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

This research addresses some of the challenges concerning Multi-Agent Reinforcement Learning. Specifically, it focuses on reducing the reality gap in a MARL environment through the training of a Deep Reinforcement Learning policy. The study also aims to enhance the skills and knowledge learned during the bachelors' degree in electronics engineering while contributing to the academic goal of bridging the sim-to-real-transfer issue. Simulation outcomes demonstrate successful learning by the agent while implementation falls short due to critical oversights in the early phases of the project concerning time management design. Personal goals, including the familiarization of several frameworks such as Gazebo, ROS 2 and Linux, as well as enhancing Python and C++ programming skills, are fully achieved. Despite challenges, resources developed while pursuing this project, such as the policy (DDPG), test files and other classes, are robust and reusable. Therefore, it is encouraged that future works in similar domains make use of them. In conclusion, the research provides valuable insights into the challenges of MARL implementation, highlighting the need for careful project management and offering reusable resources for future endeavours.

Keywords: Multi-Agent Reinforcement Learning, Deep Reinforcement Learning, Reality-Gap, Project Management.

Contents

1. 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.

1.1. Literature review

Reinforcement learning (DRL) has taken centre stage in recent years, revolutionizing how agents can learn optimal control policies in complex, dynamic environments. Its rapid evolution, while exciting, has also incorporated complexity to its increasing challenges.

On one hand, simulation DRL agents have mastered intricate robotics tasks, from playing Atari games to bipedal locomotion, within meticulously designed virtual environments. Several researches [9], [10] showcase remarkable accomplishments in simulated domains. On the other hand, the quest for real-world applicability has provided exciting breakthroughs. Other researchers [4] demonstrates real-world robotic manipulation learned solely through DRL. For instance "Learning to Run" [11], sees a bipedal robot learning and adapting to navigate real-world terrain.

However, challenges remain. The reality gap, which is the disparity between simulated and real environments can affect other important issues such as generalization, leading to agents struggling to adapt to unforeseen complexities. Some researchers already put their hands on it by applying and developing techniques such as domain randomization [12] or online adaptation [13]. In the first case, controlled variations are injected into the simulated environment enhancing robustness and adaptability, while in the second real-time adjustments are enabled, allowing agents to refine their policies when encountering unforeseen real-world dynamics.

This brief overview provides a wide picture of DRL's trajectory, navigating between simulated environments and real-world applicability. The papers mentioned serve as stepping stones, but many exciting works wait to be explored when delving deeper into this field.

1.2. 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. Among these issues are found a lack of sample efficiency, an efficient way of 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 algorithm learned in a simulation environment 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.

Why does this gap exist in the first place? A correct answer to this question could reveal the issues that, when addressed, might improve, or solve the main issue. It is essential to consider both evident and less apparent factors. Simulators, often used for training RL algorithms, have inherent imperfections. The simulation process becomes increasingly challenging particularly with complex robots, custom-made designs, or the absence of Unified Robot Description Format (URDF) or Simulation Description Format

(SDF) representations. Furthermore, the reality gap is intensified by factors such as inaccuracies in simulation physics, disparities in sensors and motors capabilities between simulation and reality, and uncertainties in modeling real-world dynamics. All these elements combined contribute to the challenges encountered when transferring knowledge acquired in simulation environments to actual robotic agents.

1.3. Objectives and rationale

The amount of time to invest in this project is very limited, therefore a study [14] 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.

As said before, this project aims to reduce the reality gap, also called sim-to-real transfer problem, with a specific task involving multiple agents (MARL). 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. Moreover, being able to bridge this gap can save time, money, and resources by minimizing the reliance on real-world training. For instance, a more efficient sim-to-real transfer reduces the need for replacing non-functional components in real robots due to friction or collisions, contributing to sustained and longer life cycles.

This project is mainly dedicated to the implementation of Reinforcement Learning (RL) algorithms within simulated and real-world scenarios alongside the construction of two collaborative robotic arms. Therefore, the project is designed around both academic and skill development goals, which are listed below.

Academic goals:

The primary academic objectives revolve around the successful application of RL algorithms. Initially, the focus lies on the implementation of these algorithms within simulated environments, with subsequent extension to real-case scenarios. While designing the model, a parallel effort involves the design and construction of two robotic arms, which will be used to test the model in a real-case environment.

The creation of RL models and robotic arms revolve around the sim-to-real-transfer issue in this field nowadays. Therefore, this project aims to, with a limited amount of resources and time, apply existing techniques [15], [11] as an intend to minimize this gap.

Personal goals:

Beyond the academic scope, this project seeks to elevate programming skills, with a particular emphasis on Python and C++. At the same time, a focus on software and framework learning is paramount, including the familiarization of Linux systems and the ROS 2 framework. Moreover, this project serves as an opportunity to apply and consolidate knowledge previously acquired in electronics, robotics, and artificial intelligence during the pursuit of a bachelor's degree in electronics engineering.

To summary, on one hand, the academic goals of this project are designing a system which, based on the current resources, minimizes the reality gap from a simulated environment to a real-case scenario. On the other hand, the main objectives in the personal domain are applying and consolidating knowledge and skills learnt the past few years as well as learning and getting familiar with frameworks and languages useful in the industry nowadays.

Key questions that provide guidance in the investigation and that are to be addressed through this project are defined below. These questions may also help the reader understand the rationale behind the goals described in this section.

- Is it possible to reduce the reality gap using the existent techniques [11], [15]?
- With limited number of resources and time, is it possible to build two working robots that communicate and cooperate with each other using MARL algorithms?

1.4. 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.

Moreover, precision is a game changer issue while dealing with sim-to-real-transfer. Both in sensory data and mechanical gears, small inconsistencies add up to create real and important errors that will affect the overall function of the system.

1.5. Overview of methodology and resources

Resources:

The study [14] 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 [16] will be used to develop the models and algorithms due to the richness of libraries and resources they provide [17], [18]. 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.

Methodology:

The project has a period of 4 months, starting on late February and ending on January 11th. The methodology followed to finish the project while respecting the deadline is described using a Gantt chart. Two charts have been developed during the course of the project, one in extreme detail, which will be attached to this document, and a simplified version that can be found on the next page in **Table 1**.

1.5.1. Gantt chart

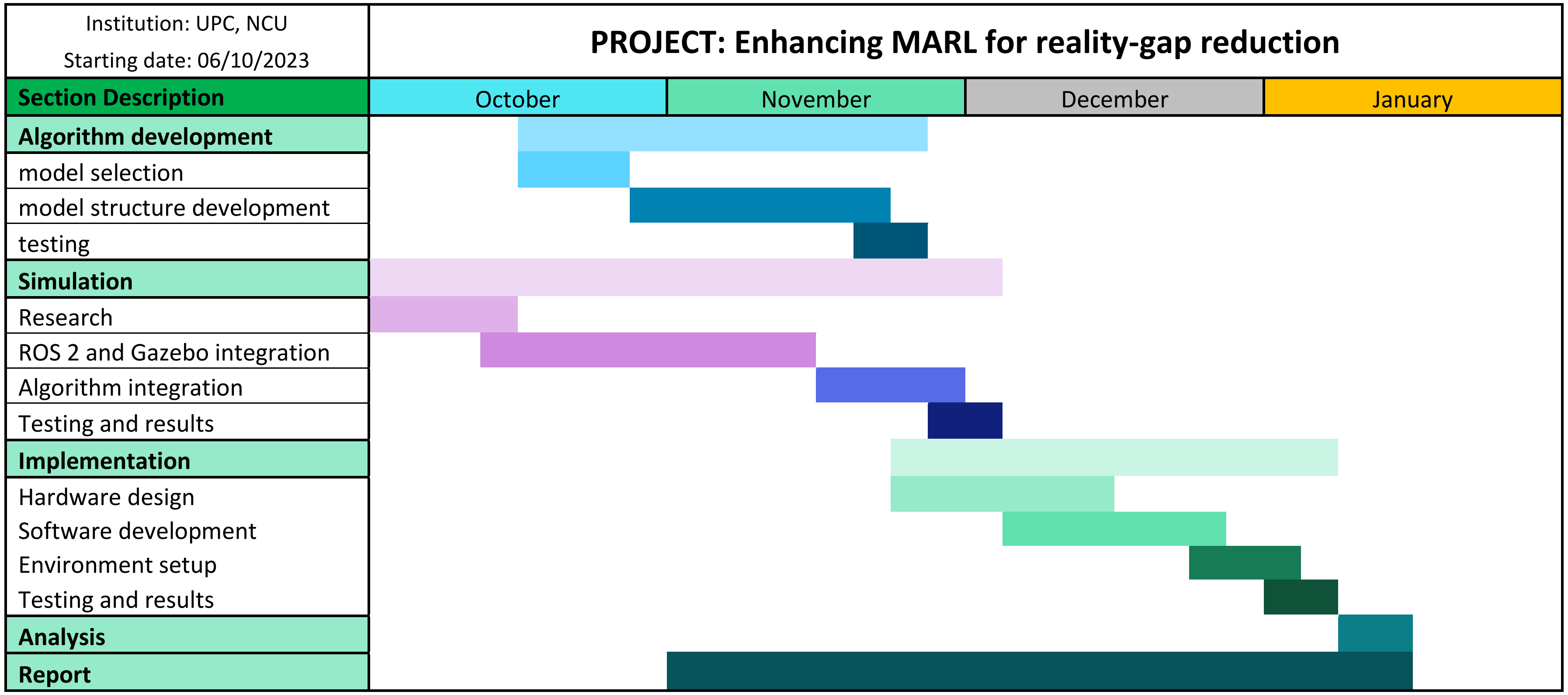


Table 1. *Simplified Gantt chart of the project.*

1.6. Organization of the Study

This research is divided into 3 main parts which are model design, simulation, and implementation. Each of these parts contains the following information:

1. Reinforcement learning design and development using state of the art literature [7], [14], [19].
2. Simulation of the environment and robots using Gazebo [20] and ROS 2 [21] for control and algorithm implementation.
3. Real implementation of the environment and robots, mimicking as much as possible Gazebo's simulation, while trying to minimize the reality gap.

Other sections are secondary, although not less important, and can be found and located in the Content section. After the general results, the economic and environmental analysis of the project can be found in which the impact of this is put into numbers, analyzed a discussed.

In each section, the rationale and resources implemented are considered the most important concepts to be located on the main body. While detailed information is provided in such cases that contribute to the understanding of the whole system, most of the time, deep explanations and technicalities are placed in Appendix I. During a section, the reader may be referred to the Appendixes if deeper information is required or just for the sake of curiosity.

In this project, two appendixes can be found, so it has been considered important to modularize information in two sections. Appendix I contains technicalities and deeper information that in the main body would add complexity to the general explanation although still being important, while Appendix II contains programming code, components' datasheet, and electronic schematics.

2. 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*", below we are going to delve right into each framework description.

2.1. ROS

ROS or Robot Operating System [21] 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 [22] 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.

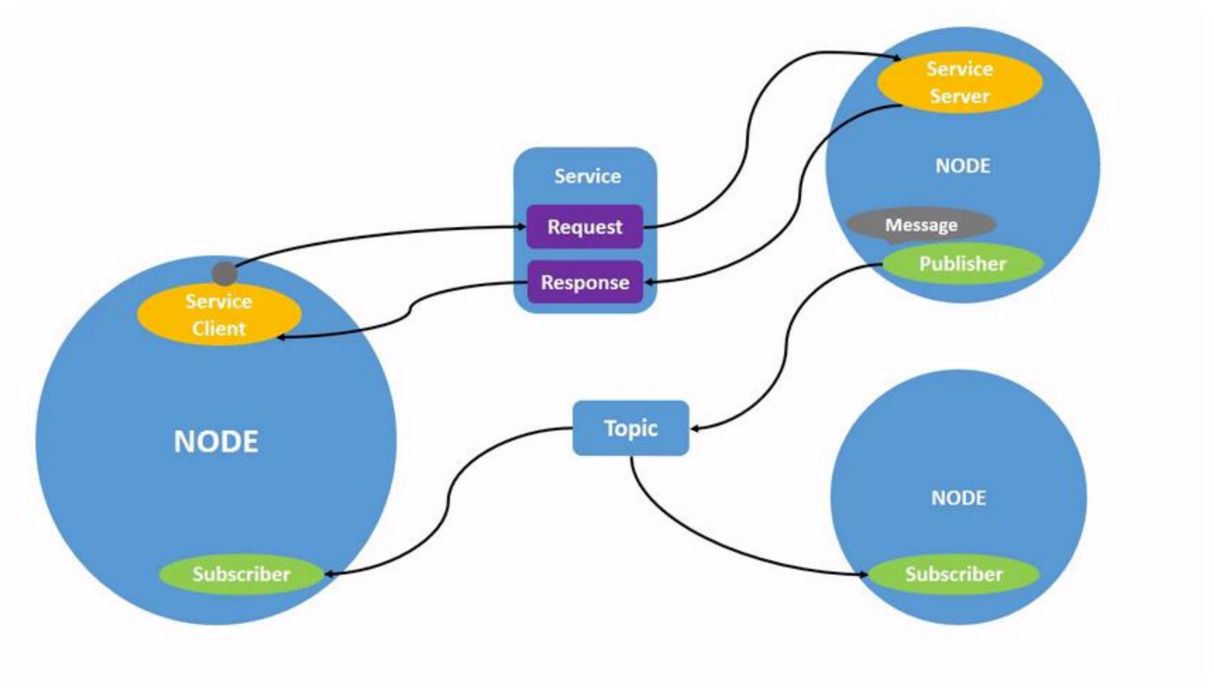


Figure 1. *Multiple nodes sharing information via topics and services.*

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.

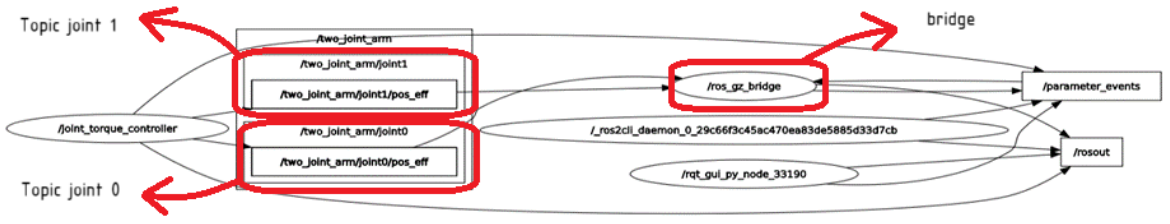


Figure 2. *Rqt_graph showing ros2 nodes connected through topics and bridge.*

2.2. 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 [11].

2.3. 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" 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" section).

2.4. 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 [23]. 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.

3. 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 [24]. 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 → Δ(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 [25] 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 [14] 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 [26], [27], and its good capabilities dealing with continuous spaces similar to those in this project.

3.1. Model definition

Deep Reinforcement Learning has been successfully applied in [4], [27], [28], 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 $\boldsymbol{\mathcal{M}}$.

$$\mathcal{M} = \{\mathcal{N}, S, (A^k)_{k\epsilon\mathcal{N}}, P, (r^k)_{k\epsilon\mathcal{N}}, (\pi^k)_{k\epsilon\mathcal{N}}, V^k(\cdot)\} \qquad \textit{Eq. 1}$$

Where:

- The set $\mathcal{N} = \{1, 2, \dots, n\}$ representing the number of agents.
- The set S representing all possible states of the environment.
- The set A^k representing available actions to the agent k.
- The state transition function $P = S \times A\hat{}1 \times \cdots \times A\hat{}n \to \mathbb{R}$.
- The policy $\pi^k : S \to PD(A^k)$for the agent $k \epsilon \mathcal{N}$.
- The accumulative reward $V^k(\cdot)$ of an agent k (also called Value function).

In this type of model, each agent $k \epsilon \mathcal{N}$ aims to maximize the value function with the starting state $\boldsymbol{s}$ at time $\boldsymbol{t}$ as shown in **Eq.** 2.

$$V^k(s) = \mathbb{E}[\sum_{i=0}^{T-t} \Upsilon^i r_{t+1}^k | s_t = s, (\pi^k)_{k\epsilon\mathcal{N}}] \qquad \textit{Eq. 2}$$

Where:

- $\mathbb{E}$ is the expected value taken over a random value which is the sum of discounted rewards.
- Υ is the discount factor that weighs the future rewards.
- r_{t+1}^k is the reward received by the agent k at the time $t + i$.

While it is true that [14] 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.

3.2. 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 $\boldsymbol{\mathcal{M}}$ (**Eq.** 1), how they are defined, and how the system comes into place both in simulation and reality.

3.2.1. 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 (s) will be sent to the policy (π) where an algorithm will decide which action to take next. Once the action (a) is executed, the environment will respond with a new state (s_{t+1}). Based on the action's (a) effectiveness, the reward function (r) will provide a reward as a scalar value. Finally, the agent learns from this data (a, s_t, s_{t+1} and r) 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 and Implementation sections, since each area uses different mechanisms to obtain states, terminal conditions and rewards.

3.2.2. States

In the reference paper [14], the state set ***S*** 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 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 the **Eq.** 3.

$$s_t = (q_t^k, p_t^k)_{k \epsilon \mathcal{N}} \qquad \textit{\textbf{Eq.}}\ \textit{3}$$

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

3.2.3. 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 k are defined as the set A^k in **Eq.** 4. Here, τ_t^i represents the torque applied to the joints at time t for the i_{th} joint. The set A^k includes all possible torque values that agent k can choose from the available actions at a specific time.

$$A^k = \{\tau_t^i\} = \{\tau_1(t), \tau_2(t), \tau_3(t), \dots, \tau_{ith}(t)\} \qquad \textit{\textbf{Eq.}}\ 4$$

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*" and "*Actions in Reality*".

3.2.4. Policy

The policy π^k takes the world state and produces a set of actions **Eq.** 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.

$$\pi^k : S \rightarrow PD(A^k) \qquad \textit{\textbf{Eq.}}\ 5$$

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.

3.2.4.1. CDDPG Architecture

Deep Deterministic Policy Gradient is based on two main neural networks called Actor and Critic, similar to the SAC [29] 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 **Eq.** 5. It is responsible for the generated actions based on the received states from the environment by maximizing its loss function.

$$L_A = -\frac{1}{N}\sum_{i=1}^{n} critic\big(s_i, actor(s_i)\big) \qquad \textit{\textbf{Eq.}}\ 6$$

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 which is computed using the mean squared error (MSE) between the estimated Q-Values (Q_i) and the target Q-Values (Q_{target_i}). How the target Q-Values are obtained is explained in the "*Reward Function*" section.

$$L_C = \frac{1}{N}\sum_{i=1}^{n} \left(Q_i, Q_{target_i}\right)^2 \qquad \textit{\textbf{Eq.}}\ 7$$

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 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.

➢ 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 e_t is the definition of an experience, then let the Replay Buffer be defined as a finite-size memory set $\mathcal{D}$ that stores a collection of size N of e_t (**Eq.** 9).

$$e_t = (s_t, a_t, r_t s_{t+1})$$ ***Eq.*** *8*

$$\mathcal{D} = \{(s_1, a_1, r_1, s_2), (s_2, a_2, r_2, s_3), \dots, (s_T, a_T, r_T, s_{T+1})\}$$ ***Eq.*** *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.

3.2.5. State transition function

The state transition function $P = S \times A^1 \times \cdots \times A^n \rightarrow \mathbb{R}$, 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.

3.2.6. 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 **Eq.** 10, **Eq.** 11 or **Eq.** 12.

$$\sum v_{\Delta t}^{i} \cong 0 \qquad \textit{Eq. 10}$$

$$\sum \left(|r_{t_0}^{k} - r_{t+\frac{n}{2}}^{k}| + |r_{t_0}^{k} - r_{t+n}^{k}|\right) \cong 0 \qquad \textit{Eq. 11}$$

$$(\theta + \phi + \psi)_{obj} < th \qquad \textit{Eq. 12}$$

$$done = (4) + (5) + (6) \qquad \textit{Eq. 13}$$

In **Eq.** 10, $\sum v_{\Delta t}^{i}$ is the summation of each velocity of the i_{th} joint therefore it becomes *True* when the velocity remains close to 0 for an incremental period Δt indicating that the robot has stopped moving and it has either reached an optimal point or a local minimum. Similarly, equation **Eq.** 11 evaluates to *True* when the summation of the difference of reward values in an incremental period of time ($\Delta t = t_0 + n$, where n is a custom variable for additional reward checks) is 0.

In the Python implementation of both equations **Eq.** 10 and **Eq.** 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 **Eq.** 12 detects if the orientation pitch, yaw, or roll (θ, ϕ, ψ) of the object is changing too much, leading to undesired object placement, or indicating the object has fallen. In this case th 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.

3.2.7. 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 does 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, **Eq.** 14 and **Eq.** 15, respectively for both robots, and secondly that which captures the object posture deviation **Eq.** 16. Here the Euler angles pitch, roll and yaw are defined as θ, ϕ, ψ, respectively and p_r^k is the location of the end-effector of the robot k.

$$r^{g1}(p_r^1, p_{target}) = -k_a \frac{1}{|p_r^1 - p_{target}| + k_{bias}} \qquad \textit{Eq. 14}$$

$$r^{g2}(p_r^2, p_{target}) = -k_b \frac{1}{|p_r^2 - p_{target}| + k_{bias}} \qquad \textit{Eq. 15}$$

$$r^{g3}(\theta, \phi, \psi) = \begin{cases} \text{-k}_c{}^*\theta_{rad} & if\ \theta < t_1 \\ \text{-k}_d{}^*\phi_{rad} & if\ \phi < t_2 \\ \text{-k}_e{}^*\psi_{rad} & if\ \psi < t_2 \end{cases} \qquad \textit{Eq. 16}$$

To encourage proximity to the object, the distance reward is based on the hyperbolic function $f(x) = 1/x$, so as the robot gets closer to its target, the reward increases proportionally. Moreover, some parameters k_n has been added to enhance flexibility and control over each reward function weights. For instance, the higher the value of k_n the more important will be the condition. A value of 1 nullifies the effect of the parameter. Finally, k_{bias} is added to avoid dividing by 0.

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 [30], [14]. The one used in this project is an example of what can be accomplished with a few sensors, and it can be further scaled and improved. The mathematical notation of the complete reward function is defined as follows:

$$RF: \quad r^{g1} + r^{g2} + r^{g3} \qquad \textbf{Eq. } 17$$

3.2.8. Value function

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 Υ. The Value function has been previously defined **Eq.** 2, but it is showed below for convenience.

$$V^k(s) = \mathbb{E}[\sum_{i=0}^{T-t} \Upsilon^i r_{t+1}^k | s_t = s, (\pi^k)_{k\epsilon\mathcal{N}}]$$

Where:

- $\mathbb{E}$ is the expected value taken over a random value which is the sum of discounted rewards.
- Υ is the discount factor that weighs the future rewards.
- r_{t+1}^k is the reward received by the agent k at the time $t+i$.

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.

3.3. Model overview diagram

Figure 3 provides a general insight of the model system and how the data is transferred from one component to another. Notice that elements such as target networks, dropouts and hyperparameters are not listed since they are not essential for understanding the core data flow in the system.

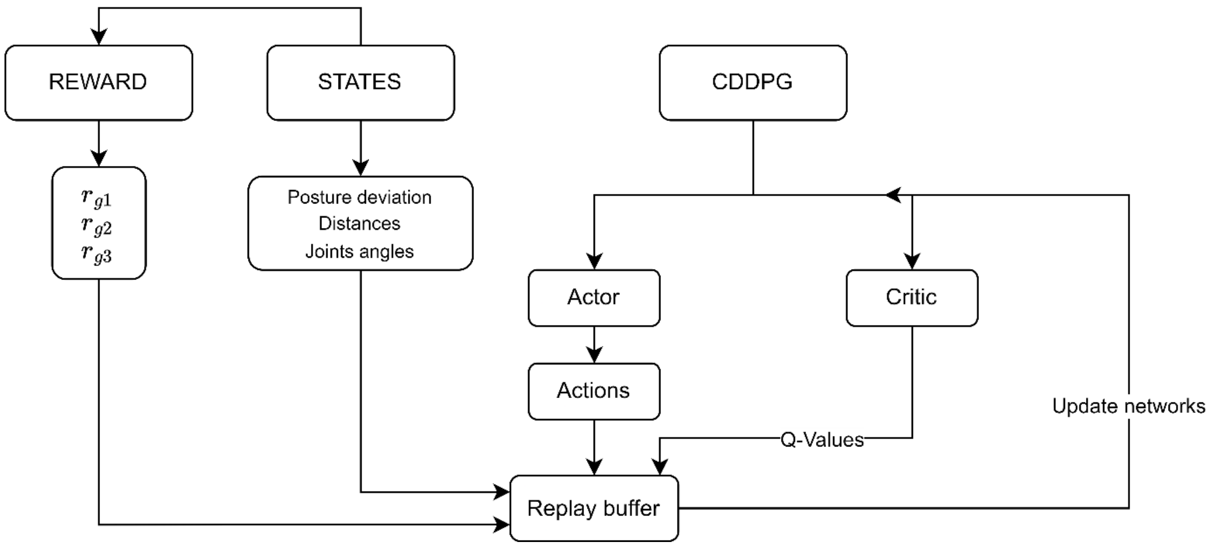


Figure 3. *Flow-chart of the models' main components*

In summary, reward values, states actions produced by the agent and Q-Values provided by the Critic are stored in the replay buffer. A batch of experiences is used to update the main networks within a certain frequency while the target networks are updated depending on the parameter τ. Always refer to the GitHub Repository where the project is stored for further information.

4. Simulation

Gazebo [20] will be used as the physics simulator due to the use of this software in [14]. 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.

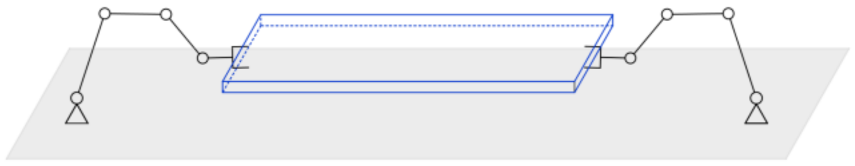


Figure 4. *Environment setup reference [9]*

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*. By combining *nodes*, *topics*, *services* and Gazebo, a system is created in which data is transferred seamlessly from ROS 2 to Gazebo and vice versa.

Next sections provide a deep understanding of this system and its main components while providing a detailed explanation about the integration of the RL algorithm in the simulation. At the end of it, the reader can find a flow chart of the system as well as the results obtained.

4.1. Simulation design

In Gazebo, a robot, also called model or agent, can be developed using the provided GUI or a Description File, such as SDF or URDF. The use of 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*" for a deeper and detailed explanation about these files.

4.1.1. 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*" 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*".

4.1.1.1. States

As explained in States section, the states set at specific time (s_t) is has been previously defined by the tuple in **Eq.** 3. How to obtain each joint angles (q_t^k) and both end-effector global coordinates (p_t^k) from Gazebo simulator is explained below.

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 ($O_{x,y,z}$) and a Quaternion ($q = a + bi + cj + dk$, 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 ($G_{x,y,z,q}$) can be directly obtained from the simulator.

Definition 2. *A quaternion is a mathematical concept [31] 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 $q = a + bi + cj + dk$. 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*** [32] and due to the fact that they are more compact and computationally efficient than rotation matrices.

4.1.1.2. 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.

4.2. Simulation data flow diagram

The figure below is a chart to visualize the flow of data in the simulation system. Here, arrows are used to represent the direction the data is following. For instance, while the States node is only receiving information from Gazebo and sending it to the DDPG algorithm, the Actions node is receiving data from the RL model and redirecting it to Gazebos' simulator to move the joints through its pertinent plugin.

Furthermore, bidirectional components also exist such as the bridge framework, to specify the topic name and data type to be transmitted from ROS 2 to Gazebo or vice versa, and the Termination Node, which is receiving data from Gazebo although if a condition is met, this node will also kill Gazebo or at least, stop the simulation.

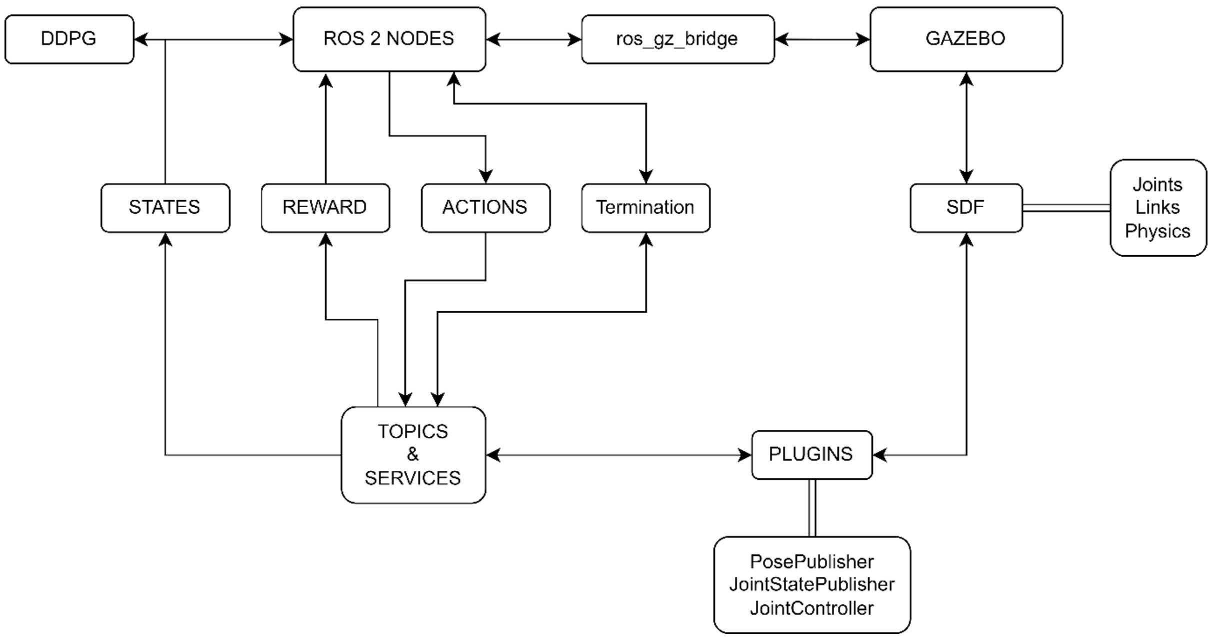


Figure 5. *Data flow in the Simulation System*

4.3. Simulation results

To evaluate the simulation behavior 3 main components have been considered. First and more important, the reward value over n 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 provide a deep understanding of the behavior of the main networks which can be analyzed and improved over iterations.

The graph shown in **Figure 6** 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, dropout or domain randomization.

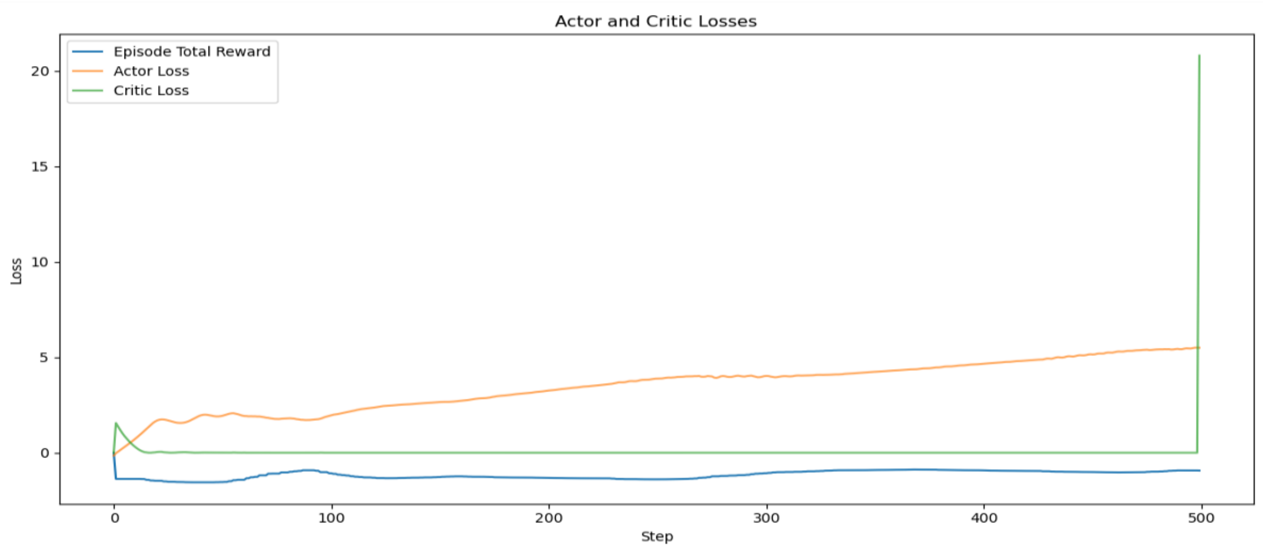


Figure 6. *Graph from a plane DDPG policy model*

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 7** 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.

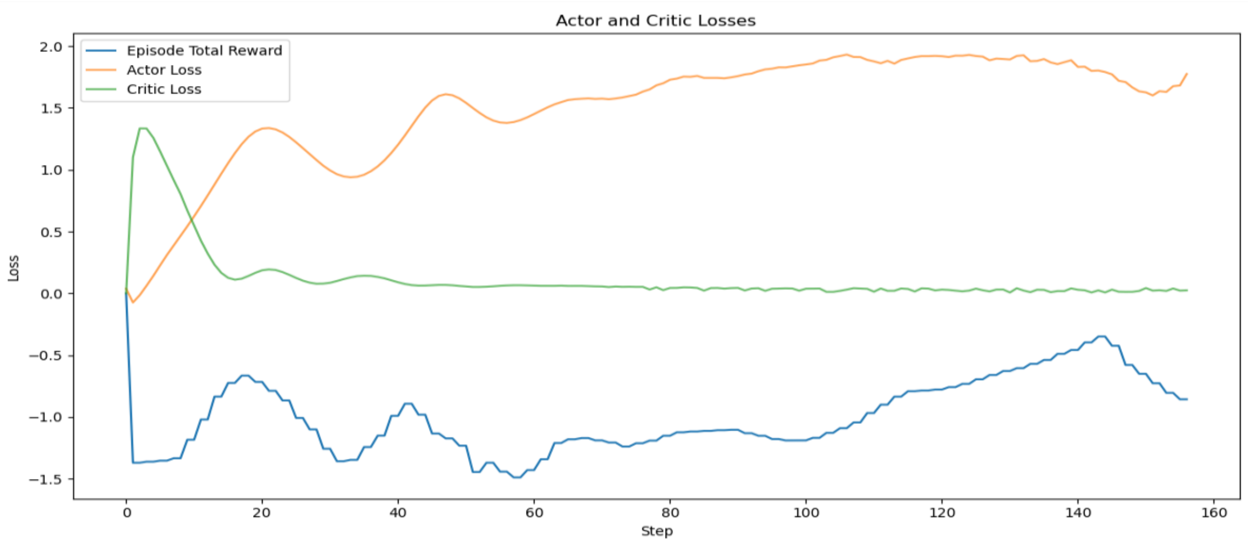


Figure 7. *Reward and loss values after adding a replay buffer.*

Finally, a dropout was incorporated into all layers of both networks besides a simpler version of domain randomization (DR). **Dropout** [33] 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. Furthermore, **domain randomization** [12] adds certain Biases to simulate unforeseen events and make the model more robust before real environment with different system dynamics. These biases have been added both in simulation and reality training.

In **Figure 8**, results after incorporating DR, 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 and improving, it 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.

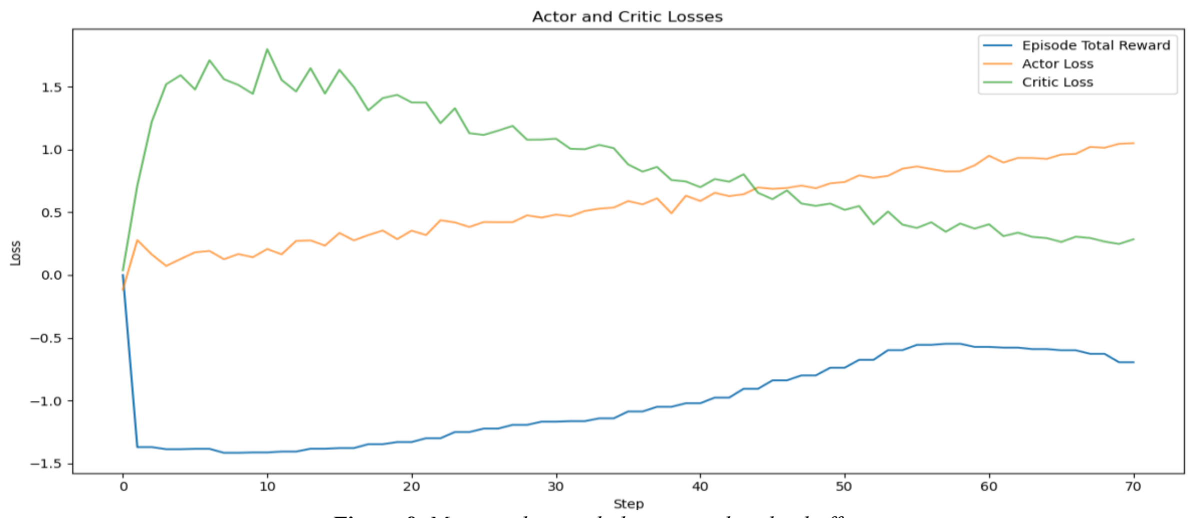


Figure 8. *Metric values with dropout and replay buffer.*

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 [14].

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 repository where this project is stored.

5. 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.

The addition of sensors to real robots serves the purpose of acquiring additional data, offering the option to fine-tune or even train the model from scratch. It's important to acknowledge that in an ideal scenario with a perfectly simulated robot, there might be no need for onboard sensors. This is because the robot could replicate its simulation behavior precisely. Although that being true, it is also obvious that the simulation is far from being 100% accurate. Therefore, integrating sensors becomes crucial to refine and guide the pre-trained model. This addition not only enhances accuracy but also provides increased flexibility and scalability in adapting to real-world conditions and unforeseen challenges.

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.

The electronics schematics and connections can be found in the Appendix II.

5.1. Components

The main focus of this project is to keep learning and applying the knowledge gained while pursuing the Electronics' degree. 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 and components crucial for the project. This encompasses the ESP32 microcontroller, HCSR04 ultrasonic sensor, MPU6050 inertial measurement unit, PCA9685 servo driver, MG996R servo motors, and aluminum parts for the robots’ structure. Each material serves a distinct purpose in shaping the real-world environment and bridging the gap between simulation and implementation.

It is worth noting that schematics and diagrams of electronic components connections can be found in Annex II.

5.1.1. 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.

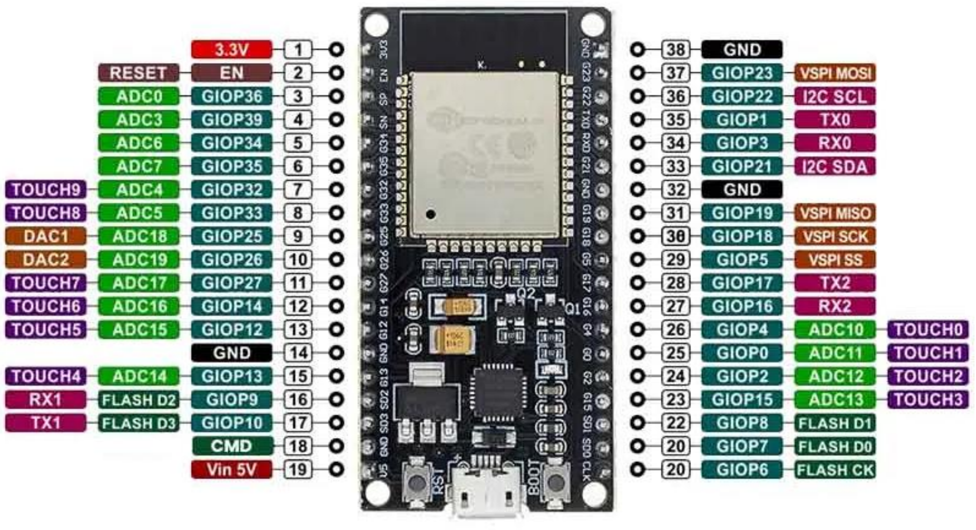


Figure 9. *ESP32 pinout*

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 9 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*".

5.1.2. 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.

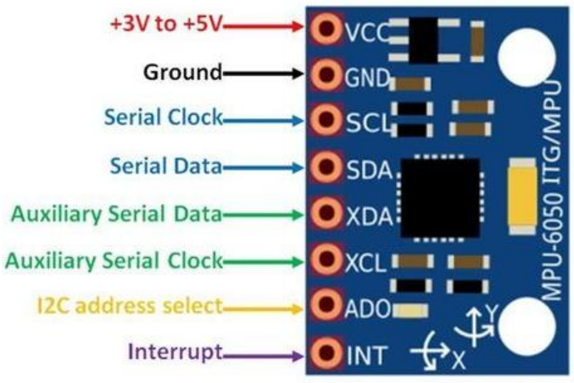


Figure 11. *MPU6050 Pinout*

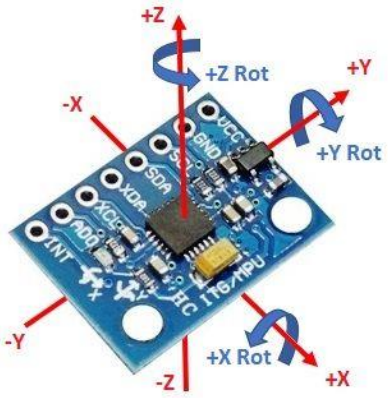


Figure 10. *MPU6050 rotation axis*

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 (see "*Implementation states*").

Technical note:

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 objects' posture deviation and trigger termination conditions or other methods in the algorithm.

5.1.3. HSCR04

After running into integrational drift issues, the HSCR04 ultrasonic sensor stands as the most cost-efficient option. While it is true that other approaches, such as using a camera, are more efficient and would provide greater outcomes, time and money are high constraints that stands in the way, as camera modules would have to be bought and algorithms based on CNN developed [34], [35].

That is why an HSCR04 is placed on each robots' end effector. The goal is to simulate the distance between each robot and the object as close as possible to the simulation environment, understanding that an error is introduced since these sensors have a 15-degree range of vision as one can see in **Figure 12**, and its precision is not always the best.

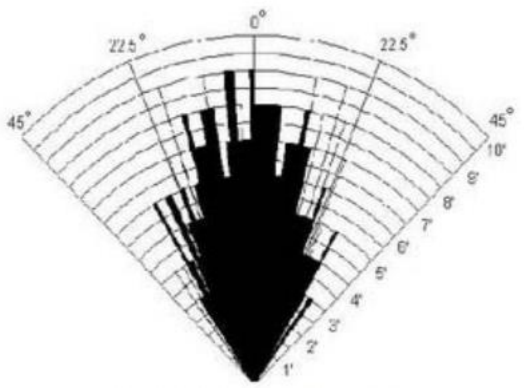


Figure 12. *Working range of HCSR04*

How the HSCR04 works is quite interesting. In simple terms, it has two input and output pins named Trigger and Echo, respectively. The trigger pin emits a short signal that triggers, as its name indicates, an ultrasonic wave that will impact the object and bounce back to the echo pin. After this, with the use of basic physics and some C++ code it is simple to retrieve the distance the ultrasonic wave has traveled and therefore, the distance to the object.

Figure 13 visually shows the interface and communication between the sensor and the microcontroller, the ESP32 in this case. Here is the user who decides when to trigger the wave from the microcontroller (refer to the GitHub repository to see the code used in the project).

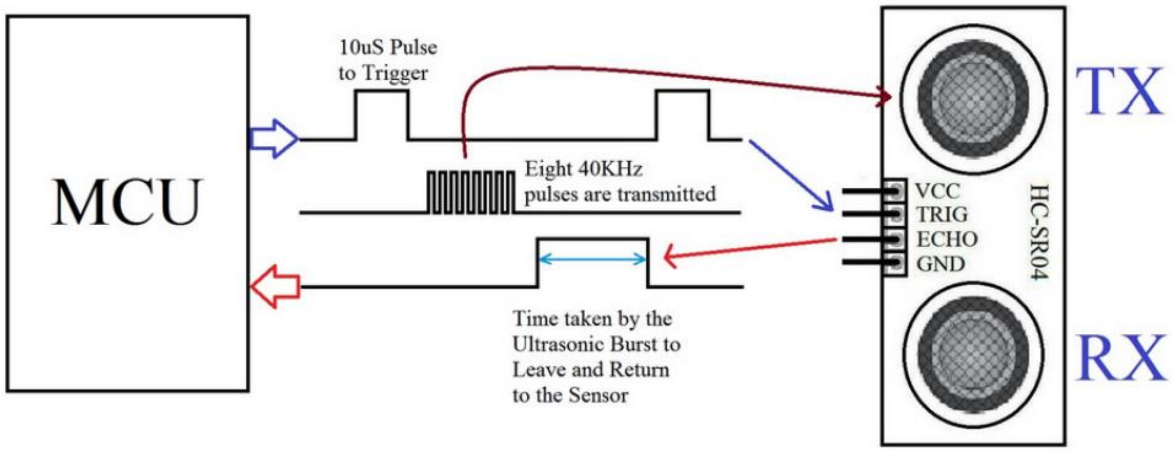

MCU
10uS Pulse to Trigger
Eight 40KHz pulses are transmitted
Time taken by the Ultrasonic Burst to Leave and Return to the Sensor
TX
VCC
TRIG
ECHO
GND
HC-SR04
RX

Figure 13. *HCS04 interface with uC*

The trigger signal is a 10μs pulse, and when produced, 8-cycles burst of ultrasound at 40kHz pulses are transmitted and set Echo HIGH, or logic '1'. The amount of time the Echo is in a HIGH state will dictate the distance travelled by the wave and can be calculated using the distance formula defined in **Eq.** 18.

$$d = E_{H_t} \frac{c_{25^{\circ}C}}{2}$$ ***Eq.*** *18*

Where d is the distance between the sensor and the closest object, E_{H_t} the time in state '1' of Echo and $c_{25^{\circ}C}$ is the speed of sound in the air at 25ºC. The division by two is due to the wave travelling forth and back thus travelling double the distance.

5.1.4. MG996R

Each robot can contain at least 4 joints which are implemented by motors. The choice of motors depends on several factors, where angle precision, monetary cost and integration with the current system stand above all other constraints.

The selection of motors for each robot involves considering various factors, with a primary focus on angle precision, economic cost, and integration compatibility with the existing system. Two potential options are considered, stepper motors and servomotors. Stepper motors operate by dividing a full rotation into a series of steps, providing precise control over angular movement. On the other hand, servomotors operate based on feedback control systems. They consist of a motor, a sensor to detect the current position, and a controller. The controller adjusts the motor's position to maintain accuracy. This design allows servomotors to offer high precision in controlling angular positions.

After some research, the chosen motors for the robotic joints are the MG996R, which dimensions are overview is shown in **Figure 14**. According to the datasheet, these motors can handle up to 11 $[kgF \cdot cm]$of torque and velocities up to 0.14 [s/60º], both values when supplied with 6 V. An interesting feature that the datasheet describes is that the motors have a rotation range of 120º, although after practical testing it has been observed that the motors can rotate up to 190º, exceeding the specified limit. Furthermore, a key factor in selecting the MG996R motor is their seamless integration with Arduino frameworks and libraries, ensuring compatibility and ease of programming through already available resources.

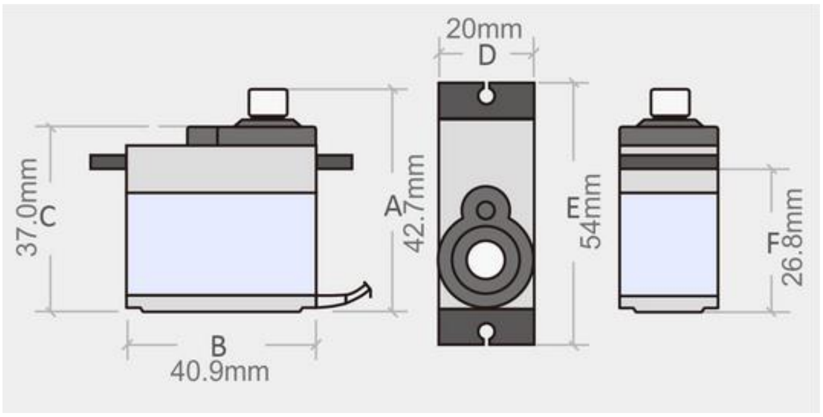


Figure 14. *Dimensions and overview of MG996R*

In general, servomotors are controlled by a PWM signal usually sent from a microcontroller. MG996R motors are no different. **Figure 15** is a visual explanation of how to use Pulse Width Modulation signals to control the servomotor.

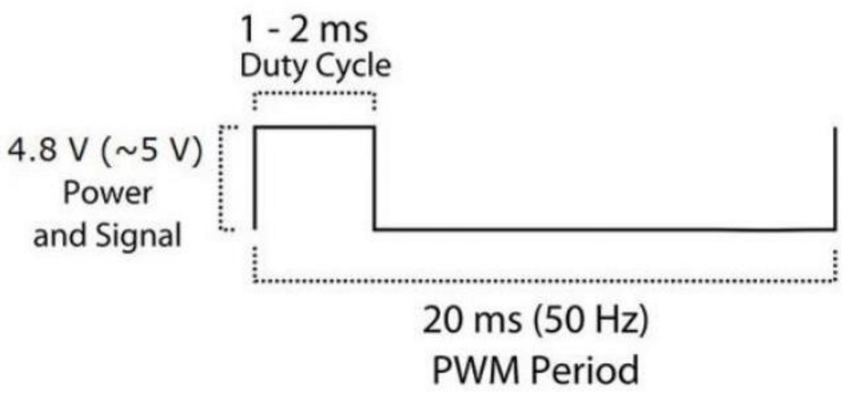


Figure 15. *MG996R control through PWM signals*

As defined in the datasheet, the period of the PWM signal should be 20ms. The duty cycle (DC) defines how much the servomotor should rotate. These motors are designed to work in a DC range of 1-2 milliseconds. Assuming a 180º rotation range ($[-\frac{\pi}{2},\frac{\pi}{2}]$):

- ➢ A 1-millisecond DC will position the joint all the way to the left ($-\frac{\pi}{2} rad = 0\ deg$).
- ➢ A 1.5-millisecond DC will position the joint in the middle ($0\ rad = 90\ deg$).
- ➢ A 2 millisecond DC will position the joint all the way to the right ($\frac{\pi}{2} rad = 180\ deg$).

To avoid collisions, limit distress and to follow safety constraints the following limitations have been added to the motors through software:

- ➢ While the motors have the capability to reach positions beyond expected, their rotation has been limited to 160º to ensure safe and controlled movements.
- ➢ The speed is governed by the reward function, which encourages lower speeds when in proximity to objects. Additionally, a safety measure has been incorporated to impose a speed limit, providing an extra layer of protection in case of unforeseen issues.

For more information about the programming implementation of the PWM signals refer to the PCA9685 section, which deals with motor control, and the GitHub repository, where the code resides.

Technical note:

After testing the motors, a slight misplacement of the rotor in the motor was noticed. Even if the motor is not being controlled, a small physical shift is introduced. This is no issue when dealing with one single motor although the problem becomes noticeable when several motors are operating along the same axis. When this happens, the slight shift is important enough to make the angle positions less precise.

5.1.5. PCA9685

The control of a single servomotor can be done directly through the ESP32 GPIO ports which allow PWM signals. However, when trying to control several servomotors, in this case 10 of them, it becomes messy and complex to control all of them from the same microcontroller. One option would be to separate each robots' motors into dedicated ESP32s, although this would add unnecessary complexity when communicating both ESP32s and unnecessary monetary expense.

This is why the module PCA9685 is being used. It can be used to control LEDs, motors or any component that is controlled by PWM signals. It has 16 output channels which means that 16 motors can be controlled using only one module. Furthermore, to receive signals from a microcontroller it uses I2C communication from a preset address 0x40. This address can be changed to add more PCA modules in cascade in case of having to control more than 16 components, which in this case is not necessary.

While I2C communication will be further explained in "Communication", notice that by using this type of communication, both MPU6050 and PCA9685 can coexist in the same environment without the need for a second microcontroller.

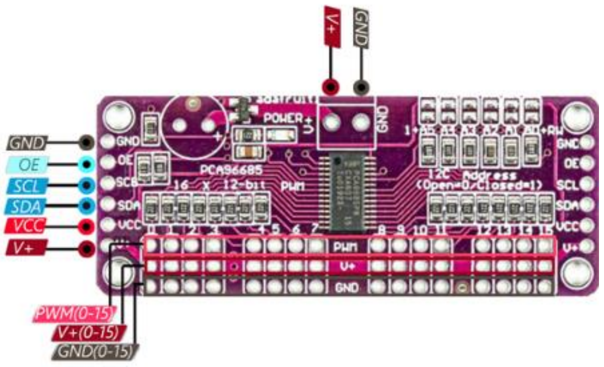


Figure 16. *PCA9685 Pinout*

In the figure above, SDA and SCL Input pins are the I2C buses that carry the data and clock signals, respectively. At the bottom of the board there are 16 PWM controllable pins and at the top an external power supply can be plugged in if more power is needed (see next section).

In this module, the PWM duty cycle is controlled by "*ticks*". These are periods of time used by the PCA to divide the pulse cycle into 4096 periods, called ticks in this context. To make the pulse have a 50% duty cycle the pulse cycle would be set HIGH for 2048 ticks and posteriorly set LOW for 2048 ticks. For a 25% DC, a HIGH state should last 1024 ticks, and so on.

5.1.6. Aluminum parts

As the main structure for both robots, prefabricated aluminum parts have been used. These modules adhere flexibility to the build, by containing several holes to fix the MG996R.

Their cost is low, around 12 € (400 NTD) and it allows for simplicity of building while providing a robust structure. Further discussion about the robots' design can be found in the section named "Robots Structure".

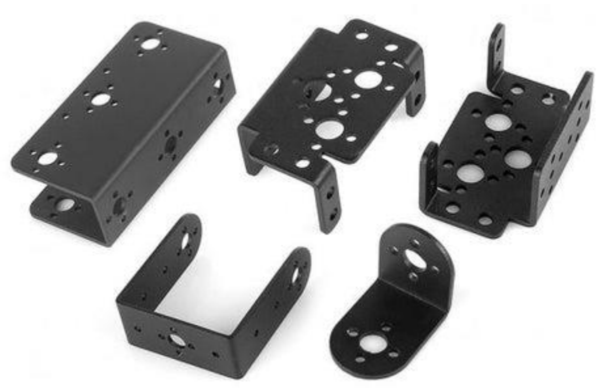


Figure 17. *Aluminum parts used in the project.*

5.2. Power consumption´

Before choosing a power supply for the system, a study of the consumed power is needed. In this case **Table 2** shows the components used with their respective maximum consumptions.

Component	Quantity	Typ	Max	Unit
ESP32	1	95	180	mA
PCA9685	1	25	-	mA
MG996R	10	0.5	0.9	A
MPU6050	1	1.6	-	mA
HC-SR04	2	6	15	mA
TOTAL	-	5.1336	9.2366	A

Table 2. *Components' power consumption*

To adhere to safety protocols, the total maximum current drawn will be used when selecting the power supply and a safety factor will be added in case of current peaks. Therefore, the power supply needs to provide at least 11 A for a $fs = 0.2$, as shown in **Eq.** 19.

$$I_{ps} = I_{max} + (fs \cdot I_{max}) = 11.084\ A \qquad \textit{Eq. 19}$$

To ensure good performance, only the power supply for the actuators will be supplied from an external source. Microcontrollers, sensors, and other logic components will be supplied from the same μC and this from the main computer.

5.2.1. Power supply

As mentioned earlier, only servomotors will be supplied by an external source. Given the MG996R's voltage requirements (4.7 V to 7 V) and a minimum total current of 11 A, a power supply that can provide regulable 6 V and 12 A would be ideal.

After thorough research, the chosen power supply is an AC-DC converter from the RS-75 series, specifically the model RS-75-5, shown in **Figure 18**.

Figure 18. *Power converter RS-75-5 110-220VAC to 5VDC 12A*

This converter can efficiently transform 110-220VAC to 5VDC with an output current of 12 A. Furthermore, it has a broad input voltage range (88 VAC to 264 VAC) and supports a frequency range of 47 to 63 Hz. This versatility ensures that the project can be seamlessly replicated in different areas with varying standard voltages and frequencies. For instance, in Taiwan where the standard voltage is 110 VAC and 60 Hz, or in Barcelona with 230 VAC and 50 Hz.

For more information about its features and characteristics, refer to Appendix II, "*RS-75-5 Datasheet*"

5.3. Robot design

The mechanical design of the robots involves decisions regarding the choice of materials, number of joints, motor placement and fixation. Initially and out of excitement, the structure was to be made with 3D printed parts. Therefore, designing all mechanical components with CAD software such as SolidWorks, a tool familiarized with while pursuing the bachelors' degree. Although that was an exciting idea, 3D printing involves several steps and extensive iterations, which demands a significant time and money investment which could escalate even more depending on the success of each part. Considering these factors, another option arises: to manufacture parts from aluminum. While the process was even more time-consuming and economically expensive than the first one, it is a robust and flexible approach. Finally, as explained in the "*Components*" section, the structure is made of aluminum parts that have been already designed to seamlessly integrate with either MG995 or MG996R servomotors.

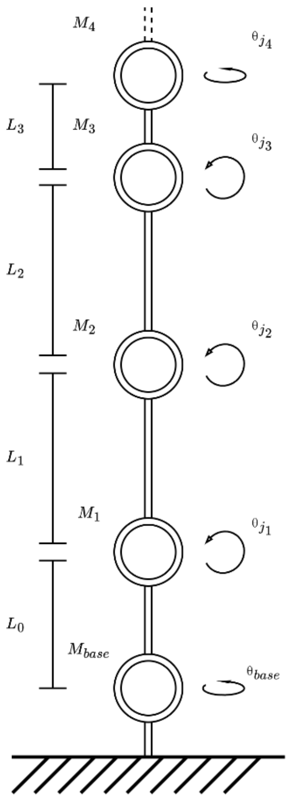


Figure 19. *Robot schematic. Angles, Joints, and Link distances*

Figure 19 illustrates the schematic of the robotic arm in a vertically aligned position representing its default configuration. Furthermore, this diagram contains all rotations (θ_n), as well as the different motors and distances are shown, defined as M_n and L_d, respectively. Due to the flexibility that the layout design provides, a sixth motor could be added if extended distances or different types of grippers were needed.

Feature	Description
DoF	5 (if an extra motor is added, 6)
Nº Motors	5 default motors (MG996R or MG995) expandable to 6
Torque (kgf·cm)	Torque range per motor: 9.4 to 11 (when [4.7, 6] V)
Joint rotation (º)	120 (theoretical), 190 (experimental)
Material	Aluminum
Power Supply	Operating voltage: [4.7, 7.2] (V). Provide 11 A at least (refer to "*Power consumption*")
Com. Interface	UART
Hardware compatibility	PCA9685, ESP32, Arduino UNO, Raspberry Pi

Software compatibility	Python, Micro-python, C++
Weight (kg)	~0.4

Table 3. *Technical characteristics of the robot*

The table above provides the main characteristics of the robot to be considered if replicating the environment setup. For more information about electronic components, connections or others, refer to their particular section or to the Appendixes.

5.4. Communication

Once the electronic components are connected and the robots built, there is a need to find an efficient way to send and receive data. Since the microcontroller used is the ESP32 which possesses Wi-Fi and Bluetooth capabilities these were first considered. After thorough testing, this option was rejected due to high interferences and difficulty of synchronization.

As explained before, the ESP32 will not be supplied by the SR-75-5 but by the computer which holds the DDPG algorithm. Taking advantage of this, serial communication will be used to transmit data to the robots and to receive their state and process it in the model. Specifically, UART serial communication will be used.

Moreover, another type of serial communication used is I2C, which enables different components to use the same bus to transmit data. Further explanation about serial communication is provided below.

5.4.1. Serial

Sending the values provided from the DDPG agent to its respective motors is being done through the Serial port using UART protocol, 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 arrive to the ESP32 using a *casting pointer* [36].

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 in the ESP32 and sent to the servomotors as angle values. See Annex I to find further information about pointer casting, endianness, and memory addresses.

5.4.1.1. I2C

This type of serial communication is often used to connect multiple peripherals devices to a microcontroller or other embedded system. Also called Inter-Integrated Circuit, I2C is a synchronous multi-master and multi-slave protocol that can be used with only two wires, a data line (SDA) and a clock line (SCL). I2C uses either a 7-bit or 10-bit address format, enabling the connection of up to 1024 devices on the same bus. However, it needs a clock signal to generate start and stop conditions which is provided by the microcontroller. Despite this requirement, I2C offers the benefit of efficient data transfer at a rate of 400 kbps, making it well-suited for onboard communication within electronic systems.

In the context of this project, I2C communication has been used to exchange data with the MPU6050 and the PCA9685 using the default addresses 0x68 and 0x40 respectively. Thanks to this, both devices can be connected to the same bus without the need for more cables or control. Moreover, there are libraries such as "<*Wire.h*>" that allows for seamless I2C integration within the project.

5.5. Real-world observations

While the system is the same, the way the observations are perceived and processed differs from simulation and reality. This section explains how the state of the environment is treated and how the actions that the DRL algorithm provides are moving the robots.

Previously, the states set s_t has been defined in **Eq.** 3. It takes the distances between each end-effector and the object, its posture deviation, and the angles of each joint. These states will be processed in the agent and the Actors' network will provide actions to get closer to the ideal state guided by the policy and the reward function.

$$s_t = (q_t^k, p_t^k)_{k \epsilon \mathcal{N}}$$

Initially 3 IMUs were considered to triangulate the distances between robots and the object. Placing one on each end-effector and the other one in the item, good representation and guidance could be provided to the agent. After thorough practical testing, it was clear that a known issue was taking place called integration drift. Let's add more context to further understanding this crucial issue.

Integration drift:

To obtain the linear and angular position from accelerations, linear and angular, respectively, two integrals must be calculated. These equations are shown in **Eq.** 20 and **Eq.** 21, where given a linear acceleration $a(t)$ and an angular acceleration $\alpha(t)$ integrating with respect to the time once gives angular and linear velocities $v(t)$ and $\omega(t)$, respectively, and integrating again gives linear position $x(t)$ and the deviation angle $\theta(t)$.

$$\theta(t) = \iint \alpha(t)\, dt \qquad \textit{Eq. 20}$$

$$x(t) = \iint a(t)\ dt \qquad \textit{Eq. 21}$$

While theoretically this is true, in practical scenarios, specifically with the MPU6050, a drifting error is introduced and amplified in every integration step. This error is mathematically introduced in the accelerations as $E_o(t)$.

$$\alpha(t)_{total} = \alpha(t) + E_\theta(t) \qquad \textit{Eq. 22}$$

$$a(t)_{total} = a(t) + E_x(t) \qquad \textit{Eq. 23}$$

Once it is introduced, a double integration is needed to obtain the linear and angular positions. Developing a little bit, equations **Eq.** 26 and **Eq.** 27 are obtained, where the error being integrated twice can be observed.

$$x(t) = \iint (a(t) + E_x(t)) \qquad \textit{Eq. 24}$$

$$\theta(t) = \iint (\alpha(t) + E_\theta(t)) \qquad \textit{Eq. 25}$$

$$x(t) = \iint a(t) + \iint E_x(t) \qquad \textit{Eq. 26}$$

$$\theta(t) = \iint \alpha(t) + \iint E_\theta(t) \qquad \textit{Eq. 27}$$

The final equations **Eq.** 28 and **Eq.** 29 show the magnitude of the error introduced by the IMU and the quadratic amplification by the integral process.

$$x(t) = \iint a(t) + E_x(t) \cdot t^2 \qquad \textit{Eq. 28}$$

$$\theta(t) = \iint \alpha(t) + E_\theta(t) \cdot t^2 \qquad \textit{Eq. 29}$$

Despite this unfortunate drifting error, already developed solutions [37], [38] have been tested to reduce the drift as much as possible. Among these solutions, offset calculation and proper calibration are introduced. Pitch and roll axis values are acceptable although yaw's values couldn't be corrected and are constantly increasing even when the object is not being moved.

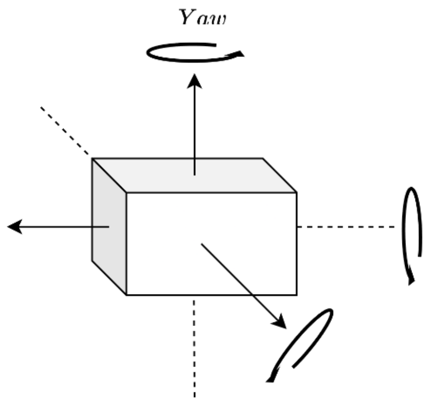


Figure 20. *Roll, Pitch and Yaw axis*

Despite that, it is considered that the most critical deviations are those happening in Pitch and Roll axis, which values are working correctly. Moreover, the drifting issue regarding linear acceleration could not be solved without a gigantic error and therefore, it will not be used to track distances between end-effectors and the object. Taking this in consideration, the MPU6050 is still used to monitor the object deviation posture, while the distances are being calculated by the sensors HCSR04, as explained in its section.

Meanwhile, the joints angles are easily obtained through the integrated control system that in each servomotor MG996R.

When the object orientation, distances and joint angles values are obtained by the ESP32, these are sent through the Serial port, using UART protocol, to be processed in the policy, together with the reward.

Once the agent has processed the states, the Actors' network will produce new actions. These actions are float values that are packed and sent through the Serial port, again using the UART protocol. Once the values are received by the ESP32, these are unpacked using pointer casting, which interprets the packed bits as float values. Furthermore, once float values are obtained, they are mapped to angle degrees, within a safety range of 0º to 160º and sent to the motors using the following library "`<Adafruit_PWMServoDriver.h>`". This library is used together with the PCA9685 driver. Mapped angle values are transformed to PWM values and sent to each output of the driver board which is connected to a servomotor.

5.6. Implementation overview diagram

In this chart, similarly to "*Simulation data flow diagram*", arrows are used to indicate where data is going to. For instance, sensory data is flowing from the robots to the ESP32 to finally be processed in the model while actions values are travelling from the agent to the robots, through the ESP32 and the module PCA9685.

Notice that the communication type used between each component is described in the arrows (I2C, UART). Moreover, the States, Terminal and Reward boxes are a visual representation of the information the robots are providing to the system. In reality, these values are just sent through the Serial port to the main Computer.

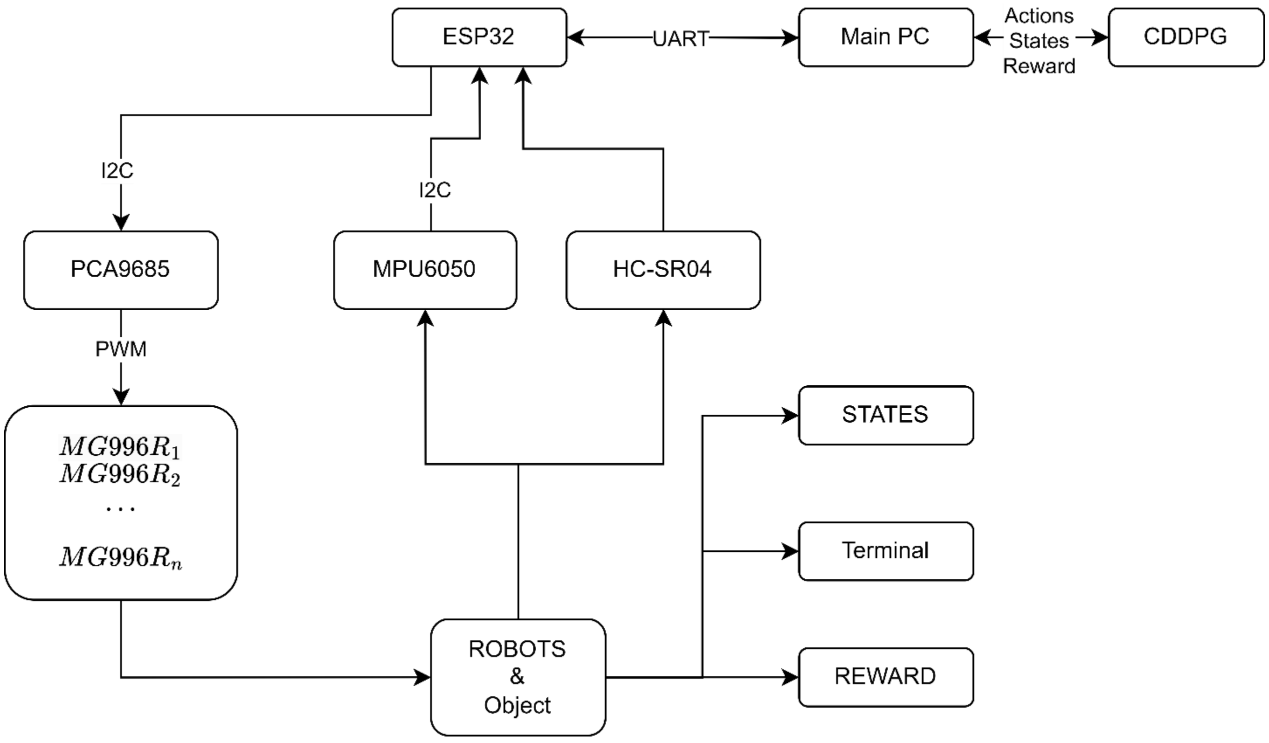


Figure 21. *Implementation data flow chart*

Please, refer to the GitHub repository where this project is stored for further information about the programs and communications. Specifically, files related to the implementation section will be found inside the /Implementation/ directory.

6. Economic analysis

Table 4 contains individual prices, quantities required, and total cost for each component used in the project. Notice that these values should be considered in potential replications of the project in the future and while the prices should be, if not identical, very similar, certain elements may vary.

Furthermore, depending on the exact setup of the replica or variant of the project, the number of passive components and active components may fluctuate. For instance, if pursuing an expansion of the robots' capabilities, additional MPU6050 sensors or camera modules would be required, considering that cameras would replace HC-SR04 and there would not be any need of them anymore.

Finally, note that the project has been developed in Taiwan (2023-2024). These prices may be different in other countries although after thorough research it can be said that the items' prices are mostly equal.

Component	NTD	€	Quantity	NTD	€
Active Components					

ESP32	280	8.4	1	280	8.4
PCA9685	125	3.75	1	125	3.75
MG996R	175	5.25	10	1750	52.5
MPU6050	75	2.25	1	75	2.25
HC-SR04	40	1.2	2	80	2.4
RS-75-5	470	14.1	1	470	14.1
Passive Components					
Capacitors	12	0.36	3	36	1.08
Resistors	7	0.21	8	56	1.68
Wiring					
Breadboard	150	4.5	2	300	9
Jumping cable	7	0.21	18	126	3.78
Structure					
Aluminum parts	515	15.45	2	1030	30.9
Screws	1	0.03	68	68	2.04
Screwdriver	110	3.3	1	110	3.3
TOTAL				4506	135.18

Table 4. *Individual and collective component price table*

7. Environmental analysis

Among the negative impact causes of carbon footprint and other sustainability issues are the active and passive electronic items, structural materials and energy dispensed while developing the algorithms.

Firstly, electronic components have a direct and indirect impact on the environment both in manufacturing and transport, respectively. In the context of this project, which in the large scale has a small impact, it is not clear how to calculate the repercussions in the environment. Despite this and after thorough research [40], [41] a UK government research in 2021 states that over a range of electronic and IT devices, the carbon intensity per kg of product is $24.8650\text{kg } CO_2$. Therefore, an indirect amount of approximately 25kg emitted carbon into the atmosphere is the result of 1kg of electronics components produced.

The electric and electronical items used in this project have been locally bought and are the listed in **Table 4**. A roughly kg value of carbon emissions can be obtained by weighting all the components used. The approximate weighting of all electronic components (excluding wires and breadboards) is of 0.85 Kg, being the MG996R motors the most important elements. Furthermore, breadboards, jumper cables and aluminum parts are analyzed as their main primary elements, polyethylene, copper, and aluminum, respectively.

Component	Weight (g)	CO2 (kg)
Breadboard	120	0.332
Jumper Cables	70	0.228
Aluminum parts	300	4.35
Electronic components	850	21.135
Total		**26.045**

Table 5. *CO2 emission per group of components*

Table 5 shows the weight of each group of items as well as the conversions to kg of CO2 emissions [42]. Therefore, the amount of carbon emitted by the robots is $R_{CO_2} = 26.045\ kg\ CO_2e.$

Moreover, another factor when calculating the carbon footprint is the power consumption of the modules. In this case, robots and pertinent electronics consume between 9.5 A and 12 A and 4.7 V to 6.7 V. Taking average values of Intensity and Voltage, which is the mean consumption of the robots, it is possible to calculate the kWh. The following equations show how to obtain the carbon emissions using an Electricity Carbon Emission Factor in Taiwan of $ECEF = 0.509\ kg \frac{CO_2e}{kWh}$.

$$E_{kWh} = \frac{P_w \cdot t_h}{1000} = \frac{I_{avg} \cdot V_{avg} \cdot t_h}{1000} = 0.0559\ kWh \qquad \textit{Eq. 30}$$

$$CO_{2,E_{kWh}} = E_{kWh} \cdot ECEF = 0.02845\ kg\ CO_2 \qquad \textit{Eq. 31}$$

Then the total carbon emissions for this project can be calculated using **Eq.** 32. Notice that the most impactful area regarding the carbon emissions are the electronic components, mainly the motors in this case, and the aluminum parts.

$$Carbon\ Emissions = CO_{2,E_{kWh}} + R_{CO_2} = 26.0734\ kg\ CO_2 \qquad \textit{Eq. 32}$$

Looking at the results, it is clear that if fewer electronic components are used, a smaller impact is propagated to the environment. Optimizing the number of motors on each robot could potentially reduce the amount of carbon as well as reducing technical complexity.

Despite the carbon impact of the robots, the overall goal of the project indirectly enhances sustainability. Diminishing the reality gap and prioritizing simulation-based training reduces the need for real-world training, resulting in less components damaged by collisions or friction which at the same time increases the life cycle of the items.

8. Reality Gap

In this project, the words "Reality Gap" have been used abundantly. One of the main goals of the project is intended to reduce the existing errors when training a real agent with a model trained in simulation. As stated in the "Literature review", other investigators already proposed several techniques [11]–[13] focused on bridging this gap, and some of these are introduced in the project.

Domain randomization:

In [12] an study of different training algorithms including domain randomization, which involves randomly perturbing various aspects of a simulated environment in order to make trained agents robust to the reality gap, is implemented. This technique provided them with better results while preparing the agents for unforeseen and a wider range of events.

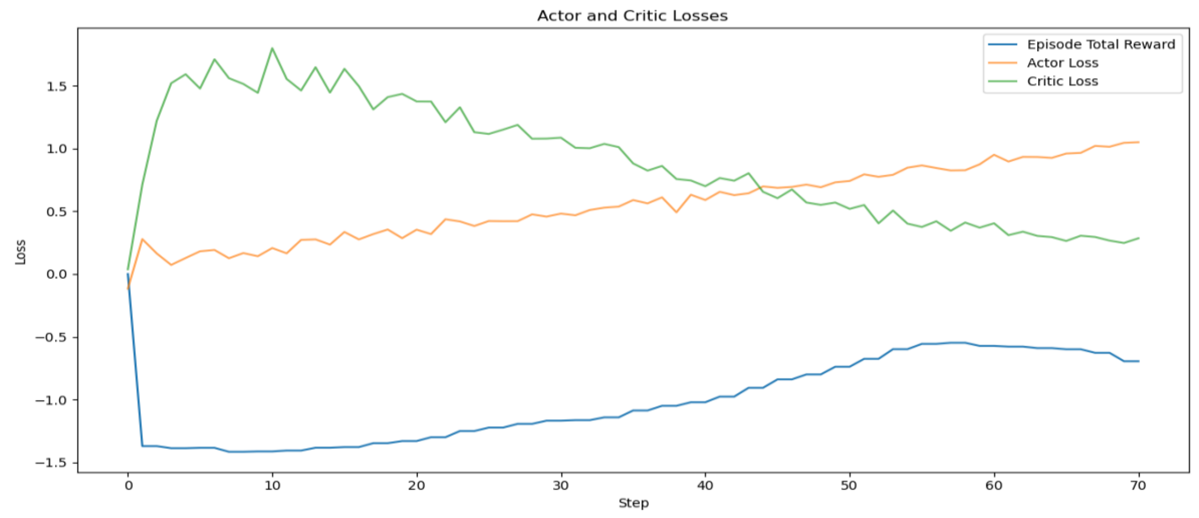


Figure 22. *Losses and reward value after adding dropout, domain randomization and other techniques.*

Therefore, in this project a simpler version of domain randomization as well as initial random state values have been added. In **Figure 22** the results when domain randomization and other techniques are added. Adding random values, or biases, to the training process, leads for longer times when converging and makes the model unstable at the beginning while in the long run the model becomes robust and more used to uncertainties.

<u>Transfer learning:</u>

This technique has been used several times involving LLMs [39]. Although this is true, it is also used in Reinforcement Learning to endow agents with context data even before the training. In this project this has been incorporated by allowing the user or researcher to choose between training a model from scratch or using a pretrained model developed in a simulated environment.

Using a pretrained model gives the real agents some context about the task to be done and how to carry it out. It is also possible to use the pretrained model only for evaluation purposes or some layers can be frozen to provide information to the robots while keeping the training going.

The number of frozen layers will depend on the resemblance between simulation and implementation. A more accurate representation of the robots in a simulator will provide better data to the real robots when transferring learning, while for different tasks or inaccurate representations of these it is recommended to unfreeze some of the layers to allow the robots to adapt to unforeseen events.

The choice of training from scratch or using a pretrained model has been automatically included in the software of this project. In the GitHub repository it is explained how to choose between the options with ease.

Online adaptation:

Real implementation of the robots allows for changes while finetuning or training from scratch which provides a controlled process and the option to introduce changes without terminating the learning. This technique is also applied in other works [13] successfully. In the context of this project, the technique has been simplified but acts as a supervisor which introduces safety and control to the training process. The implementation of this technique can be found in the GitHub repository, inside the file /Implementation/Inference/main.py.

Although it does not directly improve the results, for now, it adds a layer of control and safety, in case unexpected events happen. A robust implementation of this component could endow the agents with even more control than it has now.

Ideally, when training a perfect representation of the robots in a simulator and transferring the learning to the real robots, these would behave in the exact same way as the simulation. Although that is the ideal, this gap needs to be dealt with in the real world. These are some techniques that have been applied to overcome the inaccuracies between the simulation and reality implementations. Finally, when analyzing the results of this project, the limited number of resources and time, as well as the learning curve of all the frameworks must be considered.

In conclusion, this project applies different techniques intending to reduce bridge the reality gap. While in some areas it works, in general this goal has not been fully accomplished. That is why, on one hand there are still several features and errors to overcome and improve, therefore the material used should be carefully analyzed before usage. On the other hand, this work could perfectly be considered as a reference or pipeline for future research due to the efficiency in some features and techniques described in previous sections.

9. Results

This project provides mainly two results. Firstly, the evaluation of the model when some techniques such as domain randomization and replay buffer, among others, are added. Secondly, the results when the model is applied to the robots. It needs to be said that unfortunately, the results from the implementation section are not as good as expected. Despite that, with more time and research, upgrades in both implementation and simulation can be introduced thus improving the results. Some of these upgrades can be found in "Future Work".

On one hand, the Simulation results in **Figure 23** show a continuous improvement of the reward function, and both Actor and Critic losses are behaving as expected. The Critic loss purses the minimization between the actions provided by Actor on a certain state and the target states while the Actor network learns to maximize the Q-Values, that is why its loss value has an upper trend. These values have been obtained after thorough iterations and implementing several techniques which are intended to improve the results.

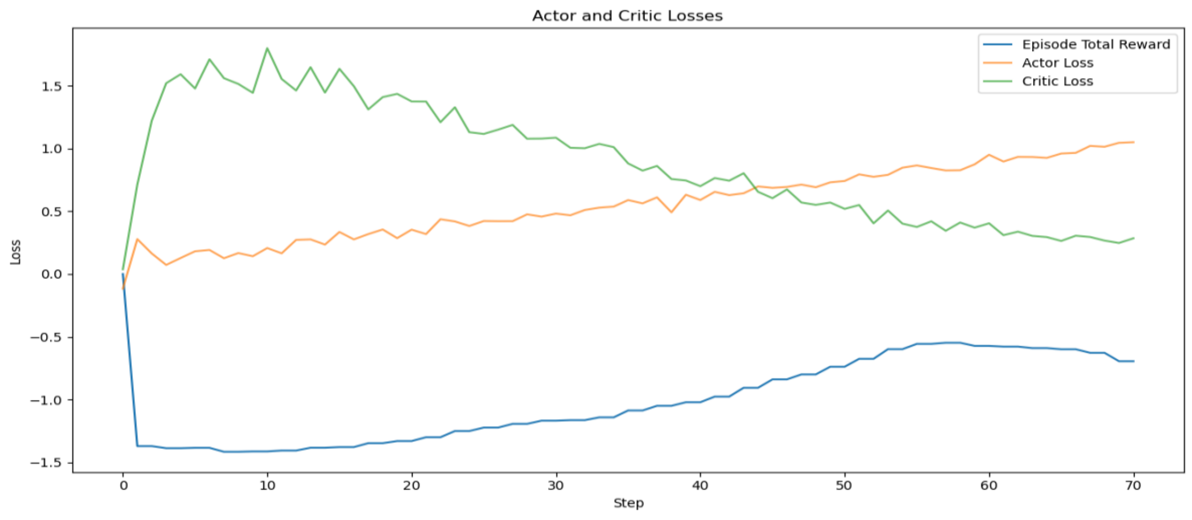


Figure 23. *Final Simulation results in Gazebo simulator and ROS 2*

On the other hand, the real-world implementation outcomes did not align with expectations. While the model shows indices of learning, based on the **Figure 24**, their actual movements are deviated from the expected behavior. This figure clearly shows an unstable learning process from the Critic network and reward, which are neither worsening nor improving, while, at the same time, shows a slight constant improvement in the Actor network. In summary, the actor manages to produce better actions for the current states but at a slow pace while the Critic is not able to guide it. Therefore, the reward value is affected by this, and it is shown through this unstable wave form.

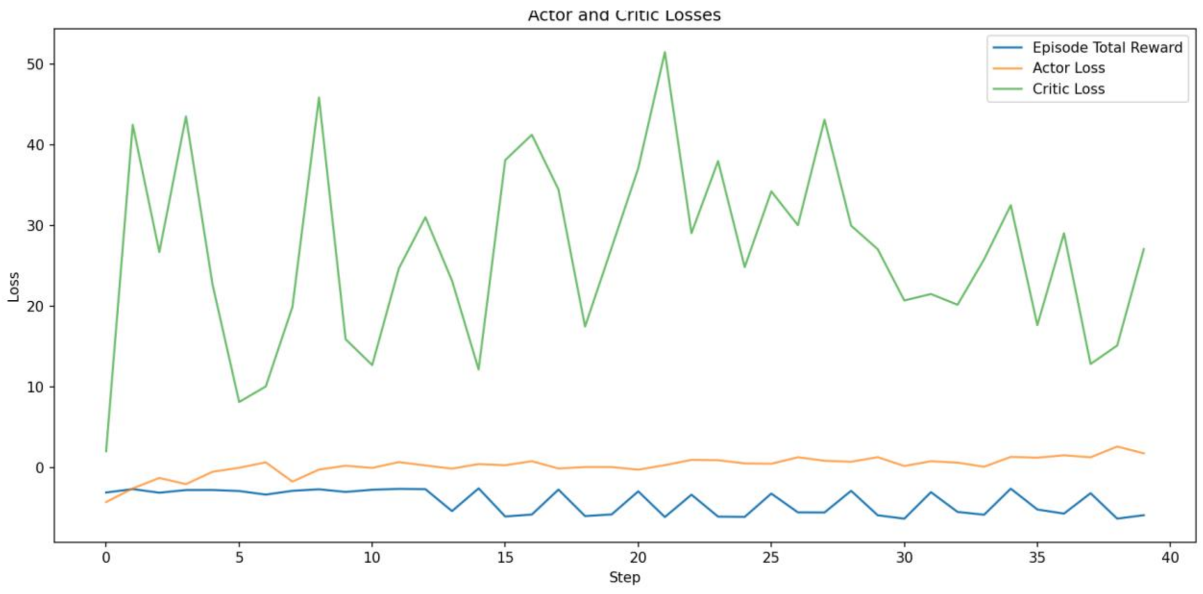


Figure 24. *Metric values when implementing the CDDPG model to a real-case scenario.*

Finally, after thorough iterations, processing steps and study of the issue, a slight improvement is accomplished, although the Critic behavior is not even close to the ideal. The last statement is made clear when looking at the reward values, which show limited improvement in **Figure 25**. Furthermore, the initial steps of the training, which can be seen in the amplified **Figure 26**, present a more stable process.

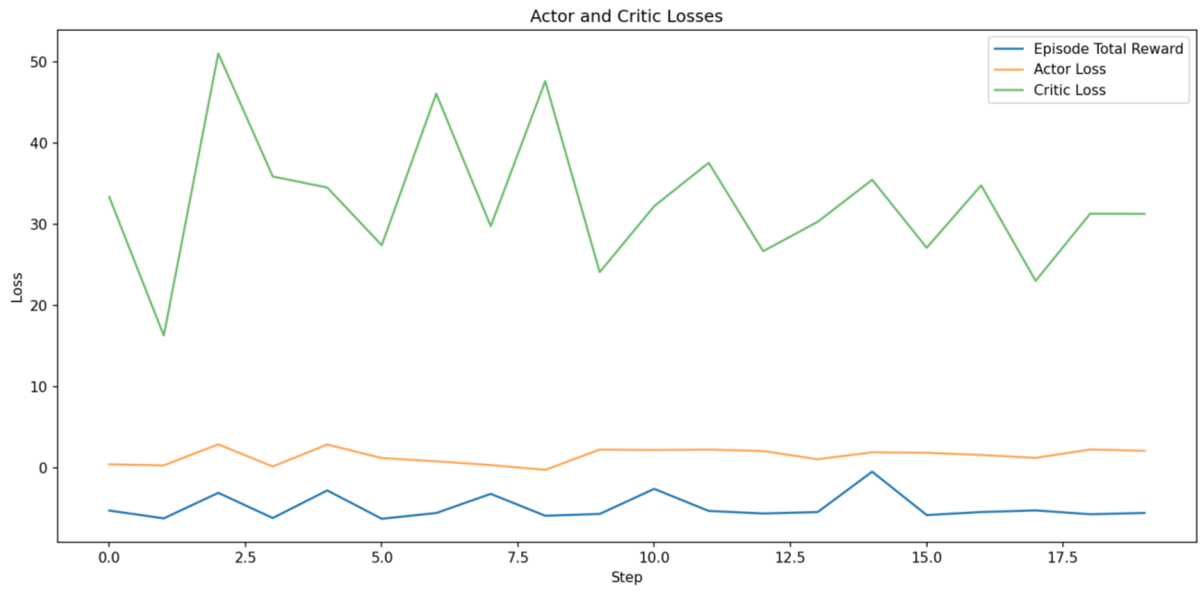


Figure 25. *Implementation outcomes after some iterations.*

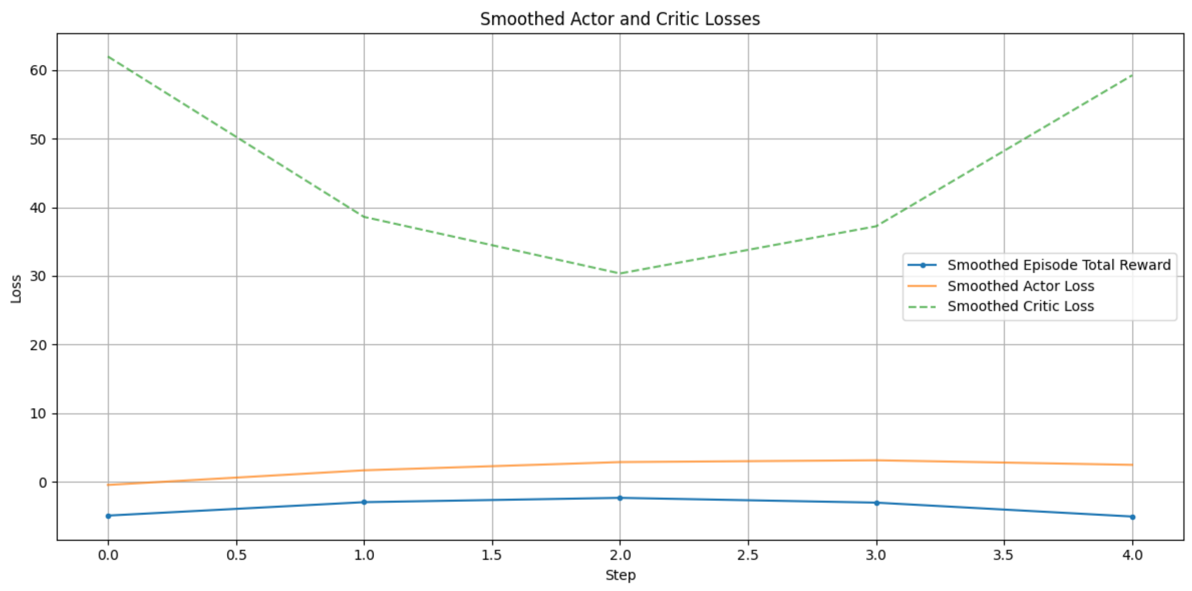


Figure 26. *Initial steps of the outcomes.*

This behavior can be explained by the possible asynchronization of some elements in the communication between the agent and the hardware. Although it could be improved over time, it needs to be said that the root of this problem is attributed to a significant oversight in the projects' design phase, attempting to encompass too many aspects within a limited amount of time. Recognizing this fundamental mistake is essential.

10. Conclusion

This project encompasses objectives in both personal and academic dimensions. On one hand, to accomplish the personal goals, this investigation aimed to enhance and apply the skills acquired during the pursuit of a bachelors' degree in electronics engineering. On the other hand, the academic goal specifically targeted the reduction of the reality gap in a Multi-agent Reinforcement Learning (MARL) environment. This also involved, among other processes, training a Deep Reinforcement Learning (DRL) policy in simulation and transferring it to a rea-world scenario.

Evaluating the success of the objectives provides a mixed outcome. Firstly, in the academic area a partial success was achieved. On one side, the simulation section provided interesting and satisfactory results allowing the agent to learn successfully from experiences. One the other side, the implementation phase did not provide the expected outcomes. Although technical explanations could be given and addressed and improved over time, the root issue was the critical oversight during the project management design, attempting to handle too many aspects within a very limited timeframe without considering personal limitations in skills, knowledge, and learning curve of the applied frameworks, which took longer than estimated in initial phases of the project. This factor mainly contributed to the suboptimal results.

Secondly and turning to personal goals, they were fully accomplished. The successful familiarization and introduction of new frameworks such as ROS 2, Gazebo, and Linux, along with a noticeable improvement in Python and C++ skills, marked significant achievements. Moreover, along the pursuit of the objectives, several iterations accompanied by a multitude of errors provided valuable and long-lasting insights and rapid skill enhancement. These factors will contribute in better designs and outcomes in future projects.

Furthermore, the project's resources, including the implemented policy (CDDPG), test files, and classes, are robust and reusable for future projects with similar implementation needs in the domain of Multi-Agent Reinforcement Learning and Deep Reinforcement Learning policies.

In conclusion, the project served its purpose in an acceptable manner, which allowed appliance of knowledge and resources to develop an investigation concerning state-of-the-art issues. Moreover, a fundamental mistake needs to be recognized in project management design, preventing the full development of both the Simulation and Implementation components. Despite the creation of a customized and robust system within the project's four-month duration, some advice for future works and projects is to carefully contemplate the resources and time possessed, and act along this information to avoid incorporating too many elements simultaneously. In the long run a focused approach as well as delving deeply into a singular topic. A mastery over a smaller set of concepts proves more beneficial than a broad understanding across various areas, in the technical world.

11. Future Work

This project can be used as a pipeline to pursue future research. In GitHub there is a step-by-step tutorial explaining how to replicate the project, both simulation and implementation parts. Although, there are some considerations to keep in mind.

Simulator:

In "Technical advice", a simulator from NVIDIA is proposed. Although Gazebo is a viable option which allows for deep customization and freedom, other simulators might be easier to control. That is why, after thorough research, we propose NVIDIA as a replacement for Gazebo, or at least, as another way to go.

Microcontroller:

In the implementation phase, the robots are controlled by an ESP32 which is a microcontroller with Wi-Fi and Bluetooth, I2C, SPI and several types of communications while providing many GPIO in a small space, while being considerably cheap. While this µC is a good choice for this project, if adding more complexity or changing the project structure, a different controller might be better.

In case of wanting to run the algorithm directly on the microcontroller, ESP32 falls back by not having enough computational power. That is why we propose upgrading it to a Raspberry Pi. Although it is more expensive, no main computer is needed, and it can be used to both manage data to and from the robots and store the DRL models. Furthermore, a hybrid approach where the sensory and actuators control is being controlled by an ESP32 and the algorithm data processing is happening in the Raspberry Pi.

This change should be made when looking for an upgrade and under the rationale of each implementation setup and project requirements.

Object detection:

Since MPU6050 was rejected as an option to triangulate the position of the object, HC-SR04 was used. This approach, although simple and effective, is prone to errors and not very precise. In future work, researchers may want to use cameras to detect the object, instead of ultrasonics sensors.

This approach enhances the robots' capabilities of obtaining data or providing further guidance when being finetuned. While it adds more hardware and software complexity, the results would be much more precise and complete. If this option is pursued, a computer vision or image detection model should be developed and trained to find the right object.

Actuators:

MG996R motors are low in cost and easy to set up. While being a good fit for the robots' structure, their precision is not always the best and its mechanical quality may conduce to unwanted movements. If an upgrade wants to be pursued, we recommend replacing servomotors with stepper motors. At least the ones at lower levels that need more torque and control, such as the base or the first joint. Stepper motors are a bit more expensive and complex to set up, although their robustness and precision compensate for the complexity added.

Reward function:

The reward function that guides the agents contains three main conditions, which can be find in "*Reward function*". In Reinforcement learning, reward functions are the core of the models. A good reward function usually produces good results. That is why we encourage finding new ways of defining reward values or pursuing different strategies. In fact, in MARL contexts, some papers [14] recommend using different reward functions for each agent that, while adding more complexity, may separate tasks and provide better results overall.

Reality gap:

One of this projects' goals was to bridge the reality gap that is generated when training a model in simulation and test it on a real scenario. Regrettably, this goal has not been fully achieved. While it is true that some elements and techniques have been incorporated intending to mitigate the gap, which at some level it has worked, it is also true that the inherent differences between real robots and their simulated counterparts produced the opposite effect.

Additionally, a factor that added another layer of complexity was trying to tackle too many things at once, concurrently unintentionally led to limitations in precision. Even though a complex and operational system has been successfully developed, wanting to pursue a deep level of understanding many different things, led to not having enough time to dig deep into any of them.

For future projects considering this one as a guide, a recommendation is to prioritize simplicity in the system design. Emphasizing simplicity allows to focus on core issues, facilitating a more thorough exploration of each aspect. This focus on simplicity is crucial for gaining a deeper understanding and achieving targeted resolutions to challenges, ultimately contributing to the advancement of project goals.

11.1. Technical advice

This project was aiming at improving the skills learned while pursuing a bachelor's degree in electronic engineering. Practical projects such as this allow for a combination of technical and theorical practices which made it a perfect opportunity to dig into.

Simplicity:

If future researchers want to continue or start projects regarding sim-to-rea-transfer in reinforcement learning it would be wise to choose a very simple practical approach. In this project, two 5 degree of freedom robotic arms have been built, adding such complexity while trying to reduce the reality gap, an important problem nowadays, becomes a tough and time-consuming task. This is why we recommend choosing a simpler robot. Such simplicity may help at the beginning to focus on having a working and clear setup and, once it is working, scale it if possible.

Regarding reality gap reduction, the most element is the simulated environment be as similar as possible to the real setup. Especially when dealing with custom robots, a perfect description of such must be developed, using either in URDF or SDF files.

Simulator:

Gazebo simulator and ROS 2 are used in this project and are good options when dealing with an environment with multiple robots. ROS 2 provides an organized setup while can communicate with Gazebo. Despite that, it needs to be taken in consideration the number of different versions that both Gazebo and ROS 2 have. Few of them can be put together and the ones that do, have several limitations.

That is why another simulator, discovered at the end of the research period, is proposed. NVIDIA developed a robot framework focused on training robots for reinforcement learning in 2021, and in February 2023 they integrated it with a simulator, called Isaac Gym [43]. Robots can be customized if a detailed URDF file is provided, and it can be controlled using APIs and Python.

Gazebo allows for flexibility and extreme customization although at the same time is more complex to communicate with. Isaac Gym, after some initial tests and once the APIs are learnt, is quite simple to use and control, although full customization would take longer time.

In conclusion, both simulators have their own advantages and drawbacks, and should be used depending on the setup and characteristics of the project.

12. Acknowledgements

Firstly, I would like to sincerely thank 柯士文, George Ke, who supervised my project during the semester, always providing me guidance and resources while sharing knowledge and wisdom through research and interesting discussions.

Secondly, special gratitude to my family and friends who always supported me both in distance and proximity. Especially towards my mother and father, Maite and Fèlix, with their endless patience and support.

Finally, this project has been possible thanks to the program and agreements between UPC and NCU institutions. I would like to thank UPC for the opportunity to travel as a student during my final stages in my bachelor’s degree while also thanking NCU for its support during the year 2023.

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] Y. Tassa *et al.*, ‘DeepMind Control Suite’, Jan. 2018.

[10] A. Rajeswaran *et al.*, ‘Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations’. arXiv, Jun. 26, 2018. doi: 10.48550/arXiv.1709.10087.

[11] 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

[12] T. Dai, K. Arulkumaran, T. Gerbert, S. Tukra, F. Behbahani, and A. A. Bharath, ‘Analysing Deep Reinforcement Learning Agents Trained with Domain Randomisation’, arXiv.org. Accessed: Jan. 07, 2024. [Online]. Available: https://arxiv.org/abs/1912.08324v2

[13] B. D. Evans, J. Betz, H. Zheng, H. A. Engelbrecht, R. Mangharam, and H. W. Jordaan, ‘Bypassing the Simulation-to-reality Gap: Online Reinforcement Learning using a Supervisor’. arXiv, Jul. 13, 2023. doi: 10.48550/arXiv.2209.11082.

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

[15] 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

[16] ‘Visual Studio Code’.

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

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

[19] Y. Bai *et al.*, ‘Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback’. arXiv, Apr. 12, 2022. doi: 10.48550/arXiv.2204.05862.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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.

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

[32] 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.

[33] N. Srivastava, 'Improving Neural Networks with Dropout'.

[34] S. Zhang, C. Wang, S.-C. Chan, X. Wei, and C.-H. Ho, 'New Object Detection, Tracking, and Recognition Approaches for Video Surveillance Over Camera Network', *IEEE Sensors Journal*, vol. 15, no. 5, pp. 2679–2691, May 2015, doi: 10.1109/JSEN.2014.2382174.

[35] A. Coates and A. Y. Ng, 'Multi-camera object detection for robotics', in *2010 IEEE International Conference on Robotics and Automation*, May 2010, pp. 412–419. doi: 10.1109/ROBOT.2010.5509644.

[36] 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.

[37] A. Valdeperes *et al.*, 'Wireless inertial measurement unit (IMU)-based posturography', *European Archives of Oto-Rhino-Laryngology*, vol. 276, Nov. 2019, doi: 10.1007/s00405-019-05607-1.

[38] Y.-L. Tsai, T.-T. Tu, H. Bae, and P. H. Chou, 'EcoIMU: A Dual Triaxial-Accelerometer Inertial Measurement Unit for Wearable Applications', in *2010 International Conference on Body Sensor Networks*, Singapore, Singapore: IEEE, Jun. 2010, pp. 207–212. doi: 10.1109/BSN.2010.47.

[39] W. U. Ahmad, X. Bai, N. Peng, and K.-W. Chang, 'Learning Robust, Transferable Sentence Representations for Text Classification', Sep. 2018, Accessed: Feb. 27, 2020. [Online]. Available: https://openreview.net/forum?id=Sye8S209KX

[40] M. Gautam, B. Pandey, and M. Agrawal, 'Chapter 8 - Carbon Footprint of Aluminum Production: Emissions and Mitigation', in *Environmental Carbon Footprints*, S. S. Muthu, Ed., Butterworth-Heinemann, 2018, pp. 197–228. doi: 10.1016/B978-0-12-812849-7.00008-8.

[41] J. Pickstone, 'The environmental impact of our devices: revealing what many companies hide', The Restart Project. Accessed: Jan. 06, 2024. [Online]. Available: https://therestartproject.org/consumption/hidden-impact-devices/

[42] A. Alsabri and S. G. Al-Ghamdi, ‘Carbon footprint and embodied energy of PVC, PE, and PP piping: Perspective on environmental performance’, *Energy Reports*, vol. 6, pp. 364–370, Dec. 2020, doi: 10.1016/j.egyr.2020.11.173.

[43] V. Makoviychuk *et al.*, ‘Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning’. arXiv, Aug. 25, 2021. Accessed: Jan. 02, 2024. [Online]. Available: http://arxiv.org/abs/2108.10470

[44] ‘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

This section contains deeper information and technicalities that are not considered important enough, for the understanding of the project, to be on the main body. The explanations you will find here are mostly expansions of an overview located in the main body.

SDF files

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, …).

To dig into SDF structure a little bit more the following link explains <tags> in an SDF file: http://sdformat.org/spec

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*" provides a network bridge that allows us to send messages between ROS 2 and Gazebo, limited to certain (but not few) types of messages. 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 the ending bracket].

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):
    a. `cd ~/<name_project>/src/`
    b. `ros2 pkg create --build-type ament_python <node_n> <package_n>`
6. Build the package:
    a. `cd ~/<name_project>`
    b. `colcon build` → Builds all packages
    c. or `colcon build --packages-select <package_n>`
7. Source the overlay ROS 2 workspace in the new terminal (so the new packages are available)
    a. `source install/setup.bash`
8. Before making a bridge between ROS2 and Gazebo, let's run the simulation with:
    a. `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.

Before pointer casting is applied, the register contains only a byte array, of length 6 in this case, denoted in **Figure 27** as **d**. The symbol "^" indicates the starting point of the pointers and the tilde "~" represents the memory occupied by each element.

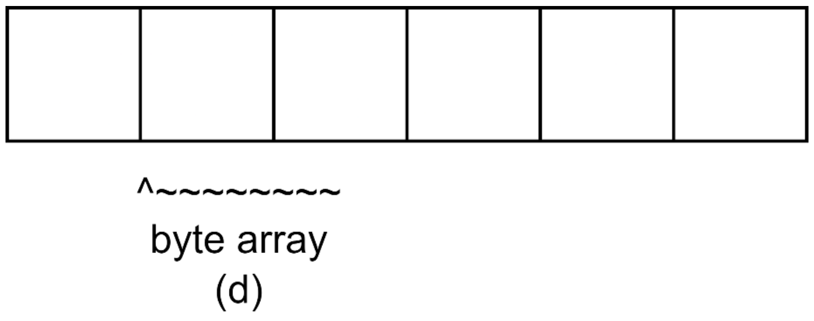


Figure 27. *Byte array before casting the pointer.*

Pointer casting involves interpreting the same memory locations as a different type, in this case, a float array **f1** to **f6**, defined as **D**.

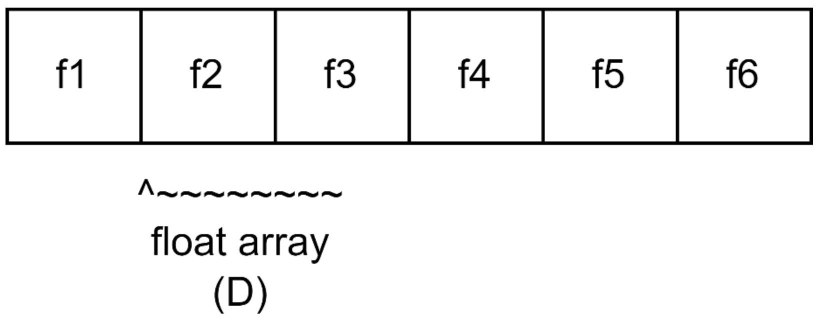


Figure 28. *Byte array interpreted as a float array.*

By casting the pointers, the data can be packed and unpacked in different devices or reinterpreted as different types of data while being in the same memory locations.

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 datasheet

The specific model being used is ESP-Wroom-32 [44]. The main features used in the project are listed below:

- Wi-Fi and Bluetooth modules.
- 24 GPIO.
- I2C and UART communications.
- PWM outputs.
- 5 V and 3.3 V outputs.
- Operating current: 80 mA.
- Operating voltage: 3.0 V ~ 3.6 V.
- Pin Layout in **Figure 29**.

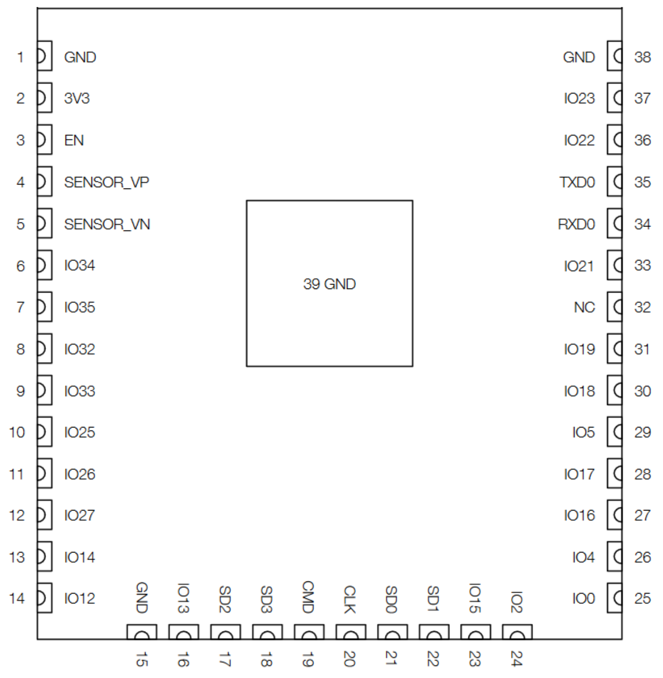


Figure 29. *ESP-Wroom-32 pin layout.*

RS-75-5 Datasheet

To free this document from unnecessary lengthy description, this is an overview of the main features of the power converter RS-75-5.

MODEL		RS-75-5
OUTPUT	DC VOLTAGE	5V
	RATED CURRENT	12A
	CURRENT RANGE	0 ~ 12A
	RATED POWER	60W
	RIPPLE & NOISE (max.)	80mVp-p
	VOLTAGE ADJ. RANGE	4.75 ~ 5.5V
	VOLTAGE TOLERANCE	±2.0%
	LINE REGULATION	±0.5%
	LOAD REGULATION	±1.0%
	SETUP, RISE TIME	500ms, 30ms/230VAC 1200ms, 30ms/115VAC at full load
	HOLD UP TIME (Typ.)	60ms/230VAC 14ms/115VAC at full load
INPUT	VOLTAGE RANGE	88 ~ 264VAC 125 ~ 373VDC (Withstand 300VAC surge for 5sec. Without damage)
	FREQUENCY RANGE	47 ~ 63Hz
	EFFICIENCY(Typ.)	79%
	AC CURRENT (Typ.)	2A/115VAC 1.2A/230VAC
	INRUSH CURRENT (Typ.)	COLD START 40A/230VAC
	LEAKAGE CURRENT	<2mA / 240VAC
PROTECTION	OVERLOAD	110 ~ 150% rated output power
		Protection type: Hiccup mode, recovers automatically after fault condition is removed
	OVER VOLTAGE	5.75 ~ 6.75V
		Protection type: Hiccup mode, recovers automatically after fault condition is removed
ENVIRONMENT	WORKING TEMP.	-25 ~ +70℃ (Refer to "Derating Curve")
	WORKING HUMIDITY	20 ~ 90% RH non-condensing
	STORAGE TEMP., HUMIDITY	-40 ~ +85℃, 10 ~ 95% RH
	TEMP. COEFFICIENT	±0.03%/℃ (0~50℃)
	VIBRATION	10 ~ 500Hz, 5G 10min./1cycle, period for 60min. each along X, Y, Z axes

Table 6. *Main characteristics of RS-75-5.*

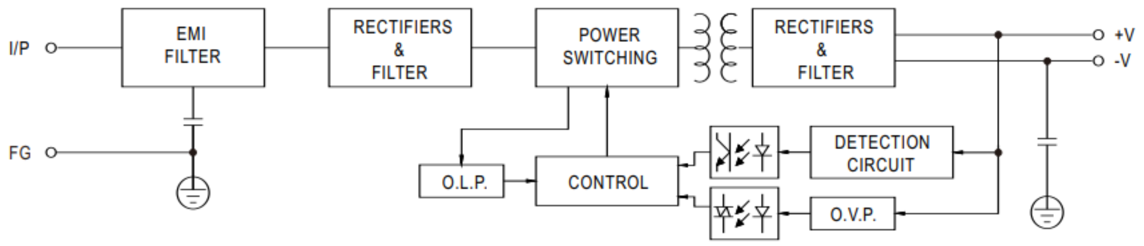
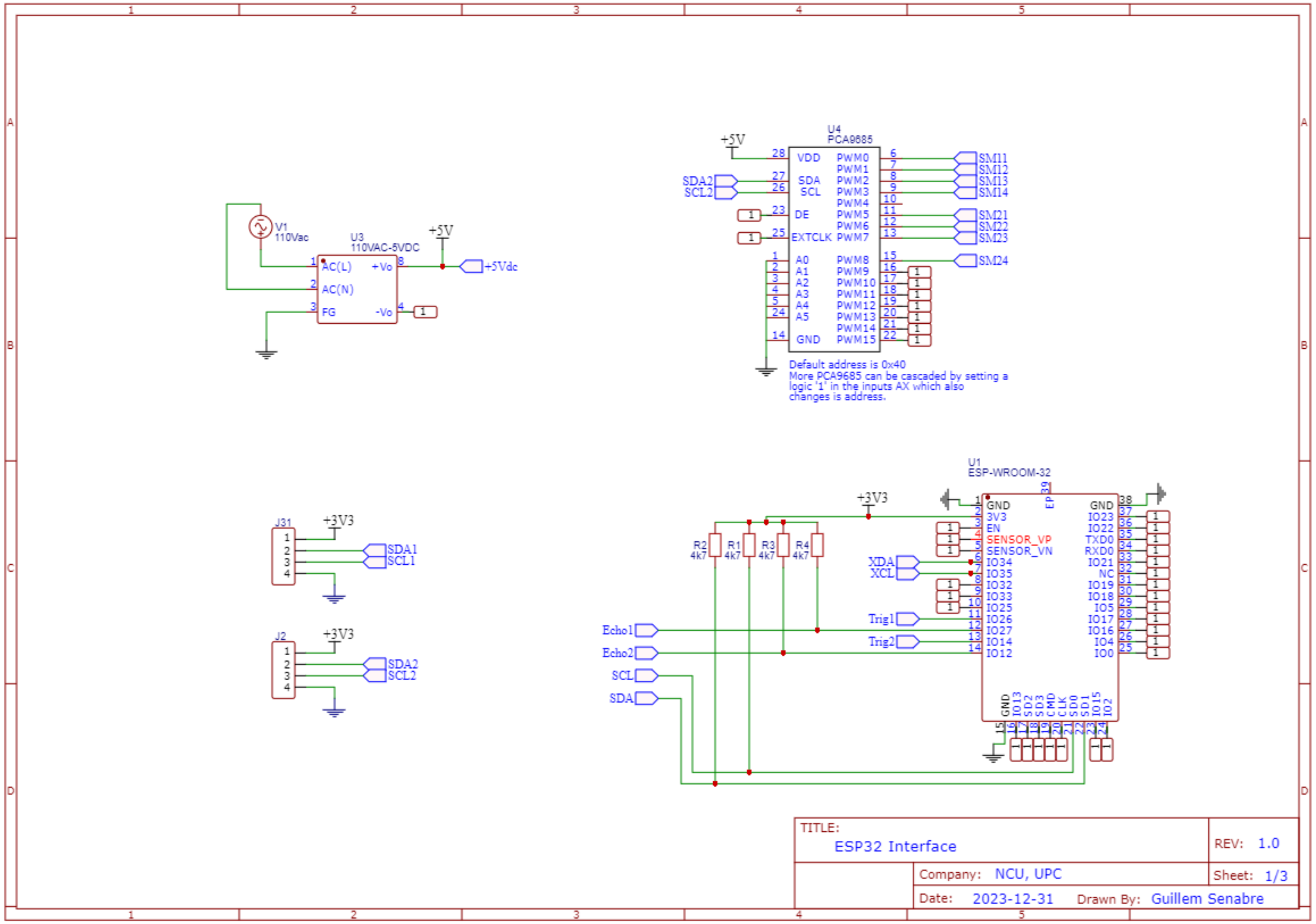
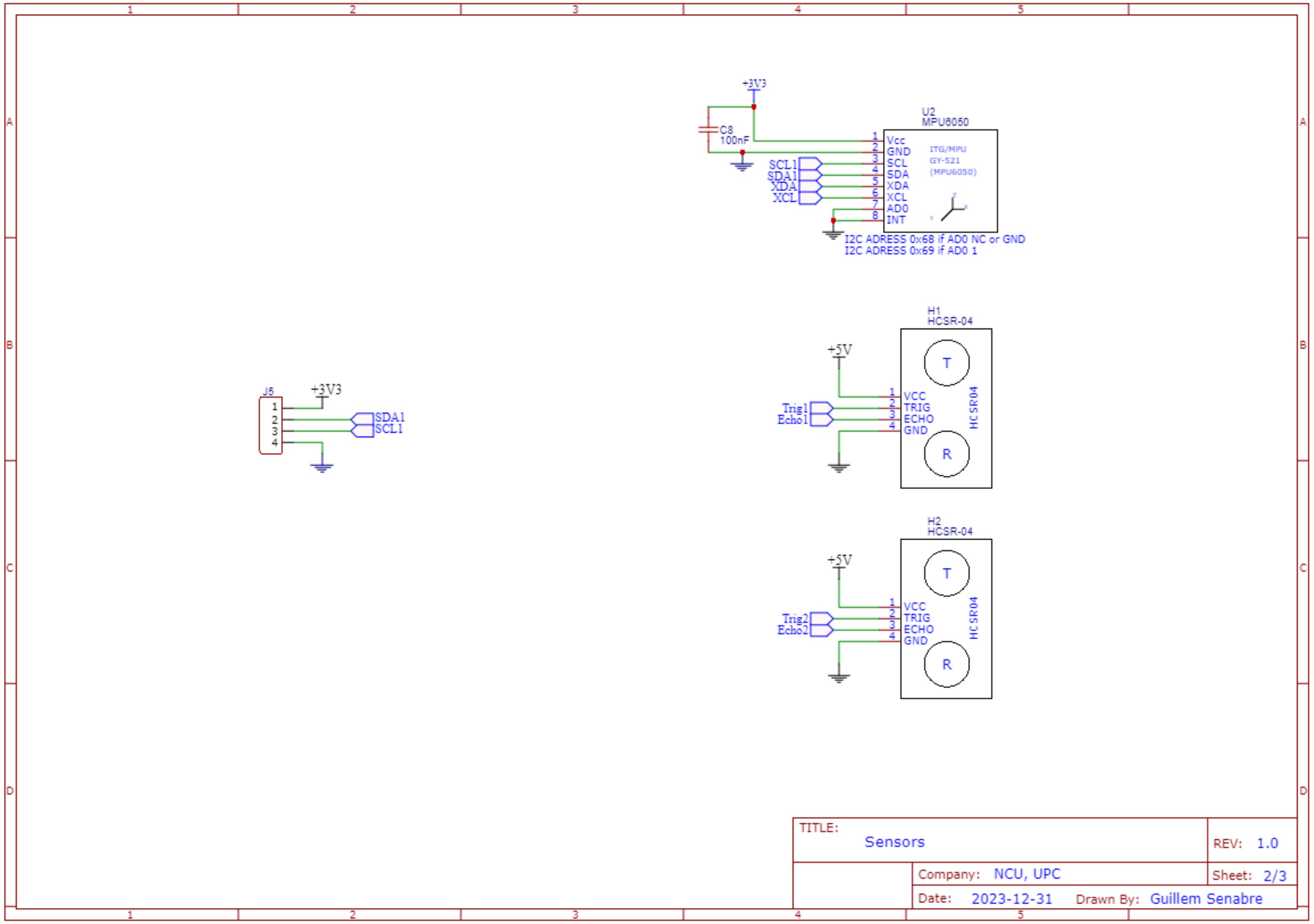


Figure 30. *Block diagram of RS-75-5.*

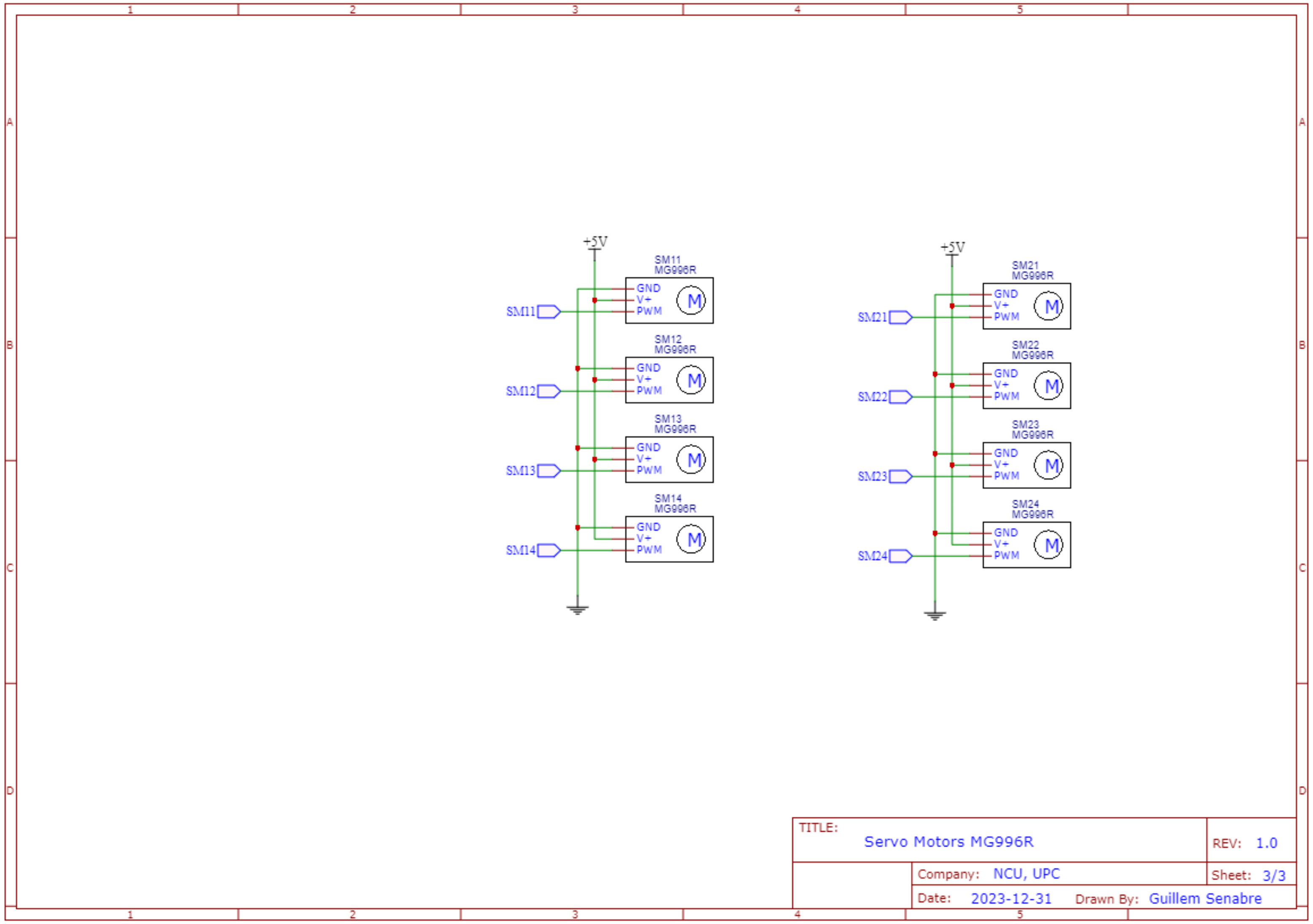
ESP32 | PCA9685 | 110VAC/5VDC Converter



HCSR-04 | MPU6050



MG996R Servo Motors


+5V
SM11
MG996R
SM12
SM13
SM14
SM21
SM22
SM23
SM24
GND
V+
PWM
M
TITLE: Servo Motors MG996R
REV: 1.0
Company: NCU, UPC
Sheet: 3/3
Date: 2023-12-31
Drawn By: Guillem Senabre