.5 Aerial Robots (Flying Robots)

Aerial robots are mobile robots that can fly using propellers, wings, or jet propulsion. They are commonly referred to as drones (Unmanned Aerial Vehicles - UAVs) and are used in various fields such as surveillance, [8] mapping, delivery, and agriculture**.** For example, DJI Phantom 4 is a widely used aerial robot equipped with GPS, high-resolution cameras, and autonomous flight capabilities. It is used for aerial photography, mapping, and inspection**.**

Figure 1.3 Example of Aerial Robots (Flying Robots)

1.4 Continuous kinematic models

These models describe the motion of bodies or materials that are considered continuous in nature, meaning they do not consist of separate parts but rather form a smooth and uninterrupted structure. These models focus on how motion variables such as position, velocity, and acceleration change across space and time within the material. Instead of analyzing individual points or particles, they look at the material as a whole, using mathematical functions to represent how different parts move and deform in relation to each other [9] . This approach is essential in fields that deal with flexible, fluid, or deformable bodies, where motion occurs in a distributed and continuous manner.

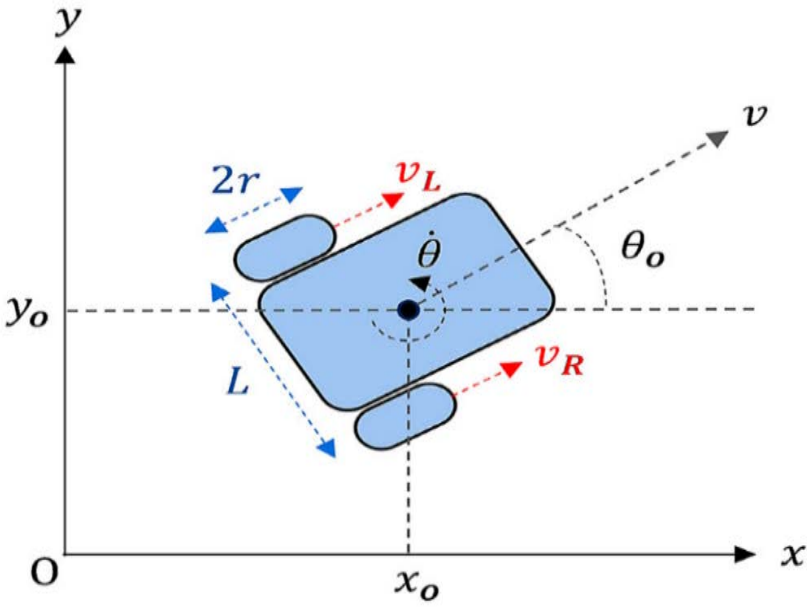

Figure 1.4 Coordinate system of the WMR[10]

A typical mobile robot with two wheels (differential drive) has its state represented by position and orientation [9]:

$$\begin{aligned} \dot{x} &= v(t) \cdot \cos(\theta(t)) \\ \dot{y} &= v(t) \cdot \sin(\theta(t)) \\ \dot{\theta} &= \omega(t) \end{aligned} \tag{1.1}$$

Where $v(t)$ is the linear velocity, $\omega(t)$ is the angular velocity and $\theta(t)$ is the robot heading.

The following is an alternative matrix representation of a differential drive robot model :

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{1.2}$$

with $v = \frac{(v_r+v_l)}{2}$; $\omega = \frac{(v_r-v_l)}{b}$

v_r and v_l are the right and left wheels velocities respectively.

$X = [x \ y \ \theta]^T$ is the state vector and $u = [v \quad \omega]^T$ is the control signal vector. This model is sufficient to describe the nonholonomic constraints of this class of robots [9].

Considering a sampling period T and a sampling instant k, we obtain a discrete-time representation of mobile motion as follow [9]:

$$\begin{cases} x(k+1) = x(k) + v(k)\cos(\theta kT) \\ y(k+1) = y(k) + v(k)\sin(\theta kT) \\ \theta(k+1) = \theta(k) + \omega(k)T \end{cases} \quad \textbf{(1.3)}$$

Where: $x(k), y(k)$ represent robots position at step k, $\theta(k)$ is the robtos orientation at instant k.

1.5 Posture Error

In this work we will be interested in calculating the error between the robot's position and the target position in the path tracking problem of a mobile robot, as shown in Figure (1.8), using the basic reference system [10].

As illustrated in figure 1.8, a reference robot is considered. It is defined with the reference state vector $X_r = [x_r \ y_r \ \theta_r]^T$ and reference control vector $u_r = [v_r \quad \omega_r]^T$. The reference robot has the same model as (1.2). So, its kinematic model can be expressed as :

$$\dot{\mathbf{x}}_r = \begin{bmatrix} \dot{x}_r \\ \dot{y}_r \\ \dot{\theta}_r \end{bmatrix} = \begin{bmatrix} v_r\cos\theta_r \\ v_r\sin\theta_r \\ \omega_r \end{bmatrix} = \begin{bmatrix} \cos\theta_r & 0 \\ \sin\theta_r & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_r \\ \omega_r \end{bmatrix} \quad \textbf{(1.4)}$$

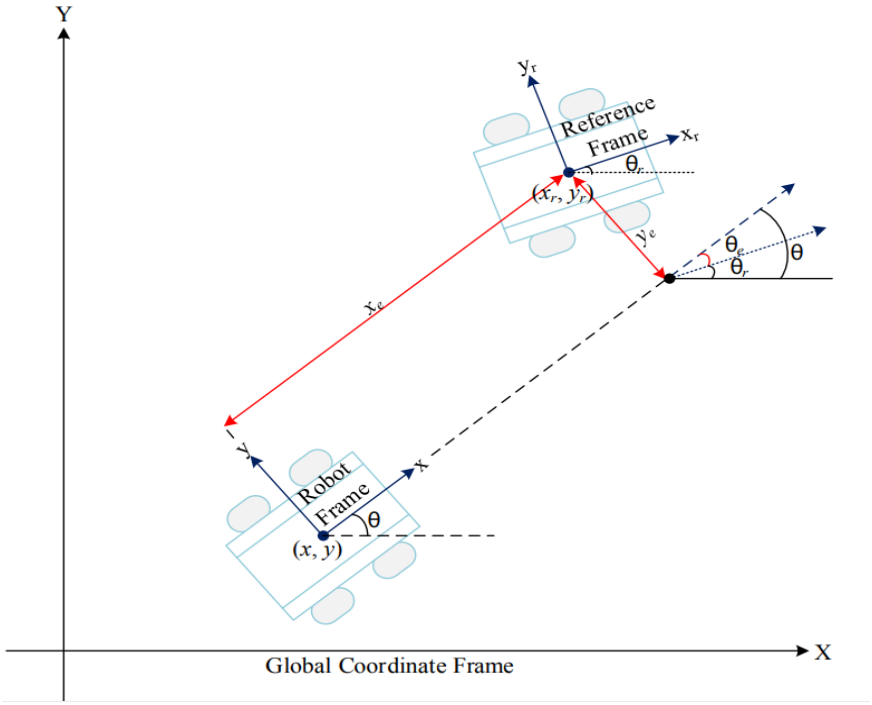

Figure 1.5 The mobile robot tracking error

The posture error is computed using the above graphic as a guide:

$$qr - q = \begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix} = \begin{bmatrix} x_r - x \\ y_r - y \\ \theta_r - \theta \end{bmatrix} \tag{1.5}$$

After implementing a coordinate change, we obtain:

$$\begin{bmatrix} x_e \\ y_e \\ \theta_e \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x_r - x \\ y_r - y \\ \theta_r - \theta \end{bmatrix} \tag{1.6}$$

$$\begin{aligned} x_e &= e_x \cos(\theta) + e_y \sin(\theta) \\ y_e &= e_y \cos(\theta) - e_x \sin(\theta) \\ \theta_e &= e_\theta \end{aligned} \tag{1.7}$$

After derivation of the system of equations (1,6) and using equation (1,4) as well as the constraint of non-holonomy $\dot{x}\sin\theta_r = \dot{y}\cos\theta_r$, we obtain the dynamics of the error as follows:

$$\begin{bmatrix} \dot{x}_r \\ \dot{y}_r \\ \dot{\theta}_r \end{bmatrix} = \begin{bmatrix} \omega y_e - v + v_r \cos\theta_e \\ -\omega x_e + v_r \sin\theta_e \\ \omega_r - \omega \end{bmatrix} \tag{1.8}$$

1.6 Odometry in Mobile Robots

Odometry is a fundamental method used in mobile robotics to estimate the robot's position and orientation over time. It relies primarily on wheel encoders to measure wheel rotations and calculate the displacement and heading of the robot. This technique is especially common in differential drive robots due to its simplicity and fast computational requirements [11].

1.6.1 Principle of Operation

Odometry works by tracking how much each wheel has rotated. Given the radius of the wheels and the distance between them (wheelbase), the linear and angular displacement of the robot can be computed.

If both wheels rotate at the same speed, the robot moves straight. If one wheel rotates faster than the other, the robot follows a curved path (see figure 1,9).

1.6.2 Basic Mathematical Model

Let r the radius of the wheel, L the distance between the wheels (wheelbase) and $\Delta_{\varphi L}$, $\Delta_{\varphi r}$ angular displacements of the left and right wheels then:

The linear displacement is given by:

$$\Delta_s = \frac{r}{2}(\Delta_{\varphi L} + \Delta_{\varphi r})$$

The change in orientation:

$$\Delta_\theta = \frac{r}{L}(\Delta_{\varphi L} - \Delta_{\varphi r})$$

These equations form the basis of differential drive kinematics [12].

1.6.3 Sources of Error and Limitations

Despite its ease of implementation, odometry suffers from several sources of error, such as:

- Wheel slippage and uneven terrain
- Inaccurate wheel encoder measurements
- Mechanical imperfections in the drive system

These errors accumulate over time, leading to significant deviations from the robot's true position (known as drift) [13].

1.6.4 Enhancing Odometry Accuracy

To improve positioning accuracy,[14] odometry is often fused with data from additional sensors such as:

- **LiDAR:** for obstacle detection and environment mapping
- **GPS:** for outdoor localization
- **IMUs or cameras:** for detecting motion and orientation

Sensor fusion techniques like the Extended Kalman Filter (EKF) or Simultaneous Localization and Mapping (SLAM) are employed to combine these data sources and reduce cumulative error

1.7 Using LiDAR in Differential Drive Mobile Robots

LiDAR (Light Detection and Ranging) is a crucial sensor in mobile robotics for mapping, localization, and autonomous navigation [2].

1.7.1 What is LiDAR?

LiDAR is a sensor that measures distances using laser beams by emitting light pulses and calculating the time it takes for them to reflect back from surrounding objects.

1.7.2 Role of LiDAR in Differential Drive Robots

In differential drive robots (robots with two independently driven wheels), LiDAR is used for:

- **Mapping SLAM** (Simultaneous Localization and Mapping): It helps the robot build a map of the environment while determining its own position.
- **Obstacle avoidance:** Detects and avoids surrounding objects in real-time.
- **Precise localization:** LiDAR data is combined with motion models (e.g., odometry) to estimate the robot's position accurately [14].

1.7.3 Integration with Other Systems

LiDAR data is typically fused with:

- Inertial Measurement Unit (IMU) readings
- Odometry data (wheel encoders)

This integration enhances the robot’s localization and reduces errors due to wheel slippage or sensor noise [15].

1.8 Conclusion

In this chapter, we introduced differential drive mobile robots, highlighting their structure and role in mobile robotics. We explored their continuous and discrete kinematic models, which describe how motion is derived from wheel speeds. We also discussed odometry as a basic localization method and its limitations. Finally, we examined the use of LiDAR as a powerful sensor for mapping, obstacle avoidance, and enhancing localization through sensor fusion. These components together form the core of navigation and autonomy in differential drive robot. To enable the mobile robot to accurately follow a predefined trajectory, avoid obstacles, and reach a target position, recent techniques such as Reinforcement Learning (RL) can be employed. This will be the primary focus of Chapter 2.

Chapter II

Reinforcement Learning

2.1 Introduction

According to Sutton and Barto, Reinforcement learning (RL) is a subfield of machine learning that is dedicated to the training of agents to make sequential decisions by interacting with an environment. reinforcement learning (RL) uses a trial-and-error process in which an agent gains the ability to maximize cumulative rewards by receiving feedback from its actions[25]. In the framework, an agent chooses an action based on the state of the environment, receives a reward signal that directs subsequent behavior, and so on. The agent gradually creates a policy, a plan that connects states to actions, that maximizes long-term results. Managing sparse or delayed rewards, as well as striking a balance between exploitation (using known effective actions) and exploration (trying new actions), are important issues in reinforcement learning (RL) [24].

RL's capacity to manage dynamic, complicated situations has led to advancements in a variety of fields, including autonomous systems, gaming, and robotics. For instance, in tasks ranging from robotic manipulation to chess, algorithms like as Q-learning and deep reinforcement learning (e.g., Deep Q-Networks) have shown human-level performance [25] ; [24]. RL's influence keeps growing as computing power and data accessibility increase, providing answers to issues that need for flexible, real-time decision-making.

In this chapter, we'll talk about the basics of reinforcement learning and how it helps an agent learn by interacting with its environment. We'll explore how Q-Learning works, why Deep Q-Learning is useful for more complex tasks, and briefly look at DDPG for continuous control problems like in robotics.

2.2 Definition

Reinforcement learning (RL) is a branch of machine learning in which an agent learns to make a sequence of decisions by interacting with its environment to maximize cumulative rewards. Unlike supervised learning, RL does not rely on labeled data; instead, the agent explores different states, takes actions, and receives feedback in the form of rewards or penalties. Through this process, the agent develops a policy—a strategy that maps states to actions—with the goal of maximizing long-term rewards [25] . The core objective is to balance exploitation of known high-reward actions with exploration of new actions to discover their potential benefits.

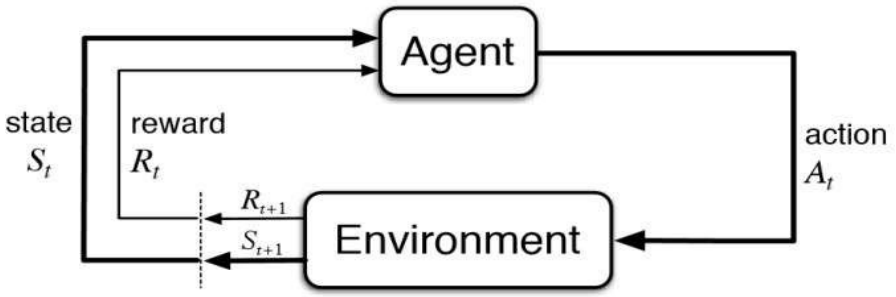

Figure 2.1 Basic Diagram of Reinforcement Learning

How Does Reinforcement Learning Work?

In RL (see figure 2.1), the agent follows the steps below:

1. Start in a state.
2. Take an action.
3. Receive a reward or penalty from the environment.
4. Observe the new state of the environment.
5. Update the policy to maximize future rewards.

Terminologies used in Reinforcement Learning

Here are some terminologies used in RL:

1.**Agent** – is the only decision-maker and learner.

2.**Environment** – a physical world where an agent learns and decides the actions to be performed.

3.**Actions** – a list of action which an agent can perform.

4.**State** – the current situation of the agent in the environment.

5.**Reward** – For each selected action by agent, the environment gives him a reward. It's usually a scalar value and nothing but feedback from the environment.

6.**Policy** – the agent prepares strategy (decision-making) to map situations to actions.

7.**Value Function** – The value of state shows up the reward achieved starting from the state until the policy is executed

8.**Model** – Every RL agent doesn't use a model of its environment. The agent's view maps state-action pairs probability distributions over the state

2.3 Reinforcement Learning Algorithms

There are 3 approaches to implement reinforcement learning algorithms

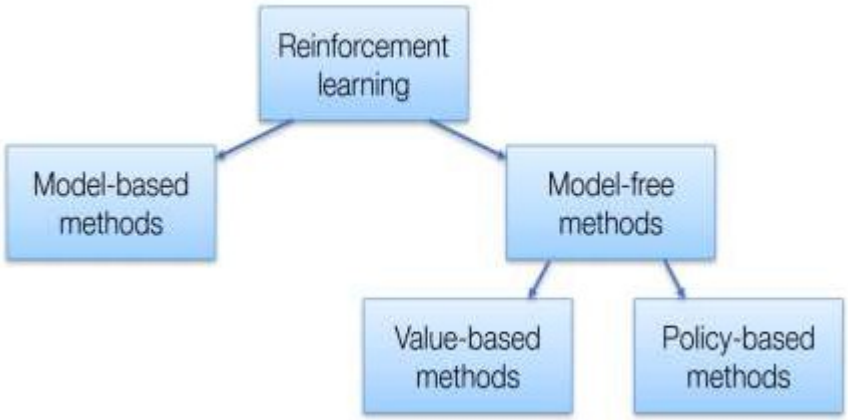

Figure 2.2 Reinforcement Learning Algorithms

Value-Based – The main goal of this method is to maximize a value function. The value- based method trains the value function to learn which state is more valuable and take action.

Policy-based – These methods train the policy directly to learn which action to take in a given state.

Model-Based – In this method, we need to create a virtual model for the agent to help him in learning to perform in each specific environment. The model-based algorithms use transition and reward functions to estimate the optimal policy and create the model. In contrast, model-free algorithms learn the consequences of their actions through the experience without transition and reward function [34] .

2.4 Q-Learning method

What is Q-Learning?

Q-learning is a reinforcement learning algorithm that determines the best action-selection policy for every finite Markov decision process (MDP). Even in situations when the model of the environment is unknown, it assists an agent in gradually learning to maximize the total reward through repeated interactions with the environment.

How Does Q-Learning Work?

1. Learning and Updating Q-values: The algorithm maintains a table of Q-values for each state-action pair. These Q-values represent the expected utility of taking a given action in a given state and following the optimal policy afterward. The Q-values are initialized arbitrarily and are updated iteratively using the experiences gathered by the agent [27].

2. Q-value Update Rule: The Q-values are updated using the formula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \quad \textbf{(2.1)}$$

Where:

- s is the current state.
- a is the action taken.
- r is the reward received after taking action a in state s.
- s' is the new state after action.
- a' is any possible action from the new state s'.
- α is the learning rate ($0 < \alpha \leq 1$).
- γ is the discount factor ($0 \leq \gamma < 1$).

3. Policy Derivation: The policy determines what action to take in each state and can be derived from the Q-values. Typically, the policy chooses the action with the highest Q-value in each state (exploitation), though sometimes a less optimal action is chosen for exploration purposes.

4. Exploration vs. Exploitation: Q-learning manages the trade-off between exploration (choosing random actions to discover new strategies) and exploitation (choosing actions based on accumulated knowledge). Techniques like the epsilon-greedy strategy, where the agent mostly takes the best-known action but occasionally tries a random action, often manage the balance between these.

5. Convergence: Under certain conditions, such as ensuring all state-action pairs are visited an infinite number of times, Q-learning converges to the optimal policy and Q-values that give the maximum expected reward for any state under any conditions [27].

Important Terms in Q-Learning

1. **Q-value (Action-Value):** This represents the value of taking a specific action within a particular state. It estimates the expected future rewards that can be obtained by starting from that state and taking that action followed by following an optimal policy.
2. **State:** This represents the status of the environment at a given time. The agent must recognize and differentiate between states in Q-learning to decide on the best actions.
3. **Action:** Actions are the possible moves or decisions the agent can make in a given state. The choice of action affects the state of the environment.
4. **Reward:** A signal returned by the environment in response to an action taken by the agent. It reflects the value of the transition from one state to another due to an action. Rewards guide the agent to its goal by reinforcing desirable actions.
5. **Policy (π):** The agent's strategy in deciding actions based on the current state. In Q-learning, the policy is often derived from the Q-values, such as choosing the action with the highest Q-value in each state.
6. **Learning Rate (α):** A factor determining how much new information overrides old information. A higher learning rate means the agent learns faster, updating its Q-values more significantly with new rewards and experiences.
7. **Discount Factor (γ):** This factor discounts the value of future rewards compared to immediate rewards. A higher discount factor means that future rewards are more valuable, encouraging long-term beneficial actions over short-term gains.
8. **Episode:** A complete sequence of states, actions, and rewards that ends when a final state is reached. Episodes allow the agent to learn from a full experience from start to finish.
9. **Exploration:** The agent tries different actions to discover their effects and learn about the environment. This is crucial in early learning or dynamic environments where the agent might need to adapt to changes.

10. **Exploitation:** Utilizing the known information to make decisions that yield the highest rewards according to the current policy. This is important for maximizing performance once the agent has adequate knowledge.

11. **Epsilon-Greedy Strategy:** This is a standard method of balancing exploration and exploitation. The agent mostly chooses the best-known action (exploiting) but occasionally chooses a random action (exploring), with the probability of random action selection controlled by a parameter epsilon (ε).

12. **Convergence:** The process by which the Q-values stabilize to the optimal Q-values as the agent continues to learn. This means that further learning will no longer significantly change the values [27].

Example of Q-Learning

In the scenario depicted on the map below (figure 2.3), a knight must rescue the princess who is imprisoned in the castle. One tile at a time can be moved. He will perish if the adversary lands on the same tile as him. His objective is to get to the castle as quickly as he can [27].

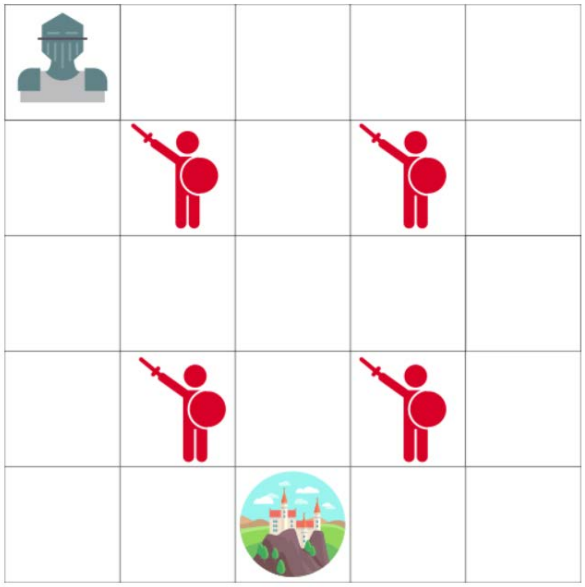

Figure 2.3 Enviroment where the prince trying to save the princess

One way to assess this would be to use a "points scoring" system. Every step he takes costs him one point, which makes the agent faster. He loses -100 points and the episode finishes if he

contacts an adversary. He receives an additional 100 points if he is in the castle when he wins. How to build an agent that can accomplish that is the question.

The question is: how does he create an agent that will be able to do that? Here's a first strategy. Let say the agent tries to go to each tile, and then colors each tile. Green for "safe," and red if not.

After that, we may instruct our agent to only accept green tiles. However, the issue is that it isn't really beneficial. When green tiles are next to one another, we are unsure which tile is superior. So, our agent's search for the castle can lead to an endless loop?

Introducing the Q-table

Here's a second strategy: create a table where we'll calculate the maximum expected future reward, for each action at each state.

Thanks to that, we'll know what's the best action to take for each state. Each state (tile) allows four possible actions. These are moving left, right, up, or down.

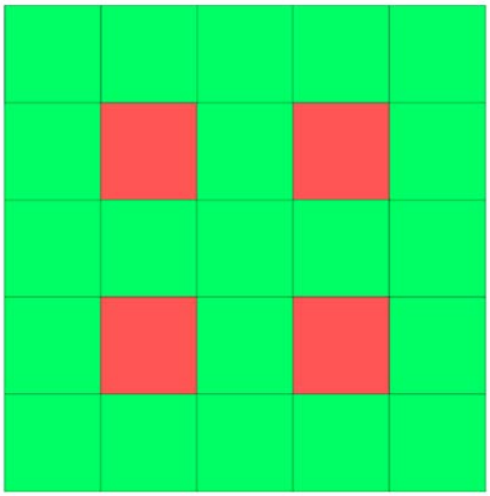

Figure 2.4 The same map, but colored in to show which tiles are safe to visit

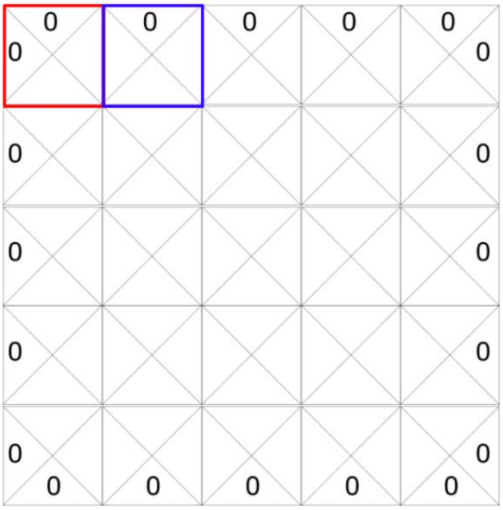

Figure 2.5 Q-Table map

0 are impossible moves (if we're in top left-hand corner we can't go left or up!). In terms of computation, we can transform this grid into a table. This is called a **Q-table** ("Q" for "quality" of the action). The columns will be the four actions (left, right, up, down). The rows will be the states. The value of each cell will be the maximum expected future reward for that given state and action.

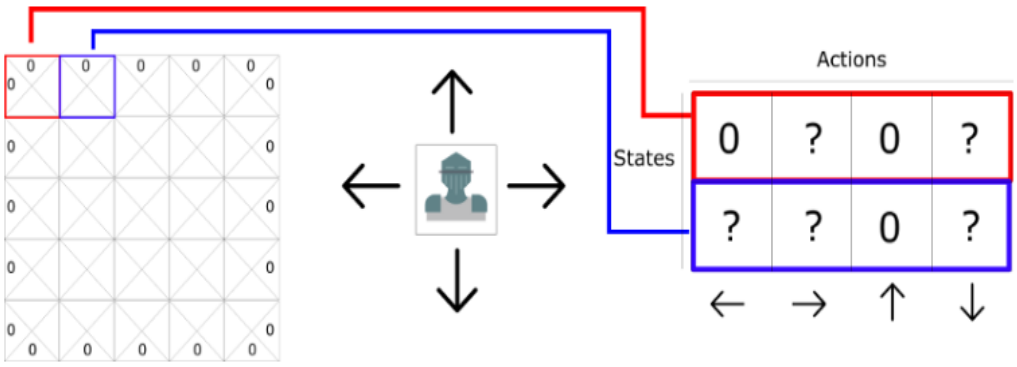

Figure 2.6 How the Q-Table works

Each Q-table score will be the maximum expected future reward that we'll get if we take that action at that state with the best policy given.

Here we don't implement a policy**.** Instead, we just improve our Q-table to always choose the best action. We can think of this Q-table as a game "cheat sheet." Thanks to that, we know for each state (each line in the Q-table) what's the best action to take, by finding the highest score in that line.

Now how do we calculate the values for each element of the Q table? To learn each value of this Q-table**,** we'll use the Q learning algorithm.

A Representative Example

Let's take a representative example to highlight how Q-Learning is put into action. Imagine a mouse in a basic maze. It has a goal of maximizing the amount of cheese it obtains. It can move in any of the four cardinal directions.

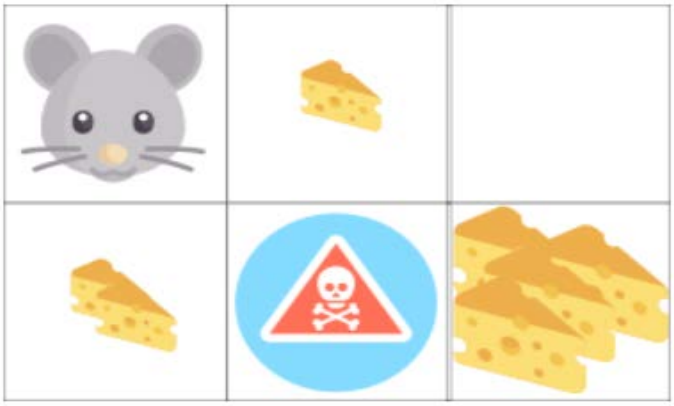

Figure 2.7 Map with mouse cheese and poison

- One cheese = +1
- Two cheese = +2
- Big pile of cheese = +10 (end of the episode)
- If it eats rat poison = -10 (end of the episode)

Step 1: We initialize our Q-table

	←	→	↑	↓
Start	0	0	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

Table 2.1 The initialized Q-tabel(mouse in first state)

Step 2: Choose an action

From the starting position, you can choose between going right or down. Because we have a big epsilon rate (since we don't know anything about the environment yet), we choose randomly. For example: move right.

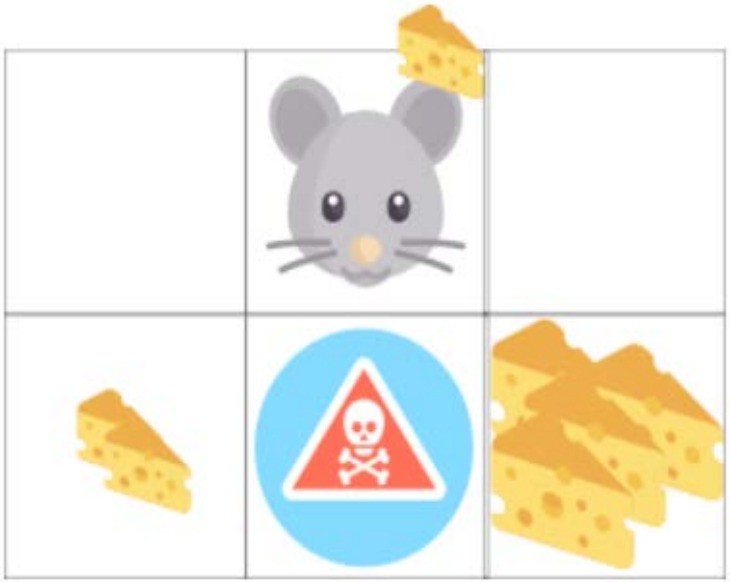

Figure 2.8 The mouse moved 1 tile to the right

	←	→	↑	↓
Start	0	0	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

Table 1.2 The after the mouse moved right

Step 3: Perform action a , get reward $R(s, a)$ and go to new state s':

By going right, the mouse gets a small cheese, so $R(s, a) = +1$ and he is in a new state.

Steps 4: Update the Q-function

We can now update the Q-value of being at start and going right. We do this by using the Bellman equation:

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma max\, Q'(s',a') - Q(s,a)]$$

New Q value for that state and that action

Current Q value

Learning Rate

Reward for taking that action at that state

Discount rate

Maximum expected future reward **given the new s' and all possible actions at that new state**

$$NewQ(start, right) = Q(start, right) + \alpha[\Delta Q(start, right)]$$

$$\Delta Q(start, right) = R(start, right) + \gamma\ max\ Q'(1cheese, a')) - \ Q(start, right)$$

$$\Delta Q(start, right) = 1 + 0.9 * max(Q'(1cheese, left), Q'(1cheese, right), Q'(1cheese, down)) - \ Q(start, right)$$

$$\Delta Q(start, right) = 1 + 0.9 * 0 - 0 = 1$$

$$NewQ(start, right) = 0 + 0.1 * 1 = 0.1$$

Here, α denotes the learning rate, set to 0.1 in this case. The learning rate controls how quickly the algorithm disregards the previous Q-value in favor of a new estimate. If the learning rate were 1, the Q-value would be completely replaced by the new estimate.

	←	→	↑	↓
Start	0	0.1	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

Table2.2 The updated Q-table

Now we need to do that again and again until the learning is stopped.

Q-Learning algorithm

The following pseudocode summarizes how Q-learning works:

Algorithm 1: $Q-learning$

$Initialize\ Q(s,a), \forall s \in S, a \in A, arbitrarly, and\ Q(terminal\ state,\cdot) = 0$

Repeat $(for\ each\ episode)$:

$Initialize\ S$

Repeat $(for\ each\ step\ of\ episode)$:

$Choose\ a_t\ from\ s_t\ using\ policy\ derived\ from\ Q(e.g.\varepsilon - greedy)$

$Take\ action\ a_t\ and\ observe\ r_{t+1}, s_{t+1}$

$$Q_\pi(s_t,a_t) \leftarrow Q_\pi(s_t,a_t) + \alpha\left(r_{t+1} + \gamma \max_a Q_\pi(s_{t+1},a) - Q_\pi(s_t,a_t)\right)$$

$s_t \leftarrow s_{t+1}$

Until $s_t\ is\ terminal$

2.5. Deep Q-Learning method

Deep Q-Learning, also known as Deep Q Network (DQN), is an expansion of the fundamental Q-Learning method that approximates the Q-values using deep neural networks. The size of the Q-table makes traditional Q-Learning difficult to use in large or continuous state spaces, but it performs well in contexts with a small and finite number of states. By substituting a neural network that can estimate the Q-values for each state-action combination for the Q-table, Deep Q-Learning gets over this restriction [30].

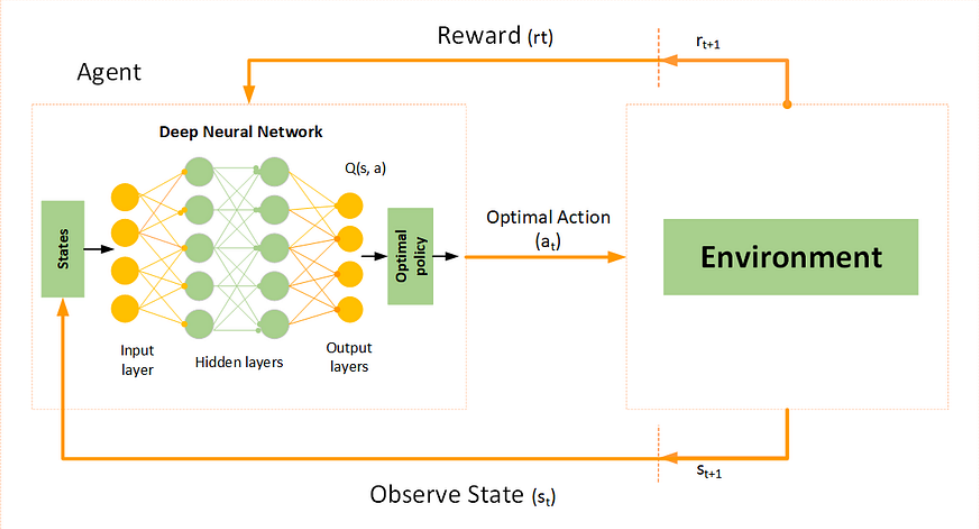

Figure 2.9 Structure of DQN

2.5.1 Key Concepts of Deep Q-Learning

1. **Q-Function Approximation**: Instead of using a table to store Q-values for each state-action pair, DQN uses a neural network to approximate the Q-values. The input to the network is the state, and the output is a set of Q-values for all possible actions.

2. **Experience Replay**: To stabilize the training, DQN uses a memory buffer (replay buffer) to store experiences (state, action, reward, next state). The network is trained on random mini-batches of experiences from this buffer, breaking the correlation between consecutive experiences and improving sample efficiency.

3. **Target Network**: DQN introduces a second neural network, called the target network, which is used to calculate the target Q-values. This target network is updated less frequently than the main network to prevent rapid oscillations in learning.

4. **Bellman Equation in DQN**: The update rule for DQN is based on the Bellman equation, similar to Q-Learning:

$$\mathrm{Q}(\mathrm{s},\mathrm{a},\theta) \leftarrow \mathrm{Q}(\mathrm{s},\mathrm{a},\theta) + \alpha[\mathrm{r} + \gamma \max_{a'} \mathrm{Q}(\mathrm{s}',\mathrm{a}',\theta^{-}) - \mathrm{Q}(\mathrm{s},\mathrm{a},\theta)]$$

Where:

- ✓ θ are the weights of the main Q-network,
- ✓ θ^{-} are the weights of the target Q-network,
- ✓ s is the current state,
- ✓ a is the action taken,
- ✓ r is the reward received,
- ✓ s' is the next state,

- $\max_{a'} Q(s', a'; \theta^{-})$ is the maximum Q-value for the next stat

2.6 DDPG method

The Deep Deterministic Policy Gradient (DDPG) is a model-free reinforcement learning (RL) approach developed for environments with continuous action spaces. DDPG, uses deep neural networks to estimate the value function (critic) and the policy (actor), combining the advantages of Q-learning and policy gradient approaches. This hybrid architecture is especially useful for robotic control, autonomous systems, and other real-world applications needing accurate, continuous outputs because it tackles the difficulties of high-dimensional state and action spaces [16].

2.6.1 Key Components of DDPG

✓ Actor-Critic Framework

The actor network learns a deterministic policy, mapping states to specific actions, while the critic evaluates the action-value function Q(s,a) . This dual- network structure enables stable learning by decoupling policy evaluation and improvement.

- ✓ **Experience Replay**

Inspired by Deep Q-Networks (DQN), DDPG uses a replay buffer to store transitions (s,a,r,s′). Random sampling from this buffer breaks temporal correlations, improving learning stability

- ✓ **Target Networks**

To mitigate divergence, DDPG employs "soft" target updates for both actor and critic networks.

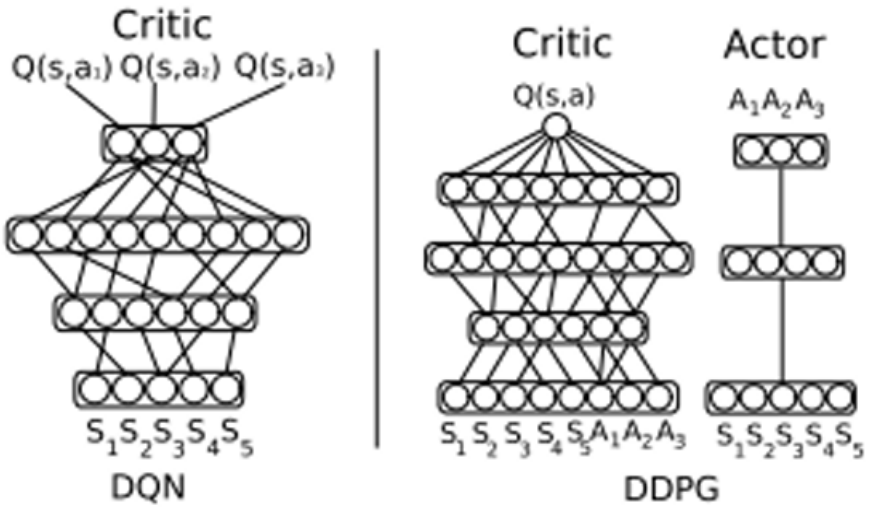

Figure 2.10 Diffrence between DQN and DDPG

2.6.2. Advantages of DDPG

Continuous Control: Unlike discrete action algorithms (e.g., DQN), DDPG excels in tasks like robotic arm manipulation or autonomous driving, where actions are continuous[32].

Sample Efficiency: Experience replay and batch normalization reduce sample complexity, making it suitable for real-world applications with limited data.

Robustness: The deterministic policy avoids the high variance of stochastic policies, improving training stability

2.6.3 Applications

DDPG has been applied to diverse domains:

- **Robotics:** Training robotic arms for precise manipulation.
- **Autonomous Vehicles:** Path planning and collision avoidance.
- **Resource Management:** Optimizing energy consumption in data centers.

2.7 Conclusion

Reinforcement learning has progressed from basic algorithms like Q-learning to more potent techniques like DDPG and DQN. DQN uses deep neural networks to improve Q-learning, which offers a strong foundation for decision-making in discrete contexts, to manage high-dimensional, difficult problems. DDPG uses an actor-critic architecture to further expand these capabilities to continuous action environments. These algorithms work together to create a powerful toolkit for creating intelligent agents in fields including resource management, autonomous systems, and robotics.

Chapter III

Implementation and Results

3.1 Introduction

The implementation process and outcomes of using the Deep Deterministic Policy Gradient (DDPG) algorithm to solve the robot trajectory tracking problem are presented in this chapter. After a brief overview of the concept, in this chapter we will be going to describe the MATLAB setup and the outcomes of training and testing the agent in various circumstances.

Validating the DDPG agent's ability to trace predetermined trajectories: square, lemniscate, circular or other types of paths under particular limitations and situations is the main objective.

3.2 Implementation in MATLAB Environment

3.2.1 Reinforcement Learning (RL) Setup

In reinforcement learning, the system is typically composed of two core components:

- **The Agent**: This is the learning component. In our implementation, the agent is based on the DDPG algorithm. It learns a deterministic policy to map observed states to continuous actions.
- **The Environment**: This simulates the world in which the agent operates, including the robot dynamics and trajectory reference system.

In our case, the MATLAB RL Toolbox was used to design both components. The DDPG agent interacts with a robot model to learn the optimal control policy for trajectory tracking.

3.2.1.1 RL Environment Components

- **State and Action Spaces**:
  States included position and orientation (x, y, θ) of the robot; continuous actions were linear and angular velocities (v, ω).
- **Observations**: are $[dx;\ dy;\ d\theta]$ where $dx = x_{ref} - x,\ y_{ref} - y$ and $d\theta = \theta_{ref} - \theta$, $[x_{ref}, y_{ref}, \theta_{ref}]$ represents the reference trajectory.

- **Robot Model**:
  The simulation used a differential drive robot governed by standard kinematic equations (1.3).
- **Reward Function**:
  The reward was designed to minimize tracking error and smooth control:

$$Reward = -D_{err} - 0.1 * |d\theta| - 0.05(v^2 + \omega^2) \quad \textbf{(3.1)}$$

Where $D_{err} = \sqrt{dx^2 + dy^2}$ is the distance between the robot and the reference trajectory.

This reward function guides the agent to minimize tracking error and take smooth actions.

- **Episode Termination Condition:**
- **An episode ends when:**
  - ✓ The number of steps reaches a fixed value: 500 steps in our case, or
  - ✓ When tracking error exceeds a threshold
  - ✓ Or the robot reaches the end of the trajectory, in other words the current time $t >= totalTime$.
- **Trajectory design**:

we trained our DDPG agent using different types of reference trajectories (e.g., circular, figure-eight, square, a waypoint-based path).

3.2.2 DDPG Agent Design

3.2.2.1 Agent configuration

The following configuration of Actor-Critic Neural Networks was used to train the DDPG agent:

- **Actor**: One input layer with 3 inputs ($state = [x, y, \theta]$), two fully connected layers with 128 and 64 neurons and relu activation functions and an output layer with 2 outputs ($action = [v, \omega]$) with tangent hyperbolic activation function that ensure output in $[-1, 1]$
- **Critic**: One input layer which concatenates the output of the Actor network (action: $[v, \omega]$) and the state ($[x, y, \theta]$), two fully connected layers with 128 and 64 neurons with relu activation functions and an output layer with one input (Q-value).

3.2.2.2 Exploration strategy

In DDPG agent, the main idea is to **inject noise** into the actions selected by the policy network to encourage exploration of the environment. During training, instead of executing the deterministic action output by the actor network directly, the agent adds exploration noise. At the beginning of training, high noise encourages the agent to explore the state–action space. As learning progresses, noise is gradually reduced to shift

focus toward exploiting learned policies.

3.2.3. Training and Evaluation

For the training, we chose the following hyperparameters:

- ✓ Number of episodes: 1000
- ✓ Steps per episode: 500
- ✓ A Sample Time dt=0.1
- ✓ Total simulation time: 40 seconds
- ✓ Replay Buffer Length = 1e6
- ✓ MiniBatch Size=64
- ✓ Discount Factor = 0.99
- ✓ Circular trajectory defined with $\omega = 2\pi/20$ completing 2 full rounds.
- ✓ Ornstein-Uhlenbeck noise was used to encourage exploration during training.

The training stops when the condition: **t >= totalTime** is met.

3.3. Discussion of MATLAB Results

Several scenarios were considered during our simulations:

1. **First Scenario**

First, we trained our DDPG agent on the circular trajectory only for 500 episodes and 200 steps per episode. Figure 3.1 below shows the **Reinforcement Learning Episode Manager** window from MATLAB during the training of the DDPG agent. It provides both visual and quantitative information about the agent's learning progress over 500 episodes. Se can see the curves of the Episode Reward, Average Reward and the Episode Q_0. From Equation (3.1) above, we note that the chosen reward is always negative; however, successful training is characterized by the reward approaching zero.

We can observe that the reward curves increase over episodes, indicating that the DDPG agent is learning the policy effectively. However, additional training episodes are needed to achieve better performance.

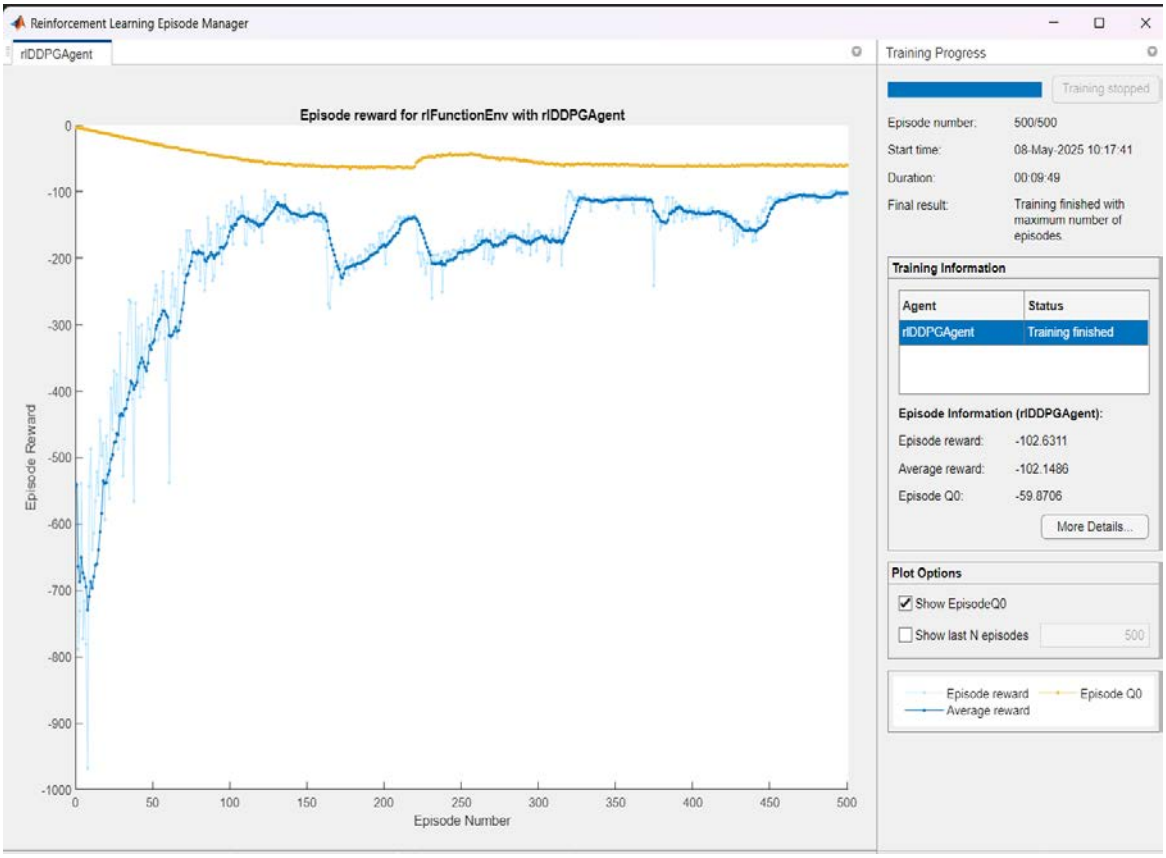

Figure 3.1 The Reinforcement Learning Episode Manager window from MATLAB during the training of a DDPG agent

Once the DDPG trained, we tested it on a reference circular trajectory. The agent acts as a controller that control the robot to track this reference trajectory. The results of this test on a circular trajectory are shown in the figure 3.2 for different initial positions and different number of rounds. The results demonstrate good tracking behavior which we can enhance by tuning hyper parameters, number of episodes and steps, number of neurons, etc.

Figure 3.3 shows the result of testing our DDPG agent (which we trained on circular trajectory only) on a lemniscate trajectory to test its generalization ability to different trajectories. We observe that the robot's trajectory initially shows a significant deviation from the reference path. However, after returning to the initial position, the robot begins to track the reference trajectory, with some error that can be reduced by tuning the training hyperparameters and increasing the number of training episodes.

Figure 3.4 gives the result for a square reference trajectory; the same remarks can be noticed here. In figure 3.5, we show the tracking position error and Heading error. We observe that the position error initially exhibits a significant deviation but decreases afterward. Both the position and heading errors increase each time the robot changes direction by an angle of $\pi/2$, but the position error remains within an acceptable tolerance of 0.5 meters which we can enhance by extending the number of the training episodes. The linear and angular velocities are shown in figure 3.6. They are smooth and constrained within the interval [-1,1].

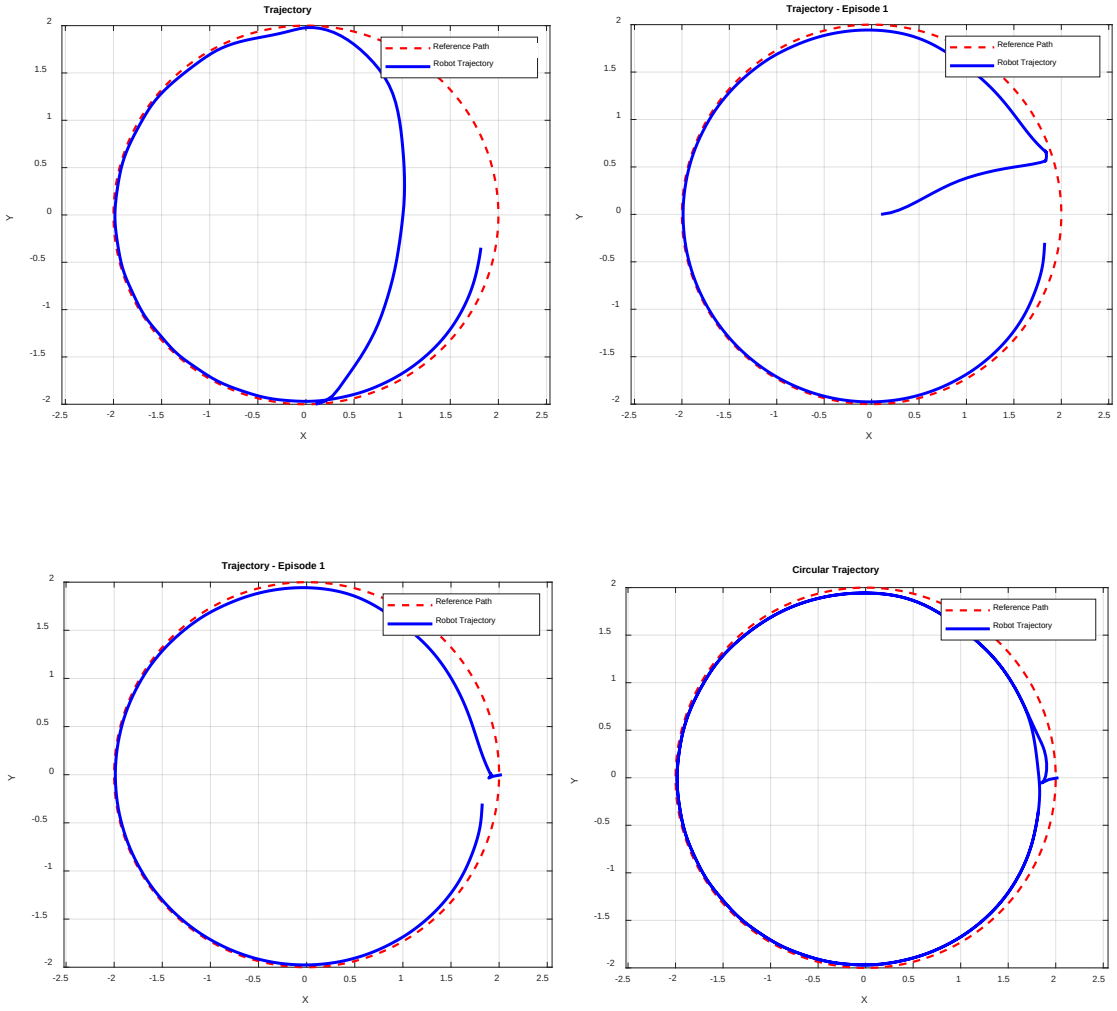

Figure 3.2 Test on a circular trajectory for different initial positions and different number of rounds

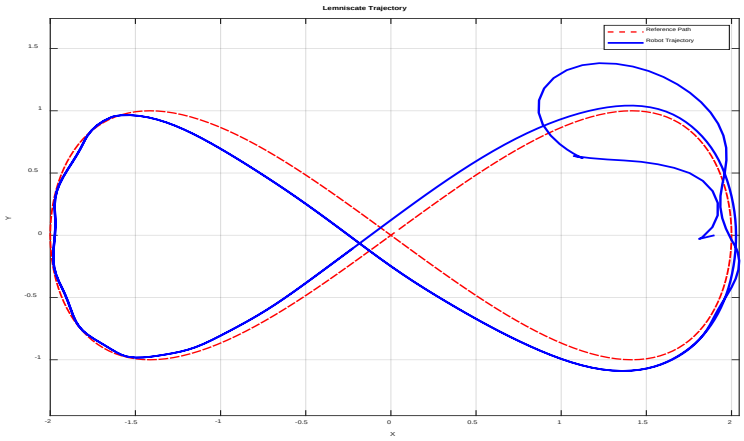

Figure 3.3 Test on a lemniscate trajectory

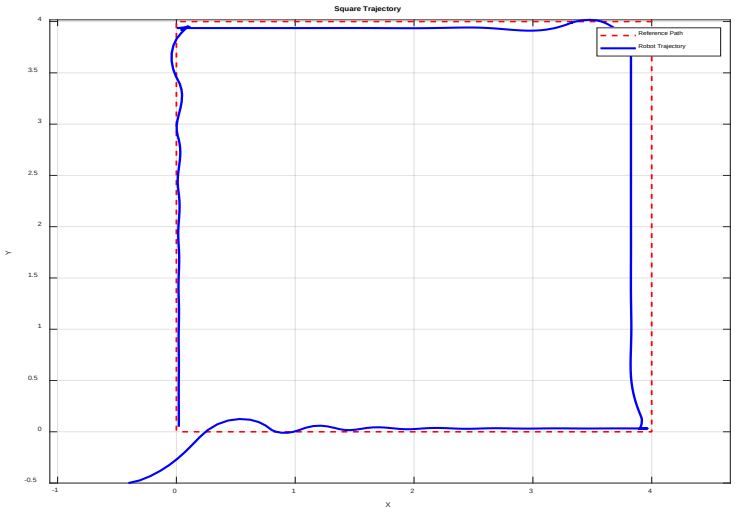

Figure 3.4 Test on a square trajectory

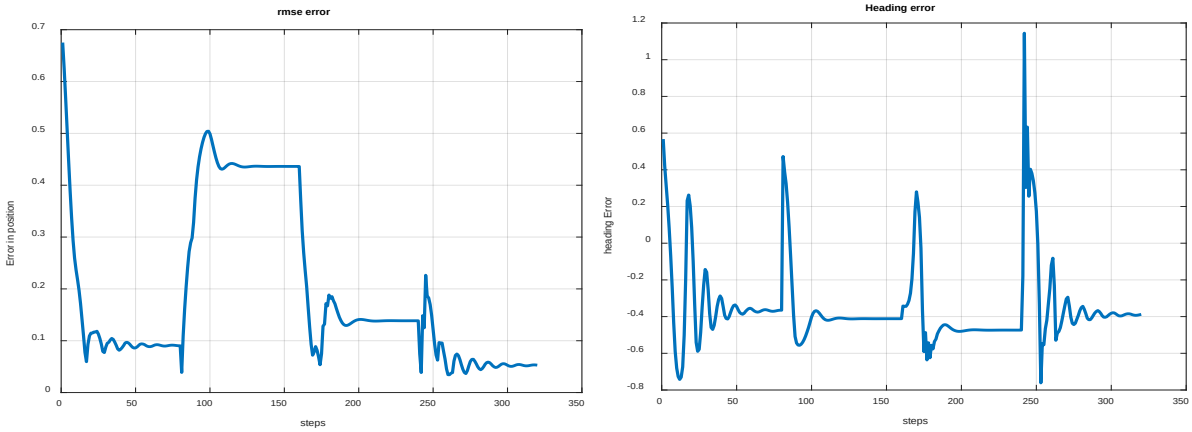

Figure 3.5 Position and Heading Errors in the case of the square trajectory

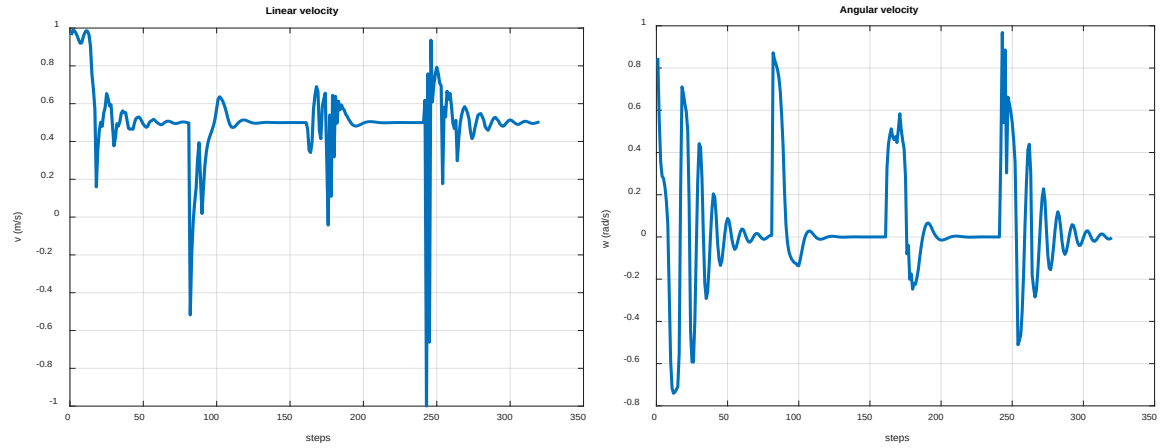

Figure 3.6 Linear and angular velocities for the square trajectory

To improve the results, we retrained our DDPG agent, this time increasing the number of episodes to 1000. The training outcome is illustrated in Figure 3.7. We can observe that the reward is very close to zero, and the deviations from the average reward are significantly reduced, indicating that the agent is learning effectively.

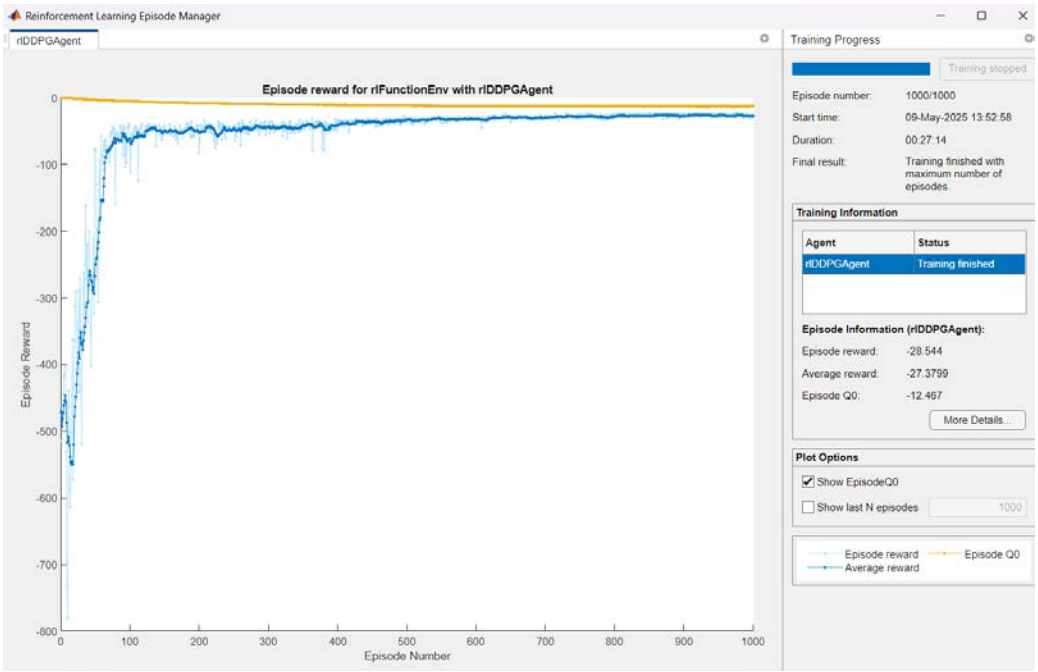

Figure 3.7 DDPG agent Training for a circular reference trajectory and 1000 episodes

Afterward, we used the retrained DDPG agent to control the robot for tracking both circular and square trajectories, as shown in Figure 3.8. We observed that the DDPG controller performed very well on the circular trajectory. However, this was not the case for the square trajectory, where slight oscillations occurred around the reference path. This can be attributed to overfitting of the agent to the circular trajectory, as the training was conducted exclusively on that shape.

This result suggests that simply increasing the number of training episodes does not guarantee generalization to different trajectory types. The position and heading errors and the control signals (v, ω) for the circular trajectory are shown in Figures 3.9 and 3.10 respectively. We observe that the errors are reduced and the velocities are smooth.

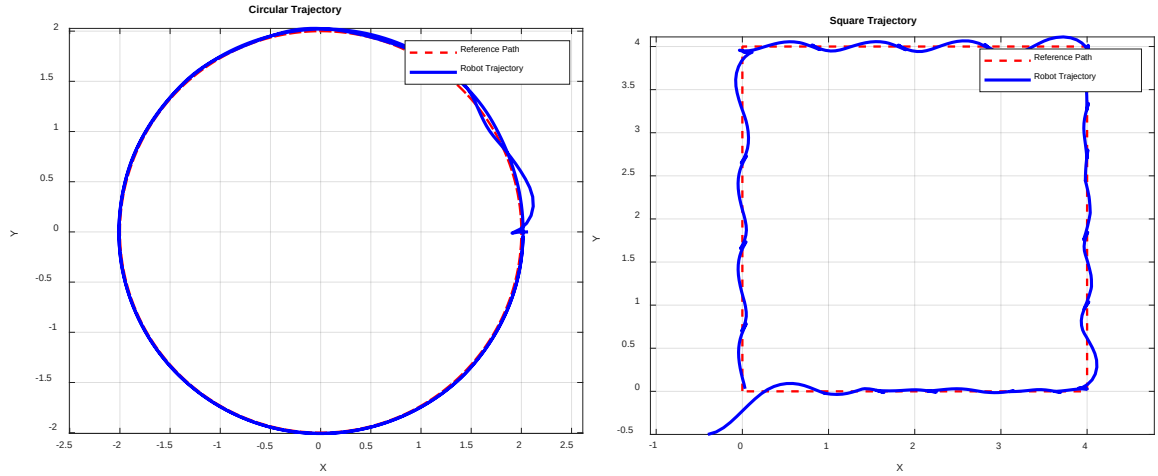

Figure 3.8 Results after using the retrained DDPG agent

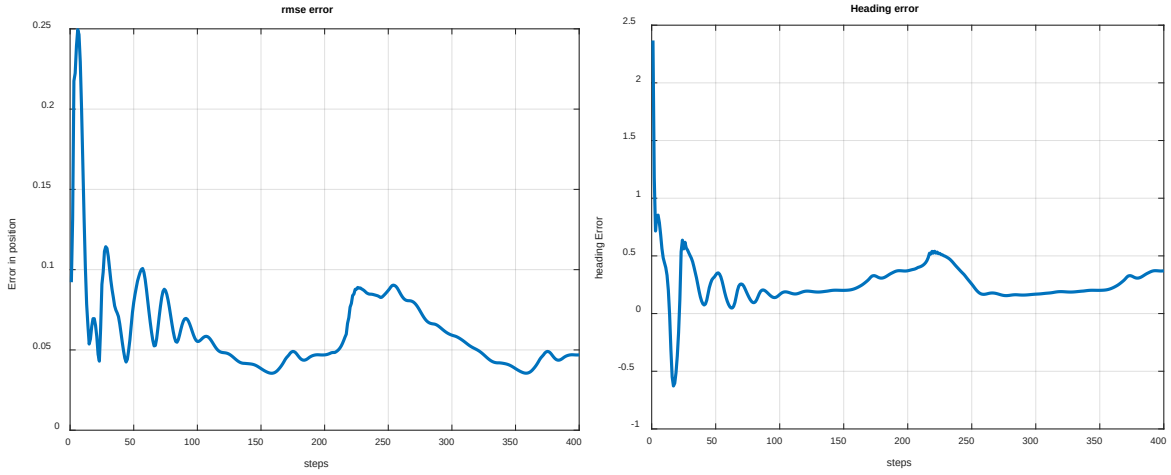

Figure 3.9 Position and heading Errors for the circular trajectory

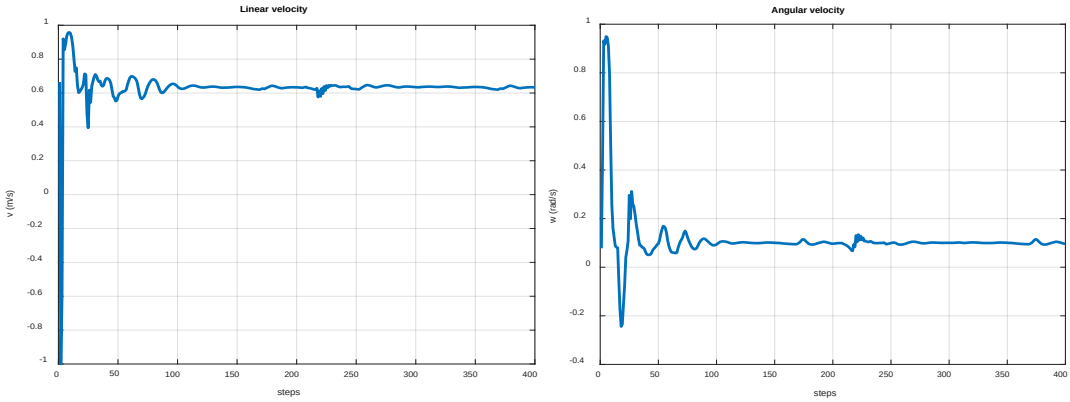

Figure 3.10 Linear and angular velocities for the circular trajectory

Second Scenario

In this scenario, we train our DDPG agent on all three reference trajectories. In each training episode, one trajectory is selected at random. The objective is to enable the agent to generalize its policy across different types of trajectories. The evolution of the curves during training is shown in Figure 3.11. They demonstrate effective policy learning.

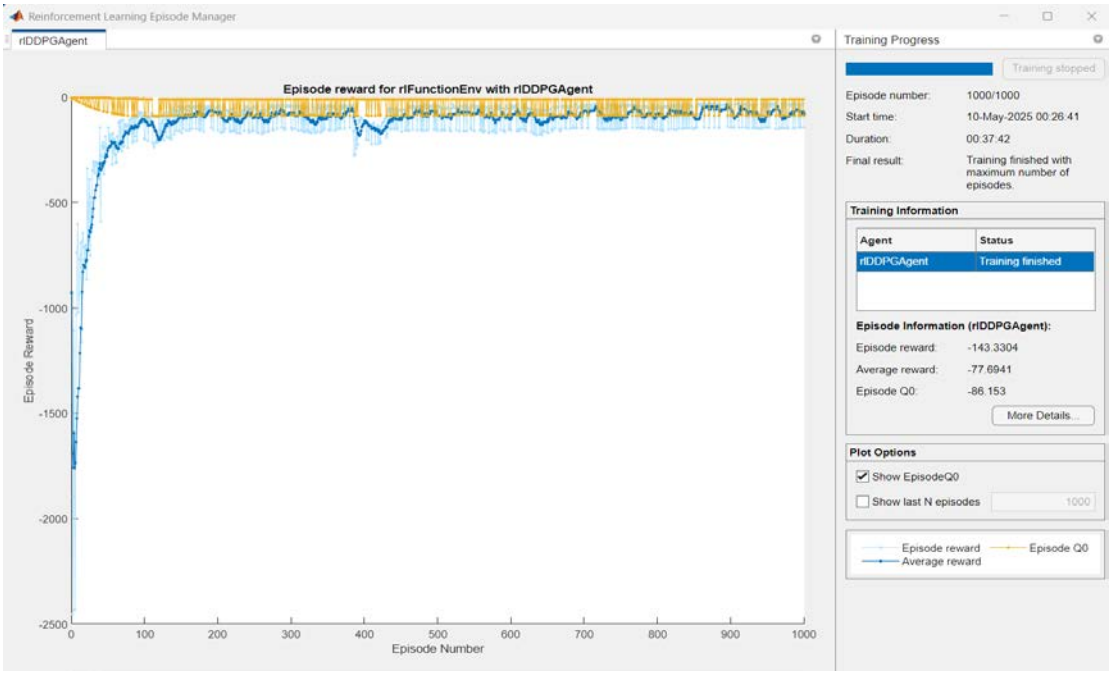

Figure 3.11 DDPG agent Training for different types of reference trajectories

Figure 3.12 gives the results for the circular trajectory with 2 initial positions, the lemniscate and square trajectories. It can be seen that the robot track accurately all the reference trajectories.

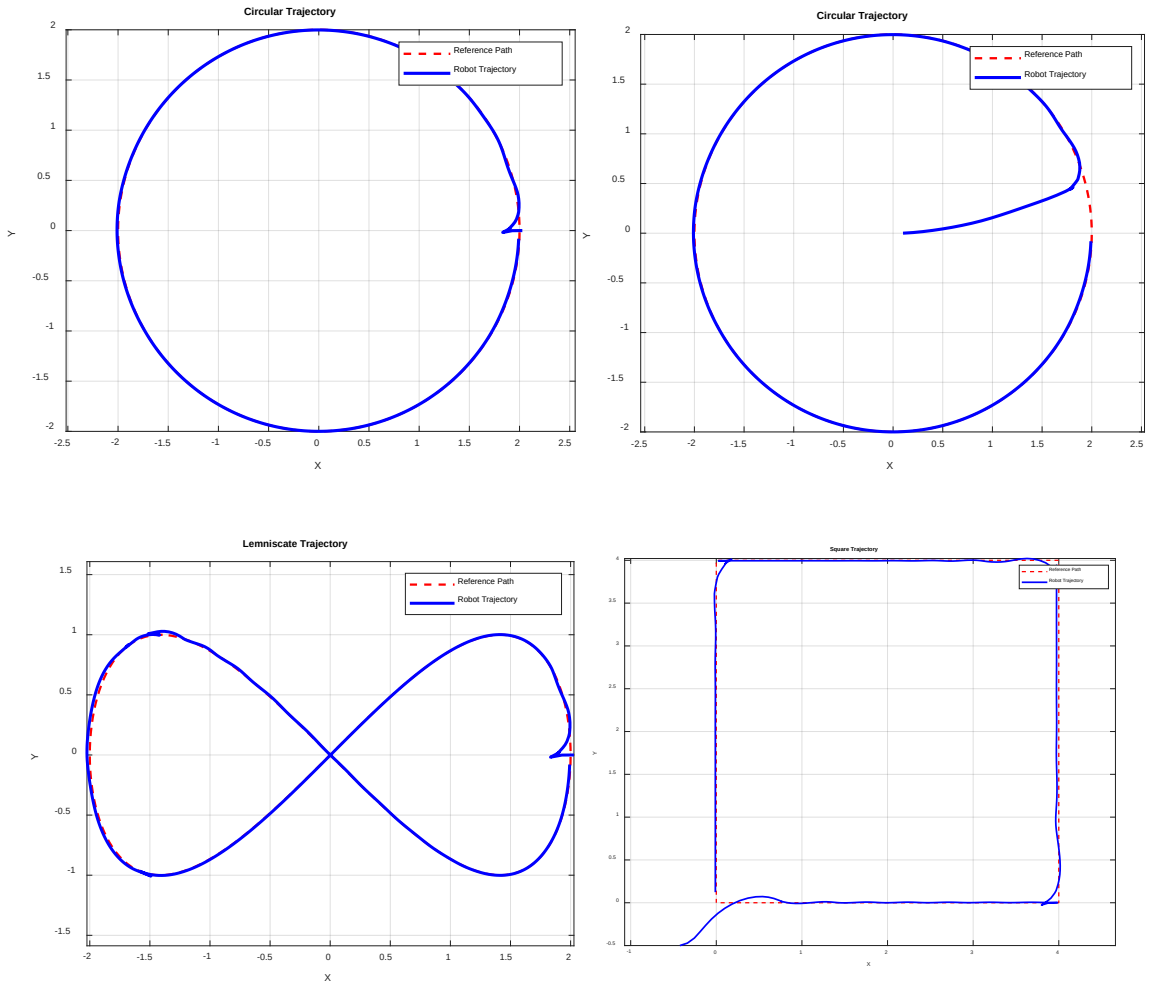

Figure 3.12 Test for all the three trajectories

3.4 Simulation Results using Matlab-ROS interface

Gazebo is an open-source 3D simulator for robotic systems, well-known for its high-fidelity physics engine, sensor simulation, and compatibility with robotic middleware like ROS [35]. It offers a dynamic testing platform that allows robotic algorithms from low-level control to high-level planning to be verified in intricate, interactive settings without the need for hardware or physical risk.

Figure 3.13 logo de Gazebo[39]

The TurtleBot mobile robot platform is used in this study because of lightweight ROS integration, and proven track record in scholarly research [37].

Figure 3.14 Turtlebot [39]

Control rules developed in MATLAB using reinforcement learning can be tested in Gazebo with no overhead because to the simplified simulation-to-deployment pipeline made possible by the integration of MATLAB and Gazebo via ROS [36].

Figure 3.15 Logo de ROS[39]

The shift from MATLAB to Gazebo is crucial for assessing the real-world practicality of taught policies. While MATLAB provides a controlled and rapid prototyping environment, Gazebo replicates real-world defects such as sensor noise, dynamics lag, and environmental randomness, acting as a stepping stone before deploying to physical robots [38].

In this section, we use the previously trained agent to control the robot in tracking a waypoint-based path within an office environment, as shown in Figure 3.16. The path is generated using

a path planning algorithm applied to the map of the office. The map of this environment is presented in Figure 3.17.

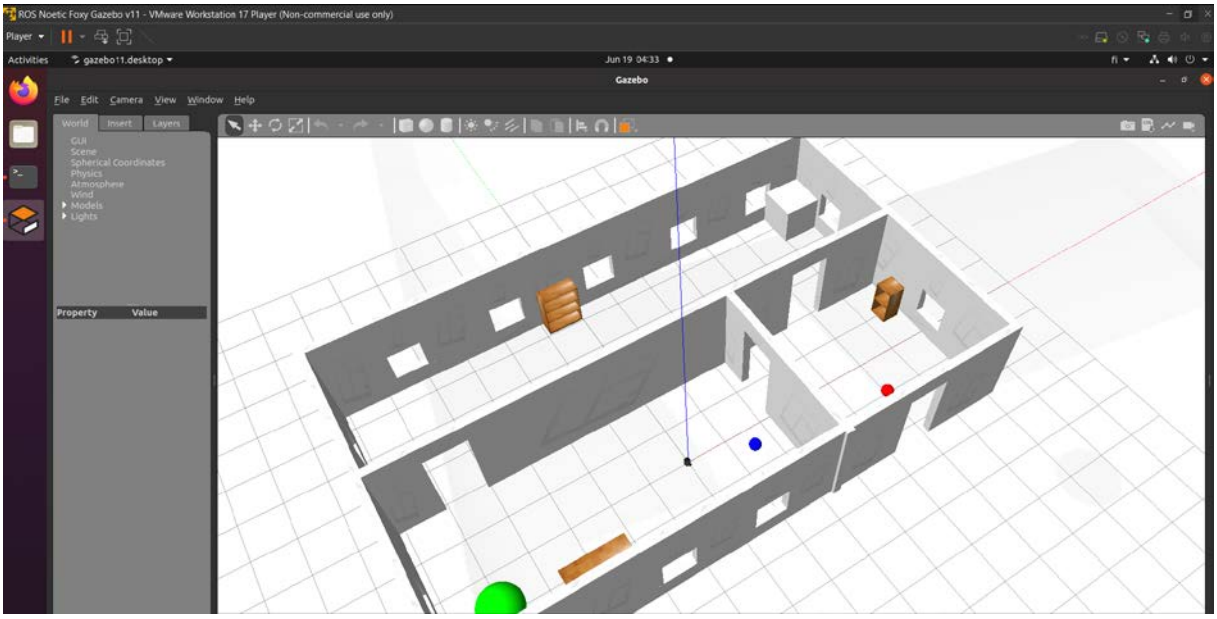

Figure 3.16 The Office environment in Gazebo

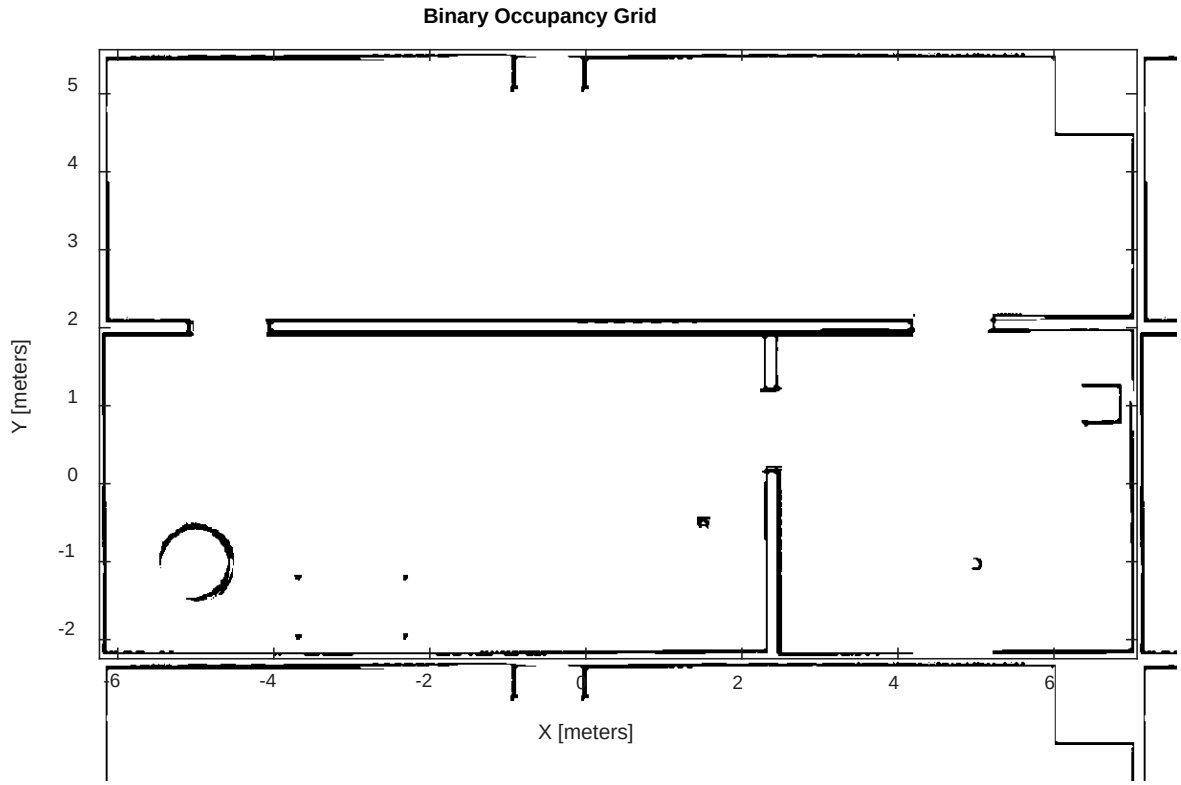

Figure 3.17 The map of the environment office

We considered two scenarios of simulation:

1. First Scenario

In the first scenario, we use the PRM classic method for path planning to find the optimal path to go from the initial position $[0, 0]$ to the Goal at position $[4.5, 4]$. Figure 3.18 shows the inflated map to skip the robot to collide with obstacles of the environment. This figure show also the path planned to go to the goal.

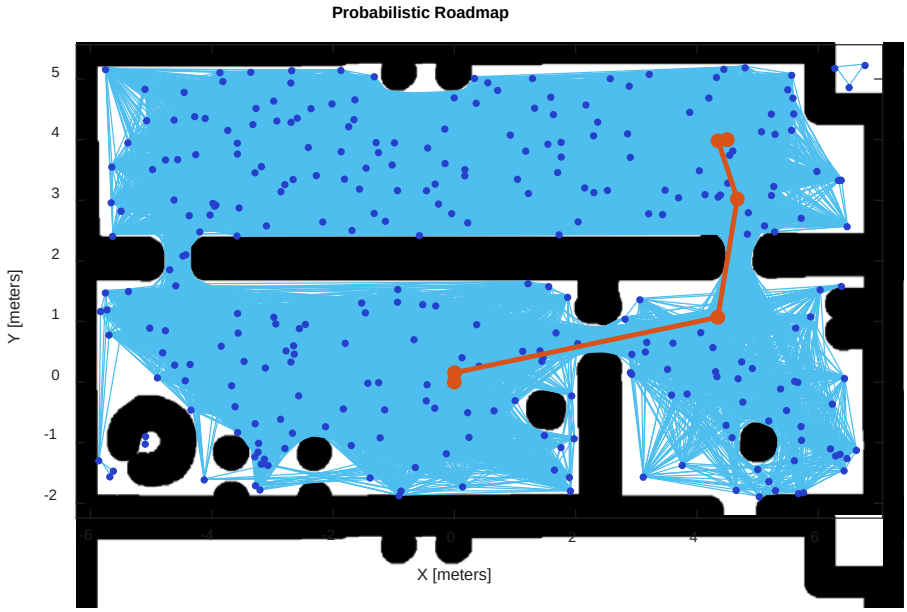

Figure 3.18the planned path using PRM in the inflated map

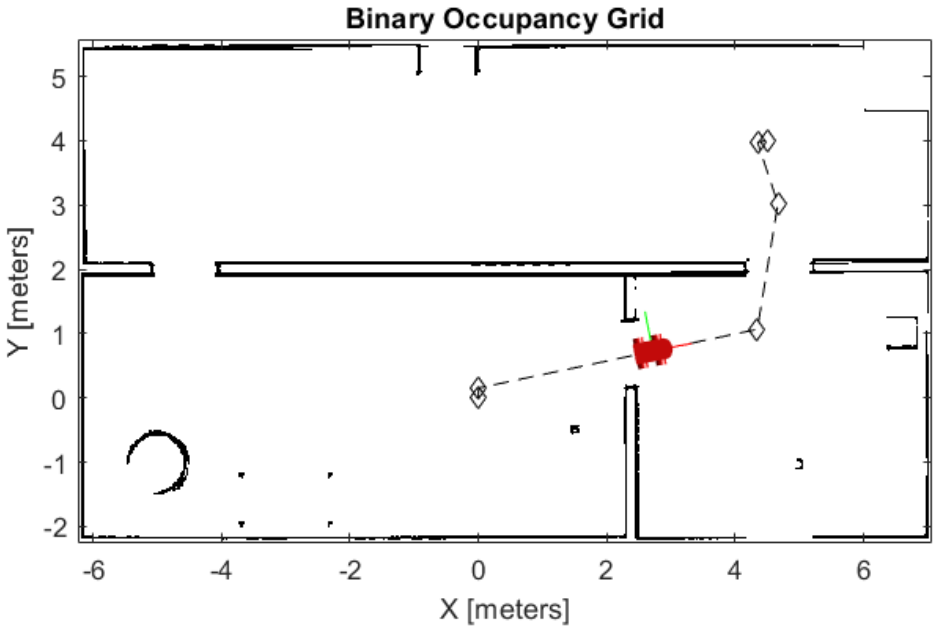

Figure 3.19 The robot following the planned trajectory

Figure 3.19 shows the robot following the planned trajectory using the DDPG controller trained in the previous section. In Figure 3.20, the robot's actual trajectory is shown in green, while the planned path's waypoints are depicted in blue. The robot accurately and smoothly tracks the reference trajectory.

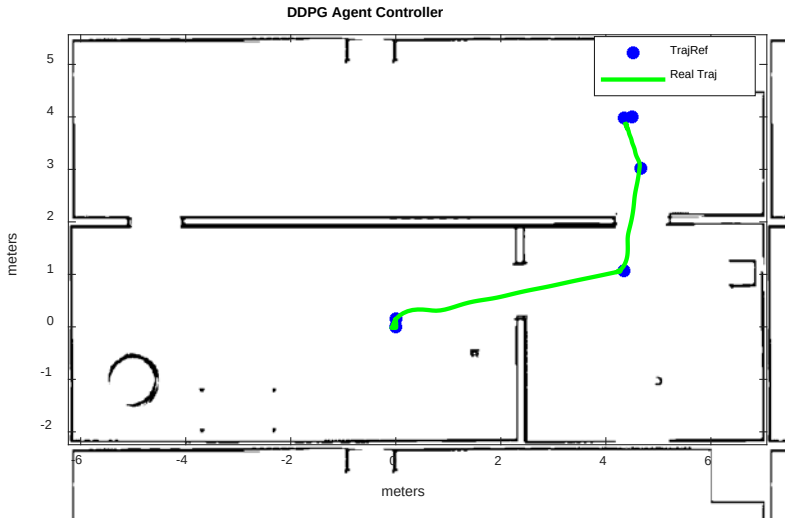

Figure 3.20 The real trajectory of the robot and the waypoints of the trajectory planned by PRM

Figure 3.21 shows the position and heading errors at the top, and the linear and angular velocities at the bottom. The errors are low, and the velocities are smooth and remain within the interval [-1, 1].

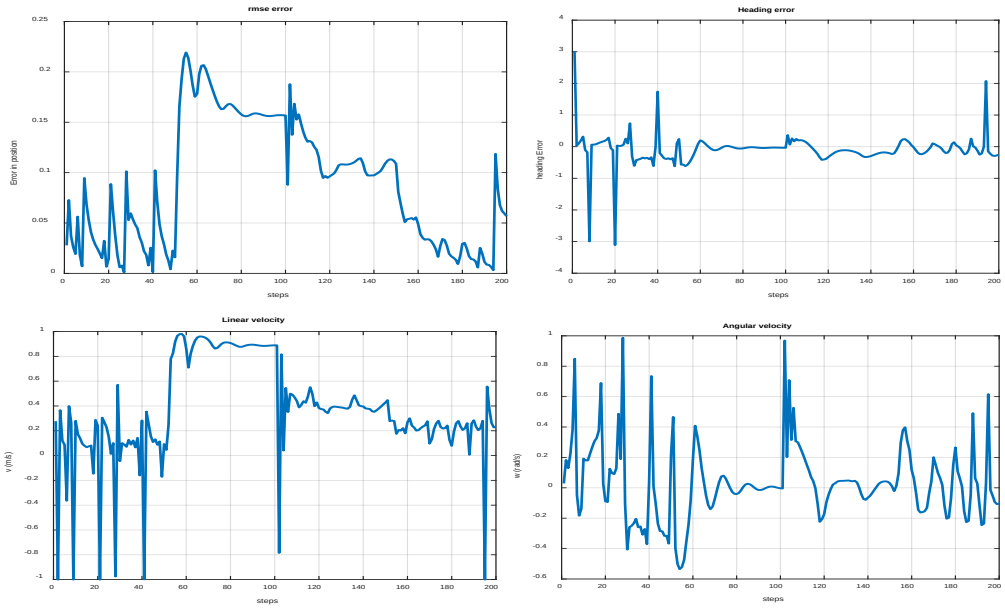

Figure 3.21 Position and heading errors at the top and linear and angular velocities at the bottom

2. Second scenario

In this scenario, we used the Q-Learning method for path planning from the initial position [0,0] to the goal at position [4.5, 4]. The Turtlebot3 is used as the base platform in Gazebo within an office environment. The DDPG controller is implemented in MATLAB and controls the robot to track the planned trajectory through the MATLAB-Gazebo interface. Figure 3.22 shows the results in MATLAB, and Figure 3.20 shows the results in Gazebo. The robot accurately tracks the planned trajectory in both MATLAB and Gazebo simultaneously.

From the simulation results shown in the figure, we can confidently say that the robot's path planning and trajectory tracking inside the Gazebo environment were both successful and reliable. The robot was able to move through a realistic indoor setting filled with walls, doors, and obstacles—while following the planned path closely and avoiding collisions.

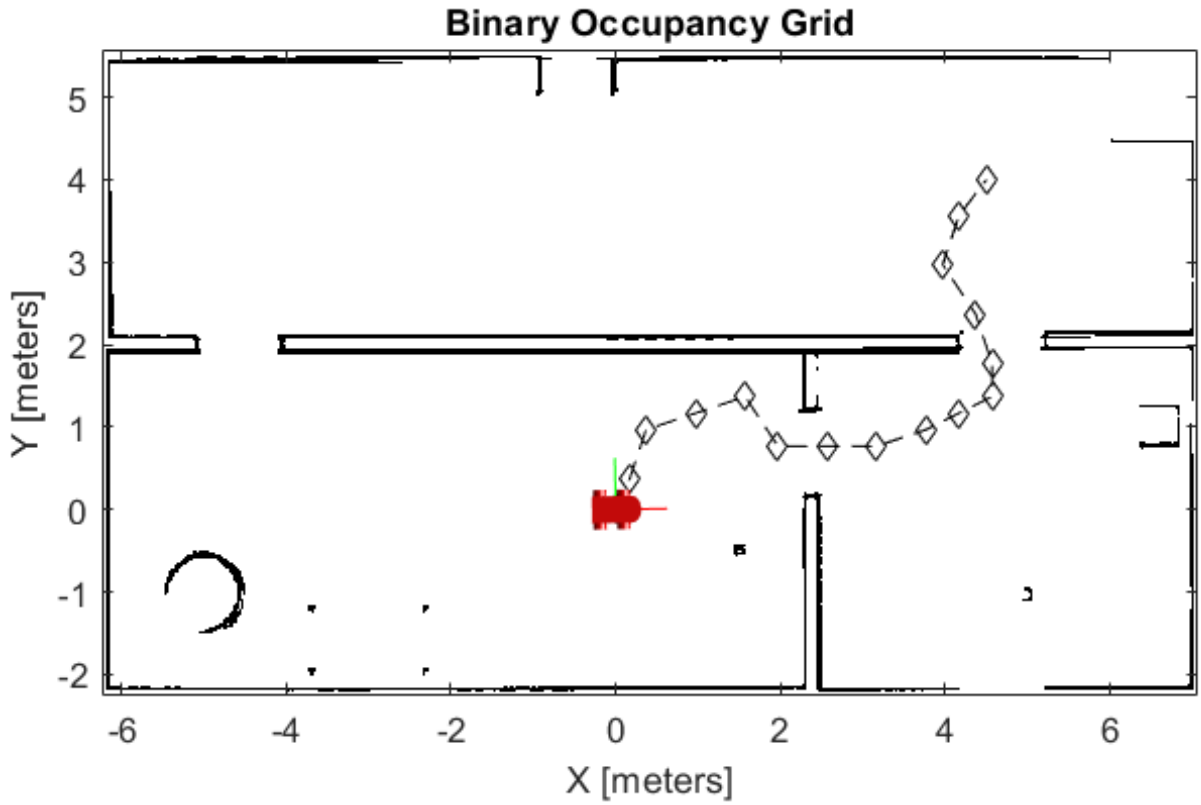

Figure 3.22 The robot moves to track the planned trajectory in Matlab

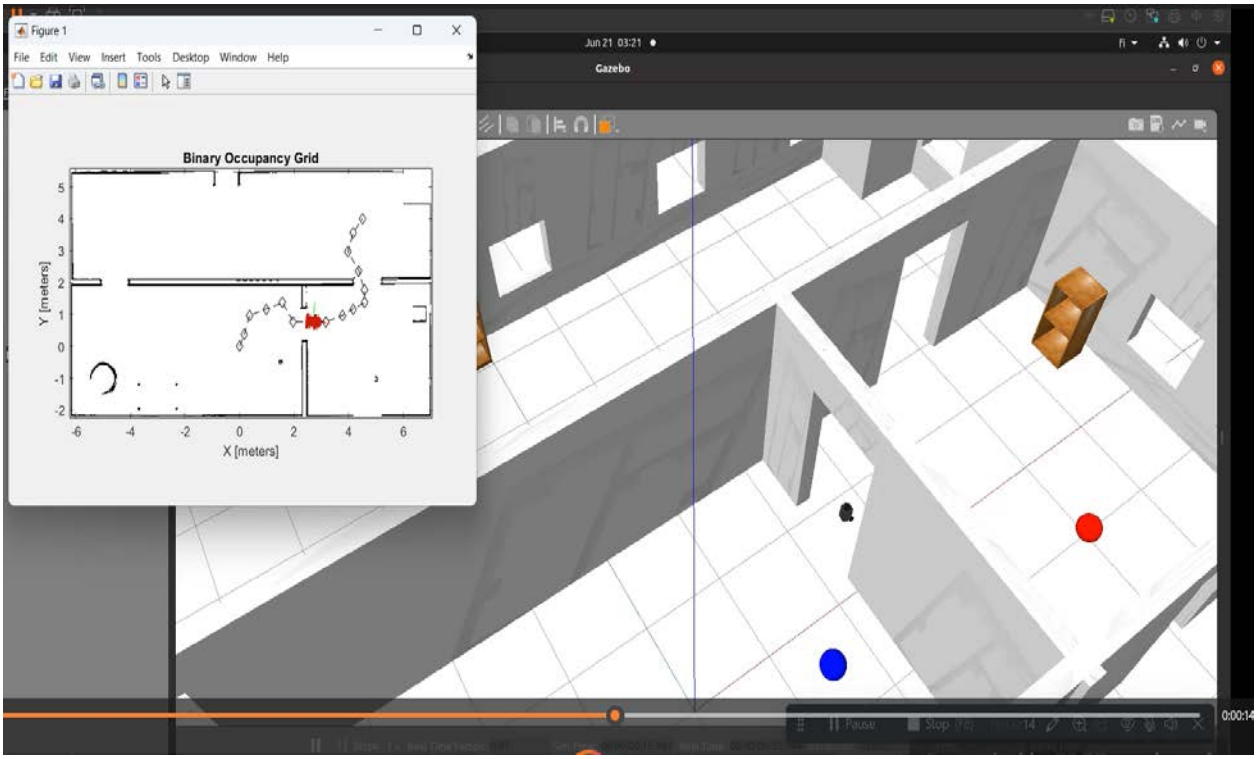

Figure 3.23 The robot moves to track the planned trajectory in Gazebo

3.5. Conclusion

In this chapter we presented the entire process, from training a DDPG agent in MATLAB to deploying and verifying it in the Gazebo simulation with ROS. The findings show that reinforcement learning can achieve reliable trajectory tracking, particularly on established patterns. Future work could offer more diversified training paths, as well as real-world deployment on physical robots.

Conclusions and Future works

In conclusion, this work successfully combined theoretical foundations with practical implementation to develop an intelligent control system for a differential drive mobile robot using deep reinforcement learning algorithms. Starting from the robot's kinematic modeling and motion analysis, we explored artificial intelligence techniques such as Q-Learning and DDPG, and implemented them in realistic simulation environments using MATLAB and the Gazebo simulator under ROS.

The results demonstrated that reinforcement learning algorithms enabled the robot to accurately track predefined trajectories, validating the effectiveness of the learned control policies. This study confirms the importance of integrating artificial intelligence with physical modeling to develop advanced and adaptive robotic systems.

This project represents a promising step toward a future where intelligent robots operate with greater autonomy. In future developments, this work could be extended to include multi-agent learning, computer vision, and deployment in more complex environments.

Bibliography

[1] Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2011). Introduction to Autonomous Mobile Robots (الطبعة الثانية). MIT Press.

[2] Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2011). Introduction to Autonomous Mobile Robots. MIT Press.

[3] Craig, J. J. (2018). Introduction to Robotics: Mechanics and Control (4th ed.) Pearson Educatio.

[4] Siciliano, B., Sciavicco, L., Villani, L., & Oriolo, G. (2010). Robotics: Modelling, Planning and Control. Springer.

[5] Boston Dynamics. (n.d.). Spot – Agile mobile robot. Retrieved from https://www.bostondynamics.com/spot.

[6] Hirose, S. (1993). Biologically Inspired Robots: Snake-Like Locomotors and Manipulators. Oxford University Press.

[7] Siegwart, R., Nourbakhsh, I. R & ,.Scaramuzza, D. (2011). Introduction to Autonomous Mobile Robots (2nd ed.). MIT Press.

[8] Austin, R. (2010). Unmanned Aircraft Systems: UAVS Design, Development and Deployment. Wiley.

[9] J. R. Barber, "Continuum Mechanics," Cambridge University Press, 2010

[10] Master’s dissertation: Q-Learning based Path Planning and Predictive Control for the Navigation of a Mobile Robot2023/2024 oussama guettaf/moncef zakaria miloudia amar telidji university of lahgouat

[11] Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2011). Introduction to Autonomous Mobile Robots. MIT Press.

[12] Dudek, G., & Jenkin, M. (2010). Computational Principles of Mobile Robotics. Cambridge University Press.

[13] Borenstein, J., Everett, H. R., & Feng, L. (1996). Where am I? Sensors and Methods for Mobile Robot Positioning. University of Michigan.

[14] Thrun, S., Burgard, W., & Fox, D. (2005). Probabilistic Robotics. MIT Press.

[15] Kelly, A. (2013). Mobile Robotics: Mathematics, Models, and Methods (Cambridge University Press).

[16] Lillicrap, T. P., et al. (2016). Continuous control with deep reinforcement learning. arXiv:1509.02971.

[17] Borenstein, J., Everett, H. R., & Feng, L. (1996). Where Am I? Sensors and Methods for Mobile Robot Positioning.

[18] Koenig, N., & Howard, A. (2004). Design and Use Paradigms for Gazebo.

[19]Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction (2nd ed.). MIT Press.

[20] Thrun, S., Burgard, W., & Fox, D. (2005). Probabilistic Robotics. MIT Press.

[21]MathWorks Documentation – Reinforcement Learning Toolbox & Robotics System Toolbox.

[22]Corke, P. (2017). Robotics, Vision and Control: Fundamental Algorithms in MATLAB (2nd ed.). Springer.

[23] Francois-Lavet, V., et al. (2018). An Introduction to Deep Reinforcement Learning. Foundations and Trends in Machine Learning.

[24] Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: A survey. The International Journal of Robotics Research, 32(11), 1238–1274. https://doi.org/10.1177/0278364913495721

[25] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd ed.). MIT Press.

[26] Simplilearn. (n.d.). What is Q-learning? Everything you need to know. Simplilearn.com. https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-q-learning

[27] Aradhya, A. L. (2023, March 23). Diving deeper into reinforcement learning with Q-learning.

freeCodeCamp.org. https://www.freecodecamp.org/news/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe/

[28] MathWorks. (n.d.). Train DQN Agent to Balance Discrete Cart-Pole System. Reinforcement Learning Toolbox Documentation. https://www.mathworks.com/help/reinforcement-learning/ug/train-dqn-agent-to-balance-discrete-cart-pole.html

[29] Bhatt, S. (2020, November 27). A beginner's guide to reinforcement learning and its basic implementation from scratch. Medium. https://medium.com/analytics-vidhya/a-beginners-guide-to-reinforcement-learning-and-its-basic-implementation-from-scratch-2c0b5444cc49

[30] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529–533. https://doi.org/10.1038/nature14236

[31] Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. Proceedings of the 35th International Conference on Machine Learning, 80, 1861–1870. http://proceedings.mlr.press/v80/haarnoja18b.html

[32] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2015). Continuous control with deep reinforcement learning. arXiv. https://arxiv.org/abs/1509.02971

[33] edium. (n.d.). Deep deterministic policy gradient(DDPG) — an off-policy Reinforcement Learning algorithm. https://medium.com/intro-to-artificial-intelligence/deep-deterministic-policy-gradient-ddpg-an-off-policy-reinforcement-learning-algorithm-38ca8698131b

[34] Kavraki, L. E., Švestka, P., Latombe, J. C., & Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation, 12*(4), 566–580.

[35] Koenig, N., & Howard, A. (2004). Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems* (Vol. 3, pp. 2149–2154). IEEE.

[36] MathWorks. (2025). *Communicate with TurtleBot using ROS in MATLAB*. Retrieved from https://www.mathworks.com/help/ros/ref/robotics.ros.turtlebot.html

[37] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., ... & Ng, A. Y. (2009). ROS: An open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.

[38] Sadeghi, F., & Levine, S. (2017). CAD2RL: Real single-image flight without a single real image. In *Robotics: Science and Systems*.

[39] Master's dissertation: *Planification de trajectoire et commande d'un robot mobile en Utilisant l'interface MATLAB-ROS* Masseauda/Zaza Ahlam amar telidji university of lahgouat