FINAL DEGREE THESIS

**Bachelor’s Degree in Automatic Industrial Electronics**

**ENHANCING MARL FOR REALITY GAP REDUCTION**



**Report and Annex**

**Author:** Guillem Senabre Prades

**Supervisor:** 柯士文 George Ke

**Department:** EIA

**Co-supervisor:** Benitez Iglesias, Raul

**Call:** 2024, January

# Abstract

Contents

[Abstract ii](#_Toc155136559)

[Introduction 1](#_Toc155136560)

[Historical overview 2](#_Toc155136561)

[Identification of the problem 2](#_Toc155136562)

[Rationale 2](#_Toc155136563)

[Objectives 3](#_Toc155136564)

[Scope and limitations 3](#_Toc155136565)

[Overview of methodology and resources 4](#_Toc155136566)

[Organization of the Study 4](#_Toc155136567)

[Frameworks and Resources 5](#_Toc155136568)

[ROS 5](#_Toc155136569)

[Gazebo 7](#_Toc155136570)

[ESP32 8](#_Toc155136571)

[C++ and Python 8](#_Toc155136572)

[Model design 9](#_Toc155136573)

[Model definition 9](#_Toc155136574)

[Custom MARL with DDPG 10](#_Toc155136575)

[System structure 11](#_Toc155136576)

[States 11](#_Toc155136577)

[Actions 11](#_Toc155136578)

[Policy 12](#_Toc155136579)

[State transition function 14](#_Toc155136580)

[Termination state 15](#_Toc155136581)

[Reward function 16](#_Toc155136582)

[Value function (expected cumulative reward) 17](#_Toc155136583)

[Simulation 18](#_Toc155136584)

[Simulation design 18](#_Toc155136585)

[Simulated observations 19](#_Toc155136586)

[States 19](#_Toc155136587)

[Actions 20](#_Toc155136588)

[Simulation results 22](#_Toc155136589)

[Implementation 25](#_Toc155136590)

[Material 25](#_Toc155136591)

[Communication 25](#_Toc155136592)

[Serial 25](#_Toc155136593)

[I2C 25](#_Toc155136594)

[Real-world observations 26](#_Toc155136595)

[States 26](#_Toc155136596)

[Actions 26](#_Toc155136597)

[Reality Gap 26](#_Toc155136598)

[Results 26](#_Toc155136599)

[Analysis and discussion 26](#_Toc155136600)

[Issues 26](#_Toc155136601)

[Integration drift 26](#_Toc155136602)

[Economic analysis 26](#_Toc155136603)

[Environmental analysis 26](#_Toc155136604)

[Conclusions 26](#_Toc155136605)

[References 28](#_Toc155136606)

[Appendix I 30](#_Toc155136607)

[ESP32 Overview 30](#_Toc155136608)

[SDF files 30](#_Toc155136609)

[Communication between ROS 2 and Gazebo 32](#_Toc155136610)

[Pointer casting 34](#_Toc155136611)

[Appendix II 35](#_Toc155136612)

[CDDPG Agent 35](#_Toc155136613)

[ESP32 | PCA9685 | 110VAC/5VDC Converter 32](#_Toc155136614)

[HCSR-04 | MPU6050 33](#_Toc155136615)

[MG996R Servo Motors 34](#_Toc155136616)

# Introduction

The growing field of Artificial Intelligence over the past decades has profoundly influenced our daily lives, altering the way we communicate with each other, learn, educate, interact with technology and numerous other aspects of life. Being able to teach machines how to learn and make predictions from data with Machine Learning opened many doors to scientists and businesses, enabling unprecedented levels of innovation, efficiency, and personalized customer services experiences across various industries.

One of the fields that has been influenced by AI is robotics, allowing the machine to learn and interact with the outside world through its actuators controlled by input sensory data processed with ML learning algorithms. Robotic Learning is what lies in the intersection of Robotics and Machine learning, and it takes advantage of Reinforcement Learning, a subfield of ML that teaches machines through trial and error allowing robots or agents in videogames to make intelligent decisions in complex environments.

Humans have the ability to learn through a process of trial and error, enabling complex locomotive tasks such as walking down the stairs and talking at the same time. This learning paradigm has already been thoroughly mimicked in virtual environments with digital agents [1], [2]. However, there are still many ongoing challenges regarding real-life scenarios, particularly in the realm of robotics and Reinforcement Learning (RL). When working with real-world scenarios, more things have to be taken into account, such as unpredictable changes [3], resilience and adaptability necessity [4] and safety operations [5], [6], among many others.

Although applying RL to a robot in real life introduces new challenges, it has already been researched [7]. In this context, Multi-Agent Reinforcement Learning [8] becomes an important area of focus. This technique naturally introduces the challenge of not only the need to adapt to the environment but considering the other agent actions. Despite this unique challenge, being able to make multiple robots cooperate with each other to accomplish a task would advance our capabilities in various fields, ranging from autonomous vehicles to collaborative robots in manufacturing, warehouses, and healthcare.

## Historical overview

[ TO BE FILLED]

## Identification of the problem

As previously mentioned, despite the extensive research conducted on RL and MARL in virtual environments and in some cases in real-life scenarios, there are still quite a few challenges opened and waiting to be solved, like a lack of sample efficiency, setting goals and specifying rewards in dynamic environments, generalization to new and different tasks and data collection without human supervision [4].

Among these ongoing challenges, one of the main issues is the **reality gap** that is introduced when trying to implement a Reinforcement Learning policy (algorithm) learned in a simulation into the corresponding real-life agent. Addressing this particular problem can help solve other issues as well, like diminishing the number of samples needed, saving time, mechanical wear and consequently, money.

## Rationale

The amount of time to invest in this project is very limited, therefore a study [9] has been taken as a reference point. This is to have a benchmark for comparing simulation results, defining the task, and narrowing down the choice of algorithms to be implemented.

This project aims to reduce the reality gap, also called sim-to-real transfer problem, with a specific task involving multiple agents (MARL) and it is divided in three main parts:

1. Simulation of the environment and robots using Gazebo [10] and ROS 2 [11] for control and algorithm implementation.
2. Real implementation of the environment and robots, mimicking as much as possible Gazebo’s simulation, while trying to minimize the reality gap.
3. Discussion, comparison, and analysis of the results.

Improving the sim-to-real-transfer in a muti-agent scenario can help the community prosper and create or improve applications where robots must cooperate with each other to solve a problem.

## Objectives

The academic goals for this project include the implementation of RL algorithms in a simulated environment as well as in a real-case scenario and the construction of two robotic arms that cooperate with each other.

Besides the research goals, improving programming abilities with Python, C, and other languages, getting familiar with software and frameworks such as Linux systems and ROS 2 as well as applying electronics, robotics and AI knowledge learnt while pursuing my bachelor are the main goals of this project. Some questions that I want to answer and may help the reader are the following:

* Is it possible to reduce the reality gap using the existent techniques [12], [13]?
* Can I build two working robots that communicate and cooperate with each other using MARL algorithms?
* How much can the reality-gap can be reduced?

## Scope and limitations

While it would be ideal to fabricate the robot via a manufacturing partner and use generative design techniques, the constrained timeframe of this project regrettably does not allow such approaches.

That is why the scope of this project is building two functional 5 joints robotic arms that interact with each other to successfully perform a task, recreating the environment in a physics simulator and in addition, implement MARL algorithms in both real-case and virtual-based while trying to minimize the reality gap.

Having a limited amount of time and resources also makes certain aspects of the project inevitably limited. This is particularly evident while training the MARL models not only in the simulation but more notably in the real-world application. The challenge extends beyond the initial development of algorithms, encompassing optimization and improvement when receiving new data and feedback.

## Overview of methodology and resources

The study [9] has been chosen as a pipeline for this project, since it provides a specific task for a MARL case scenario application and therefore, the same physics simulator will be used. In this case they used Gazebo which is mor supported in Linux Ubuntu OS. This software will be thoroughly explained during all the project and in further sections.

Python, C++, and Visual Studio Code [14] will be used to develop the models and algorithms due to the richness of libraries and resources they provide [15], [16]. To be able to communicate Python in an organized and efficient way, Robot Operating System 2 (ROS 2) will be used as a bridge. ROS 2 will also be deeply commented on its own section.

Regarding the implementation of the robots, the hardware control will be an ESP32, the PCA9685 to control all servomotors (MG996R) and diverse sensors such as MPU6050 and HSCR04.

## Organization of the Study

# Frameworks and Resources

This project is built upon a custom system where different programming languages, frameworks, and software coexist, communicate, and collaborate. In this chapter, we aim to familiarize the reader with the resources used in the project. The explanation will be detailed but not overly extensive, allowing the reader to understand how the system works and why these specific components are chosen. If additional specific and technical details are required to understand a concept, refer to Appendix I.

Since an overview of the frameworks and resources used can be found in “[*Overview of methodology and resources*](#_Overview_of_methodology)”, below we are going to delve right into each framework description.

## ROS

ROS or Robot Operating System [11] is a useful framework to control robots both in simulation and real life. To avoid confusion, there is a need to clarify that ROS is a set of open-source software frameworks, not an operating system, as one may think at first instance. Before continue explaining ROS features, it needs to be said that in this project ROS 2 [17] is being used, specifically ROS 2 Humble, due to its compatibility with other software of interests such as Gazebo.

The use of ROS 2 in this project comes from benefitting from several powerful features which makes communication between certain software and frameworks more convenient. Among these features, in hierarchical order, **workspaces**, **packages** and **nodes** will be used to organize different parts of the algorithm.

All recent ROS2 versions (Foxy, Galactic, Humble and Iron) rely on workspaces, which is a ROS term for the location of the development space on the system. This is quite convenient as it allows different ROS2 distributions to run on the same computer, switch between them and keep track of all the ongoing changes in a project. In addition, the workspace is organized in packages. Packages offer a controlled way to separate and execute files that are usually stored into the same package if similar functionalities are presented.

Now one may ask, but how can the user control robots or agents and communicate with other software, such as Gazebo, using this software? The answers to this question are **nodes**, **topics,** and **services**.

A **node** is a core concept in ROS 2 and an important element of what is referred to as the “ROS 2 Graph”, which is a network of ROS 2 elements processing data in a collective way at the same time. Each node is and should be responsible for a single, modular purpose, (e.g., controlling joint motors or publishing sensor data from an ultrasonic sensor). Although, how can the nodes communicate and share data between them and other frameworks?

This is where **topics** and **services** come into the system. Topics serve as a communication mechanism within the ROS 2 Graph. Nodes can publish data to a topic, and other nodes can subscribe to that topic to receive and process the information. This publisher-subscriber model facilitates a decentralized and modular architecture, allowing nodes to communicate seamlessly without direct dependencies on each other. Topics are crucial for real-time data exchange in robotic systems, enabling coordination between diverse components, such as sensor data input and motor control commands as explained earlier.

In contrast, services offer a request-response pattern of communication. A node providing a service advertises its availability, and other nodes can send requests to it. The service-providing node then processes the request and sends a response back. This mechanism is particularly useful for scenarios where a specific task needs to be executed on-demand, such as querying a sensor for specific information or requesting a robot to perform a specific action.

A diagram of a service

Description automatically generated

**Figure 1.** Multiple nodes sharing information via topics and services.

A full robotic system, as in this project, contains multiple nodes working in concert. In ROS 2, a single package can contain multiple nodes written in different programming languages such as C++ or Python.

A very useful tool to visualize active nodes, topics and services is *rqt\_graph*. With this command it is possible to see in a graph real time changes and connections between actives nodes in a project. Below, there is an example with several active nodes that share data between them and Gazebo, the simulation software used.

**A diagram of a diagram

Description automatically generated**

**Figure 2**. Rqt\_graph showing ros2 nodes connected through topics and bridge.

## Gazebo

Since the aim of this project is to minimize the reality gap, an error that is introduced when transitioning from simulations to real-world applications, Gazebo Sim emerges as a core tool. Gazebo, an open-source robot simulation software, plays a crucial role in bridging this gap by offering a realistic and dynamic environment for testing and refining robotic algorithms and control systems before their implementation on physical hardware.

Not only can Gazebo generate accurate 3D simulations but incorporate physics engines that faithfully replicate the dynamics of various robotic platforms. This realistic and dynamic simulation allows researchers to, among other things, train algorithms and test its performance across a wide variety of scenarios, ensuring a more accurate representation of real-world challenges.

In line with the project's goal, Gazebo seamlessly integrates with ROS2. This integration becomes extremely useful when deploying ROS 2 nodes within Gazebo simulations, providing a virtual environment for debugging and refining algorithms before they are executed on physical robots. The symbiotic relationship between Gazebo and ROS 2 enhances the overall capabilities of both platforms.

Furthermore, it is possible to construct multi-robot environments and the interactions between numerous robotic agents, which works perfectly in the current case of a MARL agent. On top of this, the simulation software supports a wide array of sensors, including cameras, lidars, and sonar.

Given these considerations, Gazebo stands out as a highly suitable choice for the current project. Its consistent integration of ROS 2, dynamic simulation capabilities, support for multi-robot (MARL) environments and diverse sensor simulation make this software a valuable tool in narrowing down the reality gap [12].

## ESP32

When it comes to control, processing, and communication between software, hardware, sensors, and actuators, the ESP32 microcontroller stands as one of the best options, if the computational cost is not too high. In this case the algorithm that has to be implemented does require more resources than the ESP32 can provide, that is why other choices such as Raspberry Pi were considered and, if this project is further developed, it is recommended to use one.

Therefore, the ESP32 is used as a middle component between sensors and actuators and a PC. The main computer takes the responsibility of training the model and sending/receiving values to the microcontroller as needed.

In “[Implementation](#_Implementation)” the reader will find how the ESP32 is being used, besides a description of the communications, sensors, and actuators. For further information about ESP32 (e.g., pinouts, features, capabilities) refer to Appendix I (“[ESP32](#_ESP32)” section).

## C++ and Python

Python and C++ play distinct roles in this project. Python, known for its user-friendly design, excels in easing the communication between devices and frameworks. Leveraging its Object-Oriented Programming (OOP) tools, extensive libraries, and concise syntax, Python contributes to the organized and readable development of Deep Learning models. Notably, Python's strengths lie in its ability to create sophisticated models with ease, thanks to powerful Machine Learning libraries such as PyTorch and TensorFlow. These libraries stand out as the primary reasons for choosing Python to develop Deep Reinforcement Learning in this project.

On the other hand, C++ takes the lead when it comes to microcontroller programming, specifically for the ESP32. While C boasts simplicity and computational efficiency, C++ elevates low-level programming with robust OOP features, akin to Python's strengths [18]. The choice of C++ is reinforced by its widespread use, comprehensive documentation, and strong support, establishing it as a reliable option for programming the ESP32.

In summary, the deliberate use of Python and C++ in their specialized roles enhances the project's efficiency. Python's user-friendly design and powerful ML libraries excel in Deep Reinforcement Learning, while C++ proves to be exceptional for microcontroller programming. For detailed and further information on libraries and methods, refer to Appendix I.

# Model design

To further understand how RL algorithms work, there is a need to first understand the Markov Decision Process. A reinforcement learning agent is designed to make a series of sequential decisions through interactions with its surroundings [19]. This environment is usually structured as an infinite-horizon discounted Markov decision process, or in a simpler way, MDP. See *Definition 1.*

**Definition 1.** A Markov decision process is defined by a tuple (S, A, P , R, γ), where S and A denote the state and action spaces, respectively; P : S × A → ∆(S) denotes the transition probability from any state s ∈ S to any state s’ ∈ S for any given action a ∈ A; R : S × A × S → R is the reward function that determines the immediate reward received by the agent for a transition from (s, a) to s’ ; γ ∈ [0, 1) is the discount factor that trades off the instantaneous and future rewards.

Parallel implementations of single-agent RL scale well on large multi-agent systems although it suffers from issues such as learning stability due to a continuous-changing environment [20] that each agent faces, therefore, to approach this challenge all agents should be jointly trained in a distributed manner.

Moreover, as previously said, one of the project’s goals is to reduce the reality gap from simulation to real-case scenario when applying a MARL algorithm. Since [9] is being taken as a reference to design the model and environment and to avoid stability issues in learning we will be using a custom deterministic policy model called Deep Deterministic Policy Gradient (DDPG), due to its promising results in other related works [21], [22], and its good capabilities dealing with continuous spaces similar to those in this project.

## Model definition

Deep Reinforcement Learning has been successfully applied in [4], [22], [23], mostly in single-agent control problems. When it comes to multiple-agent control, managing the large number of degrees of freedom, heterogeneous physical constraints and partial or asymmetric observations for different robots, demands scalability. That is why DDPG is used.

The DDPG agent can handle many inputs and outputs in its networks. It is also possible to build several agents with different reward functions which are then coupled and aligned to the goal of the cooperative task. DDPG is a multi-agent reinforcement learning policy, and the defined system for a MARL algorithm is, as Markov Decision Process (MDP) dictates, the tuple (1).

(1)

Where:

* The set representing the number of agents.
* The set representing all possible states of the environment.
* The set representing available actions to the agent .
* The state transition function .
* The policy for the agent .
* The accumulative reward of an agent (also called Value function).

In this type of model, each agent aims to maximize the value function with the starting state at time as shown in (2).

(2)

Where:

* is the expected value taken over a random value which is the sum of discounted rewards.
* is the discount factor that weighs the future rewards.
* is the reward received by the agent at the time *.*

While it is true that [9] provides the definition of an algorithm, not code is provided, making it difficult to exactly replicate their model. That is why a custom version of a MARL algorithm using a DDPG policy is created.

## Custom MARL with DDPG

First, it needs to be clear that this model is still under development. That does not mean that it is not working, but that it is still extremely scalable, since its correct completion would solve problems such as generalization, reality gap and other main issues regarding RL nowadays.

Let the custom model be **CDDPG** (Custom Deep Deterministic Policy Gradient) for ease of use and reference. Now, CDDPG is working with one single agent that can control as many robots (or actuators) as the user commands due to its variable input and output network dimensions. To further understand the features of CDDPG, below there’s an explanation about what are the elements of the tuple (1), how are they defined, and how does the system come into place both in simulation and reality.

### System structure

To apply CDDPG in the simulated environment multiple steps will be followed consequently. The first step is to acquire the real-time state of each robot as well as the state of the object to be manipulated. The observed current state () will be sent to the policy () where an algorithm will decide which action to take next. Once the action () is executed, the environment will respond with a new state (). Based on the action’s () effectiveness, the reward function () will provide a reward as a scalar value. Finally, the agent learns from this data (, , and ) and the cycle is repeated.

While the reader can find the definition of these parameters below, how they are obtained is explained in the [Simulation](#_Simulation) and [Implementation](#_Implementation) sections, since each area uses different mechanisms to obtain states, terminal conditions and rewards.

### States

In the reference paper [9], the state set is defined taking into account only the joints angles and the end-effector global position. Because of the limitations of real-world number of sensors and other drawbacks (see [Integration drift](#_Integration_drift) in Appendix I), using only two parameters is a good way to go and a good beginning. It is also true that the more parameters or states are defined, the more precise will be the simulation, with a higher computational cost as well. This may be suitable for other applications although not so much for the current approach since one of the aims of this project is to minimize the reality gap, being the number of sensors in real world very limited due to lack of space, dynamics, number of samples and money expenses.

These reasons point to using the fewer and most critical number of sensors that will become the states of the robots. Therefore, taking [9] as a reference the states set will be defined as (3).

(3)

Consequently, there is a need to find a way to extract each joint angles () and both end-effector global coordinates ().

### Actions

The environment is updated and provides new states as a consequence to the executed actions, which change the way the world is perceived by the agent, both in Gazebos’ simulator and reality. The actions to be taken are derived from the policy, the custom DDPG in this context. To enhance realism, these actions are expressed as torque values applied to each joint servomotor.

Therefore, the available actions to the agent are defined as the set in (4). Here, represents the torque applied to the joints at time for the joint. The set includes all possible torque values that agent can choose from the available actions at a specific time.

(4)

While being true that the outcome is the movement of the robots’ joints, it is essential to note that in the Simulation the actions are applied in a different way than in reality. Detailed explanations of how actions are applied on each case can be found in their respective sections, “[*Actions in Simulation*](#_Actions_in_Simulation)” and “[*Actions in Reality*](#_Actions_in_Reality)”.

### Policy

The policy (5), takes the world state and produces a set of actions (4) using a specific set of instructions or, equally said, an algorithm. In this case, the algorithm chosen is a custom version of Deep Deterministic Policy Gradient due to its good results in other works and more important, the ability to handle continuous spaces and large state dimensions on its networks.

(5)

The policy remains constant across both real-world and simulation applications. It is worth explaining its structure, components, and the rationale behind its customized implementation. Following, a breakdown of the policy and exploration of its system is provided.

#### CDDPG Architecture

Deep Deterministic Policy Gradient is based on two main neural networks called Actor and Critic, similar to the SAC [24] algorithm. The Actor provides actions while the Critic, as the name indicates, criticizes them, and gives feedback.

Furthermore, to optimize the process two sub-networks with the exact same architecture as the Actors’ and Critics’ but are slowly updated, are used, namely Target Networks. Moreover, a Replay Buffer has been added to train offline, taking batches of data and feeding them to the networks to update them. Here is how it works behind the scenes:

* Actors’ Network

The Actors’ network is the main component of the model, which follows the equation (5). It is responsible for the generated actions based on the received states from the environment by maximizing its loss function (11).

(6)

The loss function for the Actors’ network aims to maximize the Q-Value that the Critics’ network produces to maximize the expected cumulative reward, which is the expected reward for taking a particular action in a given state following a specific policy. Therefore, the Actor objective is to produce actions that lead to bigger Q-Values from the Critics’ network or, in other words, to produce actions that take the robots closer to their goals.

* Critics’ Network

On the other hand, the Critics' Network evaluates the actions chosen by the Actor estimating a Q-Value which represents the expected cumulative reward associated with those actions. The network updates itself by minimizing the Critic Loss function (12) which is computed using the mean squared error (MSE) between the estimated Q-Values () and the target Q-Values (). How the target Q-Values are obtained is explained in the “[*Reward Function*](#_Reward_function)” section.

(7)

In summary, the Critics’ Network contributes to the training of the Actor by providing feedback on the chosen actions while the associated loss function guides the optimization process, leading to more accurate Q-Value estimation and, ultimately, improving decision-making by the Actor.

* Target Networks

Target networks have the same architecture as Actor and Critic do. They are exactly the same although they are updated slowly to provide more consistent target values, helping stabilize the learning process, reducing the potential for divergence and improving overall convergence. For more information about these networks, refer to the [GitHub](https://github.com/BakiRhina/Reality-Gap-reduction-TFG) repository where this project is located, in the “*\_\_init\_\_()*” method of the “*Implementation/Inference/sub\_modules/ddpg.py*” class. The same file has been added to the [Appendix II](#_Appendix_II).

* Replay Buffer

The replay buffer stores and manages past experiences allowing the agent to learn from a more diverse set of data and can be defined as follows. If the tuple is the definition of an experience, then let the Replay Buffer be defined as a finite-size memory set that stores a collection of size of . See (14).

(8)

(9)

It is widely used in off-policy reinforcement learning algorithms since it provides several advantages and solves certain issues. Firstly, it breaks down temporal correlations, which in reinforcement learning happens due to the similarities between sequential experiences. That is why allowing the agent to sample random past experiences mitigates this issue and reduces the risk of overfitting to recent events.

Secondly, the replay buffer enables the agent to learn from a batch of experience which enhances training efficiency by making better use of parallel computation.

Finally, the environment may change over time leading to a non-stationary learning problem. Thanks to the replay buffer, the agent can now learn from a diverse set of experiences which help him better adapt to environmental changes while maintaining stability.

To sum up, the Replay Buffer plays an important role in off-policy reinforcement learning algorithms such as DDPG, contributing to stability, efficiency, and improved generalization during the learning process.

### State transition function

The state transition function , is not explicitly implemented in the provided CDDPG due to the challenges posed by continuous state spaces as Gazebo and reality are. Unlike discrete spaces where transitions can be explicitly defined, continuous spaces involve an infinite number of possible states, making it impractical to represent and compute transition probabilities. As a result, the CDDPG algorithm adopts a model-free approach, focusing on learning policies and value functions directly from interactions without modeling the exact state transition dynamics.

### Termination state

The terminal state is a crucial concept that defines the conditions under which an episode concludes. Once the system reaches a terminal state, the ongoing episode ends, and the environment will be reset to its initial state for the start of a new episode. The design of the terminal state is essential for shaping the learning process and achieving specific goals in the training of an agent and can be triggered by the fulfillment of one or many conditions, such as task completion, fatal states, safety concerns, learning process stagnation or run out of time.

Since all conditions expressed before are important enough to reset the simulation if accomplished, the design of the terminal state will be the following state. Let *done* be a Boolean variable triggered by the veracity of (10), (11) or (12).

(10)

(11)

(12)

(13)

In (10), is the summation of each velocity of the joint therefore it becomes *True* when the velocity remains close to 0 for an incremental period indicating that the robot has stopped moving and it has either reached an optimal point or a local minimum. Similarly, equation (11) evaluates to *True* when the summation of the difference of reward values in an incremental period of time (, where is a custom variable for additional reward checks) is 0.

In the Python implementation of both equations (10) and (11), a margin control variable is utilized. This variable ensures that if the velocity or reward condition is in proximity to 0 but not precisely 0, it is still considered true.

Moreover, inequation (12) detects if the orientation pitch, yaw, or roll () of the object is changing too much, leading to undesired object placement, or indicating the object has fallen. In this case is being used as a threshold that can be customized by the user.

Finally, the variable *done* becomes true when any of the conditions is also true, resulting in the termination of the current episode and the initiation of a new one.

### Reward function

The reward function is the core of the reinforcement learning algorithm. It shows the agent which path is to be followed to reach a certain goal, by providing rewards, which evaluate the actions’ impact on the environment. Since both simulation and reality are continuous spaces, the reward will also be continuous. It needs to be clear that not only the option of giving rewards when certain states are reached exist, but it is recommended to implement it therefore having a more rich and comprehensive learning experience.

The main constituents for the reward task, in both simulation and reality, are defined as, firstly, those that capture the object displacement from target, (13) and (14) respectively for both robots, and secondly that which captures the object posture deviation (15). Here the Euler angles pitch, roll and yaw are defined as , respectively and is the location of the end-effector of the robot .

(13)

(14)

(15)

To encourage proximity to the object, the distance reward is based on the hyperbolic function , so as the robot gets closer to its target, the reward increases proportionally. Moreover, some parameters has been added to enhance flexibility and control over each reward function weights. For instance, the higher the value of the more important will be the condition. A value of 1 nullifies the effect of the parameter.

Note that the reward value is negative, implying that the agent is motivated to maximize its cumulative reward. In this context, achieving a perfect score corresponds to obtaining a reward of 0, as the agent aims to minimize the negative values and move towards optimal performance. Thus, in such scenarios, the goal is to approach or reach a cumulative reward of zero, indicating successful task completion or optimal behavior in the given environment.

The reward function can be as complex as one can imagine. Actually, it is advised to use a rich and diverse reward function structure [25], [9]. The one used in this project is an example of what can be accomplished with few sensors, and it can be further scaled and improved. The mathematical notation of the complete reward function is defined as follows:

*RF:* (16)

### Value function (expected cumulative reward)

The value function, also called expected cumulative reward, is a fundamental concept in Reinforcement Learning. It represents the expected long-term reward an agent can achieve from a given state, considering its current policy for selecting actions. It guides the agent in making decisions to maximize cumulative rewards over time.

The value function is often defined using the Bellman equation, which expresses the relationship between the value of a state and the value of its neighboring states. The Bellman equation is a recursive formula that decomposes the value of a state in two components: the immediate reward based on the current state and the expected reward of the following states discounted by a factor .

The Value function has been previously defined (2), but it is showed below for convenience.

(2)

Where:

* is the expected value taken over a random value which is the sum of discounted rewards.
* is the discount factor that weighs the future rewards.
* is the reward received by the agent at the time *.*

Twisting the discount factor gives weight to future rewards and current rewards. Depending on the task, it is useful to value current rewards more than subsequent rewards and vice versa. For instance, in scenarios where immediate outcomes significantly impact the overall performance, a higher discount factor may be chosen to prioritize current rewards. Conversely, if long-term considerations are crucial, a lower discount factor may be employed to emphasize the significance of future rewards.

The choice of discount factor often falls within the range of 0 to 1, exclusive. Commonly used values include:

* Close to 1: When the discount factor is close to 1, it implies a strong consideration for future rewards. This setting is suitable for tasks where long-term consequences carry significant importance.
* Close to 0: When the discount factor is close to 0, it emphasizes immediate rewards, downplaying the impact of future rewards. This can be useful in tasks where short-term gains are more critical.

# Simulation

Gazebo [10] will be used as the physics simulator due to the use of this software in [9]. Specifically, Gazebo Fortress will be used, due to its compatibility with the software robot control, ROS 2 Humble. Newer versions of Gazebo and ROS 2 (such as Gazebo Garden and ROS 2 Iron, respectively) have been used, although they have presented many versions’ incompatibilities and errors that their use was ultimately avoided.

Since [9] does not provide the resources for the simulation, this has been built from scratch, only taking as a reference the environment setup shown in the figure below.



**Figure 3**. Environment setup reference [9]

To be able to represent and run a simulation in Gazebo two things are needed. Firstly, an *.sdf* file that describes the world (physics, models, plugins) and secondly an external software dedicated to controlling the simulation.

As previously said, the control software used is ROS 2, which allows the user to communicate with Gazebo through Python programmable *Nodes*. Nodes are the place where control and receiving and publishing data is happening.

[ … ]

## Simulation design

We can build a robot, also called a model or an agent, in Gazebo using the provided GUI or an Description File, such as SDF or URDF. Using the GUI provides faster but limited results while building a world and models from description files allows more control over the system and better understanding about what is going on behind the scenes.

Gazebo uses SDF (more supported by Gazebo’s community) and URDF (more supported by ROS community) files to design environments and models. The choice between SDF and URDF files for the simulation design recalls upon the fact that SDF is more focused on robot simulation while URDF is centered around robot modeling, control and can only describe one robot/model per file. Since the aim of this project is to simulate a multi-agent environment, SDF files will be used instead of URDF.

An SDF file allows us to describe objects and environments for robotic simulations, visualization, and control. The basics structures of an SDF are **world**, **physics**, **plugins**, **GUI**, **light** and **models**. Refer to Appendix I “[*SDF files*](#_SDF_files)” for a deeper and detailed explanation about these files.

## Simulated observations

Since the nature of the simulated and real-world environments is different, the way states, rewards, and actions are obtained and processed varies considerably. To enhance clarity, the “[*Custom MARL with DDPG*](#_Custom_MARL_with)” section provides explanations and general definitions for these terms, offering a comprehensive understanding of the system in both contexts.

This section provides an overview of the simulation system's functioning, emphasizing the seamless communication between Gazebo and ROS 2 to ensure robust data flow and adherence to best practices. For detailed technical information and specific details regarding the communication protocols employed, please refer to Appendix I titled “[*Communication between ROS 2 and Gazebo*](#_Communication_between_ROS)”.

### States

As explained in [States](#_States) section, the states set at specific time () is defined by the tuple (3). How to obtain each joint angles () and both end-effector global coordinates () from Gazebo simulator is explained below.

(3)

To obtain these values, several components are required from both Gazebo and ROS 2. Firstly, the “*PosePublisher*” and “*JointStatePublisher*” plugins for Gazebo are utilized, which provide the General Coordinates () and a Quaternion (, see *Definition 2*) describing the orientation of each link and the angle in radians for each joint, respectively. These plugins are also defined below to provide a visualization of the communication.

      <plugin

        filename="libignition-gazebo-pose-publisher-system.so"

        name="ignition::gazebo::systems::PosePublisher">

        <publish\_link\_pose>true</publish\_link\_pose>

        <update\_frequency>0.5</update\_frequency>

      </plugin>

      <plugin

        filename="libignition-gazebo-joint-state-publisher-system.so"

        name="ignition::gazebo::systems::JointStatePublisher">

      </plugin>

Additionally, a ROS 2 node is necessary, subscribing to the topic where Gazebo publishes the *PosePublisher* data. Lastly, a bridge between ROS2 and Gazebo is established using the ros\_gz\_bridge (for more information about ros\_gz\_bridge refer to Appendix I, “Bridge between ROS 2 and Gazebo”), facilitating data exchange between these systems. Then, assuming that *'segment4\_1'* and *'segment4\_2'* represent the grippers of the respective robots, their global coordinates, and quaternions () can be directly obtained from the simulator.

**Definition 2.** A quaternion is a mathematical concept [26] that extend complex numbers first described by Sir William Rowan Hamilton in 1843. It is composed by four components, one real part and three imaginary parts, and can be written in the form . It is used in different fields, robotics among them, to represent three-dimensional rotations and orientations since it provides certain advantages over other methods such as Euler angles.

In this case quaternions are used instead of Euler notation to avoid ***gimbal lock*** [27] and due to the fact that they are more compact and computationally efficient than rotation matrices.

### Actions

Actions are produced by the Actors’ Network and take the form of torque values directed to each joint servomotor. The communication between ROS 2 and Gazebo is made using the plugin “*JointController*”. For instance, below you can see the plugin definition of the base joint in the *.sdf* file.

      <plugin

        filename="libignition-gazebo-joint-controller-system.so"

        name="ignition::gazebo::systems::JointController">

        <joint\_name>joint0\_1</joint\_name>

        <topic>/arm/joint0\_1/wrench</topic>

      </plugin>

The plugin tells Gazebo that the topic “*/arm/joint0\_1/wrench*” will be used to receive float data values, representing torque forces, to control the joint named “*joint0\_1*” from the model. To ensure good and efficient communication a bridge from ROS 2 to Gazebo is necessary. The bridge command can be defined as:

ros2 run ros\_gz\_bridge parameter\_bridge /arm/joint0\_1/wrench@std\_msgs/msg/Float64]gz.msgs.Double

Where “*/arm/joint0\_1/wrench*” is the topic where Gazebo is subscribing and ROS 2 is publishing data, and both “*Float64*” and “*Double*” are the data type definitions for ROS 2 and Gazebo, respectively.

In summary, the simulation system involves obtaining states from Gazebo, forwarding them to the policy, and receiving float values representing torque forces as a result. These torque values are then transmitted through a ROS 2 topic back to Gazebo. The “*JointController*” plugin in Gazebo interprets these values, enabling the manipulation of the robot's joints to execute the desired movements. This closed-loop communication loop facilitates the integration of policy-based control into the simulation environment.

## Simulation results

To evaluate the simulation behavior 3 main components have been considered. First and more important, the reward value over episodes until a termination condition is met. This metric is crucial when evaluating reinforcement learning models so as it shows in a very visual and concise manner if your agent is learning and how fast it is doing so.

Secondly and thirdly, the Actor and Critic network losses, respectively. Although other metrics could have been used, these provides a deep understanding of the behavior of the main networks which can be analyzed and improved over iterations.

The graph shown in ***Figure 4*** presents the reward values accumulated over multiple episodes and the other metrics until a termination condition is met. This graphical representation offers a visual and concise depiction of the learning progress of the agent. A rising trend in the reward graph indicates positive learning outcomes, while fluctuations or a plateau may suggest challenges in the learning process. This initial graph was obtained with a plain model, without twisting hyperparameters, adding the replay buffer or dropout.



**Figure 4.** Graph from a plane DDPG policy model

Here is clear that the model, although it is not too bad, it is also not learning as it should be. The accumulated reward value is constant during the whole episode, creating a plateau which shows a low ratio of improvements. It is also interesting to notice that, while the Actors’ loss is behaving as expected since its work is to maximize the expected cumulative reward (the Q-Value), the Critics’ loss value converges to 0 incredibly fast, which shows that there might be an overfitting issue with the Actors’ network.

To overcome these issues, the replay buffer, explained in the previous section, was added. Looking at ***Figure 5*** here it is clear that adding the replay buffer has improved the learning process. Although the Critics’ loss is still converging quickly due to a possible overfitting issue, the reward value has clearly improved, being closer 0 in less steps. In this case, the episode was terminated due to reaching its objective, finding the object. Although this is good news, the reward has lots of oscillations and the Critics’ loss, as said before, is probably overfitted.

A graph of a person with a beard

Description automatically generated with medium confidence

**Figure 5.** Reward and loss values after adding a replay buffer.

Finally, a dropout was incorporated into all layers of both networks. Dropout is a technique that enhances exploration while preventing overfitting. This is achieved by randomly deactivating (dropping out) some neurons during training, effectively reducing their influence on the model. The dropout probability, converted into a hyperparameter, determines the likelihood of a neuron dropping out. This hyperparameter can be fine-tuned and experimented with various values to enhance the overall behavior of the model.

In ***Figure 6***, the final results with dropout and replay buffer are shown. It is clear that the Critics’ network is no longer overfitting and the Actors’ is behaving as expected. Despite that, the reward value, although consistently increasing, is never reaching the ideal point. This may be due to the continuous reward function structure designed and it would be interesting to implement sparse rewards together with the continuous reward and observe if this improves the model.

A line graph with different colored lines

Description automatically generated

**Figure 6.** Metric values with dropout and replay buffer.

It needs to be clear that the reward function can be further improved and finetuned in future works by adding different mechanisms such as sparse rewards, different methods or specific rewards for each agent as some works advice[9].

In conclusion, it has been proven that the more the model is tuned, the better results it provides, as it should be. Both replay buffer and dropout techniques helped improve the reward value and mitigate issues like overfitting in the Critics’ network.

It must be said that many different hyperparameter values have been tested in between the results, and the finetuning of these is left to future works or other research that wants to play with them. All hyperparameters are programmed to be easily tuned, always defined in the initialization method of each class. Finally, for more information regarding the code, hyperparameters and methods implemented, refer to the [GitHub](https://github.com/BakiRhina/Reality-Gap-reduction-TFG) repository where this project is stored.

# Implementation

This is the second part of the project where the same policy is applied for continuous training, evaluation, or fine-tuning of the pretrained model within the simulation environment. Ideally, the real environment should be as similar as possible to the simulation, however, achieving perfect conditions poses several challenges. Practical constraints, such as limited space for sensors, economic considerations, and the influence of cost on the size and precision of sensors and motors, introduce inherent limitations. Additionally, time is a critical factor to consider. Recreating a precise digital representation of a real and custom robot is a time-intensive process. It is essential to acknowledge these constraints and recognize that the actual operating conditions of the real environment will play a major role in minimizing the reality gap between the simulation and implementation.

This section details the environment design, the mechanical and electrical materials used, and the process of acquiring and transmitting observations to the algorithm. Furthermore, challenges and limitations encountered during material selection and implementation will take place, offering valuable insights into the decision-making process and adaptations made to ensure project success.

## Material

The main focus of this project is to keep learning and applying the knowledge gained during the pursuit of a Bachelor's degree in Electronics. That is why the robots and their electronics control are customized. Although this customization process takes more time than using pre-built control robots, it aligns perfectly with our goal of hands-on learning, applying theoretical knowledge in practical scenarios, and actively learning from any mistakes or challenges that may arise.

This section meticulously outlines the selection and utilization of key materials crucial for the project. This encompasses the ESP32 microcontroller, HCSR04 ultrasonic sensor, MPU6050 inertial measurement unit, PCA9685 servo driver, MG996R servo motors, and aluminum components for the robots’ structure. Each material serves a distinct purpose in shaping the real-world environment and bridging the gap between simulation and implementation.

## Esp32

The core controller election has several constraints, and it depends on the aim and structure of the project itself. In the first instance, Raspberry Pi was considered as a main option to store and process the RL algorithm and control the robots at the same time. While this is a valid approach, factors such as cost, learning curve and the fact that the number of states to be processed are below 20, a decision was made to find a better fit for the project requirements.

Consequently, both Arduino UNO and ESP32 emerged as strong candidates for the control task, driven by factors such as familiarity gained during the bachelor's degree, user-friendly interfaces, and a good number of GPIO pins that facilitate the control of numerous actuators and sensors. It's worth noting that although the processing capacity of these microcontrollers is not as high as that of the Raspberry Pi, a main computer will be employed to host and process the RL algorithm. This way, the microcontroller acts as a bridge between the robots and the algorithm, receiving data from the robots' sensors and sending signals to the motors for robot movement.

In the end, ESP32 was selected as the preferred choice due to its superior computational capacity, ample memory storage, built-in Wi-Fi and Bluetooth capabilities, and compatibility with programming languages like C++ and μPython. Additionally, the ESP32 boasts a large and active community, providing valuable support and resources for the project's development.



**Figure 7.** ESP-Wroom-32 pinout

The figure above shows the Pinout of the board as well as its main capabilities. For more information and deeper features about the ESP32, refer to Appendix I “[*ESP32 Overview*](#_ESP32_Overview)”.

## MPU6050

The MPU6050, or Inertial Measurement Unit (IMU), is a sensor used to measure and report physical movement. An IMU combines accelerometers and gyroscopes to track both linear and angular movements. Accelerometers measure linear acceleration, while gyroscopes measure angular acceleration or changes in orientation, as one can see in the figures below.



**Figure 9.** MPU6050 rotation axis



**Figure 8.** MPU6050 Pinout

The choice of the MPU6050 for this project is based on several factors. Firstly, it's affordable, making it cost-effective for various applications. Additionally, it's compatible with Arduino systems and has well-established libraries, making it user-friendly, especially for those familiar with Arduino. The MPU6050 has a large community of users, providing ample support and resources for troubleshooting.

Initially intended for determining object orientation, the sensor was also employed in triangulating the positions of objects and end-effectors. To achieve this, the second integral of both angular and linear accelerations is necessary. The mathematical representation of this involves integrating acceleration with respect to time twice.

However, custom calibrations, offsets, and libraries were required to address issues, particularly in measuring angular acceleration. Despite these efforts, there were still slight errors. The major challenge encountered was the integration drift, which will be explained in depth in the Issues section. Due to this problem, the use of IMUs for object location and end-effector triangulation was rejected. Instead, and for simplicity, the ultrasonic sensor HSCR04 is used, as a replacement for the MPU6050, which will be explained in the next section.

It is worth noticing that the MPU6050 is still being used to detect the object posture deviation and trigger termination conditions or other methods in the algorithm.

Hscr04

Pc9685

Mg996r

Aluminum parts

## Communication

### Serial

Sending the values provided from the DDPG agent to its respective motors is being done through Serial port, due to its proximity to the main computer and the ease of use that both Arduino IDE and Python provide. These values are sent as packed bytes representing floats that will be unpacked when they get to the esp32 using *casting pointer* [28].

Casting pointer allows the user to change the interpretation of the bits in a particular memory location, which is useful when receiving a group of bytes that must be reinterpreted as float values, to be further processed and sent as servo motor angle values. See Annex I to find more information about pointer casting.

### I2C

## Real-world observations

### States

### Actions

# Issues

## Integration drift

# Reality Gap

# Results

# Analysis and discussion

# Economic analysis

# Environmental analysis

# Conclusions

# References

[1] B. Osiński *et al.*, ‘Simulation-Based Reinforcement Learning for Real-World Autonomous Driving’, in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, May 2020, pp. 6411–6418. doi: 10.1109/ICRA40945.2020.9196730.

[2] D. Silver, ‘Reinforcement Learning and Simulation-Based Search in Computer Go’, ERA. Accessed: Oct. 29, 2023. [Online]. Available: https://era.library.ualberta.ca/items/12fc0cb8-c990-4759-8fa1-5e12af33116a

[3] S. Padakandla, ‘A Survey of Reinforcement Learning Algorithms for Dynamically Varying Environments’, *ACM Comput. Surv.*, vol. 54, no. 6, p. 127:1-127:25, Jul. 2021, doi: 10.1145/3459991.

[4] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, ‘How to Train Your Robot with Deep Reinforcement Learning; Lessons We’ve Learned’, *The International Journal of Robotics Research*, vol. 40, no. 4–5, pp. 698–721, Apr. 2021, doi: 10.1177/0278364920987859.

[5] J. Garcıa and F. Fernandez, ‘A Comprehensive Survey on Safe Reinforcement Learning’.

[6] S. Gu *et al.*, ‘A Review of Safe Reinforcement Learning: Methods, Theory and Applications’. arXiv, Feb. 20, 2023. Accessed: Oct. 02, 2023. [Online]. Available: http://arxiv.org/abs/2205.10330

[7] X. Huang *et al.*, ‘Creating a Dynamic Quadrupedal Robotic Goalkeeper with Reinforcement Learning’. arXiv, Oct. 10, 2022. Accessed: Sep. 26, 2023. [Online]. Available: http://arxiv.org/abs/2210.04435

[8] L. Busoniu, R. Babuska, and B. De Schutter, ‘A Comprehensive Survey of Multiagent Reinforcement Learning’, *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 2, pp. 156–172, Mar. 2008, doi: 10.1109/TSMCC.2007.913919.

[9] G. Ding *et al.*, ‘Distributed Reinforcement Learning for Cooperative Multi-Robot Object Manipulation’.

[10] N. Koenig and A. Howard, ‘Design and use paradigms for gazebo, an open-source multi-robot simulator’, *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, pp. 2149–2154, 2004, doi: 10.1109/IROS.2004.1389727.

[11] A. Koubaa, Ed., *Robot Operating System (ROS)*, vol. 625. in Studies in Computational Intelligence, vol. 625. Cham: Springer International Publishing, 2016. doi: 10.1007/978-3-319-26054-9.

[12] J. Tan *et al.*, ‘Sim-to-Real: Learning Agile Locomotion For Quadruped Robots’. arXiv, May 16, 2018. Accessed: Oct. 29, 2023. [Online]. Available: http://arxiv.org/abs/1804.10332

[13] W. Yu, J. Tan, Y. Bai, E. Coumans, and S. Ha, ‘Learning Fast Adaptation with Meta Strategy Optimization’. arXiv, Feb. 15, 2020. Accessed: Oct. 29, 2023. [Online]. Available: http://arxiv.org/abs/1909.12995

[14] ‘Visual Studio Code’.

[15] G. Brockman *et al.*, ‘OpenAI Gym’. arXiv, Jun. 05, 2016. Accessed: Oct. 29, 2023. [Online]. Available: http://arxiv.org/abs/1606.01540

[16] T. Developers, ‘TensorFlow’, *Zenodo*, May 2021, doi: 10.5281/zenodo.4758419.

[17] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, ‘Robot Operating System 2: Design, architecture, and uses in the wild’, *Science Robotics*, vol. 7, no. 66, p. eabm6074, May 2022, doi: 10.1126/scirobotics.abm6074.

[18] F. Zehra, M. Javed, D. Khan, and M. Pasha, ‘Comparative Analysis of C++ and Python in Terms of Memory and Time’. Preprints, Dec. 21, 2020. doi: 10.20944/preprints202012.0516.v1.

[19] K. Zhang, Z. Yang, and T. Başar, ‘Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms’, in *Handbook of Reinforcement Learning and Control*, K. G. Vamvoudakis, Y. Wan, F. L. Lewis, and D. Cansever, Eds., in Studies in Systems, Decision and Control. , Cham: Springer International Publishing, 2021, pp. 321–384. doi: 10.1007/978-3-030-60990-0\_12.

[20] P. Hernandez-Leal, B. Kartal, and M. E. Taylor, ‘A survey and critique of multiagent deep reinforcement learning’, *Auton Agent Multi-Agent Syst*, vol. 33, no. 6, pp. 750–797, Nov. 2019, doi: 10.1007/s10458-019-09421-1.

[21] N. Casas, ‘Deep Deterministic Policy Gradient for Urban Traffic Light Control’. arXiv, Aug. 02, 2017. Accessed: Nov. 15, 2023. [Online]. Available: http://arxiv.org/abs/1703.09035

[22] S. Amarjyoti, ‘Deep Reinforcement Learning for Robotic Manipulation-The state of the art’. arXiv, Jan. 30, 2017. Accessed: Nov. 08, 2023. [Online]. Available: http://arxiv.org/abs/1701.08878

[23] I. V. Serban *et al.*, ‘A Deep Reinforcement Learning Chatbot’, *arXiv:1709.02349 [cs, stat]*, Nov. 2017, Accessed: Jul. 01, 2021. [Online]. Available: http://arxiv.org/abs/1709.02349

[24] T. Haarnoja *et al.*, ‘Soft Actor-Critic Algorithms and Applications’. arXiv, Jan. 29, 2019. Accessed: Nov. 15, 2023. [Online]. Available: http://arxiv.org/abs/1812.05905

[25] R. T. Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, ‘Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning’, *Journal of Artificial Intelligence Research*, vol. 73, pp. 173–208, Jan. 2022, doi: 10.1613/jair.1.12440.

[26] J. Voight, *Quaternion Algebras*. Springer Nature, 2021. doi: 10.1007/978-3-030-56694-4.

[27] E. G. Hemingway and O. M. O’Reilly, ‘Perspectives on Euler angle singularities, gimbal lock, and the orthogonality of applied forces and applied moments’, *Multibody Syst Dyn*, vol. 44, no. 1, pp. 31–56, Sep. 2018, doi: 10.1007/s11044-018-9620-0.

[28] S. H. Yong, S. Horwitz, and T. Reps, ‘Pointer analysis for programs with structures and casting’, *SIGPLAN Not.*, vol. 34, no. 5, pp. 91–103, May 1999, doi: 10.1145/301631.301647.

[29] ‘esp32-wroom-32\_datasheet\_en.pdf’. Accessed: Dec. 18, 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32\_datasheet\_en.pdf

# Appendix I

## ESP32 Overview

The specific model being used is ESP-Wroom-32 [29], with the above pinout. The main features used in the project are listed below:

* Wi-Fi and Bluetooth modules.
* 24 GPIO

## SDF files

The following link explains <tags> in an SDF file: <http://sdformat.org/spec>

An SDF file allows us to describe objects and environments for robotic simulations, visualization, and control. The basics structures of an SDF are **world**, **physics**, **plugins**, **GUI**, **light** and **models**. Below there is an example of a basic SDF file defining an empty world:

<?xml version="1.0" ?>

<sdf version="1.10">

<world name="car\_world"> 🡪 World definition

<physics name="1ms" type="ignored"> 🡪 Physics definition

<max\_step\_size>0.001</max\_step\_size>

<real\_time\_factor>1.0</real\_time\_factor>

</physics>

<plugin 🡪 Plugins (libraries)

filename="gz-sim-physics-system"

name="gz::sim::systems::Physics">

</plugin>

<plugin

filename="gz-sim-user-commands-system"

name="gz::sim::systems::UserCommands">

</plugin>

<plugin

filename="gz-sim-scene-broadcaster-system"

name="gz::sim::systems::SceneBroadcaster">

</plugin>

<light type="directional" name="sun"> 🡪 Lights definition

<cast\_shadows>true</cast\_shadows>

<pose>0 0 10 0 0 0</pose>

<diffuse>0.8 0.8 0.8 1</diffuse>

<specular>0.2 0.2 0.2 1</specular>

<attenuation>

<range>1000</range>

<constant>0.9</constant>

<linear>0.01</linear>

<quadratic>0.001</quadratic>

</attenuation>

<direction>-0.5 0.1 -0.9</direction>

</light>

<model name="ground\_plane"> 🡪 Model definition

<static>true</static>

<link name="link">

<collision name="collision">

<geometry>

<plane>

<normal>0 0 1</normal>

</plane>

</geometry>

</collision>

<visual name="visual">

<geometry>

<plane>

<normal>0 0 1</normal>

<size>100 100</size>

</plane>

</geometry>

<material>

<ambient>0.8 0.8 0.8 1</ambient>

<diffuse>0.8 0.8 0.8 1</diffuse>

<specular>0.8 0.8 0.8 1</specular>

</material>

</visual>

</link>

</model>

</world>

</sdf>

In this case the model is just an empty plane. The **model** is where we define our robot and its characteristics such as links, joints, physics (mass, inertia, …) and its main components (tags) are the **model definition** and the **links**.

The <**links>** compose the <**model**> (in a wheeled robot the links will be the chassis, the wheels, sensors, …) and its main components (tags) are **<pose>**, **<inertial>**, **<visual>**, **<collision>**, **<frame>** and **<joint>**. Each of these tags has more sub-tags to define its features. For instance, in the <**visual**> tag we can define the **<geometry>** of the model (shape, size, …) and the **<material>** (colors, render, …).

## Communication between ROS 2 and Gazebo

To be able to communicate Gazebo with ROS 2 and retrieve sensor data, joint positions, robot state and more, we need a bridge between both systems. The “*[ros\_gz\_bridge](https://github.com/gazebosim/ros_gz/tree/ros2/ros_gz_bridge)*” provides a network bridge that allows us to send messages between ROS 2 and Gazebo, limited to certain (but not few) [types of messages](https://github.com/gazebosim/ros_gz/blob/ros2/ros_gz_bridge/README.md). Find below the command definition to initialize a bidirectional bridge where ROS 2 is the publisher and Gazebo the subscriber or vice versa.

ros2 run ros\_gz\_bridge parameter\_bridge /TOPIC**@**ROS\_MSG@GZ\_MSG

Here the first symbol (@) is separating the topic name from the message types and the second symbol can be either:

* @ for a bidirectional bridge
* [ for a bridge from Gazebo to ROS 2
* ] for a bridge from ROS 2 to Gazebo

For instance, take the following command:

ros2 run ros\_gz\_bridge parameter\_bridge /model/vehicle\_blue/cmd\_vel@geometry\_msgs/msg/Twist]ignition.msgs.Twist

In this case, the **node** name is /model/vehicle\_blue/cmd\_vel whereas the **topics** are geometry\_msgs/msg/Twist from ROS2 and ignition.msgs.Twist from Gazebo. The **bridge** is only from ROS 2 to Gazebo as we used ].

Furthermore, to create a good workflow and not miss any detail, here’s a step-by-step system to correctly build nodes and communications between Gazebo and ROS 2.

1. Create parent/child folders in the root directory: mkdir -p ~/<name\_project>/src/
2. Create a remote github repository in ~/<name\_project
3. Add a sdf folder inside src: mkdir ~/<name\_project>/src/sdf\_files/
4. Build ROS2 workspace: cd ~/<name\_project> 🡪 colcon build
5. Create a package with a node (refer to \* and \*\* below):
   1. cd ~/<name\_project>/src/
   2. ros2 pkg create --build-type ament\_python <node\_n> <package\_n>
6. Build the package:
   1. cd ~/<name\_project>
   2. colcon build 🡪 Builds all packages
   3. or colcon build –-packages-select <package\_n>
7. Source the overlay ROS 2 workspace in the new terminal (so the new packages are available)
   1. source install/setup.bash
8. Before making a bridge between ROS2 and Gazebo, let’s run the simulation with:
   1. Ign gz ~/path/to/our/sdf\_file.sdf
9. Now there should be a working workspace, with a package, a node, and a running simulation. To communicate with Gazebo, we’ll have to use ros\_gz\_bridge (explained above) to *publish* and *subscribe* to **topics** and receive or send messages.

\*When adding new *nodes* (*Python* files inside the package) create an instance of them inside *setup.py*

\*\*When adding new dependencies on *nodes*, add them in both *package.xml* and *setup.py*

These commands might be useful to debug topic publishing:

ign topic -l

ign topic -e -t /topic/name

ros2 topic list

ros2 topic echo /topic/name

## Pointer casting

Casting a pointer allows the user to reinterpret the content of a specific memory location, altering its interpretation without changing the actual data or its location. This is particularly useful when receiving a stream of bytes that needs to be reinterpreted as a different data type, such as converting a sequence of bytes into float values.

In the context of this project, pointer casting is employed to instruct the compiler to treat a block of bytes as a float array, enabling the conversion of raw byte data received from the serial port into meaningful float values. This is essential for subsequent processing, such as mapping these float values to servomotor angles for control.

-+----+----+----+----+----+----+-

| | | | | | |

-+----+----+----+----+----+----+-

^~~~~~~~

| byte array

d

```

After Pointer Casting:

-+----+----+----+----+----+----+-

| f1 | f2 | f3 | f4 | f5 | f6 |

-+----+----+----+----+----+----+-

^~~~~~~~~

| float array

D

# Appendix II

## CDDPG Agent

import numpy as np

import torch

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

from sub\_modules.rbuffer import ReplayBuffer

class Actor(nn.Module):

    def \_\_init\_\_(self, state\_dim, action\_dim, actor\_dropout\_p):

        super(Actor, self).\_\_init\_\_()

        self.dropout = nn.Dropout(p=actor\_dropout\_p)

        self.fc1 = nn.Linear(state\_dim, 256)

        print(self.fc1)

        self.fc2 = nn.Linear(256, 128)

        self.fc3 = nn.Linear(128, 64)

        self.fc4 = nn.Linear(64, action\_dim)

    def forward(self, state):

        x = F.relu(self.dropout(self.fc1(state)))

        x = F.relu(self.dropout(self.fc2(x)))

        x = F.relu(self.dropout(self.fc3(x)))

        action = torch.tanh(self.fc4(x)) # normalise [-1, 1]

        return action

class Critic(nn.Module):

    def \_\_init\_\_(self, state\_dim, action\_dim, critic\_dropout\_p):

        super(Critic, self).\_\_init\_\_()

        self.dropout = nn.Dropout(p=critic\_dropout\_p)

        self.fc1 = nn.Linear(state\_dim + action\_dim, 256)

        self.fc2 = nn.Linear(256, 128)

        self.fc3 = nn.Linear(128, 64)

        self.fc4 = nn.Linear(64, 1)

    def forward(self, state, action):

        x = torch.cat([state, action], dim=1)

        x = F.relu(self.dropout(self.fc1(x)))

        x = F.relu(self.dropout(self.fc2(x)))

        x = F.relu(self.dropout(self.fc3(x)))

        value = self.fc4(x) # Estimated Q-Value for a given state-action pair

        return value

class DDPGAgent:

    def \_\_init\_\_(self, state\_dim, action\_dim, buffer\_size = 10000):

        self.actor\_lr = 0.5e-4

        self.critic\_lr = 1e-4

        self.discount\_factor = 0.95

        self.soft\_update\_rate = 0.01

        self.actor\_dropout\_p = 0.5

        self.critic\_dropout\_p = 0.5

        self.batch\_size = 64

        self.replay\_bufer = ReplayBuffer(buffer\_size)

        self.actor\_losses = []

        self.critic\_losses = []

        self.actor = Actor(state\_dim, action\_dim, self.actor\_dropout\_p)

        self.actor\_target = Actor(state\_dim, action\_dim, self.actor\_dropout\_p) # Has the same architecture as the main actor network but it's updated slowly --> provides training stability

        self.actor\_target.load\_state\_dict(self.actor.state\_dict()) # Get parameters from main actor network and synchronize with acto\_target

        self.critic = Critic(state\_dim, action\_dim, self.critic\_dropout\_p)

        self.critic\_target = Critic(state\_dim, action\_dim, self.critic\_dropout\_p)

        self.critic\_target.load\_state\_dict(self.critic.state\_dict())

        self.actor\_optimizer = optim.Adam(self.actor.parameters(), lr=self.actor\_lr)

        self.critic\_optimizer = optim.Adam(self.critic.parameters(), lr=self.critic\_lr)

    #SECTION - Select action

    def select\_action(self, state):

        state = torch.FloatTensor(state)

        action = self.actor(state)

        # remove gradients from tensor and convert it to numpy array

        return action.detach().numpy()

    def update(self, state, action, reward, next\_state, terminal\_condition):

        # Add the real-time experience to the replay buffer

        self.replay\_bufer.add((state,

                               action,

                               reward,

                               next\_state,

                               terminal\_condition)

        )

        # Sample a batch from the replay buffer

        batch\_size = self.batch\_size

        buffer\_batch = self.replay\_bufer.sample(batch\_size)

        # Unpacking buffer\_batch into separate lists for each variable

        buffer\_states, buffer\_actions, buffer\_rewards, buffer\_next\_states, buffer\_terminal\_condition = zip(\*buffer\_batch)

        # Convert lists to NumPy arrays for efficency

        buffer\_states = np.array(buffer\_states)

        buffer\_actions = np.array(buffer\_actions)

        buffer\_rewards = np.array(buffer\_rewards).reshape(-1, 1)

        buffer\_next\_states = np.array(buffer\_next\_states)

        buffer\_terminal\_condition = np.array(buffer\_terminal\_condition).reshape(-1, 1)

        # Convert lists to PyTorch tensors

        buffer\_states = torch.FloatTensor(buffer\_states)

        buffer\_actions = torch.FloatTensor(buffer\_actions)

        buffer\_rewards = torch.FloatTensor(buffer\_rewards)

        buffer\_next\_states = torch.FloatTensor(buffer\_next\_states)

        buffer\_terminal\_condition = torch.FloatTensor(buffer\_terminal\_condition)

        # Buffer data

        buffer\_values = self.critic(buffer\_states, buffer\_actions)

        buffer\_next\_actions = self.actor\_target(buffer\_next\_states)

        buffer\_next\_values = self.critic\_target(buffer\_next\_states, buffer\_next\_actions.detach())

        # BELLMAN EQUATION

        buffer\_target\_values = buffer\_rewards + self.discount\_factor \* buffer\_next\_values \* (1 - buffer\_terminal\_condition)

        # Critic loss for buffer data

        critic\_loss = F.mse\_loss(buffer\_values, buffer\_target\_values)

        # Actor loss for buffer data

        actor\_loss = -self.critic(buffer\_states, self.actor(buffer\_states)).mean()

        # Append losses to the history

        self.actor\_losses.append(actor\_loss.item())

        self.critic\_losses.append(critic\_loss.item())

        # Update networks

        self.actor\_optimizer.zero\_grad()

        actor\_loss.backward()

        self.actor\_optimizer.step()

        self.critic\_optimizer.zero\_grad()

        critic\_loss.backward()

        self.critic\_optimizer.step()

        # Update target networks with soft updates

        self.soft\_update(self.actor, self.actor\_target, self.soft\_update\_rate)

        self.soft\_update(self.critic, self.critic\_target, self.soft\_update\_rate)

    def soft\_update(self, local\_model, target\_model, tau):

        for target\_param, local\_param in zip(target\_model.parameters(), local\_model.parameters()):

            target\_param.data.copy\_((1.0 - tau) \* target\_param.data + tau \* local\_param.data)

## ESP32 | PCA9685 | 110VAC/5VDC Converter

### HCSR-04 | MPU6050

### MG996R Servo Motors