UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Reinforcement Learning for Controlling Cable Driven Parallel Robots

THÈSE

présentée et soutenue publiquement le 13 Décembre 2024

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention automatique, traitement du signal et des images, génie informatique)

par

Abir Bouaouda

Composition du jury

Président :	Jacques Gangloff	Professeur des universités, Université de Strasbourg
Rapporteur :	Abdel-Illah Mouaddib	Professeur des universités, Université de Caen Normandie
Examinateurs :	Ouiddad Labbani-Igbida	Professeur des universités, Université de Limoges
	Laëtitia Matignon	Maîtresse de conférences, Université Claude Bernard Lyon 1
	François Charpillet	Directeur de recherche, Inria Nancy
	Rémi Pannequin	Ingénieur de recherche, CRAN, Université de Lorraine
Encadrants :	Mohamed Boutayeb	Professeur des universités, Université de Lorraine
	Dominique Martinez	Directeur de recherche, CNRS, Aix-Marseille Université

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503
Centre de Recherche en Automatique de Nancy — UMR 7039

Mis en page avec la classe thesul.

Acknowledgment

Throughout my thesis, I have learned that the journey is more important than the destination. This lesson came to me twice: first, when I realized that in reinforcement learning, the environment plays a more crucial role than the reward function, and second, when I understood that the journey of the thesis is more important than the thesis itself. The joy of completing this journey, along with the knowledge and growth I've gained, wouldn't have been possible without someone who choses you for this mission, someone who motivates you to keep going, someone who believes in you, someone from who to learn, someone who helps you sharpen your theoretical approach, someone who helps you to develop your practical skills, someone who helps you to overcome the obstacles and someone to help you see the light at the end of the tunnel. I am honored to have a whole team of these someones to thank for their support and guidance throughout this journey. I would like to express my deepest gratitude to my official supervisors: Prof. Mohamed Boutayeb and Dr Dominique Martinez, for their guidance, support, and encouragement throughout my thesis. I would also like to thank Dr François Charpillet and Dr Rémi Pannequin for their informal yet equally important support. I am grateful for the opportunity to work with such a talented and dedicated team.

I would like to express my sincere gratitude to the members of the jury for their time, effort, and valuable feedback.

I am grateful to Loria/Inria and CRAN for providing the support and resources that made my research possible. I would like to thank also the LARSEN team for their support and for providing me with the necessary tools and resources to complete my research.

Finally, I am deeply grateful to my family and friends who always been there for me and supported me throughout this journey. I would like to thank especially my parents for preparing me for this journey, for their sacrifices and for always prioritizing my education.

Contents

Abstract

In this thesis, the use of reinforcement learning for controlling cable-driven parallel robots has been investigated. This class of robots is known for its complex dynamics and the nonlinearity of the system, which offers an interesting environment for the implementation of reinforcement learning algorithms. These algorithms require a lot of data to learn the optimal policy, which is not always possible in real-world applications. To overcome this issue, we propose a *sim-to-real* approach. First, the Newton-Euler equation of the robot is used to derive its dynamic model, and by setting the parameters to the real values of the robot, we validated the model by comparing the simulation results with the real data. To ensure a high precision of the simulation data and a reduced execution time, it was implemented using Matlab/Simulink and then converted to `C++` library for easier integration with the `gym` environment in `python`. Additionally, in order to learn the optimal policy using reinforcement learning, the objective of the controller should be specified. As most of the use cases of cable-driven parallel robots could be summarized as a trajectory tracking problem, a reward function that align with this objective is designed and a process for target trajectories generation was developed. In addition to that, a limitation on the action space was introduced to ensure the cable tension is within the limits during the training process. These key components along with the most known reinforcement learning algorithms for continuous space: DDPG, PPO, and SAC make a full-fledged training platform for the generation of reinforcement learning-based controllers for cable-driven parallel robots. A comparison between the three algorithms during the training process and the performance of the trained controllers was conducted. A side-by-side evaluation of the reinforcement learning controller with a PID-based controller developed for insect tracking purposes was also performed and many aspects were compared such as the tracking error, the energy consumption, and the robustness of the controller. One of the main challenges of this work is the transition to different configurations of the robot, as the trained policy was developed for a specific configuration, a new training process is required for each different configuration. To overcome this issue, a new method to learn an actuator-level policy has been developed and comparative analysis with the conventional policy has been hold. Finally, the trained controller was tested on the robot to ensure the transferability of the policy from the simulation to the real world.

Keywords: Cable-driven parallel robot, Reinforcement learning, Deep learning, Trajectory tracking, Dynamic model, Sim-to-real RL.

Résumé

Dans cette thèse, l'apprentissage par renforcement appliqué au contrôle des robots parallèles à câbles a été étudié. Cette catégorie de robots se distingue par sa dynamique complexe et la non-linéarité de son système, offrant ainsi un cadre idéal pour l'implémentation d'algorithmes d'apprentissage par renforcement. Toutefois, ces algorithmes nécessitent de vastes quantités de données pour apprendre la politique optimale, ce qui n'est pas toujours réalisable dans des scénarios réels. Pour contourner cette limitation, nous avons proposé une approche *sim-to-real*. Tout d'abord, l'équation de Newton-Euler a été utilisée pour modéliser la dynamique du robot, et en fixant les paramètres aux valeurs réelles, le modèle a été validé en comparant les résultats de la simulation avec les données expérimentales. Afin d'assurer une grande précision des simulations tout en réduisant les temps de calcul, le modèle a été implémenté sous Matlab/Simulink, puis converti en bibliothèque `C++` pour une intégration plus fluide avec l'environnement `gym` en `python`. Par ailleurs, pour déterminer la politique optimale via l'apprentissage par renforcement, l'objectif du contrôleur doit être défini. Étant donné que la majorité des applications des robots parallèles à câbles se rapportent au suivi de trajectoires, une fonction de récompense alignée sur cet objectif a été conçue, accompagnée d'un processus de génération de trajectoires cibles. De plus, une limitation de l'espace d'action a été introduite afin de garantir que la tension des câbles reste dans les limites acceptables durant l'apprentissage. Ces éléments clés, associés aux algorithmes d'apprentissage par renforcement les plus répandus pour les espaces continus – DDPG, PPO et SAC – forment une plateforme complète dédiée à la génération de contrôleurs pour les robots parallèles à câbles. Une comparaison approfondie entre ces trois algorithmes a été réalisée, tant durant l'apprentissage que lors de l'évaluation des performances des contrôleurs entraînés. En parallèle, une comparaison a été effectuée entre le contrôleur d'apprentissage par renforcement et un contrôleur PID développé pour le suivi des insectes, en prenant en compte divers critères tels que l'erreur de suivi, la consommation d'énergie et la robustesse du système. Un des défis majeurs de cette étude concerne la transition vers différentes configurations du robot, car la politique apprise est spécifique à une configuration donnée, nécessitant un nouveau processus d'apprentissage pour chaque configuration différente. Pour pallier cette difficulté, une nouvelle méthode d'apprentissage d'une politique au niveau des actionneurs a été développée et comparée à la méthode conventionnelle. Enfin, le contrôleur entraîné a été testé sur le robot afin de valider la transférabilité de la politique de la simulation au monde réel.

Mots-clés: Robot parallèle à câbles, Apprentissage par renforcement, Apprentissage profond, Suivi de trajectoire, Modèle dynamique, sim-to-real RL.

List of Figures

Introduction

1 Cable driven parallel robots

Cable-driven parallel robots (CDPRs) are a class of parallel robots that use cables instead of rigid links to transmit forces and motion 1. This special structure offers many advantages over other types of parallel robots, such as large and flexible workspace, the ability to handle heavy payloads, the reduced maintenance cost as cables are inexpensive to produce and easy to replace compared to rigid links, and the ability to adapt to different environments and tasks. All these strengths make CDPRs the optimal choice to handle many tasks where traditional robots are not suitable, such as the Skycam robot [2] used in sports events or the STRING-MAN used for gait rehabilitation [3]. However, most of the applications are limited to the prototype stage, and the industrial applications are still rare. This is mainly due to the complexity of the control and the modeling of CDPRs. The cables are flexible and have a nonlinear behavior, which makes the control task more challenging. Also the safety of the system is a critical issue, as the cables can break and cause serious accidents, especially when moving at high speeds. This makes CDPRs limited to human-free environments, which is a significant drawback for many applications.

Many researchers have worked on the modeling and control of CDPRs, and many control

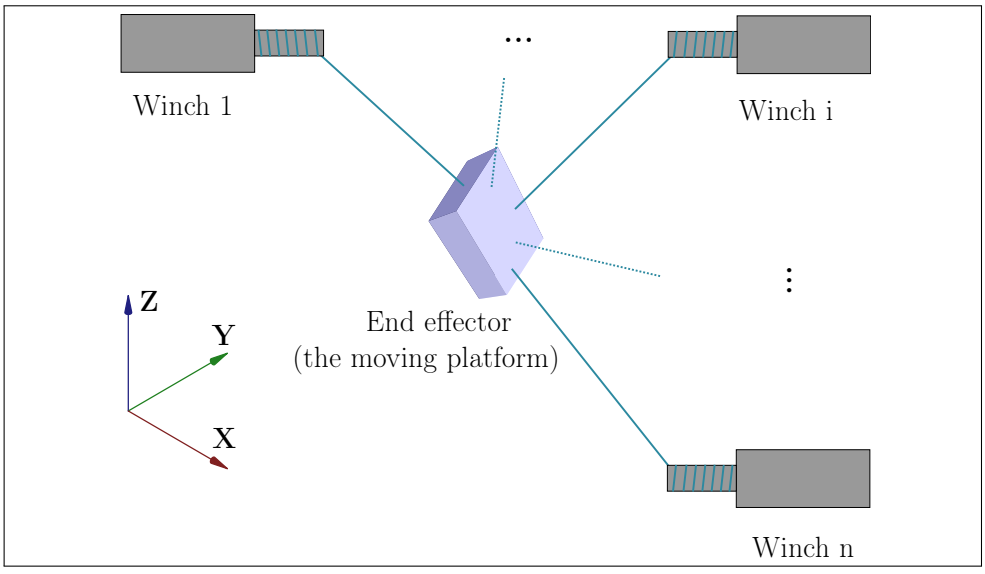


Figure 1: CDPR with n actuators, the end effector is the mobile platform, it is connected to the base by n cables, each cable is actuated by a winch motor.

aspects specific to CDPRs have been addressed, such as the tension distribution in the cables, the workspace analysis, the trajectory planning, and the safety issues [4, 5, 6, 7]. However, the

use of reinforcement learning (RL) in the control of CDPRs is still limited. The main reason is that RL requires a large amount of data to learn the optimal policy, the training process is time-consuming and computationally expensive, and the traditional control methods are still more efficient in many cases. However, RL has shown great potential in many applications, especially when the objective is to learn a complex task that is difficult to model analytically while an associated reward function can be defined. In the next section, the project Lab-On-cables that revealed the need for further investigation of the use of RL in the control of CDPRs is presented.

2 Context and motivation

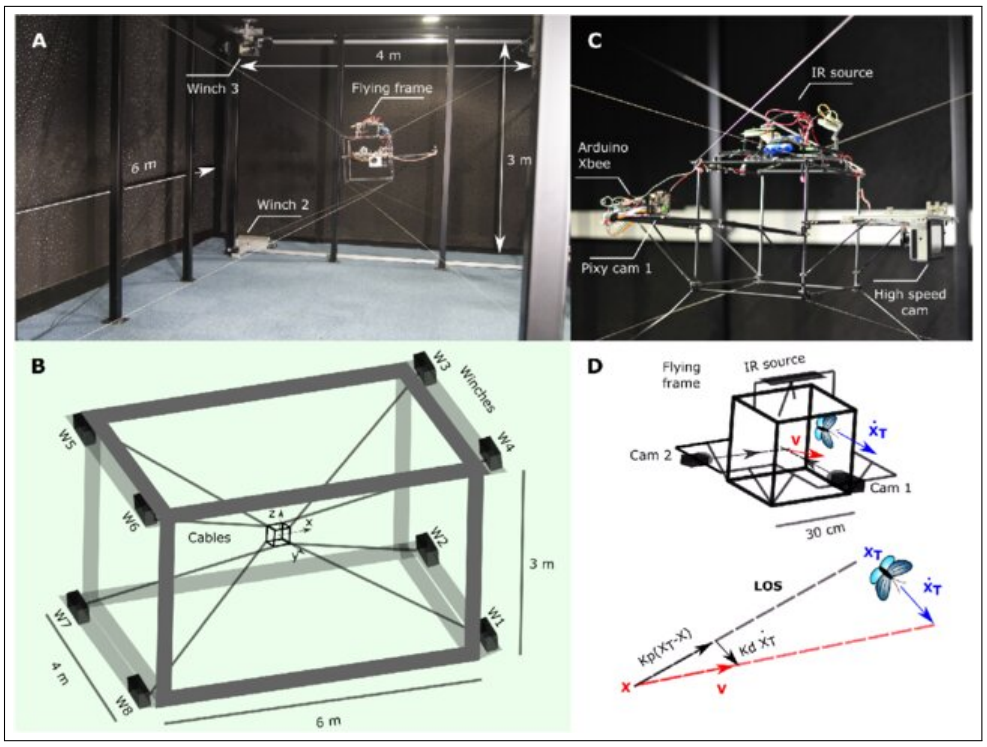


Figure 2: Lab-on-cables setup. (A) Photo of the cable robot (6 m by 4 m by 3 m) and (B) schematic view of the setup. (C) Photo of the flying frame (30 cm by 30 cm by 30 cm). The flying frame supports the lab equipment, i.e., an infra-red source and a pair of calibrated cameras (Pixy cam 1 and 2), for online insect location. The flying frame moves automatically to keep the insect within the detection range of the cameras. (D) Robot control as deviated pursuit. Robot and insect locations are X and X_T , respectively. The insect speed is denoted $\dot{X}_T$. The tracking speed V is the sum of a pure pursuit term pointing along the line of sight (LOS) toward the target current location and a corrective term, taking into account the direction of motion of the target. This is like anticipating where the target will be to point ahead of the target and cover less distance. [1]

In the Lab-on-Cables (LoC) project, the cable driven parallel robot is used to track a free flying insect using the cameras mounted on a cubic end effector 2. Such application could not be achieved using any other type of robot because of the high speed and acceleration required to track the insect, the large workspace that must be covered by the robot to have similar aspect to the natural environment of the insect, and the noiseless movement of the robot that do not

scare the insect. The latter is tracked using 3D depth-sensing camera that compute its position in the moving frame, then a control algorithm is used to move the robot to keep the insect in the center of the end effector. It is based on a PID controller with geometric model of the robot to compute the desired speed of the end effector combined with QP optimization solver to adjust the tension in the cables. The main objective of the project is to capture the movement of the insect using high speed cameras to study its behavior. The sequence of images collected during this experiment is basically 4 seconds video recorded in slow motion mode, so losing the insect for some milliseconds from the view of the camera could lead to a loss of important information. This is why a control algorithm that can handle high speed trajectories more efficiently and may predict the movement of the insect is needed. This is where the use of reinforcement learning (RL) comes into play. This discipline has shown great potential in the last years enabling robot to learn directly from visual data and to perform complex tasks that are difficult to model analytically [8]. The use of RL in the control of CDPRs is still limited, and the few works that have been done are mainly focused on the low speed trajectories or pick and place tasks. The main objective of this thesis is to investigate RL for the control of CDPRs, and more specifically in the tracking of high speed trajectories, this will be done using the cable driven parallel robot in the Lab-On cables project 2. In the next section, the contribution of this thesis is presented.

3 Contribution

The first chapter of this thesis is dedicated to introducing the fundamentals of reinforcement learning, deep learning, and their intersection in deep reinforcement learning(DRL). The most prominent DRL algorithms used for continuous control tasks are presented, some of the most successful applications of DRL in robotics are discussed and, at the end of the chapter, the state of the art of the use of RL in the control of CDPRs is presented.
In chapter two attention is directed toward the development of the simulation environment that will be used to train the RL agent. Basic concepts of CDPRs like the configuration, the workspace, the forward and inverse kinematics and the Jacobian matrix are presented, before moving to the modeling of the cables and the end effector. Finally a simulation of this model is developed using Matlab/Simulink from which `C++` library is generated to be used in `gym` environment for training the RL agent.
Chapter three is centered around the development of the RL framework for the control of CDPRs. The objective of the controller is defined, the reward function is designed, and the process for generating the target trajectories is developed. At the end, the entire training algorithms including the RL agents: DDPG, PPO, and SAC are presented.
In chapter four, the tuning of the hyperparameters of the RL agents is discussed. The reward function is analyzed, and the results of the training process are presented. The trained policies are then transferred to the real robot and tested. The results are analyzed, and the performance of the trained policies is compared with the PID controller.
In chapter five, the generalization of the trained policy to different configurations of the robot is addressed. As the trained policy is developed for a specific configuration, a new training process is required for each different configuration. To overcome this issue, a new method to learn an actuator level policy has been developed and comparative analysis with the conventional policy has been hold using the results on the real robot.
Finally, the last chapter is dedicated to the conclusion and the perspectives of this work. The main results of this thesis are summarized, the limitations of the work are discussed, and the perspectives for future work are presented.

Chapter 1

Deep reinforcement learning: From concepts to algorithms

Contents

1.1 Introduction

In recent years, there has been an increasing interest in the use of artificial intelligence (AI) in robotics. AI, defined by John McCarthy, one of the founders of the discipline of artificial intelligence, as "the science and engineering of making intelligent machines, especially intelligent computer programs." [9], encompasses a wide range of techniques, the most revolutionary one on robotics is machine learning. The latter may be divided into three main categories: supervised learning, unsupervised learning, and reinforcement learning. While supervised learning techniques are based on the use of labeled data, unsupervised learning does not require labeled data. Which make some people think that all techniques could be classified under one of these two categories. However, reinforcement learning is a different paradigm. Rather than using collected data, it is based on the interaction between an agent and an environment. The agent learns by trial and error, and his goal is to maximize a reward signal. The idea of maximizing a reward function, which is equivalent to minimizing a cost function, is also the main goal of optimal control field of control theory. This is why reinforcement learning is the most suitable field of machine learning for control problems. While supervised and unsupervised learning are used in wide range of applications in robotics, like computer vision, natural language processing (NLP), voice recognition, etc., reinforcement learning presents a great potential in the field of robot control. Especially with the recent advances in deep learning, the combination of deep learning and reinforcement learning has led to the development of deep reinforcement learning (DRL). This field was notably advanced by the work of DeepMind which developed the deep Q network (DQN) algorithm that was able to achieve human-level performance in playing Atari games [10]. The purpose of this chapter is to present the concepts for reinforcement learning and how it could be used in the control of cable-driven parallel robots.

1.2 Reinforcement learning

The modern field of reinforcement learning (RL) as presented by Richard S. Sutton and Andrew G. Barto in their book "Reinforcement Learning: An Introduction" is the result of the convergence of several fields of research in 1980s [11]. The main idea of RL is learning from experience to achieve a defined objective. It is the same way humans (and animals) learn: it is almost impossible to teach a child how to succeed a new skill without letting him fail several times: the child learns from his mistakes and tries to avoid them in the future. To exploit this idea, a RL framework is defined, its main components are: the agent, the environment, the state, the action, the reward, and the policy: figure 1.1.

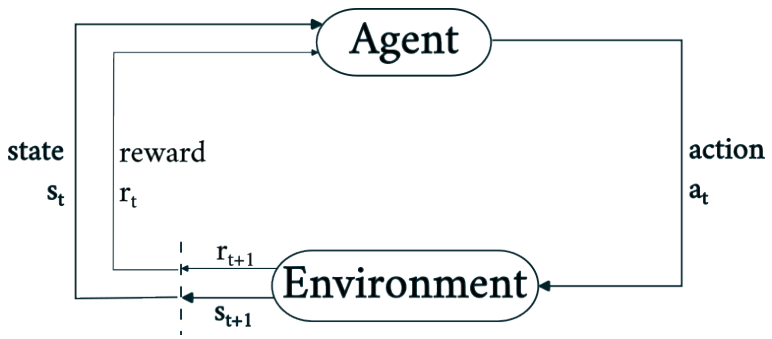


Figure 1.1: Interaction between the agent and the environment in reinforcement learning

The environment: it represents the set of rules according to which our problem is defined. It is the world in which the state evolves and changes. For example in the case of games like chess, the environment is the chessboard and the rules that define the possible moves and their outcome. In the case of a robot, the environment is the physical world in which the robot evolves. It is the physical laws that govern the robot's dynamics.

The agent: It is the component responsible for deciding which action to send to the environment, taking into account its state and the rewards that it receive from it. It is the decision maker, the algorithm that will make us achieve our goal. The agent could try many strategies that aren't optimal, but it will learn from its mistakes and try to avoid them in the future.

State: It is the information about the environment estimated by the the agent through it sensors at a given time. It is the representation of the environment that change over time and according to the agent's actions. For example, in the case of a robot, it is basically the measurements collected using sensors from the real world.

Action: It is the decision made by the agent at a given state. It is the way the agent act on the environment. For example, in the case of a robot, it is the control signals sent to the motors.

Policy: The agent acts according to certain rules and logic, this is what we call the policy. It is the strategy that the agent uses to select the action at a given state. It is the function that maps the state to the action.

Reward: It is considered as part of the environment even if it is something we can change according to our problem and our objective. It is a function that maps each state-action pair to a real number. It is the feedback that the agent receives from the environment to evaluate its actions. While it is up to the designer to define the reward function, it is not always easy how to do it, especially in the case of multi-objective problems. The reward function is an important part of the RL problem. It is what the agent tries to maximize. The reward will be designed using the variable R_t, which is the reward received at time step t.

Return: The agent objective is to maximize the reward received during the whole episode. In most cases, the instant reward is not enough to evaluate the action, as this action could lead to a better reward in the future compared to another action which gives a better instant reward. For example, in chess, playing the move that seems to maximize the instant reward (taking the opponent piece's) without consideration for the consequence of this move is obviously a very bad strategy.This is why we use the concept of return. The return is the weighted sum of the rewards received during the whole episode. In the case of an infinite episode, this sum could be infinite, so the discount factor γ is set between 0 and 1 to make the return finite and at the same time to give more importance to the instant reward compared to the future reward. The return is defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.1}$$

Now, that we defined the main components of RL, we can introduce the concept of Markov decision processes (MDP) that is a formal framework where the properties of RL can be further analyzed.

1.2.1 Markov decision processes

MDP is defined by the tuple (S, A, P, R) [11] where:

- S is the set of states which could be finite or infinite
- A is the set of actions which could be finite or infinite

- P is the transition probability function $P : S \times A \times S \to [0, 1]$
- R is the reward function $R : S \times A \to \mathbb{R}$

The MDP respects the Markov property which means that the future state depends only on the current state and the current action, so all the information needed to make a decision is contained in the current state. This property could be written using mathematical notation as:

$$P[s_{t+1}|s_t, a_t] = P[s_{t+1}|s_1, ..., s_t, a_1, ..., a_t] \tag{1.2}$$

By looking at this property, we could make the difference between the observation which is the information given by the environment and the state which is normally chosen by the designer to respect the Markov property. For example if we need both the information gathered at the current time step and the last time step to make decision, the designer could make a state that contains both last observations.

1.2.2 Value function and policy (the discrete case):

As we defined previously, the policy is a function that maps the state to the action [11]. The policy could be either deterministic or stochastic. In the case of deterministic policy, the action is chosen according to the state, while in the case of stochastic policy, the action is chosen according to the state and a probability distribution. The policy is denoted by π, and it is defined as:

$$\pi(a|s) = P[a_t = a|s_t = s] \tag{1.3}$$

$P[a_t = a|s_t = s]$ is the probability of taking the action a at the state s. To evaluate the quality of being in a given state when following a certain policy, we use the concept of value function. The state value function is defined as the expected return when starting from a given state s and following a certain policy π. The state value function is defined as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s] \tag{1.4}$$

G_t is the return at time step t 1.1. There is also the action value function which is the expected return when starting from a given state s and taking a certain action a and following a certain policy π. The action value function ("Q function") is defined as:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] \tag{1.5}$$

The value function is the most important concept in RL. It is the function that the agent tries to maximize. The solution of the RL problem is to find the optimal policy π^* that maximizes the value function. This policy is defined as:

$$\pi^*(s) = \arg\max_{\pi \in \Pi} v_\pi(s) \tag{1.6}$$

And the corresponding value function is denoted by $v^*(s)$, and it is defined as:

$$v^*(s) = v_{\pi^*}(s) \tag{1.7}$$

1.2.3 Bellman equation and temporal difference learning

The most important property of the value function is recursive property. This property is the result of recursive property of the return. The return is defined as the sum of the rewards received during the whole episode. This sum could be written as:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = R_{t+1} + \gamma G_{t+1} \tag{1.8}$$

By using this property, we could write the value function as [11]:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{1.9}$$

Where $\pi(a|s)$ is the policy that maps the state s to the action a, $p(s', r|s, a)$ is the transition probability function, it is the probability of going to the state s' and receiving the reward r when taking the action a at the state s. This equation is called the Bellman equation. It is the most important equation in RL. It determines the value of a state based on the received reward and the value of the subsequent state. The Bellman optimality equation is the same equation but with the optimal policy. It is defined as:

$$v^*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v^*(s')] \tag{1.10}$$

Using the Q function, we could write the Bellman optimality equation as:

$$q^*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q^*(s', a')] \tag{1.11}$$

By solving the Bellman optimality equation, we could find the optimal policy and the optimal value function. The problem is that solving this equation for large problems is not always possible. It requires a lot of computation. This is why instead of finding the value function, we try to approximate it. So we use the Bellman equation as an update rule:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{1.12}$$

Let us assume that we have a policy π, and the model of the environment is known. Starting from a random value function, we could use the Bellman equation to update the value function, and the sequence v_k will converge to the value function of the policy π. This is the idea of dynamic programming, and this algorithm is called policy evaluation. Now, suppose that we have found the value function of the policy π, we could use it to improve this policy. At the state s, instead of using the policy π, we could use the action that maximizes the action value function $q_\pi(s, a)$ using the equation:

$$\pi'(s) = \arg\max_a q_\pi(s, a) \tag{1.13}$$

And for computing the action value function of the new policy in this specific state, we could use the equation:

$$q_{\pi'}(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{1.14}$$

So the new policy which is nothing else than the old policy in the other states and the new policy in the specific state is better than the old policy. The extension of this idea to the whole state

space is called policy improvement. And it has been shown that [11] the new greedy policy is better than the old one. After improving the policy, we could use the policy evaluation again to find the value function of the new policy, and then we could use the policy improvement again to find a better policy. This process is called generalized policy iteration (GPI). In all the previous algorithms, we assumed that the distribution model of the environment, the transition probability function P, is known, but in most of the cases, we only have the sample model of the environment. So the agent has to do the update using only the information that he gets from interacting with the environment at each time step. Assume that the agent visited the state s many times, and each time we computed the return G_i which is an estimation of the value function of the policy π and, we associate with each return a weight W_i, we could use the weighted average of these returns as the estimation of the value function of the policy π. This could be written as:

$$v_n = \frac{\sum_i W_i G_i}{\sum_i W_i} \tag{1.15}$$

W_i is defined based on the importance of the sampled return G_i in the estimation of the value function v_n. At the next time step, this could be written as:

$$v_{n+1} = \frac{\sum_i W_i G_i + W_{n+1} G_{n+1}}{\sum_i W_i + W_{i+1}} \tag{1.16}$$

And if we set $C_{n+1} = C_n + W_{i+1}$ and $C_0 = 0$ we could write the previous equation as:

$$v_{n+1} = v_n + \frac{W_{n+1}}{C_{n+1}}(G_{n+1} - v_n) \tag{1.17}$$

and by setting $\alpha_{n+1} = \frac{W_{n+1}}{C_{n+1}}$ we could write the previous equation as:

$$v_{n+1} = v_n + \alpha_{n+1}(G_{n+1} - v_n) \tag{1.18}$$

By using this equation we could update the value function at each time step using only G_{n+1} and the previous value function. This is what we call Monte Carlo learning. The equations above are suitable for any kind of target of the value function. By target we mean the desirable value of the value function. We could write the equation in more generalized form as[11]:

$$NewEstimate = OldEstimate + \alpha(Target - OldEstimate) \tag{1.19}$$

In the case of one step temporal difference learning, we want only to use the reward received at the next time step to find the new target. This is why we use the equation:

$$v_{t+1}(s) = v_t(s) + \alpha[R_{t+1} + \gamma v_t(s_{t+1}) - v_t(s)] \tag{1.20}$$

By using it combined with the generalized policy iteration algorithm we could find the optimal policy and the optimal value function. So in the case of finite state and action space, the value function could be stored in a table, and the optimal policy could be found by using the action that maximizes the action value function. This is not the case of continuous state and action space. This is why we need to use methods of function approximations like neural networks to approximate the value function. In the next section, we will present the concept of deep learning and how it could be used in RL.

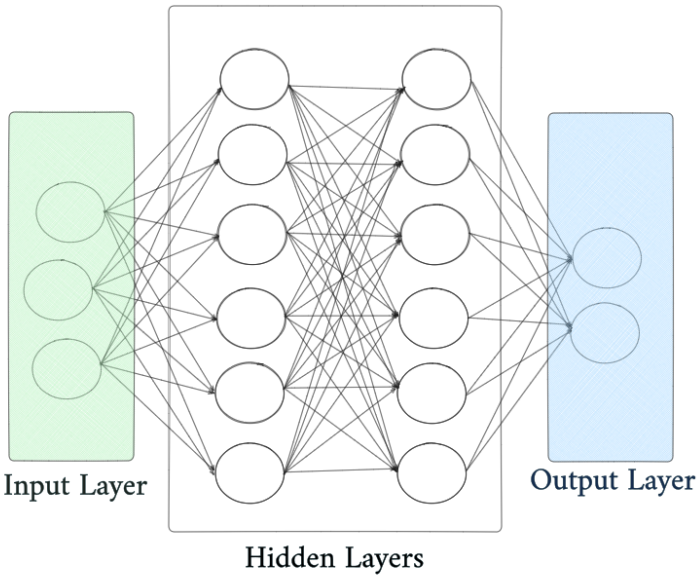


Figure 1.2: Neural network architecture

1.3 Deep reinforcement learning (continuous case):

1.3.1 Function approximation and neural networks

Deep neural networks are a powerful tool for function approximation. Also called deep feedforward networks, artificial neural networks (ANN) or multilayer perceptrons (MLPs), they can approximate any continuous function, as stated by the universal approximation theorem [12]. The universal approximation theorem states that a feedforward network with at least one hidden layer and with any activation function can approximate any Borel-measurable function from one finite-dimensional space to another with any desired accuracy, given enough neurons in the hidden layer. Practically speaking, the use of deep neural networks with multiple hidden layers allows better approximation of complex functions. Moreover, the ANN could learn to approximate a given function where data is collected in online manner and the function is changing over time. This is why ANN is the most used function approximation method in recent years, and when we talk about deep reinforcement learning we talk about using ANN to approximate the value function or the policy.

1.3.2 Neural networks architectures

The neural network is composed of layers, each layer is composed of neurons. The input layer is the first layer, and the output layer is the last layer. The layers between the input and the output layers are called hidden layers 1.2. The number of hidden layers is called the depth of the network. The number of neurons in the hidden layers is called the width of the network. The output of the neuron is computed by the activation function. The most used activation functions are the sigmoid function, the hyperbolic tangent function, and the rectified linear unit (ReLU)

function [13].
Neural network is mapping a value from the input space x to the output space y, this mapping is done by a set of parameters *theta* called weights $f(x;\theta) = y$. In function approximation, the objective is to change the set of parameters θ in such way that the output of the network is close to the target. This closeness to the target is measured by a *loss function*, that needs therefore to be minimized. In RL, ANN could learn the value function by minimizing the temporal difference (TD) error and the policy by maximizing the return [11].

1.3.3 Gradient based optimization

Minimizing the loss function also called the objective function requires the implementation of an optimization algorithm. For function $f(x)$ with one variable, the derivative of the function at the point x is the slope of the tangent line to the curve at this point. It shows how changing the input will affect the output.

$$f(x + \epsilon) \approx f(x) + \epsilon f'(x) \tag{1.21}$$

$f'(x)$ is the derivative of the function at the point x and ϵ is a small number. By changing the input in the opposite direction of the derivative, we could minimize the function. This is how gradient descent works. The gradient is the generalization of the derivative to the case of function with multiple variables. It is the vector of the partial derivatives of the function. In machine learning (ML), computing the gradient using large datasets is computationally expensive. Since the gradient is an expectation over the data, one might use a subset of the data to estimate its value. This is the idea of stochastic gradient descent (SGD). The SGD update is done by using the equation:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta J(\theta) \tag{1.22}$$

Where α is the learning rate, $J(\theta)$ is the loss function that we want to minimize, and $\nabla_\theta J(\theta)$ is the gradient of the loss function with respect to the parameters of the network θ. The learning rate is a hyperparameter that controls the step size of the update. Keeping this parameter constant is not always a good idea. This is why there are many optimization algorithms that adapt the learning rate during the learning process. The most used optimization algorithms are the Adam, RMSprop, and Adagrad [13].

1.3.4 Back-propagation

Computing the gradient of a function using the analytic expression can be very difficult. This is why the back-propagation algorithm was introduced. The back-propagation algorithm is a method for computing the gradient of the loss function with respect to the weights of the network. It is based on the chain rule of calculus and computational Graphs. The idea is to compute the gradient of the loss function with respect to the output of the network, and then to compute the gradient of the output with respect to the input of the activation function, and so on until we reach the input of the network. The back-propagation algorithm is the most used algorithm in deep learning. It is the algorithm that allows the training of deep neural networks. By using the back-propagation with the gradient-based optimization algorithm, we could train the neural network to approximate the value function or the policy. In the next section, we will present some of the most used algorithms for continuous state and action space.

1.4 Algorithms and architectures for continuous state and action space: DDPG, PPO, SAC

Deep reinforcement learning could be used to solve problems where the state space and the action space are continuous. There is a wide range of algorithms that could be used to solve this kind of problems. The Deep Q Network (DQN) algorithm was the first algorithm that was able to solve problems with continuous state space [14, 10]. In this section, we will define some notions that are used in these algorithms, and we will present some of the most used algorithms in this field.

1.4.1 General concepts

Replay Buffer

The concept of replay buffer was used for the first time in the *Deep Q Network (DQN)* algorithm [14]. The idea is to store the experiences (s_t, a_t, r_t, s_{t+1}) in a memory buffer, then randomly sample from this buffer during the learning to train the agent. This technique helps in breaking the correlation between the samples. It also allows the agent to learn from the same experience multiple times increasing the data efficiency.

On-policy vs off-policy

In general, the learning by reinforcement is done online while interacting with the environment. The agent uses the data collected during the interaction to update its policy. If the data used to update the policy is the same data collected using this policy we call this on-policy learning. If the data used to update the policy is collected using another policy, the behavior policy, we call this off-policy learning [11, Chapter 5]. The idea of using replay buffer in the learning process allows the agent to use the data collected using the old policy to update the new policy. This is why the algorithms that use replay buffers are off-policy algorithms.

Deterministic vs stochastic policy

In the learning process, the agent could learn a deterministic policy or a stochastic policy. In the case of deterministic policy, the action is the output of the actor neural network. Whereas for the stochastic policy, the action is sampled from a probability distribution. Usually, we use the Gaussian distribution to sample the action. The mean of the distribution is the output of the actor neural network, and the standard deviation could be a learnable parameter or an output of the actor neural network. Practically, the deterministic version of the stochastic policy is used in the deployment phase, as it is more stable [15].

Target networks

The concept of target networks was also introduced in the *Deep Q Network (DQN)* algorithm [14]. In the learning process of the value function in algorithms like DQN, the updates to the value function are chasing a constantly moving target. This could lead to divergence. To solve this problem, the idea of target networks was introduced. A target network is a copy of the original network that is updated slowly. The parameters of the target network are updated according to the equation:

$$\theta^{'} \leftarrow \tau\theta^{'} + (1-\tau)\theta \tag{1.23}$$

θ' is the parameters of the target network, θ is the parameters of the original network, and τ is a small number. The target network is used to compute the target value in the learning process. This is why the target network is also called the target value network.

1.4.2 Actor critic methods

In reinforcement learning we could distinguish between two main classes of methods: value-based methods and policy-based methods. In value-based methods, the agent tries to learn the value function and then derive the policy from the value function. While in policy-based methods, the agent tries to learn the policy directly. The actor-critic methods are a combination of these two methods. We have two neural networks: an actor network responsible for learning the policy and a critic network responsible for learning the value function. To update the critic network we use the temporal difference learning, and to update the actor network we use the policy gradient theorem with the critic network as a baseline. The actor-critic methods are more stable than the value-based methods and more sample efficient than the policy-based methods [11, Chapter 13].

1.4.3 Deep deterministic policy gradient (DDPG)

The *Deep Deterministic Policy Gradient* is a deterministic, off-policy reinforcement learning technique that uses a replay buffer for sample efficiency. It is an adaptation of the *Deep Q Network (DQN)* technique which is based on two *Actor/ Critic* models. [16] For stability reasons, each of the two models are associated with another neural network called: *Target actor network* and *Target critic network*.

To understand the DDPG algorithm, we will start with the DQN which makes it possible to solve problems with a continuous state space before moving on to the DDPG which is an extension of the DQN to problems with continuous action space. In DQN, the Q function is approximated by a neural network, and the optimal action at state s $\mu(s)$ is chosen by selecting the action that maximizes the Q function:

$$\mu(s) = \underset{a}{\operatorname{argmax}}\, Q(s, a) \tag{1.24}$$

Thus, it is just the Q network which is represented by a neural network in DQN. For the DDPG since it is designed for problems where the action space is also continuous, the policy is also represented by a neural network, *Actor network*. As we've seen in the previous section, most of the reinforcement learning algorithms are based on the concept of GPI (Generalized Policy Iteration), so we alternate between policy evaluation and policy improvement. In the case of DDPG, the policy improvement is done by using the policy gradient theorem to find the optimal policy according to the current Q function. Policy evaluation is done by using the temporal difference learning to update the Q function according to the data collected by the new policy. So let's start with the Q function learning.

Action-value function(Q function) learning

The Q function at the state s and the action a is denoted by $Q(s, a|\theta^Q)$, it is the expected return when starting from the state s and taking the action a and following the policy β. It is approximated by a neural network whose parameters are θ^Q, and updated by minimizing the loss function:

$$L(\theta^Q) = E_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}[(Q(s_t, a_t|\theta^Q) - y_t)^2] \tag{1.25}$$

β: "*behavior policy*" the policy we use for data collection. ρ^β Distribution of visited states following the β policy. E: Environment, and the notation $s_t \sim \rho^\beta$ means that the state s_t is sampled from the distribution ρ^β.
With:

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q) \tag{1.26}$$

y_t is called *the target*, it is the new estimate of $Q(s_t, a_t)$ using the reward just obtained at time t and the old estimate $Q(s_t, a_t)$

Optimal policy learning

The current policy is approximated with a neural network whose parameters are θ^μ. To find the optimal policy, we need to maximize the expected return. The expected return is defined as:

$$J(\mu_\theta) = \mathbb{E}_{s\sim\rho^\mu}[r(s, \mu_\theta(s))] \tag{1.27}$$

with $\theta = \theta^\mu$ Using the Deep policy gradient (DPG) theorem we get:

$$\nabla_{\theta^\mu} J = \mathbb{E}_{s\sim\rho^\beta}[\nabla_a Q(s, a|\theta^Q)_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu}\mu(S|\theta^\mu)|s = s_t] \tag{1.28}$$

So by using the policy gradient theorem, we could update the policy's weights.

Exploration noise

The policy learned by the actor network is deterministic, so to explore the action space, we add to the output of the actor network an exploration noise. In the original paper [16], the noise used is based on the "*Ornstein-Uhlenbeck process*" which is a stochastic process that generates temporally correlated noise. But during the experiments, it was found that the "*Ornstein-Uhlenbeck process*" does not really have great advantage over the Gaussian noise, which is already mentioned in other works [17, 18]. So the Gaussian noise is used for this purpose. Thus the action applied by the agent is:

$$a_t = \mu(s_t|\theta_t^\mu) + \mathcal{N}_t \tag{1.29}$$

with $\mathcal{N}_t$ the Gaussian noise. The standard deviation of the Gaussian noise is an hyperparameter to be tuned.

Algorithm 1: DDPG algorithm [16]

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ;
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$;
Initialize replay buffer $\mathcal{R}$;
for *episode* $= 1, M$ **do**
  Initialize a random process $\mathcal{N}$ for action exploration;
  Receive initial observation state s_1;
  for $t = 1, T$ **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise;
    Execute action a_t and observe reward r_t and observe new state s_{t+1};
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{R}$;
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{R}$;
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$;
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$;
    Update the actor policy using the sampled policy gradient:
    $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$;
    Update the target networks:
    $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$;
    $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$;
  end
end

1.4.4 Proximal Policy Optimization (PPO)

The *Proximal Policy Optimization* algorithm is a policy gradient method. It is an on-policy algorithm that could be used to solve problems with discrete or continuous action space. It was first introduced by John Schulman et al., in 2017 [19]. The main idea of PPO is to ensure that the new policy is not changing too much from the old policy. While it benefits from the advantages of the Trust Region Policy Optimization (TRPO) algorithm [20], it is more sample efficient and easier to implement. The PPO algorithm uses the advantage function to estimate the quality of the action instead of the Q function. So let's define what is the advantage function.

The advantage function

The advantage function is defined as the difference between the Q value of the action at the state s and the value function of the state s. It is defined as:

$$A(s, a) = Q(s, a) - V(s) \tag{1.30}$$

Using the advantage function instead of the Q function has many advantages. The first one is the variance reduction, as the advantage function measures the relative quality of the action at the state s compared to the average value of all possible actions in state s which is the value function. The second advantage is that the advantage function is more stable to policy changes compared to the Q function [21]. This is why the advantage function is used in many policy gradient methods.

Policy update

When using the advantage function as the target for the policy update, the most commonly used loss function is:

$$L^{PG}(\theta) = \hat{E}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{1.31}$$

By differentiating this loss function, we find the gradient estimator:

$$\nabla_\theta L^{PG}(\theta) = \hat{E}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{1.32}$$

In the PPO and TRPO algorithms, instead of using the objective function as it is defined, we use a surrogate objective function that is easier to optimize. The surrogate objective function is a type of function used to approximate or replace the true objective function during optimization, and it is easier to optimize. The surrogate objective function used in the PPO algorithm is defined as:

$$L^{CPI}(\theta) = \hat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] \tag{1.33}$$

To prevent large policy updates, which could lead to unstable learning, PPO-clip version uses a modified version of the surrogate objective function. The modified version is defined as:

$$L^{CLIP}(\theta) = \hat{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{1.34}$$

Where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, and ϵ is a hyperparameter that controls the size of the policy update.

There is also another version of the PPO algorithm called PPO-Penalty. In this version, the penalty is added to the loss function to prevent the policy from changing too much. In the application chapter of this thesis, we will be only using the PPO-clip version as it has better performance compared to the other version.

Value function learning

In PPO algorithm the critic network represents the value function which is nothing else than an estimation of the expected return R_t when starting from the state s and following the policy π. At the end of each episode, the return is computed as the discounted cumulative sum of the rewards received during the episode. Then, the value function is updated by minimizing the loss function:

$$L^V(\phi) = \hat{E}_t[(V_\phi(s_t) - R_t)^2] \tag{1.35}$$

The value function update and the policy update are done in an alternating way, but they are done at the end of each episode unlike the DDPG algorithm where the policy update is done at each time step. The PPO learns a stochastic policy, so the action is sampled from a probability distribution. The most commonly used distribution is the Gaussian distribution. The mean of the distribution is the output of the actor neural network, and we decided to use the standard deviation as learnable parameter.

Algorithm 2: PPO-Clip Algorithm [22]

Input: Initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, \ldots$ **do**

Collect set of trajectories $D_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment;

Compute rewards-to-go $\hat{R}_k$;

Compute advantage estimates, $\hat{A}_t$, (using any method of advantage estimation) based on the current value function V_{ϕ_k};

Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_k}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right)$$

typically via stochastic gradient ascent with Adam;

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm;

end

1.4.5 Soft Actor Critic (SAC)

The soft actor critic algorithm is an off-policy actor-critic algorithm that could be used to solve problems with continuous state and action space. It was first introduced by Haarnoja et al., in 2018 [23]. It optimizes a stochastic policy while using some tricks already used in the DDPG algorithm like the target networks and the replay buffer. The main idea of the SAC algorithm is the entropy regularization. The entropy is a measure of the randomness of the policy. The entropy regularization is used to encourage the policy to explore the action space. The entropy is defined as:

$$H(\pi) = E_{s \sim \rho^\pi}[-\log \pi(a|s)] \tag{1.36}$$

Maximum entropy reinforcement learning

In the maximum entropy reinforcement learning, the agent tries to maximize the expected return while maximizing the entropy of the policy. The objective function is defined as:

$$J(\pi) = E_{s \sim \rho^\pi, a \sim \pi}[\sum_{t=0}^{\infty} \gamma^t (r(s_t, a_t) + \alpha H(\pi(a|s_t)))] \tag{1.37}$$

α is a hyperparameter that controls the importance of the entropy in the objective function. It is called the temperature parameter.

Value function

The SAC algorithm uses two value functions: the state value function V and the Q function. While the Q function is used in the policy update, the state value function is used in the Q

function update. The value function is updated by minimizing the loss function:

$$L^V(\psi) = E_{s\sim\rho^\pi}[(V_\psi(s) - E_{a\sim\pi}[Q(s,a) - \alpha \log \pi(a|s)])^2] \tag{1.38}$$

The Q function is updated by applying the modified Bellman equation:

$$L^Q(\phi) = E_{s\sim\rho^\pi, a\sim\pi}[(Q(s,a|\phi) - r(s,a) - \gamma E_{s_{t+1}\sim\rho^\pi}[V_\psi(s_{t+1})])] \tag{1.39}$$

Policy update

The policy is updated by maximizing the expected return while maximizing the entropy of the policy. The objective function is defined as:

$$J(\pi) = E_{a\sim\pi_\theta}[Q^{\pi_\theta}(s,a) - \alpha \log \pi_\theta(a|s)] \tag{1.40}$$

Action bounds

In the SAC algorithm, the action is computed by sampling from a Gaussian distribution. Its mean is an output of the actor neural network, and its standard deviation is also an output of the actor neural network. So the standard deviation is a complex function that depends on the state. In most cases, the actions are bounded. To enforce the action bounds on the output of the actor neural network, they apply *tanh* to the sample from the Gaussian distribution. The *tanh* function is used to map the output of the actor neural network to the interval $[-1, 1]$. The action is then scaled to the desired interval. This bound enforcement change the probability distribution of the action. To compute the new probability of the sampled action, they employ the change of variable formula. Let's denote the action sampled from the Gaussian distribution as u, the action after applying the *tanh* function as a, and the probability density function of the Gaussian distribution as μ and the probability density function of the action as π. The log probability of the action is defined as:

$$\log \pi(a) = \log \mu(u) - \Sigma_{i=1}^{d} \log(1 - a_i^2) \tag{1.41}$$

Where d is the dimension of the action space.

Algorithm 3: Soft Actor-Critic [23]

Input: θ_1, θ_2, ϕ
Output: θ_1, θ_2, ϕ

Initialize parameters θ_1, θ_2, ϕ ▷ Initial parameters
$\bar{\theta_1} \leftarrow \theta_1, \bar{\theta_2} \leftarrow \theta_2$ ▷ Initialize target network weights
$D \leftarrow \emptyset$ ▷ Initialize an empty replay pool
for *each iteration* **do**
  for *each environment step* **do**
    $a_t \sim \pi_\phi(a_t|s_t)$ ▷ Sample action from the policy
    $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$ ▷ Sample transition from the environment
    $D \leftarrow D \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$ ▷ Store the transition in the replay pool
  end
  for *each gradient step* **do**
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i) for i \in \{1, 2\}$ ▷ Update the Q-function parameters
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$ ▷ Update policy weights
    $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$ ▷ Adjust temperature
    $\bar{\theta_i} \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta_i}$; ▷ Update target network weights
  end
end

1.5 Application of reinforcement learning in robotics

Most of the robotics problems could be formulated as a Markov Decision Process (MDP). The state is the configuration of the robot, the action is the control input, and the reward is the cost function. The goal is to find the optimal policy that minimizes the cost function. This cost function could be the distance between the end effector and the target, the energy consumption, or any other cost function that represents the task to be solved.

While robotics present good candidate for the application of reinforcement learning, most reinforcement learning algorithms,the field has historically focused on games, such as board games and video games, as ideal testing grounds for RL algorithms [10, 24]. This is because applying reinforcement learning in video games is much easier than in robotics: the environment is simulated, we have direct and noise-free access to the state, the agent can interact with the environment indefinitely, and the resulting policy is applied in the same controlled environment. Which is not the case in robotics as the environment is the real world, the state is noisy, the agent could not interact with the environment as much as it wants, and the resulted policy will be used in different environment than the one used during the learning process. Those challenges make the application of reinforcement learning in robotics more difficult. But the recent advances in deep reinforcement learning have made it possible to solve complex robotics problems, which was not possible before [8, 25, 26].

One of the main challenges in applying reinforcement learning in robotics is the sample efficiency: the amount of data needed to learn the optimal policy. The agent has to interact with the environment to learn the optimal policy. This could be very expensive in terms of time and money. To overcome this challenge, we could use a simulation instead of the real robot to train the agent. While this could be a good solution, having a simulation that is close to the real environment is not always possible. Sometimes we could try to learn the

model of the environment during the learning process, but this could be also very expensive in terms of computation. This approach is called model-based reinforcement learning. Unlike the model-free reinforcement learning, the model-based reinforcement learning uses the model of the environment to plan the actions instead of learning the policy directly [27]. This approach is more sample efficient than the model-free reinforcement learning, but it requires the model of the environment to be accurate. Another solution to the sample efficiency problem is to use the off-policy algorithms. The off-policy algorithms use the data collected by the previous policy to update the new one. This could be done by using the replay buffer. Then the data stored in the replay buffer could be used many times to update the policy. This approach proved to be several times more sample-efficient than the on-policy algorithms. When data collection is done outside the learning process by the old policy, we call this approach offline reinforcement learning. The problem with offline reinforcement learning is that the data collected by the previous policy could be not representative of the new policy. This could lead to divergence of the learning process. That is why the offline reinforcement learning algorithms are suffering from instability [28].

Although the use of simulations presents a good solution to this problem, there is always a gap between the simulation and the real environment, especially in the case of vision tasks. Many solutions have been proposed to overcome this challenge: domain randomization [29], domain adaptation [30], and transfer learning [31], but none of them could solve the problem completely and learning on the real environment would always stay the objective of the reinforcement learning community.

A second challenge in applying reinforcement learning is the safety in the real environment. To learn the optimal policy directly in real environment, the agent has to explore the environment. This could lead to catastrophic failures and hardware damage. To overcome this challenge, many safe reinforcement learning algorithms have been developed. The main idea of these algorithms is to learn the optimal policy while ensuring that the agent will not take actions that could lead to catastrophic failures [32]. Whereas these algorithms could solve the safety problem, they are less sample efficient than the other algorithms, and the training process could take more time.

Further, a third significant challenge is the need for a human intervention during the learning process in real environment. Even when using the safe reinforcement learning algorithms, the agent could take actions that lead to a state where it could not recover. In this case, the human intervention is needed to bring the agent back to a safe state. Sometimes the long learning process could lead to hardware damage, and the human intervention is needed for maintenance purposes.

All those challenges make the application of reinforcement learning in robotics an interesting and challenging field of research. In [33], Ibarz et al. present a survey of the application of reinforcement learning in robotics, the challenges, and the solutions proposed to overcome those challenges. Before moving to the application of reinforcement learning in the control of cable-driven parallel robots, we will present some compelling applications of reinforcement learning in robotics.

One of the most interesting application of reinforcement learning in robotics is the control of legged locomotion. In [25] Lee et al. introduce a highly effective control strategy that leverages proprioceptive feedback and sophisticated learning algorithms to enable autonomous navigation over complex terrains. The training was done only in simulation and the learned policy was transferred to the real robot. The results show that the learned policy retains its robustness in situations never seen during training. Robotic manipulation is also an intriguing use case for reinforcement learning. In [8] Levine et al. develop a method that learns a policy that maps raw image pixels to torques at the robot's joints. The policy is trained using a combination of trajectory-centric reinforcement learning algorithm and supervised learning. The results show

that the learned policy could perform complex manipulation tasks using only raw image pixels as input. Reinforcement learning has also been used in the control of aerial vehicles. In [26], Sadeghi et al. focus on using simulation environments to train deep reinforcement learning models that can then be applied to real-world autonomous drone flight. The training employs a Monte Carlo policy evaluation method within a highly randomized simulation environment to enhance the model's generalizability to real-world conditions. The experimental results demonstrate that the model, trained only on synthetic data, can effectively navigate real indoor environments, handling varied lighting conditions and obstacle configurations.

1.6 Reinforcement learning based-control of cable-driven parallel robots

Because of their great flexibility, large payload, and high precision, cable-driven parallel robots offer a promising platform for the application of reinforcement learning (RL). These robots might be employed in numerous scenarios where RL could be used in the control algorithm to deal with the high nonlinearity and the uncertainties of the system or to adapt the control strategy to a specific task. A large and growing body of literature has looked at those applications. These studies can be grouped into three main categories: those where RL is applied to a specific task within a particular application, those that combine RL with other control strategies to address trajectory tracking, and those that use end-to-end RL (i.e. where the agent learns to control actuators directly from raw sensor data, without intermediate processing or representations) for trajectory tracking.

For the first category, in [34] where the authors present a Q learning based method to identify the manipulators' geometry for calibration purposes, to our knowledge this is the first study to investigate the use of RL in cable driven parallel robot. Alex Grimshaw and John Oyekan focus on enhancing the robot's control using DRL to maintain the balance of unstable loads in [35]. A cable driven parallel robot has been utilized in [36] for rehabilitation, where RL is employed to find the optimal admittance to adapt to human voluntary force. In [37], the cable driven parallel robot is used in different way, the length of the cables is fixed while the actuators are on mobile bases, so instead of moving the end effector by winching the cables, the end effector is moved by moving the bases. As this configuration is more useful for manipulation tasks where the environment is cluttered, an obstacle avoidance algorithm based on SAC was used to enable real-time trajectory adjustment. Although all those studies provide innovative uses of cable-driven parallel robots, and some of them were enhanced with test on a real robot, they all lack the generalization of the approach to other problems. As the main goal of controlling CDPR is moving the end effector to a desired position, we will be more interested in the trajectory tracking problem. In the following, we will present some works that solve the trajectory tracking problem using hybrid control strategies before moving to the end-to-end RL.

The task of controlling cable driven parallel robot is a complex task that could be divided into many subtasks. One of the most challenging subtasks is the tension distribution for redundant cable driven parallel robot. As there is many possible ways to distribute the tension in the cables to achieve the same end effector position, the problem is considered to be an optimization problem under constraint. In [38], the authors propose a deep Q-network (DQN) approach to manage the tension of four cables that control the robot's end effector, enabling it to follow a 3D trajectory derived from human motion: the robot is specifically designed for upper-limb rehabilitation. Another interesting use of RL alongside basic controller is presented in [39], where it is used to enhance the performance of cable-driven parallel robots under the presence

of parameter uncertainties, the control performance was verified on real 3-DOF CDPR with three cables. A comparison between the performance of hybrid RL control strategies and end-to-end RL is presented in [40]. The authors use the DDPG algorithm to calculate the target wrench then calculate the optimal tension distribution using the inverse dynamics equation of the robot. They compare the performance of this approach named hybrid DDPG with the end-to-end DDPG, where the target cable tensions is directly calculated by the actor network. They apply the two approaches on a CDPR for ankle rehabilitation where only rotational motion is allowed. Although the end-to-end DDPG approach shows similar performance to the hybrid DDPG during normal tests, the hybrid DDPG outperforms the end-to-end DDPG when the model is subject to uncertainties, and it takes less training time. Even though this study provides a good comparison between the two approaches, the end-to-end DDPG approach was not trained to handle uncertainties, moreover the testing was basically done on a simulation environment, and the trajectory test tracking was done using a simple sinusoidal trajectory with a period of more than 100 seconds.

Even if hybrid methods could help to reduce the training time drastically, they are not always the best choice, especially when there is possibility to learn on the real system. Even when it is not the case, end-to-end RL would still be considered as the best choice to reveal the full potential of the RL and to discover strategies that are not apparent through traditional control approaches. In the control of CDPR using end-to-end RL, rather than the work cited above, to date and to our knowledge there are three other studies that investigated this aspect.
The first one was conducted by Dinh-Son Vu and Ahmad Alsmadi in 2020 [41]: the robot is 2 DOF underacted CDPR, they use SAC algorithm, and they propose three different reward functions to train the robot for a pick and place task, so trajectories are basically point to point trajectories.
In contrast, the second study, carried out by Sancak et al. in 2021 [42], focuses more on the trajectory tracking problem. They use the DDPG algorithm to learn the optimal policy and they propose two different approaches, one where the training algorithm learn from point to point reference trajectories ("point-to-point agent") and the other where the reference training trajectories were sinusoidal references ("dynamic agent"). Both agents were tested on both trajectories and the results show that each agent outperforms the other on his corresponding task. A comparison between the two approaches and PID controller was also presented, showing that the PID results are slightly better than the dynamic agent for sinusoidal trajectories, but the point-to-point agent outperforms the PID controller for point-to-point trajectories. The period of the sinusoidal trajectory used for test is about 10 seconds, but still the maximum speed was not really so high as it was less than 0.2 m/s in the x axis and even less in the y-axis.
Concluding our review, the final study by Raman et al. in 2023 [43], use the TD3 algorithm on reconfigurable CDPR (rCDPR) where the anchor points of the cables could slide along the side of the rectangular base. First, they apply the TD3 algorithm on a CDPR where the anchor points are fixed, then they apply the algorithm on the rCDPR, so the policy outputs alongside the cable tension the new position of the anchor points that maximize the wrench quality and the manipulability of the robot. Additionally, to those two agent they propose a decoupled training approach where they use two different agents one to learn the cable tension and one for the anchor points position. First they train the pose quality agent to learn the anchor points position by incorporating an aspect of imitation learning in the reward function, then they train the tension agent to learn the cable tension in two different ways, one where the anchor points are delivered by the learned policy, and the other where the anchor points are calculated from optimization. The results show that the decoupled training approach outperforms the other two approaches: the error RMSE was significantly reduced, and the manipulability was increased. The training

and the testing trajectories are structured as Bezier curves through a set of random points selected at the start of every episode during training, the maximum speed was not presented in the paper, but we could conclude from the figures that the speed is really low as the robot takes about 200 seconds to accomplish a trajectory of less than 1 meter.

The main limitation of the studies presented above is the low speed of the robot, indeed, the maximum speed of all those studies is less than 0.2 m/s and the reference trajectories are mostly the same one used in the training. Moreover, for the four end-to-end RL studies, the training and the testing were done only on a simulation environment, and the results were not tested on a real robot, which questions the generalization of the results to more sophisticated unknown trajectories with high speed and acceleration and the ability of the learned policy to be transferred to the real robot.

1.7 Conclusion

In this chapter, we introduced the main concepts of reinforcement learning, before presenting the main algorithms used in RL in the case of continuous state and action space. This introduction to RL shows the difficulty of implementing RL and how the most simple algorithm involve a lot of mathematical concepts to implement, numerous hyperparameters to tune and many design choices to make [44]. After that, we presented the main challenges of applying RL in robotics, and we discussed the solutions proposed to overcome those challenges. Finally, we presented the state of the art of the application of RL in the control of cable-driven parallel robots with focus on the end-to-end RL for trajectory tracking. The presented studies are limited by the low speed of the robot, the lack of generalization of the results to more sophisticated unknown trajectories with high speed and acceleration, and the lack of testing on a real robot. In the next chapter, we will develop a reinforcement learning (RL) environment modeled for the cable-driven parallel robot (CDPR) we have in the lab. That will serve as a test bench for the control of CDPR before we move on to the methodology used to address the trajectory tracking problem using RL.

Chapter 2

Cable-Driven Parallel Robots: Modeling and Simulation

Contents

2.1 Introduction

Cable-driven parallel robots are a type of parallel robots (CDPRs) that use cables to transmit forces and torques to the end effector. They are used in various applications such as rehabilitation [3], manufacturing industry [45], construction [46] and logistics [47]. In the context of the project *Lab on Cable* [1], we are using a cable-driven parallel robot to track flying insects. The end effector is a cubic structure equipped with cameras (or other instruments) and a light source to capture images of the insect and a depth sensing camera to estimate the position of the insect. So basically we have a small lab moved by cables following the insect flying freely in the room. The cable-driven parallel robot used in this project could reach a speed of 3.6 m/s and an acceleration of 17 m/s^2. Although CDPRs offer high speed and acceleration, they are complex to control and model due to the non-linearity of the way actions on the cables are transferred to end effector movement. A control algorithm based on PID controller requires the integration of a position estimator as the position of the end effector is computed from the lengths of the cables and there is no analytical solution in the case of rotational CDPRs(i.e. robot which end effector has rotational degrees of freedom). Moreover, in the case of redundant/over-actuated CDPRs(i.e. which have more cables than the degrees of freedom of the end effector), the control algorithm should take into account the energy consumption to chose the optimal control strategy. It requires an optimization, and (under some assumptions) it has been shown to be a quadratic problem. The complexity of CDPRs and the wide range of applications make them an interesting candidate for artificial intelligence-based control. In this chapter, the existing CDPR used in the Lab-on-Cables project will first be presented, along with its original control (PID-based), algorithms, and various blocks. Then, a dynamic modeling of the CDPR will be proposed, and a simulation tool developed using Matlab/Simulink will be presented and validated using real data. This simulation is the basis of the environment used for the development of the artificial intelligence-based control algorithm in the next chapters.

2.2 Configuration

One of the main advantages of CDPRs is the possibility to reconfigure the robot. As the configuration of the CDPR is defined by the number of cables, the position of the motors and the shape of the end effector, replacing the end effector, using less or more cables, changing the position of the motors or the attachment points of the cables can change the configuration of the CDPR. Although the choice of the configuration depends basically on the application and the workspace, there are some tools to help the designer to choose the optimal one. One of the most used aspects to study CDPRs configuration is the wrench-feasible workspace: it is the set of positions in the workspace where the robot can apply a desired wrench [48, Chapter 5]. Another important aspect is the singularity analysis: it is the study of the singularities in the workspace such as the robot loosing one or more degrees of freedom [48, Chapter 4]. A further significant aspect to consider is the collision between the cables, the end effector and the structure. In the case of the CDPRs subject to collision constraints, one could use collision detection algorithm like the one presented in [49]. Finally, it is worth highlighting the aspect of stiffness "characterized by the infinitesimal displacements δy of the mobile platform that are generated by infinitesimal wrenches δw_P applied to it" [48, Chapter 3]. The stiffness of the CDPR is an important aspect to consider, in order to reduce the impact of perturbations on the precision of the end effector positioning. For example in the case of the CDPR used in the project *Lab-on-Cable*, the configuration used is the one with 8 cables offering 6 degrees of freedom. Because this CDPR is only used for translational

motion, we are more interested in the translational stiffness than the rotational one. Changing the attachment points of the cables have a direct impact on the stiffness of the CDPR hence the robustness to disturbance and the tracking accuracy. Passing from configuration 1 (figure 2.1) to configuration 2 (figure 2.2) has decreased the rotational stiffness of the CDPR drastically, while the mean of translational stiffness only change slightly with more uniform distribution over the workspace (figure 2.3).

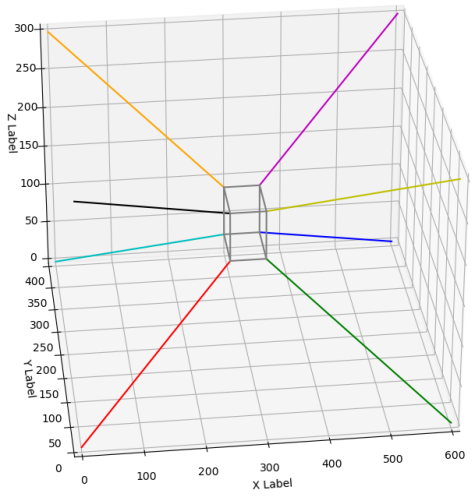


Figure 2.1: Configuration 1: CDPR with 8 cables where every cable is attached to the end effector directly without crossing

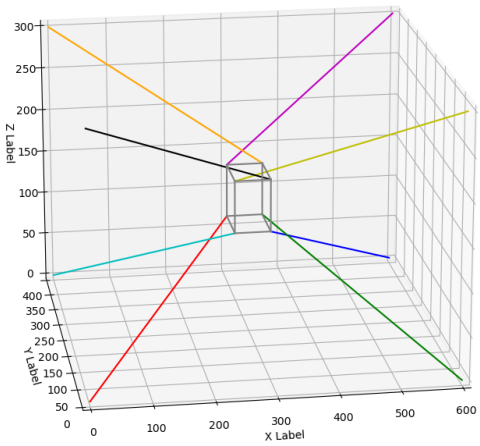


Figure 2.2: Configuration 2: CDPR with 8 cables where the cables are crossed

To eliminate the rotation of the end effector, a new configuration with 12 cables is proposed in [46] as shown in figure 2.4. This configuration is the most suitable one for the insect tracking application since it allows the end effector to move in the 3D space without any rotation. Each pair of cables on top have the same tension and the same length, so it forms a parallelogram with the side of the end effector, this ensures that the end effector does not rotate. As it has been decided to adopt this new configuration in the project *Lab on Cable*, we will be only interested in the transitional configurations. From the CDPR that already exists in the lab we will build two purely translational CDPRs with 4 and 8 cables, one where we consider the end effector as a point mass moving in 2D space actuated by 4 cables figure 2.5 and the other where the end effector is still a point mass but moving in 3D space and actuated by 8 cables.

2.3 Geometric model and inverse kinematics

The geometric model defines the relationship between the position of the end effector and the lengths of the cables.

On the one hand, the end effector's movement is described by the user in the fixed base frame using the vector:

$$X = [x, y, z, \alpha, \beta, \gamma]$$

where x, y, and z represent the position of the end effector, and α, β, and γ are the rotation angles around the x, y, and z axes, respectively.

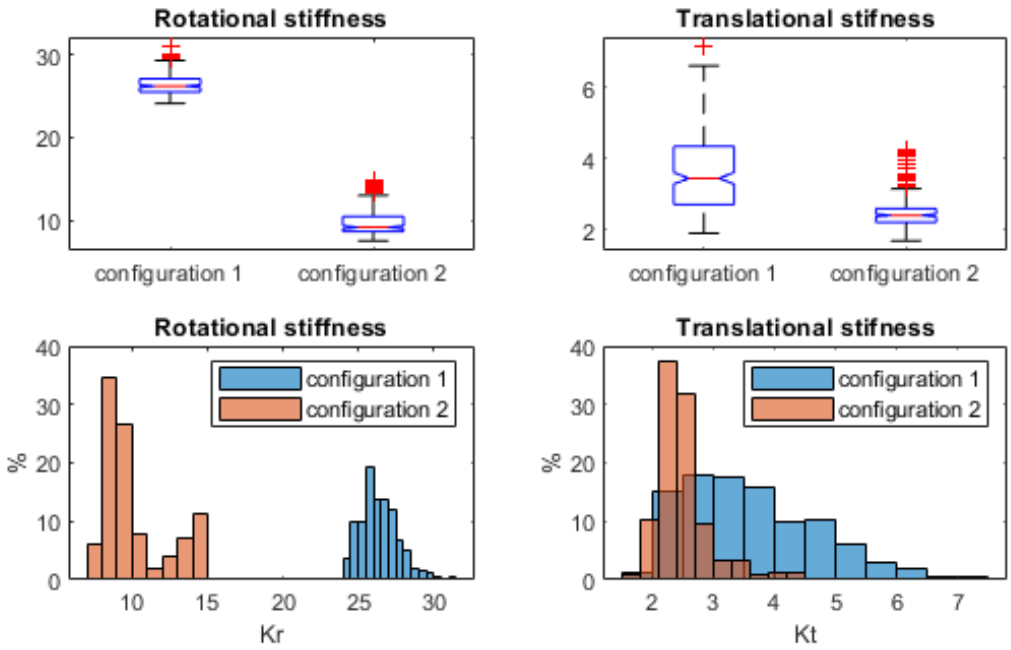


Figure 2.3: Comparison of the stiffness of the CDPR in configuration 1 and configuration 2. K_r is the rotational stiffness, K_t is the translational stiffness

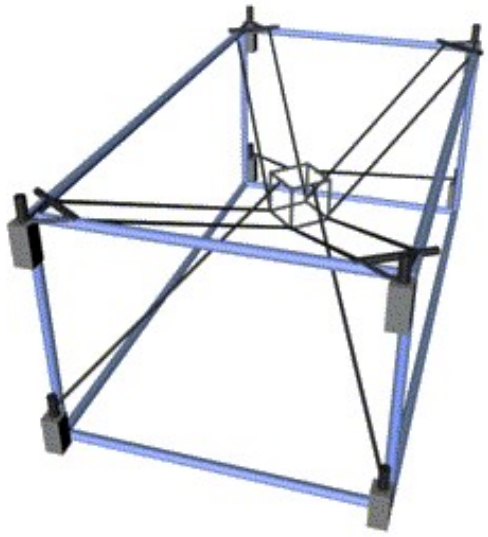


Figure 2.4: Configuration 3: CDRP with 12 cables : each pair of the cables on top have the same length, so it forms a parallelogram with the side of the end effector, this ensures that the end effector do not rotate.

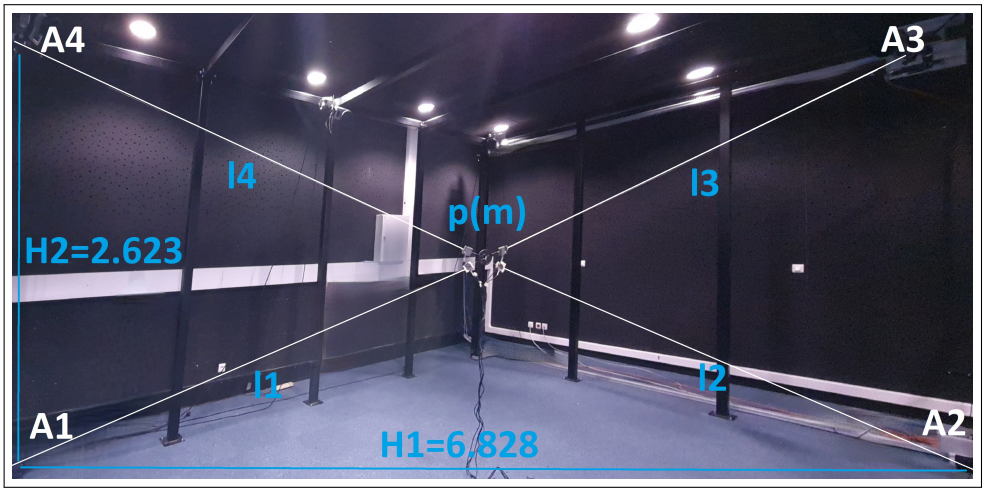


Figure 2.5: Two-DOFs point-mass CDPR with four cables, l_k is the length of the cable k, A_k is the position of the motor k, p is the position of the end effector.

On the other hand, from the system's perspective, the movement is characterized by the lengths of the cables, as only these lengths are measurable, and the position of the end effector is not directly observable. Determining the cable lengths based on the end effector's position is known as the inverse kinematics problem. For cable-driven parallel robots, this is a straightforward process, as the cable lengths are calculated using the norm of the vector between the attachment point and the winch position.

In contrast, the forward kinematics problem, where the end effector's position is determined from the cable lengths, only has a numerical solution. The forward kinematics problem will be discussed in the next section. To solve the inverse kinematics problem, we need to compute the lengths of the cables based on the position of the end effector. Thus a function $IK(X) = L$ is defined, where L is the vector of the lengths of the cables. The Jacobian matrix associated with this function at position X is a $n \times 6$ matrix, where n is the number of cables. It is used by definition to compute the variation of the lengths of the cables based on the variation of the position of the end effector $\dot{L} = J(X)\dot{X}$. The Jacobian matrix is defined by:

$$J(X) = \begin{pmatrix} \frac{\partial l_1}{\partial x} & \frac{\partial l_1}{\partial y} & \frac{\partial l_1}{\partial z} & \frac{\partial l_1}{\partial \alpha} & \frac{\partial l_1}{\partial \beta} & \frac{\partial l_1}{\partial \gamma} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial l_n}{\partial x} & \frac{\partial l_n}{\partial y} & \frac{\partial l_n}{\partial z} & \frac{\partial l_n}{\partial \alpha} & \frac{\partial l_n}{\partial \beta} & \frac{\partial l_n}{\partial \gamma} \end{pmatrix} \tag{2.1}$$

To compute the lengths of the cables, the geometric model of the CDPR could be used directly (equation: 2.2). This model describes the configuration of the system, including the positions of the motors, the attachment points of the cables, and the shape of the end effector. The geometric model for the CDPR used in the *Lab-on-Cable* project is illustrated in figure 2.6. The cable lengths are computed using the following equation:

$$\vec{l_i} = \overrightarrow{O_f O_m} + \overrightarrow{O_m B_i} - \overrightarrow{O_f A_i} \tag{2.2}$$

where $l_i = \left\| \vec{l_i} \right\|$ is the length of the cable i, $\overrightarrow{O_f O_m}$ is the position of the mobile platform in the fixed base, $\overrightarrow{O_m B_i} = R(X)\overrightarrow{O_m B_{iref}}$ is computed using the rotation matrix $R(X)$ and the position of the attachment point of the cable i in the mobile platform denoted by B_{iref}, and $\overrightarrow{O_f A_i}$ is the

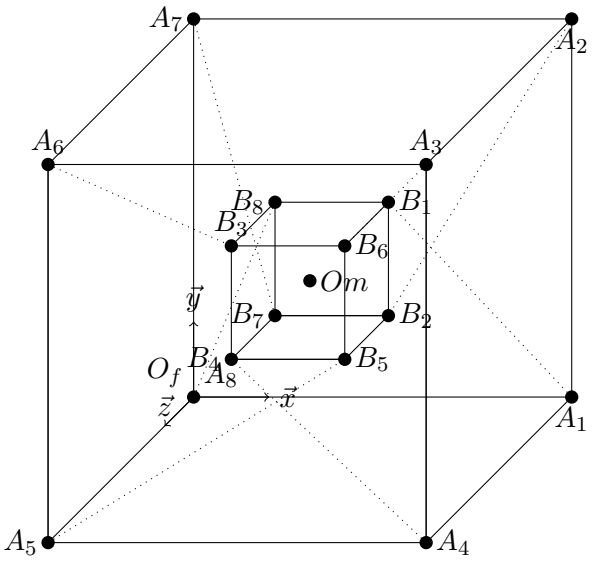


Figure 2.6: Geometry of the lab-on-cables. The vertices of the end effector are at the points B_i, i = 1 ... 8. The origin of the fixed and mobile frame of reference is O_f and O_m, respectively. The dashed lines represent the cables changing the pose of the flying frame (end effector) via motorized winches. The i-th cable connects the points A_i corresponding to the position of the pulley at the entrance of the i-th winch to the distal anchor point B_i.

position of the winch in the fixed base.
O_mO_f is given by the translational components of the pose (x, y, z), R is given by the Euler angles (α, β, γ), and the B_{iref} and A_i are fixed and correspond to the geometry of the robot.

$$\vec{l_i} = (x, y, z) + R(\alpha, \beta, \gamma) \cdot O_m B_{iref} + O_m A_i \tag{2.3}$$

From this equation, we can compute the Jacobian matrix by differentiating the lengths of the cables with respect to the position of the end effector.

In the case of the CDPR with 4 cables (figure 2.5), where the end effector is supposed to be a point mass moving in 2D space, the equation 2.3 becomes:

$$l_i = \sqrt{(x - a_i)^2 + (y - b_i)^2} \tag{2.4}$$

x and y are the position of the end effector in the fixed base, a_i and b_i a are the coordinates of the attachment point A_i and B_i of the cable i in the fixed base.

2.4 Forward kinematics and position estimation

The forward kinematics problem is the computation of the position of the end effector from the lengths of the cables. In general, the forward kinematics problem is not directly solvable for CDPRs, so the use of numerical methods is required. In [1], they use a simple optimization technique (gradient descent) to minimize the error between the lengths of the cables computed from the estimated position of the end effector and the measured lengths of the cables. Other position estimation approaches for CDPRs are presented in [48, Chapter 4]. In the case for redundant translational CDPRs, the forward kinematics problem is directly solvable, as the

position of the end effector is computed by the intersection of the spheres of the attachment points of the cables.

In the case of the 2D CDPR with 4 cables, the position of the end effector is computed by the intersection of the circles of the attachment points of the cables. The intersection of two circles of two successive cables is always two points, one in the workspace and the other one outside the workspace, so by using each pair of successive cables, we have 4 intersection points. Ideally, all circles (or spheres in the 3D case) should be concurrent (i.e. intersect all at the same point). Due to modelling and measurements inaccuracies, they do not and we need a way to approximate the intersection point. By computing the median of these points, we have the position of the end effector while avoiding the outliers. In the case of $3D$ CDPRs: 8 or 12 cables, the position of the end effector is computed by the intersection of three spheres of the attachment points of the cables. When the end effector is not a point mass but a rigid body, the center of the spheres is the translational displacement of the attachment points of the cables by the vector $\overrightarrow{B_iO_m}$, this translational displacement is applied to the cable vector $\overrightarrow{l_i}$ so that all the points B_i are displaced to O_m and the intersection of the spheres gives the position of the end effector.

2.5 PID-based control

This section describes the controller that was developed before the beginning of this thesis' project. This control scheme, that we name "PID" (after the way the end effector motion is computed with respect to the location of the target), represent an useful reference point, that we will use as a performance landmark to evaluate RL-based control.

This approach represents a classical control method that does rely only on the inverse kinematics model of the robot and the Jacobian matrix without the use of the dynamic model. It utilizes a PID controller to calculate the tracking speed based on the error between the robot's current position and the desired position (PID controller block). The tracking speed is then transformed into winding speed using the robot's Jacobian matrix (Inverse Kinematics block). Subsequently, a quadratic optimization problem is solved to determine a correction term to be added to the winding speed in order to make sure they respects limits on cable tensions estimated through the motors currents (Tension Control block). Finally, the winch speeds are converted into motor commands (Winches I/O block) [1]. The control scheme is illustrated in figure 2.7.

2.6 Dynamic modeling

The dynamic model of the robot aims to represent the "operating system" of the robot, i.e. the way the robot reacts to the inputs (motor commands) and the external forces (gravity, friction, etc.). It refers to the system to be controlled and the way it evolves over time. It can be considered as digital twin of the robot, as it allows to simulate the robot's behavior in a virtual environment, allowing to experiment different control strategies quickly and safely. It is an essential tool for training reinforcement learning controller. It is important to emphasize that this model does not include the control algorithm itself, rather, it include all elements in the system from actuators to sensors, and the way they interact with each other.

While certain components, such as sensors and the communication network are assumed to be perfect, others like the internal proportional control loop of the winch are modeled in detail.

The system consists of an end effector connected to the structure by cables, which are driven by DC motors. The dynamic model of the robot involves deriving the equations that govern the

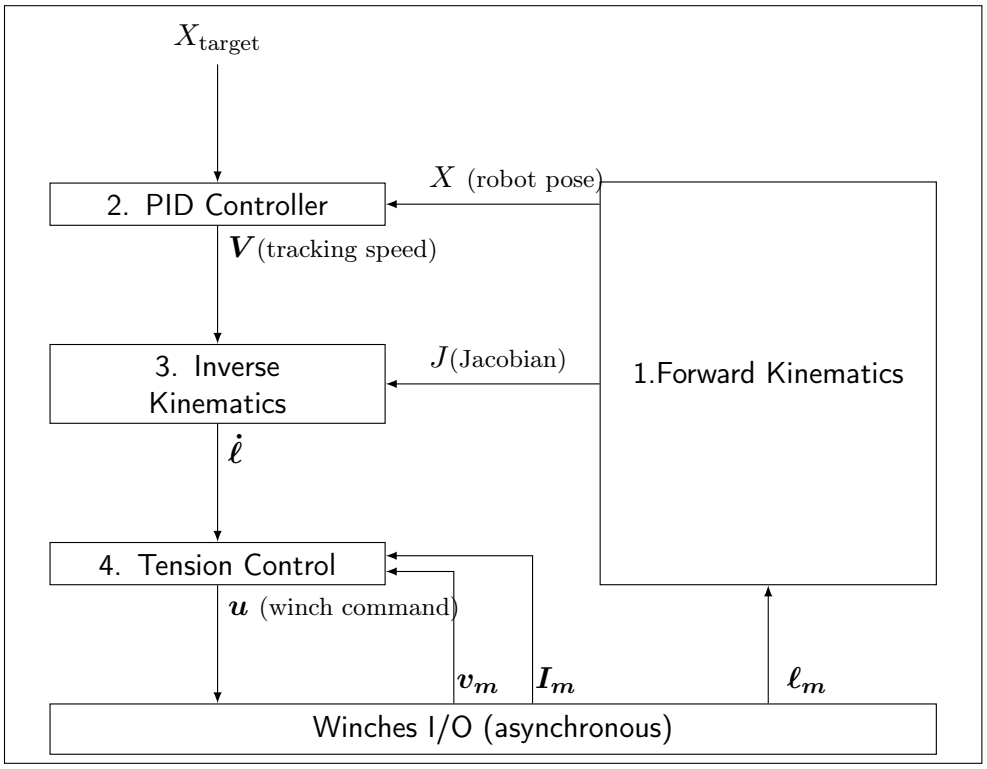


Figure 2.7: The PID-based control scheme of the robot consists of four steps: (i) estimation of the robot pose by using forward kinematics, (ii) computation of the tracking speed to minimize the tracking error, (iii) transformation into a winding speed vector by using the Jacobian and inverse kinematics, and (iv) conversion to motor command with constraints on the cable tensions. The vectors $I_m = (I_{1m}, \ldots, I_{8m})$, $l_m = (l_{1m}, \ldots, l_{8m})$, $v_m = (v_{1m}, \ldots, v_{8m})$ are measurements of the motor currents, of the winding/unwinding speeds, and of the cable lengths, respectively [1]

relations between the inputs of the system (the set speeds of the motors) and the outputs of the system (the lengths of the cables).

2.6.1 Dynamic equations of transitional CDPRs

In this section, we will be interested only in the translational CDPRs, where the end effector is moving in the 3D space. The dynamic model of the robot is derived from the Newton-Euler formalism. For more details about the dynamic modeling of CDPRs, the reader can refer to [50].

$$M(X)\ddot{X} = WT + w_p \tag{2.5}$$

with:

$$M(X) = m_e I_{3\times 3}, w_p = w_g + w_e \tag{2.6}$$

- $X = \left[x, y, z\right]^T$ pose of the end effector in the fixed base
- $\dot{X}$ velocity vector of the end effector in the fixed base
- $\ddot{X}$ acceleration vector of the end effector in the fixed base
- $M(X)$ is the inertia matrix of the effector
- m_e the mass of the end effector
- w_p the vector representing the external forces and torques applied to the end effector
- w_g the vector representing gravity itemw_e the vector representing other internal forces than gravity
- $T = \left[T_1, T_2, T_3, ..., T_n\right]^T$ cable tensions
- n number of cables
- $W = -J^T$ the vector describing the direction of the forces applied by the cables, it is the transpose of the Jacobian matrix
- J the Jacobian is defined by $\dot{l} = J\dot{X}$
- l cable lengths

To simplify our model, we will consider that the forces applied on the end effector are the forces applied by the cables and the gravity force.
Thus our simplified model is:

$$m_e I_{3\times 3}\ddot{X} = -J^T(X)T + \left[0, 0, -m_e g\right]^T \tag{2.7}$$

2.6.2 Motor dynamic model

Electrical Equation

The electrical equation of the motor is written as

$$U = E + Ri + L\frac{di}{dt} \quad avec \quad E = K_e\Omega \tag{2.8}$$

It is the application of Kirchhoff's Law-mesh method in the electric circuit

Mechanical Equation

Using the Fundamental Equation of Dynamics for rotating systems :

$$J_{moteur}\frac{d\Omega}{dt} = C_m - C_f - C_{fs} - C_r \tag{2.9}$$

with

$$C_m = K_c i \tag{2.10}$$

and

$$C_f = f\Omega \tag{2.11}$$

with:

- K_c: Torque constant
- J_{motor}: inertia of the motor
- Ω: angular speed in rad/s of the motor
- C_m: motor torque
- C_f: viscous friction torque
- C_{fs}: dry friction torque
- C_r: resistive torque

for our system we have

$$C_r = rT \tag{2.12}$$

- r: pulley radius
- T: cable tension

To build our model we will use the mechanical equation and we will neglect the friction torques so the mechanical equation of the motor n becomes:

$$J_{moteur}\frac{d\Omega_n}{dt} = K_c i_n - rT_n \tag{2.13}$$

In the case of n motors we have :

$$J_{moteur}\dot{\Omega} = K_c i - r.T \tag{2.14}$$

with $\Omega = \left[\Omega_1, \Omega_2, \Omega_3, ..., \Omega_n\right]^T$, $i = \left[i_1, i_2, i_3, ..., i_n\right]^T$ et $T = \left[T_1, T_2, T_3, ..., T_n\right]^T$

Speed control loop

For speed control, an internal proportional loop embedded in the motor is used. The motor speed is controlled by the motor current, which is proportional to the difference between the set speed and the measured speed. The equation of the motor speed control loop is given by:

$$i = k_{moteur}(u - v) \tag{2.15}$$

- i: the motor current

- k_{motor}: Speed loop proportional constant
- u: the set speed in rpm
- v: the measured speed in rpm

In the case of n motors we note : $i = \left[i_1, i_2, i_3, ..., i_n\right]^T$, $U = \left[u_1, u_2, u_3, ..., u_n\right]^T$ et $V = \left[v_1, v_2, v_3, ..., v_n\right]^T$
so the equation (2.15) for n motors is :

$$i = k_{moteur}(U - V) \tag{2.16}$$

With

$$V = \frac{\Omega}{2\pi} \tag{2.17}$$

2.6.3 Cables model

For cables model we used the model in [51] with some simplifications. Thus, the relation between the cable tension and the motor speed is :

$$T_i = c_{ref}\Delta l_i + d_{ref}\Delta \dot{l_i} \tag{2.18}$$

with

$$\Delta \dot{l_i} = J(X)dotX - r\Omega \tag{2.19}$$

2.7 Simulation with Matlab/Simulink

To simulate the CDPR system, the model is implemented in Matlab/Simulink. Two simulations are performed: the first one with a simplified model where the cable tensions are directly linked to the motor currents by a constant α, and the second one with a more complex model where the mechanical model of the motor and dynamic model of the cables are integrated.

2.7.1 Simulation 1: Simplified model

The scheme of the simulation is shown in figure 2.8. The input is the desired speed of the motors U and the output is the position of the end effector X and the cable lengths l. The current i is calculated using the equation (2.16). Assuming that the motor speed is constant, the equation (2.14) can be simplified to $T = \frac{K_e}{r}i$ and using the dynamic equation of the robot (2.7) we can calculate the position of the end effector using the resulting acceleration and the initial conditions. In the case of 4 cables CDPR, the equation (2.7) becomes:

$$m\vec{a} = T_1.\vec{n}_1 + T_2.\vec{n}_2 + T_3.\vec{n}_3 + T_4.\vec{n}_4 + \vec{P} \tag{2.20}$$

$\vec{a}$ represents the acceleration vector of the end effector, $\vec{n}_i$ represents the unit vector of the cable i, T_i represents the tension of the cable i and $\vec{P}$ represents the Earth's gravitational force applied to the end effector. This equation can be rewritten as:

$$m\ddot{x} = T_1.n_{1x} + T_2.n_{2x} + T_3.n_{3x} + T_4.n_{4x} \tag{2.21}$$

$$m\ddot{z} = T_1.n_{1z} + T_2.n_{2z} + T_3.n_{3z} + T_4.n_{4z} - mg \tag{2.22}$$

n_{kx} and n_{kz} are the components of the unit vector $\vec{n}_k$ in the x and z axis, respectively. The Jacobian matrix of the robot is given by [48]:

$$\vec{J} = \begin{bmatrix} n_{1x} & n_{2x} & n_{3x} & n_{4x} \\ n_{1z} & n_{2z} & n_{3z} & n_{4z} \end{bmatrix} \tag{2.23}$$

The dynamic model of the robot can be rewritten as:

$$m\ddot{\vec{X}} = \vec{J}(X).\vec{T} + \begin{bmatrix} 0 \\ -mg \end{bmatrix} \tag{2.24}$$

$$where \quad \vec{X} = \begin{bmatrix} x \\ z \end{bmatrix} \quad and \quad \vec{T} = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix}.$$

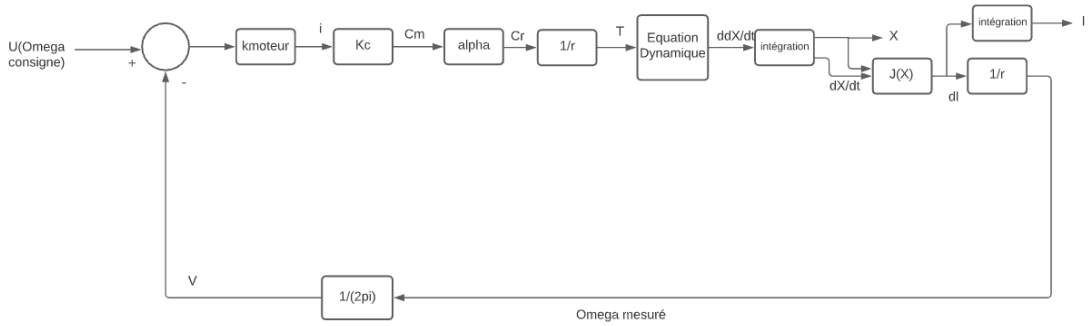


Figure 2.8: Simulation 1: Simplified model, the cable tensions are directly linked to the motor currents by a constant α , the input is the desired speed of the motors U and the output is the position of the end effector X and the cable lengths l

2.7.2 Simulation 2: Simulation with the mechanical model of the motor and the dynamic model of the cables

The scheme of the simulation is shown in figure 2.9. Same as simulation 1, the input is the desired speed of the motors U and the output is the position of the end effector X and the cable lengths l. The difference is that the tension of the cables is calculated using the equation (2.18). This requires the calculation of the $\Delta\dot{l}_i$ using the Jacobian matrix, the speed of the end effector and the motor speed (2.19). The motor speed is calculated using the mechanical model of the motor (2.16) and the acceleration of the end effector is calculated using the dynamic equation of the robot (2.7).

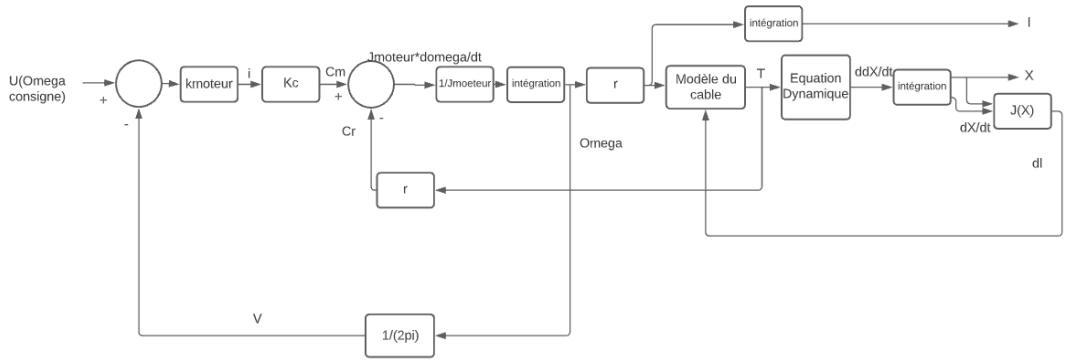


Figure 2.9: Simulation 2: Simulation with the mechanical model of the motor and the dynamic model of the cables, the cable tensions are calculated using the equation (2.18), the input is the desired speed of the motors U and the output is the position of the end effector X and the cable lengths l

2.7.3 Results: validation of the model in simulation with real data

The parameters of the CDPR are shown in Table 2.1, we compared both versions of the simulation using the real data. The second simulation requires identification of the parameters of the cable model c_{ref} and d_{ref}, while for the first simulation, we have access to the value of all the parameters. The results of both simulations were so close that we decided to use only the first simulation for the rest of the work as the second simulation is more complex and requires parameters identification.

We simulated the developed model in figure 2.8 of the robot using Simulink. To validate the model, we used real data that we collected during a trajectory tracking using PID-based controller with the real CDPR.

We used the same motors speed generated during the real trajectory tracking experiment to compare the real and simulated end effector position for the same inputs. The results are shown in figure 2.10. The simulation and the real data are very close. The maximum error is 0.04 m in both x and z axis. We also measured the cable tensions experimentally using Phidget 22 force sensors during trajectory tracking and found that the current is proportional to the cable tension (Pearson correlation $R^2 = 0.98$, figure 2.11). Thus, the measured motor currents i_t can be used as estimators of the cable tensions.

Table 2.1: CDPRs parameters

Parameter	Value	
Nominal current	I_{nom}	6.8 A
Nominal speed	V_{nom}	3420 rpm
Nominal torque	T_{nom}	0.8 Nm
Control signal frequency	f_e	100 Hz
Motor torque constant	K_c	0.123 Nm/A
Motor Current proportional gain	K_{motor}	1 A/rps
Drum radius	r	0.0178 m
End effector mass	m	1 kg

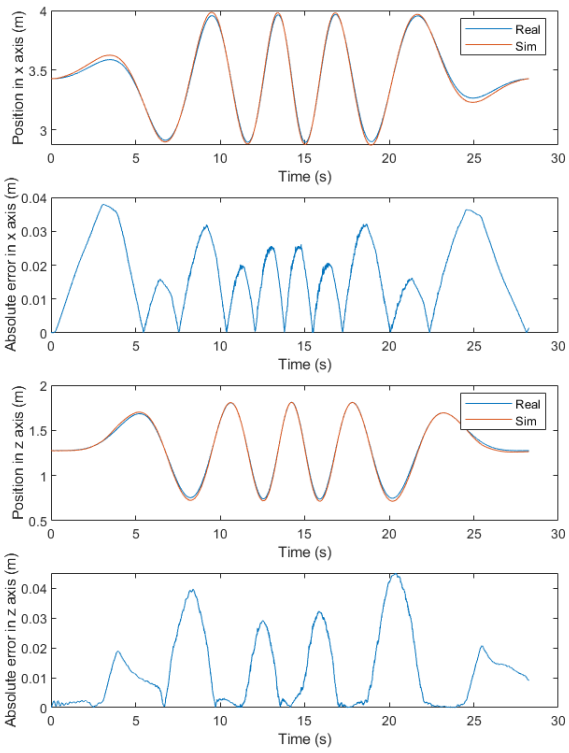


Figure 2.10: Comparison between the real and simulated end effector trajectories using the same inputs.

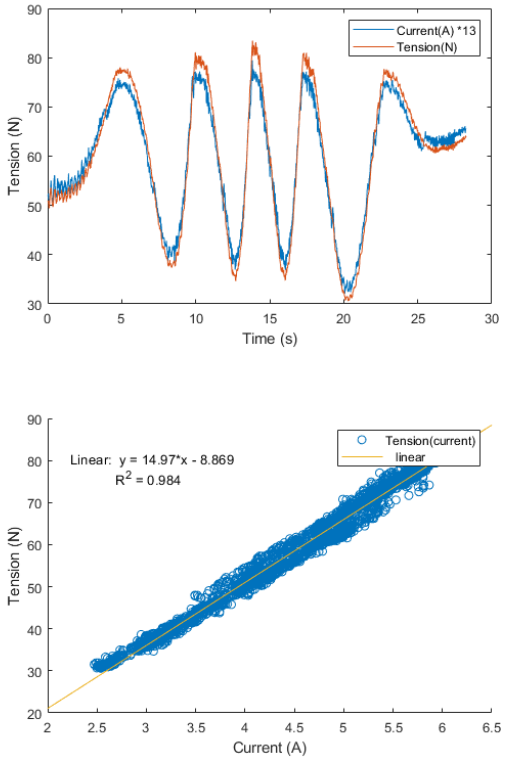


Figure 2.11: Correlation between the cable tension and the motor electrical current. Experimental data recorded for one motor.

2.8 Conclusion

In this chapter, the Cable-Driven Parallel Robot (CDPR) used in the project *Lab on Cable* have been thoroughly presented. Key aspects, including both the forward and inverse kinematics of CDPRs, have been discussed in detail, providing a solid theoretical foundation. Additionally, the dynamic model of the robot has been developed and subsequently simulated using Matlab/Simulink, ensuring a comprehensive computational framework for future control applications. The accuracy of the simulation results has been rigorously validated by comparing them with real-world data collected during a trajectory tracking experiment using the PID-based controller, demonstrating the robustness and reliability of the model.

These foundational elements, including the kinematic analysis, dynamic modeling, and simulation, will serve as the basis for the development of a reinforcement learning (RL) algorithm for the control of CDPRs in the upcoming chapters. The scalability of the developed simulation is a key feature, as it can be adapted to various CDPR configurations, regardless of the number of cables or their arrangement. This flexibility is critical for extending the RL algorithm's application across different CDPR setups, enabling experimentation and performance evaluation on configurations that differ from the current setup in the lab. Specifically, it will allow us to test and refine the RL algorithm for a 12-cable CDPR configuration, which remains a future goal due to the unavailability of the physical system in the lab at this time.

Chapter 3

A deep reinforcement learning approach for the control of cable-driven parallel robots

Contents

3.1 Introduction

The use of deep reinforcement learning (DRL) in control has been the subject of several studies in recent years, as presented in chapter 1. Some of these studies have focused on the use of RL side by side with traditional control techniques. This approaches are called *Hybrid control*, while others approaches that use RL as the only control technique are called *End-to-End reinforcement learning*.
In this work we rather focus on the second option, as it is the most suitable for highlighting the potential of DRL in control [8, 16]. The main advantage of this approach is the ability to learn the control policy from scratch, without the need for a priori knowledge of the system to be controlled. Moreover, it treats the complex systems as whole, without the need to decompose them into sub-systems, which helps to overcome the limitations of traditional control techniques, such as taking into account the non-linearity of the system, the presence of disturbances, and the uncertainties in the model. This chapter first establishes the environment where the control problem is defined, then it presents some of the challenges that we faced in the training process and how we solved them, and finally, it presents the results of the training process and the comparison between different RL agents and a PID-based control.

3.2 Training environment, state and action space

One of the advantage of using DRL in the control of complex systems is the ability to treat the system as a black box where we ignore the internal dynamics of the system. So instead of trying to solve each sub-problem separately as in the traditional control, the policy is learned directly from the interaction with the environment to achieve the desired objective. This is done by defining the system as an environment, and the control policy as an agent that interacts with the environment. This interaction is done through the exchange of the action a_t and the observation s_t at each time step t. For CDPRs, the action a_t is the control signal that we send to the motors, it could be the torques, the currents, or the desired speed of the motors as in our case [1]. The observation s_t is the state of the system at time t, it is a vector chosen according to the objective of the control, and the measurements available. In our case we defined the state space by:

$$s_t = (X_t, X_{t-1}, e_t, e_{t-1}, I_t) \tag{3.1}$$

where X_t is the position of the end effector at time t, e_t is the error between the current position and the target position at time t, and I_t is the current of the motors at time t.

The state space is the set of all possible states that the system can take, sometimes it is different from the space defined by the state variables, because it is not always possible to reach all the states of the state variables. For example, in the case of CDPRs, if we decided to use the cables length instead of the position of the end effector as state variables, we will have a state space that is not continuous, because not all the combinations of cable lengths are possible.

The action space is the set of all possible actions that the agent can take, for CDPRs the dimension of the action space is equal to the number of motors, and the action space is continuous, but have a limited range of values that the motors can take. We will see later, each action depends on the other actions and the state of the system, so some combination of actions are not safe for the system.

The environment is the system that we want to control, it depends on the dynamics of the system, the objective of the control, the measurements available, and the control signal. Since we have no control over those parameters. This results in many challenges that we need to overcome

to succeed in the training process.
The first challenge is the desired trajectory generation. We are solving a tracking problem. The desired trajectory is part of the state, it is something that we generate, which is good thing as we could use it to explore the state space. However, the desired trajectory affects significantly the training process as the agent needs to learn to control the robot in all the states of the environment, so we need to generate trajectories that are coherent with the dynamic limits of the robot, and that visit all the states of the environment.
The second challenge is in the action space exploration. For our system the control signal is the desired speed of the motors, and the current of the motors is proportional to the difference between the desired speed and the measured speed of the motors. That results in a negative desired current when the desired speed is less than the measured speed, which is not possible, but our agent ignore that and will choose this action, thus the tension of the cable will be zero, which is not safe for the system. To solve this problem, we propose to use a saturation function to limit the action space, so the agent will not choose actions that are not safe for the system.
The last challenge is the reward function. It is used to guide the agent to learn the best policy for a given task to achieve, designing a good reward function is not an easy task, in the reward engineering section we will introduce the reward function that we used in our training process.

3.3 Desired trajectory generation: the case of trajectory tracking in bounded workspace

Visiting all the states of the environment is extremely important for the training process, as the agent needs to learn the best action for each state. In the case of trajectory tracking, the desired trajectory is part of the state, and as it is something that we generate, we have control over it. But we still need to generate trajectories that are coherent with the dynamic limits of the robot, and that visit all the states of the environment. To solve this problem we decided to generate random trajectories with random speed and acceleration. The trajectories are generated using the following procedure: First, we set the maximum acceleration a_{max} and the maximum speed v_{max}. Then, we establish a random initial position X_0 at the start of each episode and reset the speed and acceleration to 0. Next, at each time step t, we generate a random acceleration a_t using a gaussian distribution such that $-a_{max} \leq a_t \leq a_{max}$, and by integrating the acceleration, we generate the speed v_t using the following equation:

$$v_t = v_{t-1} + a_t * dt \tag{3.2}$$

We check that the speed is between the limits. If not we set it to the limit. Finally, we generate a trajectory using the following equation:

$$X_t = X_{t-1} + v_t * dt \tag{3.3}$$

This approach works very well for low speed or unbounded workspace, but for high speed, we often reach the workspace limits before the end of the episode. To overcome this problem we tried to reduce the episode length, but it leads to instability as the samples of rewards used to estimate the average return is reduced. So instead of reducing the episode length, we decided to change the desired speed v_t to random speed in the other direction when the end effector reaches the workspace limits, so that the end effector stay in the workspace. This approach works well for high speed even if it is not the best solution, as the desired speed is not coherent with the dynamic limits of the robot and the generated trajectories are not smooth. But it is acceptable

in the training phase as the robot will visit all the states of the environment and the agent will learn to control the robot in all the states.

We use the parameters shown in table 3.1 to generate the trajectories for our CDPRs. The velocity limits are chosen to match the velocity limits of the robot. The nominal speed of the motors is 3000 rpm which is equivalent to 4 m/s for the cables. The acceleration limits are chosen higher than the maximum acceleration of the robot to reach maximum speed during the limited episode length and also because the acceleration is generated using a random gaussian distribution, so the robot will not reach the maximum acceleration in all the states of the environment. In figure 3.1 we show an example of the variation of the end effector's position over

Table 3.1: Target generation parameters

Parameter	Value
Maximum acceleration in x-axis	35 m/s^2
Maximum acceleration in z-axis	35 m/s^2
Maximum velocity in x-axis	4 m/s
Maximum velocity in z axis	4 m/s

time for a generated trajectory using the above procedure. When the target pose reaches the workspace limits, the speed is inverted to keep the target position of the end effector in the workspace.
In figure 3.2 we show the distribution of the generated target poses in 2D workspace over 500 episodes with 1000 steps each. The density of points is uniform at the most parts of the workspace, which is good as the agent will learn to control the robot in all the states of the environment. There is a high density of points at the center and the edges of the workspace that we could not avoid basically for two reasons: For the real robot resetting the position to a random one at the start of each episode will require a controller to bring the robot to the desired position which contradict the objective of the training, the development of a controller from scratch without the use of a priori knowledge of the system. Even with some basic position control based only on the cable lengths, the robot will take a lot of time to reach the desired position which will add more time to the training process. So we decided to rely on the random last position of the robot at the end of the episode when it is possible, otherwise we reset the position to the center of the workspace. The second one is that the workspace is bounded, so to visits all the states of the environment while having smooth target trajectories that are coherent with the dynamic limits of the robot, the workspace limits will be visited more than the other states.
In figure 3.3 we show the distribution of the generated target speeds over 500 episodes with 1000 steps each. The episode always start with zero speed to have coherent objective trajectories for the robot to follow, this influences the distribution of the speed to be more concentrated around the zero speed. Because the acceleration is generated using a random gaussian distribution, the probability of having a high speed is low. The speed limits are more visited than the other states, as every point in the trajectory exceeding the speed limit will be set to this limit.

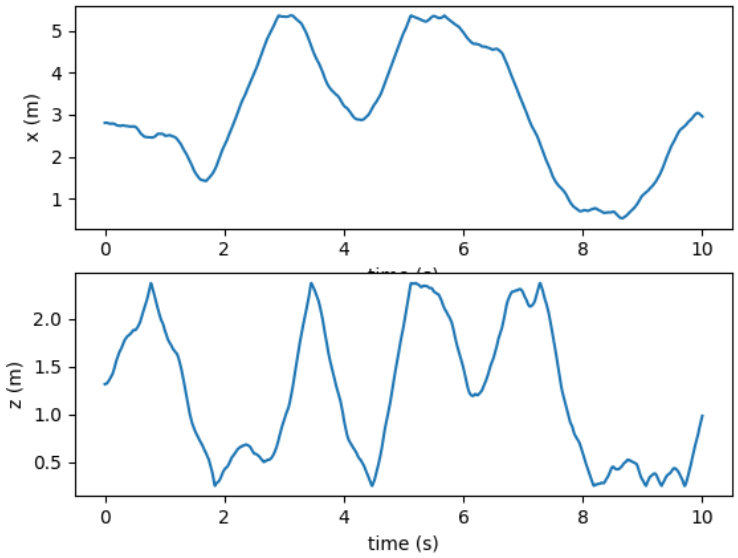


Figure 3.1: Example of a generated trajectory for training, the trajectory does not reach the workspace limits as the speed is inverted when it reaches virtual limits.

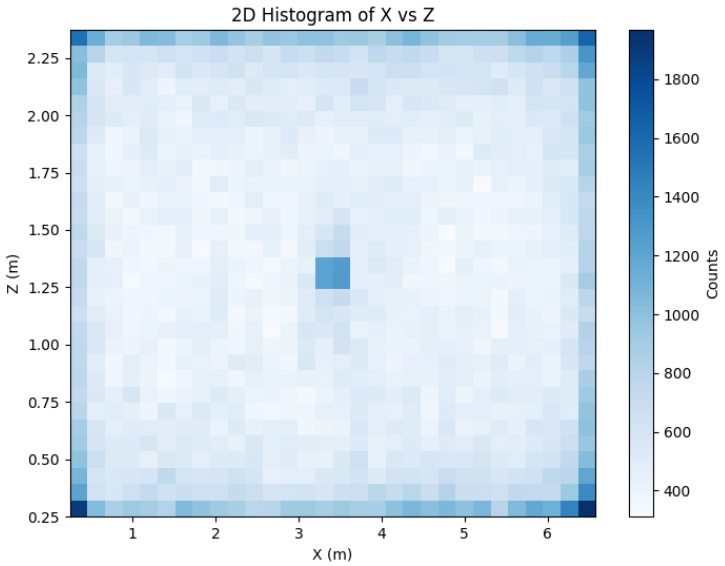


Figure 3.2: Distribution of the generated target poses over 500 episodes with 1000 steps each. There is a high density of points both at the center and the edges. This is because the position is reset to the center at the start of each episode when the last episode ends at the workspace limits, and the end effector's speed is inverted upon reaching the workspace limits at the end of an episode.

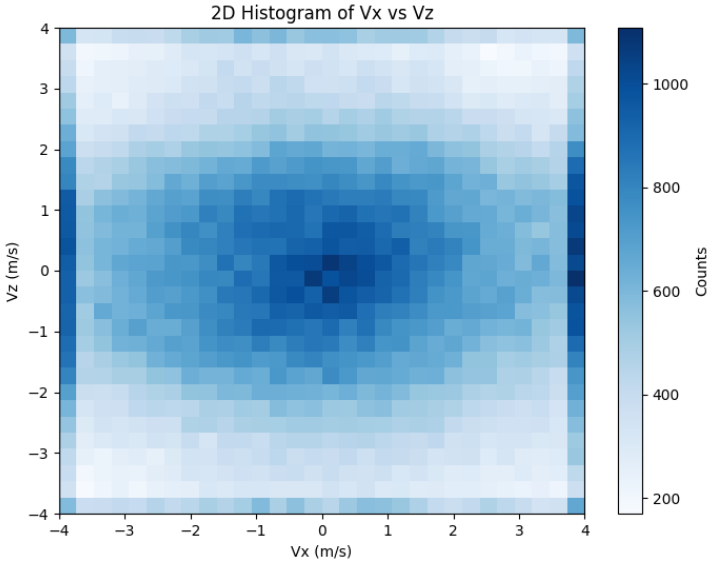


Figure 3.3: Distribution of the generated target speeds over 500 episodes with 1000 steps each. The speed follows a gaussian form, with a peak at the zero speed because the acceleration is generated using a random gaussian distribution. The speed limits are more visited than the other states, as every point in the trajectory exceeding the speed limit will be set to this limit.

3.4 Current constraints

3.4.1 Current control loop

Cable driven parallel robots are robots that use cables to transmit forces from the motors to the end effector, so the control signal is normally the tension of the cables, but as the tension is not directly measurable, the current of the motors is used instead. The CDPR that we have in the lab uses a proportional control loop where the current of the motors is proportional to the difference between the desired speed and the measured speed of the motors. The current is computed using the following equation:

$$i = K_{motor}(u - v_m) \tag{3.4}$$

where i is the current of the motors, K_{motor} is the motor constant, u is the desired speed of the motors, and v_m is the measured speed of the motors. From this control loop we could deduce that any desired speed that is less than the measured speed will lead to a negative desired current, which is simply the motor rotating in the other direction, and the tension of the cable will be zero which is not safe for the system. Normally we have only control over the desired speed of the motors, but as we have access to the measured speed of the motors, we could act on the current of the motors to respect the current limits by modifying the desired speed of the motors.

3.4.2 Current bounds

The cables in the CDPR should always be in tension, because when winching slack cables, there is a high risk that they get entangled around the drum, and the robot will lose its ability to

move. In this situation we should stop the robot and fix the cables to bring the system to its safety state. To avoid this, we set a minimum current limit i_{min}. On the other hand, the current should not exceed a maximum limit i_{max}, as there is a risk of damaging the motors, the cables, or the end effector. So we set a maximum current limit i_{max}. As we use the desired speed of the motors as the control signal, the current limits are translated to speed limits at each time step using the following equation:

$$i_{sat} = \begin{cases} i_{max} & \text{if } i > i_{max} \\ i & \text{if } i_{min} < i < i_{max} \\ i_{min} & \text{if } i < i_{min} \end{cases} \tag{3.5}$$

And based on the current control loop the action of the agent is deduced using this function:

$$u_{sat} = \frac{i_{sat}}{K_{motor}} + v_m \tag{3.6}$$

By computing the action of the agent using this function, we ensure that the current of the motors is between the limits, and the cables are always in tension. Moreover, as we map each unsafe action to the nearest safe action, we ensure that the agent will not choose actions that are not safe for the system and will learn to control the robot in all the states of the environment.

3.5 Reward design

3.5.1 Reward-engineering

The reward function is an important part of the reinforcement learning algorithm, it is used to guide the agent to learn the best policy for a given task to achieve. In the case of redundant cable driven parallel robots, there are many solutions to reach the same target position, so a reward function that only takes into account the position error is not enough to achieve a good policy, because the policy oscillates between different solutions, leading to noisy control. Furthermore, Our control signal is the speed of the motors, as there is a proportional control loop of the current of the motors, so the reward function need also to take into account the control signal to visit only the states that are significant for the control.

Error based reward

The first part of the reward function is the position error and the derivative of the position error:

$$r_{1,t} = -(\alpha_1 \|e_t\| + \alpha_2(\|e_t\| - \|e_{t-1}\|)) \tag{3.7}$$

where e_t is the position error at time t, α_1 and α_2 are constants that determine the importance of the position error and the derivative of the position error in the reward function.

Energy optimization reward

The second part of the reward function penalizes the energy consumed by the motors. In this reward function, we follow the classical approach of energy optimization: the energy consumed by the motors is penalized, and as the energy is related to the current, the current of the motors is penalized as well:

$$r_{2,t} = -\alpha_3 \|i_t\| \tag{3.8}$$

where i_t is the current of the motors at time t, α_3 is a constant that determine the importance of the current in the reward function.

Current limits reward

The third part of the reward function focuses on the current limits, as we limit the current using a saturation function after getting the action from the agent, we want to encourage the agent to choose actions that keep the current between the limits as other actions will be saturated, so there is no point in choosing them. To do so we use the following reward function:

$$r_{3,t} = -\alpha_4 \left\| a_i - u_{sat,i} \right\| \tag{3.9}$$

a_i is the action of the agent, $u_{sat,i}$ is the action after applying the saturation function, α_4 is a constant that determine the importance of this part in the reward function. If the action of the agent is not modified by the saturation (i.e. $a_i = u_{sat,i}$) this function is null.

So our final reward function is:

$$r_t = r_{1,t} + r_{2,t} + r_{3,t} \tag{3.10}$$

$$\begin{aligned} r_t = &- \alpha_1 \left\| e_t \right\| - \alpha_2 (\left\| e_t \right\| - \left\| e_{t-1} \right\|) - \alpha_3 \left\| i_t \right\| \\ &- \alpha_4 \left\| a_i - u_{sat,i} \right\| \end{aligned} \tag{3.11}$$

Where $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ are the coefficients of the reward function. By tuning these coefficients we change the importance of each part of the reward function until we get the best performance.

3.6 Hyperparameters

In the training process, many parameters need to be tuned to get the best performance. Those parameters could be classified into two categories: the parameters of the environment and the parameters of the training algorithm.
In the first class, are the coefficients of the reward function, the parameters of the target trajectories, the current limits and the speed limits of the motors. Most of these parameters are fixed after the characteristics of the robot and the objective of the training process. For example if the tracking of trajectories is done at low speed, the speed limits of the motors are set to low values, and target trajectories with low speed and acceleration are used for training.
In the second class, are the parameters of the training algorithm, called hyperparameters. The reason why they are called like that is to make the difference between them and the parameters of the network that are learned during the training process. The hyperparameters operate at at an upper level than the networks parameters so they are fixed before the training process. Some hyperparameters are best understood as fixed values like: the learning rate, the batch size, the number of training iterations, the number of episodes, the discount factor, the replay buffer, the number of hidden layers, the number of neurons in each layer, etc. Others involve strategic choices, like: the activation function, the optimizer, the exploration noise process, the structure of the networks, the use of batch normalization, the initialization of the last layer of the actor network and many others.
The choice of the hyperparameters is crucial for the training process, as they can affect significantly the performance of the agent [52]. However, tuning all these hyperparameters is not an easy task. First, the independence of the hyperparameters is not always verified, for example the learning rate and the batch size are correlated. Second, the effect of each hyperparameter on the training process is not always clear and could be affected by the choice of the seed of the random generator. Third, the training process is time-consuming, so it is not possible to test all the possible combinations of hyperparameters. Moreover, the algorithm implementation could have some bugs that affect the training process, so it is not always clear if the bad performance of the

agent is due to the choice of the hyperparameters or to the implementation of the algorithm. Many hyperparameter optimization techniques have been proposed to find the best one for the training process: the grid search where we test all the possible combinations of hyperparameters, the random search [53] where we test randomly some combinations of hyperparameters, and more advanced techniques like the Population-Based Training (PBT) [54] where the hyperparameters are adjusted during the training process. However, in this work, we rely on manual tuning for deeper understanding of how each hyperparameter affects the model behavior, to have more flexibility and adaptability to the specificities of the problem, and to avoid the complexity of the optimization process.

3.7 Environment and reward setup

During the training process, the agent interacts with the environment by sending the action a_i and receiving the tuple (s_i, a_i, r_i, s_{i+1}). All those variables take their values in very different ranges, requiring normalization (i.e. a data-conditioning function that maps the whole variable range to the interval [-1, 1]) for homogeneity. Normalization of the state and action is proved to be crucial element for good performance of the agent [55]. For position $X_t = (x_t, z_t)$, the normalization is done using the following equation:

$$x_t = [\frac{x_t - x_{min}}{max(x_{max} - x_{min}, z_{max} - z_{min})} - 0.5] \times 2 \tag{3.12}$$

$$z_t = [\frac{z_t - z_{min}}{max(x_{max} - x_{min}, z_{max} - z_{min})} - 0.5] \times 2 \tag{3.13}$$

Same for $X_{target,t}$ and the error $e_t = (X_{target,t} - X_t)/2$. For the current I_t, the normalization is done using the following equation:

$$I_t = [\frac{I_t - I_{min}}{I_{max} - I_{min}} - 0.5] \times 2 \tag{3.14}$$

The action is normalized using the following equation:

$$a_i = [\frac{a_i - a_{min}}{a_{max} - a_{min}} - 0.5] \times 2 \tag{3.15}$$

A truncated gaussian noise is added to the all the normalized states of the environment to encourage the exploration of the state space and the robustness of the agent, then clipping is applied to the states to keep them between -1 and 1.
In the reward function, we use the normalized terms of the state and the action, and each term of the reward function is normalized to be between -1 and 0. The final reward is the sum of the normalized terms of the reward function divided by the sum of the coefficients of the reward function. This normalization is done to ensure that the reward is between -1 and 0. A reward scale is used as hyperparameter during training to scale the reward to the desired range, as the reward is used to estimate the average return, and the scale of the reward could affect the training process. During the training process, we test the agent on the validation trajectory (no learning is done during the validation phase), this is done with a validation frequency hyperparameter, to avoid testing the agent at each episode, as it takes time. When the cumulative reward of the episode is greater than the threshold, the actor network is saved, and the threshold is increased. This is done to avoid overfitting (i.e. the agent learns to control the robot only for the training trajectories, capturing noise or irrelevant details, which leads to poor generalization

Table 3.2: Environment parameters

Parameter	Value
Position limits in x-axis	[0,6.8289] m
Position limits in z-axis	[0,2.623] m
Current limits	[0.3,6] A
Speed limits	[-50,50] rps
Reward scale	20
Noise standard deviation	0.00005
Noise truncation	0.0005
Time step	0.01 s
Episode length	1000
Validation frequency	50

on new trajectories) and to have the best policy for the task to achieve. The parameters of the environment are shown in table 3.2.

To find the best hyperparameters for the training process, we started by testing the hyperparameters used for the pendulum environment in the OpenAI gym, as the pendulum problem is the closest to our problem, with an adjustment on some hyperparameters to match the characteristics of our environment. The episode length is set to 1000 steps which is the equivalent of $10s$ the same as the pendulum environment. The learning rate is set lower than the one used for the pendulum environment as the CDPR environment is more complex and the action/state space is larger. More details about the hyperparameters used in the training process will be presented in the next chapter.

3.7.1 Validation trajectory during the training process

During the training process, we need to periodically test the agent, and save the best-performing network configuration. Indeed, during the learning, performance drop could be observed, due to noise or overfitting.
Validating the performance of the agent is a challenging task as the performance of the agent could be different from one trajectory to another. For instance, on a specific trajectory that happen to be slow, an agent that is good for low speed will perform better (but might be unable to manage high speeds). So to select the actor network that could operate with equal effectiveness on all the speed, a sinusoidal trajectory characterized by varying speed and acceleration, was employed. The generation of the trajectory is done using the following equations:

$$x_{target,t} = x_{target,0} + 0.5 \times (j/24) \times \sin(2\pi \times \frac{t}{T}) \tag{3.16}$$

$$z_{target,t} = z_{target,0} + 0.5 \times (j/24) \times \cos(2\pi \times \frac{t}{T}) \tag{3.17}$$

with j varying from 1 to 24, $T = 0.8s$ the period of the trajectory, and $x_{target,0}$ and $z_{target,0}$ are the initial position of the trajectory (the center of the workspace). The trajectory is generated for 24 periods with 80 steps per period.

3.8 Agents Configuration

Reproducing the results of reinforcement learning approach could be a challenging task. Even with the same environment and the same hyperparameters, the results could be different from one run to another. This is due to the randomness of the training process, the environment dynamics and the wide range of implementation of the algorithms [44]. The code is implemented using the Keras library, and the training process is done using Grid5000 servers. The training process is done using the following agents:

3.8.1 Deep Deterministic Policy Gradient (DDPG)

In section 1.4.3 we introduced the DDPG algorithm as presented in the original paper [16]. Whereas the fundamentals of the algorithm remain the same, there are many variant implementations of the algorithm. In this work, we use the implementation presented in [56], where the actor and critic networks are implemented using the Keras library. We adapt this implementation to our environment, the interaction between the different parts of the environment and the agent is shown in figure 3.4. The algorithm is presented in algorithm 4.

Algorithm 4: Implementation of the DDPG algorithm for CDPR control.

Initialize the critic network $Q(s, a)$ parameters, and replicate them onto the critic target network $Q'(s, a)$;
Initialize the parameters of the actor network $\mu(s)$ and replicate them to the actor target network $\mu'(s)$;
for *episode* $\leftarrow 1$ **to** *Maximum number of episodes* **do**
  Set initial state s ;
  Create a random trajectory;
  for *step* $\leftarrow 1$ **to** *Maximum number of steps* **do**
    Add a random noise N_t to $\mu(s_t)$ for exploration, $u_t = \mu(s_t) + N_t$;
    Calculate u_{sat} based on the equations (3.5, 3.6) to avoid current saturation;
    Apply the saturation function to u_t to get $u_{sat}(t)$;
    Implement $u_{sat}(t)$ on the robot and get the next state $s_{t+1} = (X_{t+1}, X_t, e_{t+1})$ and the reward r_t ;
    Save the transition (s_t, a_t, r_t, s_{t+1}) in the replay buffer;
    Randomly select a minibatch of transitions (s_i, a_i, r_i, s_{i+1}) from the replay buffer. ;
    Perform updates on the Q network to reduce training loss;
    Perform updates the μ network policy using the sampled policy gradient ;
    Gradually adjust the target networks' parameters with an update rate τ;
    if *the next position is out of the workspace limits* **then**
      End the current episode;
    end
  end
end

3.8.2 Proximal Policy Optimization (PPO)

The PPO algorithm is presented in section 1.4.4, we use the implementation presented in [57] with an adaptation for continuous action space, as the original implementation is for discrete action space. The algorithm is presented in algorithm 5.

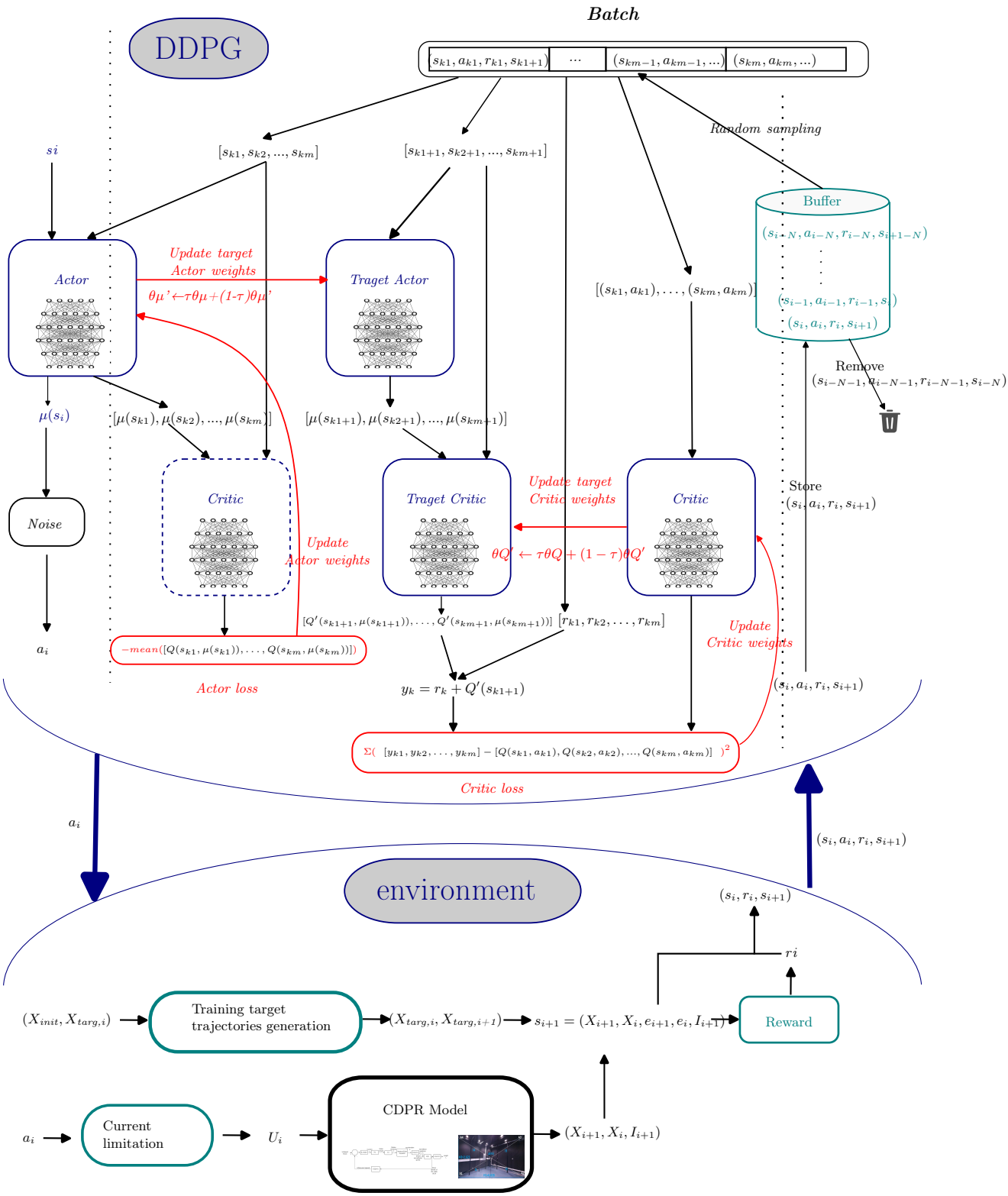


Figure 3.4: DDPG agent architecture, the agent interact with the environment by sending the action a_i and receiving the tuple (s_i, a_i, r_i, s_{i+1}), this tuple is stored in the replay buffer, then at each gradient step, the agent samples a batch from the replay buffer and updates the actor and the critic networks.

Algorithm 5: Implementation of the PPO algorithm for CDPR control.

Initialize the critic network $v(s)$ parameters and the parameters of the actor network $\mu(s)$ and the standard deviation σ;
for *episode* $\leftarrow 1$ **to** *Maximum number of episodes* **do**
  Set initial state s ;
  Create a random trajectory;
  for *step* $\leftarrow 1$ **to** *Maximum number of steps* **do**
    Sample an action a_t from a gaussian distribution with the mean $\mu(s_t)$ and the standard deviation σ;
    Calculate u_{sat} based on the equations (3.5, 3.6) to avoid current saturation;
    Apply the saturation function to u_t to get $u_{sat}(t)$;
    Implement $u_{sat}(t)$ on the robot and get the next state $s_{t+1} = (X_{t+1}, X_t, e_{t+1})$ and the reward r_t ;
    Save the transition (s_t, a_t, r_t, s_{t+1}) in a buffer;
    if *the next position is out of the workspace limits* **then**
      End the current episode;
    end
    if *the buffer is full* **then**
      Compute the advantage estimates $\hat{A}_t$ using the Generalized Advantage Estimation (GAE) ;
      Compute the returns R_t using the dicounted cumuliative sums of the rewards;
      for *iteration* $\leftarrow 1$ **to** *Number of training iterations* **do**
        Split the buffer into minibatches and shuffle them;
        for *each minibatch* **do**
          Perform updates on the μ network policy by maximizing the PPO objective ;
          Perform updates the v network by minimizing the mean squared error between the predicted returns and the computed returns ;
        end
      end
      Clear the buffer;
    end
  end
end

3.8.3 Soft Actor-Critic (SAC)

The SAC algorithm was elaborated in section 1.4.5, as we didn't find an implementation of the SAC algorithm in the Keras library, we decided to implement it from scratch. The algorithm is presented in algorithm 6 and the interaction between the different parts of the environment and the agent is shown in figure 3.5. The version of the SAC algorithm used excludes the dual Q-network, utilizes a target value network, and employs a fixed temperature coefficient.

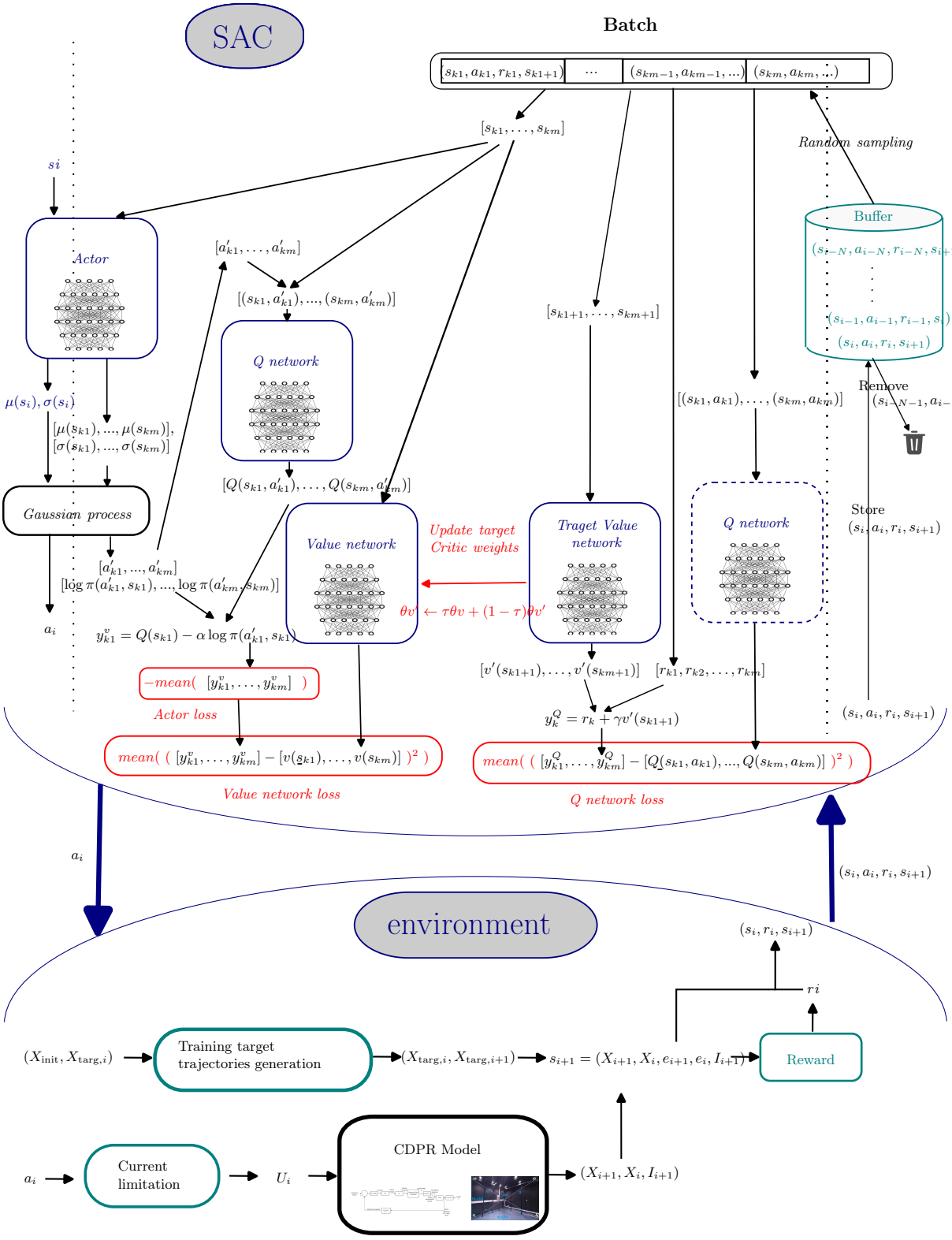


Figure 3.5: SAC agent architecture, the agent interact with the environment by sending the action a_i and receiving the tuple (s_i, a_i, r_i, s_{i+1}), this tuple is stored in the replay buffer, then at each gradient step, the agent samples a batch from the replay buffer and updates the actor and the critic networks.

Algorithm 6: Implementation of the SAC algorithm for CDPR control.

Initialize the Value network $v(s)$ parameters, and replicate them onto the critic target network $v'(s)$;
Initialize the Q network $Q(s, a)$ parameters;
Initialize the parameters of the actor network $\mu(s)$;
for *episode* $\leftarrow 1$ **to** *Maximum number of episodes* **do**
  Set initial state s ;
  Create a random trajectory;
  for *step* $\leftarrow 1$ **to** *Maximum number of steps* **do**
    Sample an action a_t from a gaussian distribution with the mean $\mu(s_t)$ and the standard deviation $\sigma(s_t)$;
    Calculate u_{sat} based on the equations (3.5, 3.6) to avoid current saturation;
    Apply the saturation function to u_t to get $u_{sat}(t)$;
    Implement $u_{sat}(t)$ on the robot and get the next state $s_{t+1} = (X_{t+1}, X_t, e_{t+1})$ and the reward r_t ;
    Save the transition (s_t, a_t, r_t, s_{t+1}) in the replay buffer;
    Randomly select a minibatch of transitions (s_i, a_i, r_i, s_{i+1}) from the replay buffer. ;
    Perform updates on the value network to reduce the value loss;
    Perform updates on the Q network to reduce the Q loss;
    Gradually adjust the target Q network's parameters with an update rate τ;
    Perform updates the μ network policy by maximizing the entropy-regularized expected return ;
    if *the next position is out of the workspace limits* **then**
      End the current episode;
    end
  end
end

3.9 Conclusion

In this chapter, a reinforcement learning framework for the control of cable-driven parallel robots has been developed. It is comprehensive reinforcement learning environnement for solving the trajectory tracking problem of CDPRs. The most crucial parts of this framework are the trajectory generation, the current constraints and the reward function. They are designed to ensure that the agent will learn to control the robot in all the states of the environment while avoiding actions that are not safe for the system. This is so important to ensure the transfer of the learned policy to the real robot. Along with the environment, the agents used in the training process are also elaborated in this chapter and their interaction with the environment is detailed in the algorithms. In the next chapter, this framework will be used to learn the optimal policy for the trajectory tracking problem of CDPRs. The training process will be explained in detail, and the experimental results on the real robot will be presented and discussed.

Chapter 4

Training process and experimental results

Contents

4.1 Introduction

The hyperparameters tuning is a time consuming process, as the training process could take hours or even days. To prevent the excessive use of the training resources, we started the tuning process by testing the hyperparameters used for the pendulum environment in the OpenAI gym, and we adjusted each hyperparameter to get the best performance and the best convergence speed. After getting the best optimal policy for each agent using the best hyperparameters, a validation process was conducted to evaluate the performance of the agent on some selected trajectories.
One of the main challenge of the reinforcement learning (RL) in robotics is the gap between the simulation and the real world. The optimal policy learned in the simulation does not always work in the real world, and the performance of the agent could be reduced significantly. To ensure the transferability of the agent from the simulation to the real robot, experiments on the real robot are conducted and a comparison between the performance of the reinforcement learning agent and the performance of the PID-based controller has been hold on many aspects like the tracking performance, the current limits respect, and the robustness to disturbances.

4.2 Training Process and Hyperparameters tuning

The training process is done using the reinforcement learning framework and the algorithms presented in the previous chapter. For each agent, a tuning process of the hyperparameters was conducted to get the best average return and the best convergence speed. The training process is done using Grid5000 servers [58] so the training time depends on the available resources and the nature of the experiment. For example, during the tuning of the hyperparameters, the experiment is ended when the average return of the agent is stable, while for the final training process, the experiment runs for a fixed number of episodes allowing the agent to learn the best policy for the task to achieve. In table 4.1, the training time per episode and per experiment are presented for each agent.
In this section the tuning process of each agent is presented concisely and the final hyperparameters used for the training process are shown. The tuning process of the hyperparameters is done manually to study the effect of each hyperparameter.

Table 4.1: Training Time

Agent	Training time per episode (seconds)	Number of episodes	Training time per experiment (hours)
DDPG /SAC	6-10	10000-30000	16-85
PPO	4.5-8	40000-80000	50-100

4.2.1 Deep Deterministic Policy Gradient (DDPG)

The final hyperparameters used in the training process are shown in table 4.2. As has been mentioned previously, the tuning process started by testing the hyperparameters used for the pendulum environment in the OpenAI gym and an adjustment of each hyperparameter was performed to get the best performance and convergence speed. The learning rate and the target update were reduced as our problem is more complex than the pendulum problem until the best

performance is achieved. The exploration noise deviation was also set to a low value 0.033 as the action space is large, this standard deviation associated with an interval of $[-1, 1]$ due to the normalization of the action space. The maximum desired speed of the motor is 50 *rps*, this is equivalent of $0.033 \times 50 = 1.65$ *rps* standard deviation for the desired speed of the motor which is acceptable to ensure a good tradeoff between exploration and exploitation. For the neural network architecture, two hidden layers were used as in the original paper. For the number of neurons in each layer, many values were tested, and the smallest size with the best performance has been selected. The batch size used is the largest value allowing the training process to be the fastest. For the buffer size and the discount factor, the same values used for the pendulum problem are kept. Finally the number of episodes is set between 10000 and 30000, as at this point the average return of the agent is stable and the agent has learned the best policy for the task to achieve.

Table 4.2: Hyperparameters

Hyperparameter	DDPG
Number of episodes	10000-30000
Critic learning rate	$4e^{-5}$
Actor learning rate	$2e^{-5}$
Update rate for target networks	5.10^{-4}
Discount factor for future rewards	0.99
Buffer size	50000
Batch size	128
Exploration noise deviation	0.033
Neural network hidden layers	2
Neural network hidden units	128

4.2.2 Proximal Policy Optimization (PPO)

The final hyperparameters used for the PPO algorithm are shown in table 4.3. The same approach of starting from hyperparameters used for the pendulum problem was conducted, and the neural network architecture was similar to DDPG for comparison purposes. After getting the best average return, the ultimate objective was to get the best convergence speed. The PPO policy is on-policy, so it suffers from sample inefficiency. Without an optimization of the hyperparameters, the training process could take 100 times more than off-policy algorithms like DDPG and SAC, which makes even more crucial the tuning process of the hyperparameters affecting the convergence speed.
The learning rate, the batch size, the number of training iterations and the number of episodes in an epoch are all hyperparameters that affect the convergence speed of the agent. Moreover, the correlation between them is not always clear. To find the best combination of these hyperparameters, many values based on the ones used for the pendulum problem were tested, and the best combination that ensures the best convergence speed while keeping good performance was selected. The initial standard deviation of the action plays an important role in the exploration of the action space. After trying many values the best value that ensures a good trade-off between exploration and exploitation was selected. The Generalized Advantage Estimation (GAE) and the discount factor were kept the same as the on used in the original paper.

Table 4.3: Hyperparameters

Hyperparameter	PPO
Number of episodes	40000-80000
Critic learning rate	$1e^{-5}$
Actor learning rate	$2e^{-5}$
Discount factor for future rewards	0.99
Generalized advantage estimation	0.95
Batch size	50
Initial log standard deviation	-1.3
Neural network hidden layers	2
Number of episodes in an epoch	10
Neural network hidden units	128
Number of training iterations	4

4.2.3 Soft Actor-Critic (SAC)

For the SAC algorithm, the final hyperparameters are shown in table 4.4. The learning rate used in the original paper was kept. For the other hyperparameters common with the DDPG algorithm like the buffer size, the batch size, the update rate for the target networks, the discount factor, the number of hidden layers, and the number of neurons in each layer, the number of episodes in the training process, the same values used for the DDPG algorithm were kept. Many values were tested for the alpha entropy coefficient, and the best value that ensures the best performance was selected.

Table 4.4: Hyperparameters

Hyperparameter	SAC
Number of episodes	10000-30000
Q learning rate	3.10^{-4}
State value learning rate	3.10^{-4}
Actor learning rate	3.10^{-4}
Update rate for target networks	5.10^{-4}
Discount factor for future rewards	0.99
Buffer size	50000
Batch size	128
Neural network hidden layers	2
Neural network hidden units	128
Alpha entropy coefficient	0.01

4.2.4 Rewards analysis

The objective of this section is to evaluate the influence of each part of the reward function on the training process. The reward function is a linear combination of four basic functions: the position error, the derivative of the position error, the energy optimization, and the current limits respect part. To evaluate the influence of each part of the reward function on the training process, we trained the agent with different combinations of the reward function parts.
The derivative of the position error proved to have a bad influence on the training process, as the performance of the agent was worse than without using this part, so it would not be used

in the final reward function. The energy optimization also affect the performance of the agent when it exceeds a certain value, so it should be used with caution. After testing many values, the best performance was achieved when the energy optimization part was used with a coefficient of $\alpha_3 = 0.001$ while $\alpha_1 = 1$. The current limits part proved to be the most challenging part of the reward function, as this part help the agent to have more smooth control but it negatively affects the performance of the agent on the tracking error level. Two options were decided to be used, one where this part of the reward was not used at all so the associated coefficient was set to zero, and one where the coefficient of this part of the reward was increased until the control signal was smooth. The value of the coefficient that ensures this behavior was $\alpha_4 = 0.5$.

4.3 Comparison between RL agents: Learning rate range and convergence time analysis

In this section, two aspects of the training process are compared. The first one concerns the tuning of the hyperparameters. As the learning rate is the common hyperparameter between the three algorithms, it could be considered as an indicator to compare the sensibility of the algorithms to the hyperparameters. Nevertheless, no general conclusion could be drawn from this comparison as the learning rate is not the only hyperparameter that affects the training process. The second aspect concerns the sample efficiency of the algorithms, as the DDPG and the SAC algorithms are off-policy algorithms, they are expected to be more sample efficient than the PPO algorithm. However, the objective is to develop certain idea about the difference in the convergence time between the three algorithms. Also the final average return of the agent is compared to evaluate the performance of the agent. The comparison is done using the hyperparameters presented in the previous section with the coefficient of the reward function set to $\alpha_1 = 1$, $\alpha_2 = 0$, $\alpha_3 = 0.001$, $\alpha_4 = 0.5$.

4.3.1 Learning rate range

To compare the sensibility of the algorithms to the learning rate, a training process was conducted for each algorithm with different values of the learning rate. The learning rate was varied by multiplying and dividing the initial learning rate by [2, 5, 10, 100]. The main reason why choosing these values instead of fixed values for all the agents is that during the tuning, one would multiply the learning rate by a fixed value to find the optimal one rather than adding a fixed value to it. The average final return in function of the learning rate is shown in figure 4.1. The learning rate is presented in logarithmic scale to have a better visualization of the results. Only the actor learning rate is presented as the critic learning rate is the same for the SAC algorithm, and is twice as high for the DDPG and the PPO algorithms. When the average return converges to a favorable value before diverging to an unfavorable one, the average return is considered to be the mean of the favorable and the unfavorable values. All the trainings are done using the same seed, and only one experiment per learning rate is conducted to avoid the excessive use of the training resources. So comparing the agents using the maximum average return is not relevant, as the average return could be different from one run to another could vary to 30% of the maximum average return.

In figure 4.1, the DDPG algorithm is the less sensitive to the learning rate, as the average return is more than -1000 for a wide range of learning rates, followed by the PPO algorithm,

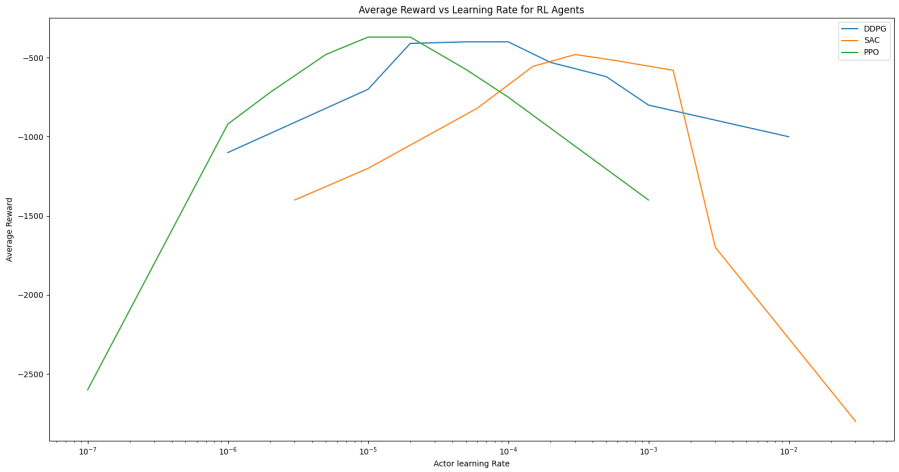


Figure 4.1: The average cumulative reward of the agent in function of the learning rate for the three algorithms: DDPG, PPO, and SAC, the learning rate is presented in logarithmic scale. An average reward around -600 is considered to be good performance.

and the SAC algorithm is the most sensitive to the learning rate. However the best performance by the SAC algorithm is achieved with a learning rate of 3.10^{-4}, the same value used in the original paper.

4.3.2 Convergence time analysis

To analyze the convergence time and the maximum average return of the agents, a training process using the hyperparameters presented in the previous section was conducted 5 times with different random seeds: [11936, 26994, 68248, 82364, 25982] for each agent. In figure 4.2, the average cumulative reward of the agent over the 5 experiments in function of the number of episodes is presented. The DDPG and the SAC algorithms have a similar convergence speed, they reach the best performance after 6000 episodes while the PPO algorithm takes more 40000 episodes to stabilize over the maximum average return. The variation of the average reward between the different experiments is more apparent for the SAC and DDPG algorithm than for the PPO algorithm. This could be explained by the fact that the PPO algorithm is an on-policy algorithm, so the training process is more stable than the off-policy algorithms. The maximum average return of the PPO algorithm is the highest among the three algorithms, followed by the DDPG algorithm, and the SAC algorithm has the lowest maximum average return. While the maximum average return could be a good indicator to compare the performance of the agents, the performance of the agent on the validation trajectory is more important as it is the final objective of the training process. It was noticed that that the best performing network on the validation trajectory is saved during the early episodes of the training. In the next section, the tracking performance of the agent on the validation trajectory is evaluated using the best saved actor network during the 5 experiments for each agent.

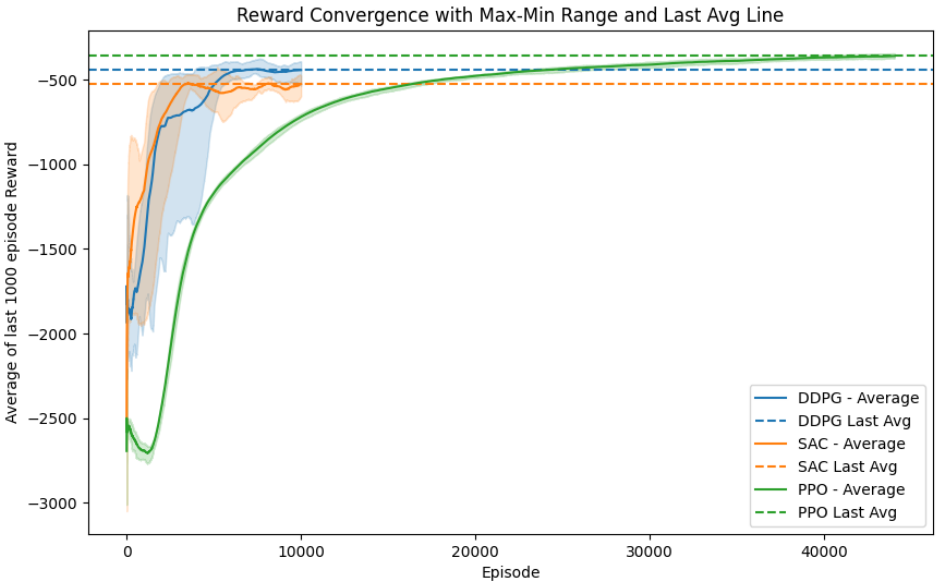


Figure 4.2: The average cumulative reward of the agent in function of the number of episodes for the three algorithms: DDPG, PPO, and SAC.

4.4 Performance evaluation and comparison between RL agents

4.4.1 Testing trajectories

To evaluate the performance of the RL agents and to compare it with the PID-based controller, three different trajectories are used:

- Trajectory 1: a cardioid trajectory with varying period, it is trajectory used on the real robot many times, so the safety is not compromised.

- Trajectory 2: an insect trajectory saved during a tracking experiment using the PID-based controller, it is characterized by stationary periods and an unpredictable change of speed and acceleration. As the trajectory is in 3D, and the robot can only move in 2D using the actual configuration, the trajectory is projected on the x-z plane.

- Trajectory 3: a sinusoidal trajectory with varying speed and acceleration, the same as the one used during the training process for validation, the only difference is that during tests on real robot we use $j_{max} = 14$ because the robot can not keep the cables taught at high speed and the maintenance of the cables is required in this case. The main advantage of this trajectory is that it allows evaluating the robot on many speeds and accelerations, and to test the limits of the control policies.

Only three trajectories are used to reduce the experimental time, and to simplify the comparison between the agents. In figure 4.3, the three trajectories in the x-z plane are presented, and also the variation of the pose in the x and z axis in function of the time is shown.

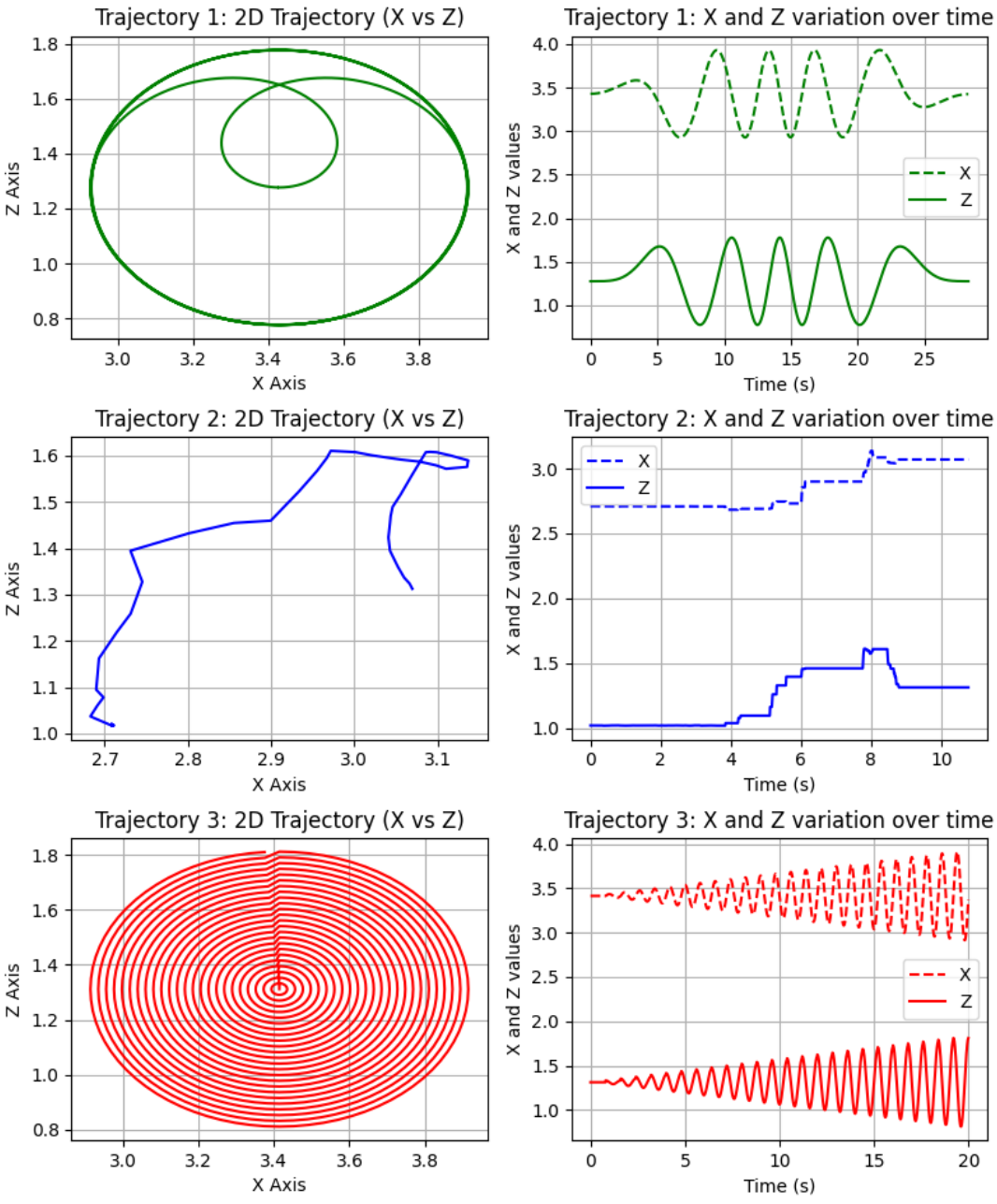


Figure 4.3: The three trajectories used for the evaluation of the agent: Trajectory 1: sinusoidal trajectory with varying period, Trajectory 2: insect trajectory, Trajectory 3: sinusoidal trajectory with varying speed and acceleration.

4.4.2 Policies used for the evaluation

Using the best hyperparameters for each agent, 5 experiments with diffrent seeds: [11936, 26994, 68248, 82364, 25982] were conducted (the same experiments used for convergence analysis). Also another 5 experiments were conducted for each agent using the same hyperparameters but without using the current limits respect part of the reward function $\alpha_4 = 0$. The PPO does not converge without using this part of the reward function, so no policy is used for this case in the evaluation phase. The best performing network on the validation trajectory is saved during the early episodes of the training process, and it is used in the next sections to evaluate the performance of the agent on the testing trajectories. The DDPG, SAC, and PPO agents learned using $\alpha_4 = 0.5$ for the current limits respect part of the reward function, and the DDPG1, SAC1 are the agents learned without using this part of the reward function.

4.4.3 Evaluation method

To evaluate the tracking performance of the agents, the tracking error is computed using the euclidean distance between the target pose and the actual pose of the robot. It is calculated for each step of the trajectory, and its violin plot representation is used to compare the performance of the agents.
To ensure the significance of this comparison we compute the p-value of the Mann Withney U test [59, 60], where the null hypothesis is rejected if the p-value is less than 0.05 which is equivalent of accepting the alternative hypothesis that the distribution underlying x is stochastically less than the distribution underlying y, i.e. $F(u) > G(u)$ for all u. F and G are the cumulative distribution functions of the two samples x and y. In this study, each sample is the tracking error of each controller on the testing trajectory, so we evaluate the comparison of each pair of controllers using this test. The "less" and the "greater" refer to the direction of the inequality in the alternative hypothesis. They are complementary to each other, so if the p-value is greater than 0.95 the null hypothesis could be accepted. Which is equivalent of the x is stochastically greater than the distribution underlying y, i.e. $F(u) < G(u)$ for all u. The results of the test are presented in the violin plot representation of the tracking error of the agents on the testing trajectories. $-^{***}$ means that the p-value is less than 0.001, so the PID controller is stochastically less than the RL agent, while $+^{***}$ means that the p-value for the greater test is less than 0.001, so the RL agent is stochastically less than the PID controller. ** means that the p-value is less than 0.01, and * means that the p-value is less than 0.05.

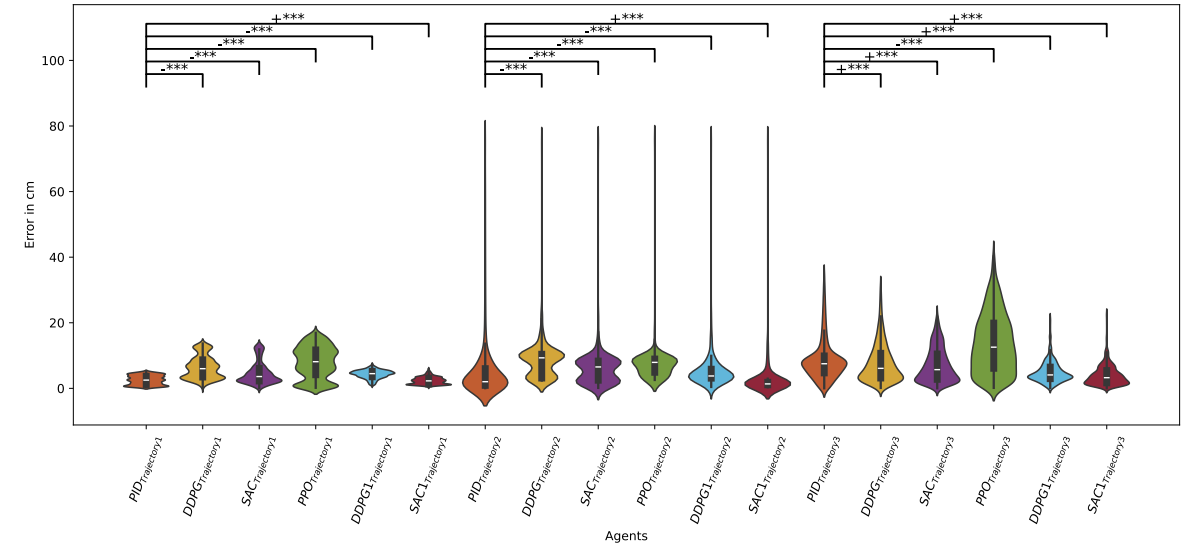
4.4.4 Tracking performance on simulation: Tracking error and current optimization

In figure 4.4, the tracking errors and the currents of the 5 RL policies along with the PID-based controller on validation trajectories are presented. The the SAC1 has the best performance over all validation trajectories, while the PPO shows the worst performance. The DDPG1 and the SAC1 have a good performance on the tracking error level, but in the current plot, we can see that their current values vary between the current limits because the current limits respect part of the reward function is not used during the training process. Applying such policy on the real robot could lead to noisy and unexpected behavior.
The DDPG and the SAC policies that have smooth signal outperform the PID controller on trajectory 3, while the PID controller has the best performance on trajectory 1 and 2. This is due to the speed of the target trajectories and the acceleration, the PID controller has a better performance on the trajectories with low speed and acceleration, while it is the opposite for the

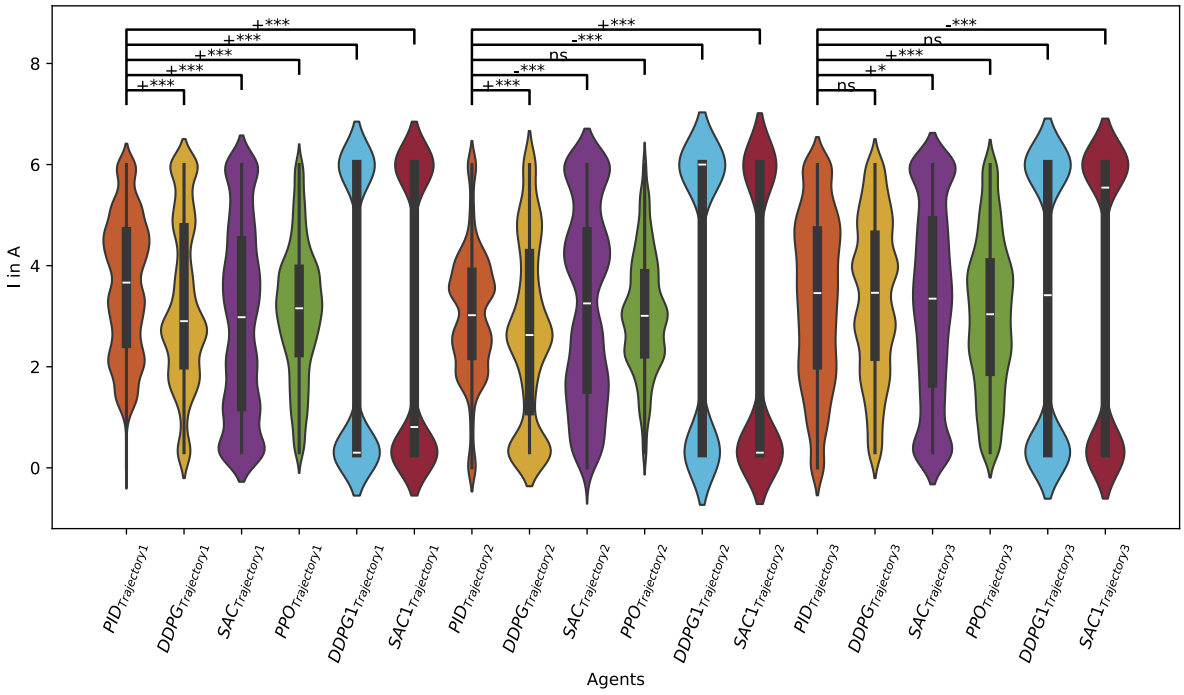
RL policies.
For the current optimization part, it is hard to evaluate the performance of the agents, specially that it is not relevant to the real robot results as the current depends on the initial conditions of the robot, and the model of the robot is not perfect. The only conclusion that could be drawn from this plot is that the RL agents with smooth current have a better performance on the current optimization than the PID controller most of the time, so the current optimization part of the reward function has a positive effect on the current optimization.

4.4.5 Summary of the results

To summarize, the PPO agent suffers from a high tracking error on the testing trajectories, while the DDPG1 and the SAC1 have a good performance on most of the trajectories comparable to the PID-based controller. The only problem with those agents is the smoothness of the control signal, which could affect the performance of the agent on the real robot The DDPG and the SAC have a good performance on trajectory 3 but their performance is compromised on low speed trajectories. To improve the performance of those agents, more tuning of the hyperparameters and the variables of the training will be performed and due to the high computational cost of the training process, the additional tuning will be done only for The DDPG agent as it has a good performance, a good convergence speed, and more stable results than the SAC implementation used in this study. But before that, the performance of the agents on the real robot will be evaluated to ensure the transferability of all the agents from the simulation to the real world.



(a) Tracking error comparison



(b) Current comparison

Figure 4.4: Simulation results: Comparison of the tracking error and current of the agent on validation trajectories for the three algorithms: DDPG, PPO, and SAC. The DDPG, SAC, and PPO are the agents learned using $\alpha_4 = 0.5$ for the current limits respect part of the reward function, and the DDPG1, SAC1 are the agents learned without using this part of the reward function.

4.4.6 Tracking performance on real robot

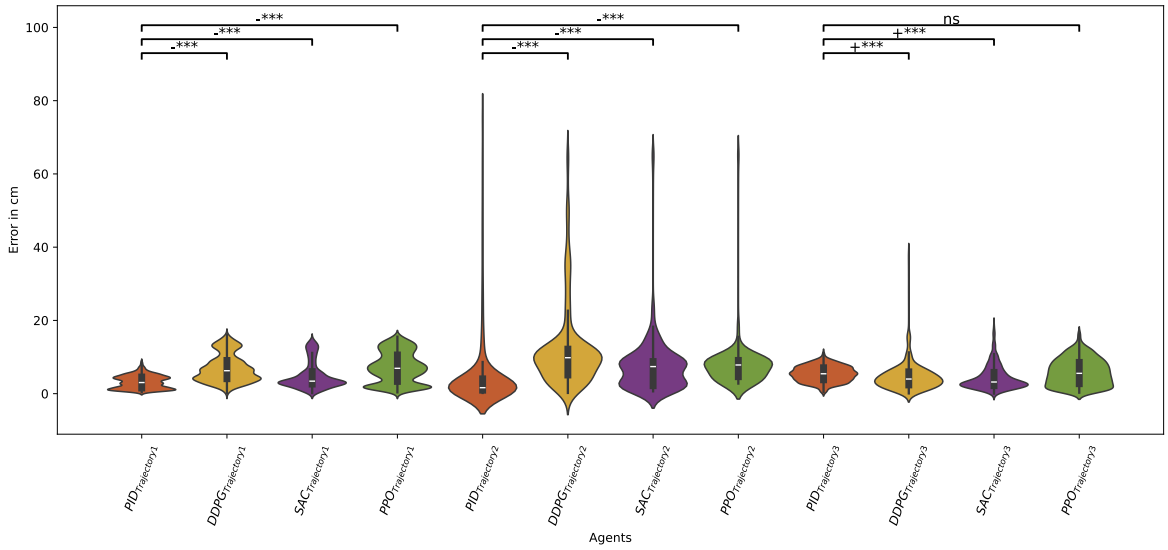
During the real robot experiments, the policies DDPG1 and SAC1 could not be used on the real robot, the control signal is so noisy that the robot could not start at all even with a filter. So the least mean current of the control signal is a crucial aspect to ensure the transferability of the agent from the simulation to the real world. The three other policies and the PID-based controller were tested on the real robot using the same trajectories used for the simulation except for the third trajectory where the speed and the acceleration were reduced to ensure the safety of the robot.
For trajectory 3, the main objective of testing it is to evaluate the performance of the agent on high speeds and accelerations. When applying this trajectory on the real robot, the cables become slack and the controller could not keep the cables taught. This requires immediate stop of the robot otherwise the cables become entangled and the maintenance of the cables is required. So we only tested the validation trajectory for $j_{max} = 12$ to ensure the safety of the robot.
In figure 4.5, we can see that the all the controllers have a tracking error close to the simulation over all trajectories. This could be explained by the fact that the simulation is closer to the real robot, the PID still outperforms all the RL controllers on the first and second trajectory which align with the simulation results. For the third trajectory, the RL controllers: DDPG and SAC have smaller error compared to the PID controller while the comparison is not significant for the PPO. The current of the PID-based controller is the highest in all the trajectories with mean value of 1.5 A while the RL controllers have a similar low mean of the current 1 A.

Summary of the results

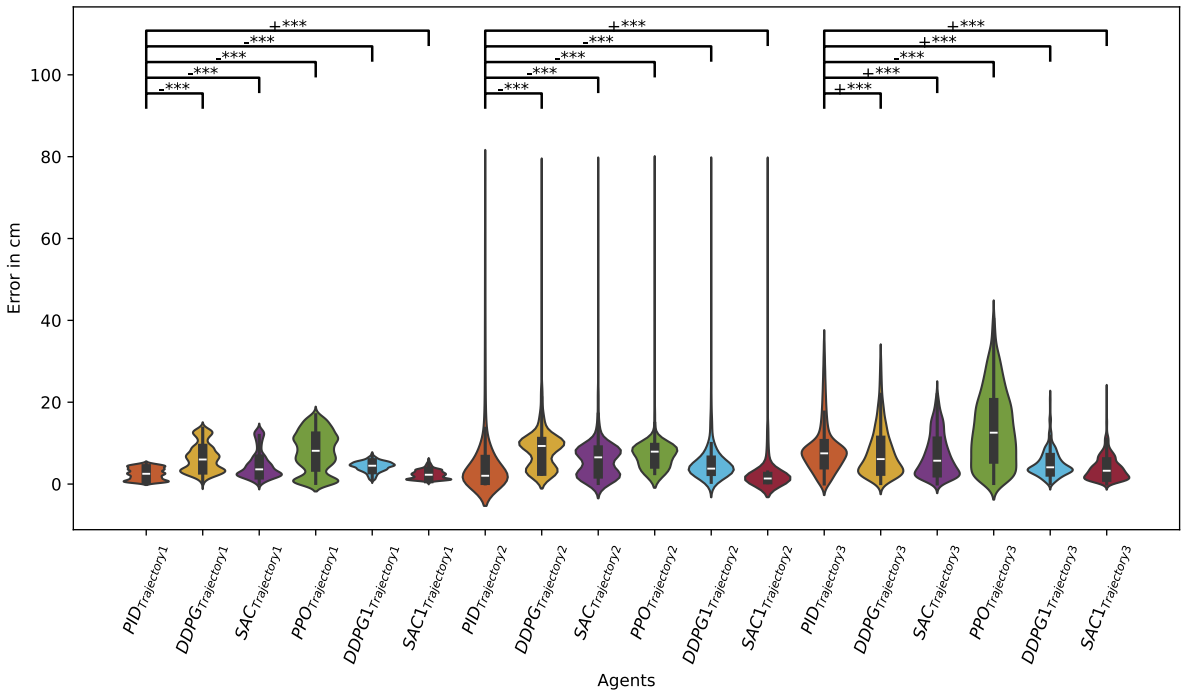
In this section we tested the transferability of the RL policies from the simulation to the real robot, the results are promising as the RL controllers have a similar performance on the real robot compared to the simulation. However, their performance is still lower than the PID-based controller for trajectory 2: the insect trajectory. The main reason for this is the static error that the DDPG and the SAC suffer from. The current of the PID-based controller is the highest in all the trajectories, while the RL controllers have a similar mean of the current. To conduct a more detailed comparison between the RL controllers and the PID-based controller, we decided to improve the performance of the DDPG agent by tuning the hyperparameters and the variables of the training process. Then, we will compare the performance of the best RL agent with the PID-based controller on the real robot using the same trajectories used in this section, on many other aspects.

4.4.7 Optimal policy improvement for the DDPG agent and comparison with the PID-based controller

The DDPG agent performance has been compromised by the static error and low performance on low speed trajectories compared to high speed one. To improve the performance of the agent, a tuning of the hyperparameters and the variables of the training process has been conducted: after using more neurons in the hidden layers (512 instead of 128), the best performance was achieved. Although the results were not consistent between the different runs (different seeds) using the same hyperparameters, this could be explained by the fact that the DDPG algorithm is an off-policy algorithm, so the training process is unstable. The best performing network on the validation trajectory was saved during the early episodes of the training, and the performance of the agent on the testing trajectories is evaluated using this network. In simulation, the DDPG agent have closer performance compared to the PID-based controller on the testing trajectories,



(a) Tracking error comparison



(b) Current comparison

Figure 4.5: Experimental results: Comparison of tracking error and current for the agent on testing trajectories for DDPG, PPO, and SAC.

but we will only focus on the results on the real robot. The comparison between the best DDPG agent and the PID-based controller on the testing trajectories will be conducted on many aspects: the tracking performance, the current optimization and the robustness to disturbances.

4.4.8 Comparison between best RL agent and PID-based controller on real robot

Tracking performance and current optimization

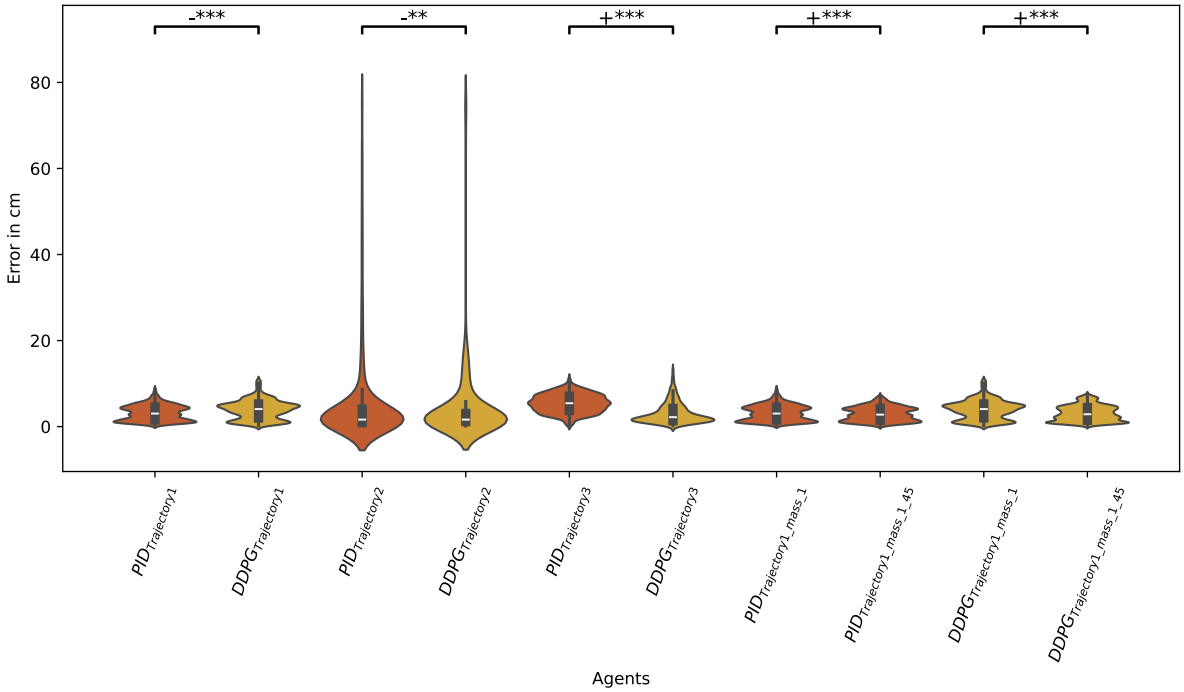
After improving the performance of the DDPG agent, the DDPG policy is compared with the PID-based controller on the real robot using the same testing trajectories used in the previous section. The results are presented in figure 4.6, the PID still outperforms the DDPG agent on the first and second trajectory, while the DDPG agent has a better performance on the third trajectory. But we can see a good improvement of the DDPG agent compared to the previous results. The difference between the mean of errors is reduced to only 1.5 cm for the first trajectory and similar as the PID for the second trajectory while on third trajectory the difference is still significant. The current of the PID-based controller is still the highest in all the trajectories with mean value of 1.5 A while the DDPG agent has a similar low mean of the current 1 A. While the mann whitney U test still shows that the PID controller is stochastically less than the DDPG agent on the first and second trajectory, the difference in the tracking error mean is not significant. So the RL agent performance could be considered as close to the PID-based controller if not better after rigorous tuning of the hyperparameters and the variables of the training process.

Robustness to mass change

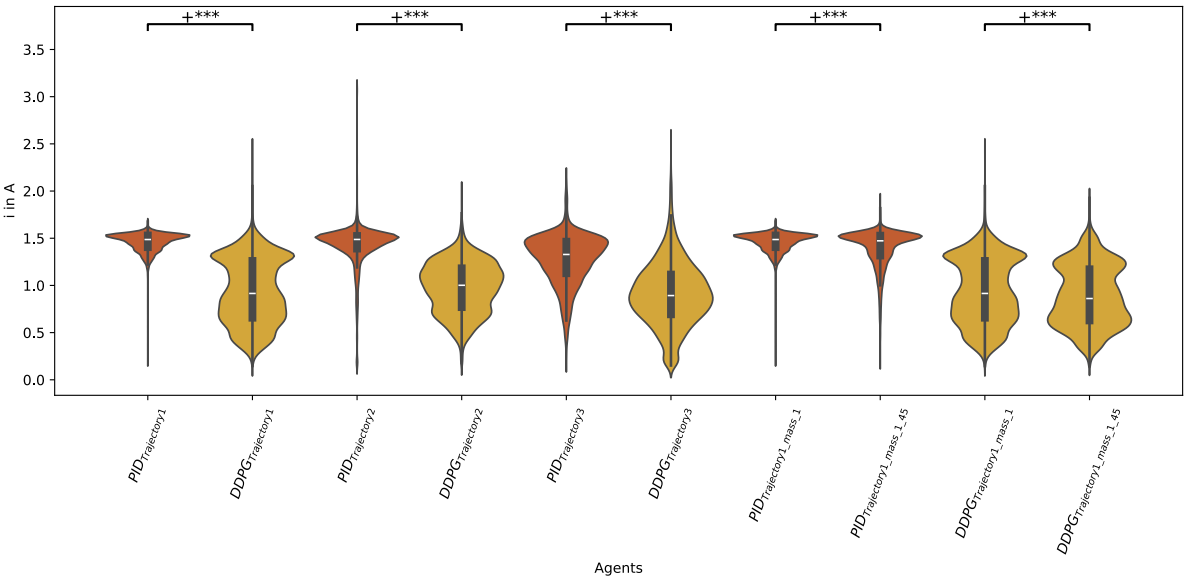
The robustness to mass change is evaluated by changing the mass of the end-effector of the robot, the mass is increased by 45%, 1 kg to 1.45 kg. The results are presented in figure 4.6, both the DDPG agent and the PID-based controller keep a good performance on trajectory 1 with the mass change, even the error is reduced after mass change, this could be explained by the fact that the mass adds more tension on the cables which help the agent keep lower tracking error.

Robustness to noisy target pose

The target trajectory is a fixed pose in the x-z plane perturbed by adding some noise. This noise is applied exclusively to the x-component of the position vector, while the y and z components remain constant. It is introduced using a random variable, where each value is generated using a uniform distribution within the range [-0.1, 0.1]. the value of this noise is updated each 10 time steps. The value of the target x-coordinate fluctuates around a base value of 3.4, with slight deviations introduced by the noise, resulting in a range of values between [3.3, 3.5]. Furthermore, the noise is held constant over intervals of 10 time steps. The PID controller failed to pass this test. As the disturbance is added to the desired position, the PID controller tries to compensate it by increasing the current which lead to current saturation and cables breakage. In Fig 4.7, we can see that the AI based control method is robust to this kind of disturbance.



(a) Tracking error comparison



(b) Current comparison

Figure 4.6: Experimental results: Comparison of tracking error and current between the DDPG agent and PID controller on three trajectories, including robustness to mass change.

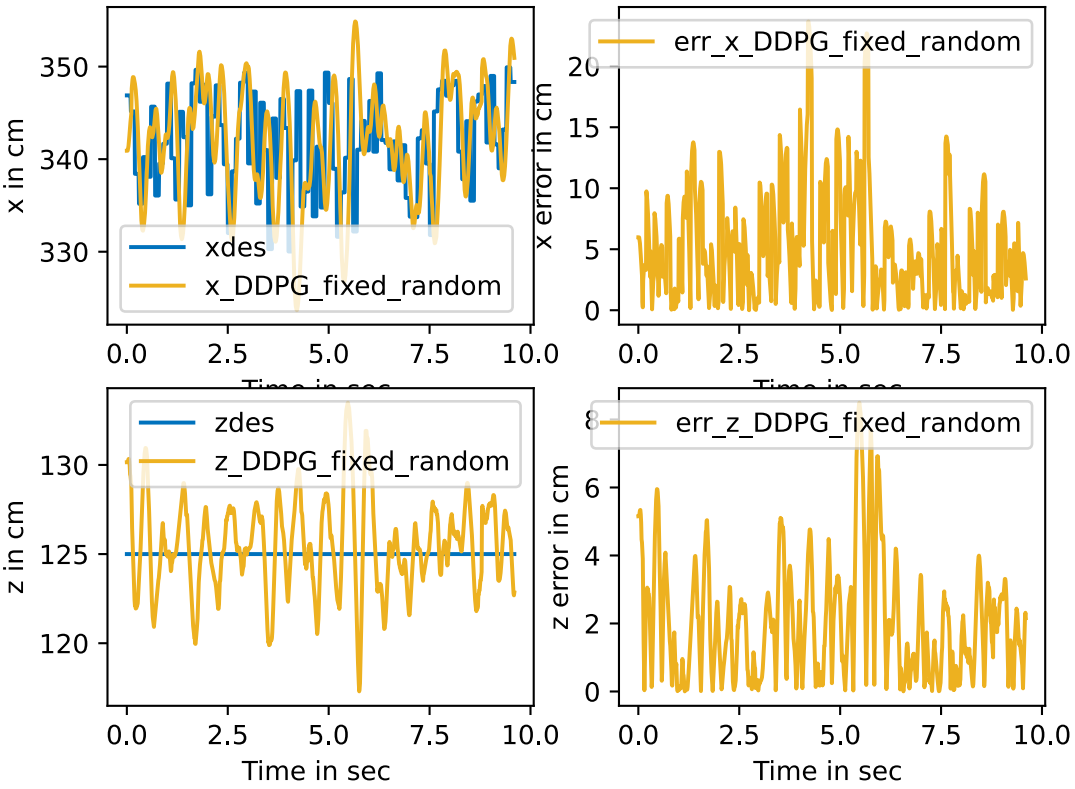


Figure 4.7: Experimental results: the x and the z components of the pose of the end-effector and the tracking error of the agent on the trajectory 1 when the x component of the target pose is perturbed by a noise for the DDPG agent only as the PID-based controller failed to pass this test.

4.5 Conclusion

In this study, the performance of the RL agents after successful training on the simulation environment was evaluated on the real robot. The transferability of the policies learned using the three agents: DDPG, PPO, and SAC is validated, the only condition for this transferability is the smoothness of the control signal. While the results of the RL agents were promising, they suffered from a static error that compromised their performance on the real robot on stationary periods. To overcome this problem, the DDPG agent was improved by more tuning of the hyperparameters. The results of the improved DDPG agent are close to the PID-based controller on simulation and on the real robot. Moreover the static error is reduced to only $2cm$ in the improved version of DDPG compared to $5cm$ in the previous version. The DDPG agent has a better performance than the PID-based controller on the third trajectory, and the current of the DDPG agent is lower than the PID-based controller in all the trajectories. The DDPG agent is also robust to the mass change and the noisy target pose, while the PID-based controller failed to pass the noisy target pose test. The ultimate objective of this study is to develop a control

policy that could be used for insect tracking, and the results of the DDPG agent are promising. The insect trajectory projected on the x-z plane was validated successfully on the real robot. The next step is to study the generalization of the proposed reinforcement learning approach for diffrent configurations of the robot.

Chapter 5

Toward n-Cable Driven Parallel Robots: A Generalized Approach

Contents

5.1 Introduction

In the last section, we have shown that the reinforcement learning controller can be used to control a CDPR, the learned policy have similar performance compared to the PID-based controller and can be used to control the robot in a real environment. The main drawback of this approach is that the learned policy is specific to the robot. While it proved to be robust to small changes in the robot parameters, it can't be used to control a different configuration of the robot. As it has been presented in the state of the art of controlling CDPRs, all the proposed end-to-end reinforcement learning approaches [41, 42, 43] suffer from the same issue, they learn a policy that outputs a vector of torques to be applied to the motors, so the number of elements in the output vector should be equal to the number of actuators on the robot. Changing the configuration of the robot will require retraining the controller from scratch. This could be described as a lack of generalization of the learned policy. Many approaches have been proposed in the literature to address the latter issue, such as domain randomization [29] and transfer learning [61], however these approaches are not effective when the input space is changing. Another innovative approach,"Gato", to address the generalization issue has been proposed by [62], "The same network with the same weights can play Atari, caption images, chat, stack blocks with a real robot arm and much more, deciding based on its context whether to output text, joint torques, button presses, or other tokens." While the Gato agent is a promising approach to address the generalization issue, the large size of the neural network required to implement the agent and the necessity of fine-tuning the agent for some tasks make it impractical for many applications. Also Gato only reaches expert level on 180 simulated control tasks out of 604. Addressing the generalization issue on large scale control tasks as the one presented by Gato could be the ultimate goal for the robotics community, however, in the meantime, we could focus on developing more efficient and practical approaches to address the generalization issue on some robotics classes, such as Cable driven Parallel Robots (CDPRs). In this chapter a new reinforcement learning approach to control the robot will be proposed, it will allow the learned policy to generalize to environments with different number of actuators without any retraining, this will be compared to the one presented in the last section, and the results will be presented.

5.2 Conventional reinforcement learning controller

In the last chapter, the algorithms: DDPG [16], PPO [19], and SAC [23], have been used to learn a policy that maps the state of the robot to the torques to be applied to the motors. To control CDPRs with different number of actuators, a retraining from scratch is required, as the number of elements in the output vector should be equal to the number of actuators on the robot. This is not the case for classical control methods such as the PID-based controller presented in 2.5 [1], normally the passage from one configuration to an other require only a change in the jacobian matrix, and maybe a change in the PID gains, but no need to design the control law from scratch. In the next section, a new approach to control CDPRs with different number of actuators will be presented, it will allow the learned policy to generalize to environments with different number of actuators without any retraining, but first we will underline the interaction between the agent and the environment in the conventional reinforcement learning so that we can compare it with the new approach.
In conventional reinforcement learning, the state is describing the whole robot (figure 5.1). We defined it as follows:

$$s_i = (X_i, X_{i-1}, e_i, e_{i-1}, I_i) \tag{5.1}$$

where X_i is the position of the end effector at step i, e_i is the error between the current position and the target position at step i, and I_i is the current of the motors at step i.
The observations and the actions are normalized to be in the range $[-1, 1]$. The agent takes the tuple (s_i, a_i, r_i, s_{i+1}) as input and outputs the action a_i to be applied to the motors where $a_i = (u_1, u_2, ..., u_n)$ is the vector of torques to be applied to the motors. To compare our new

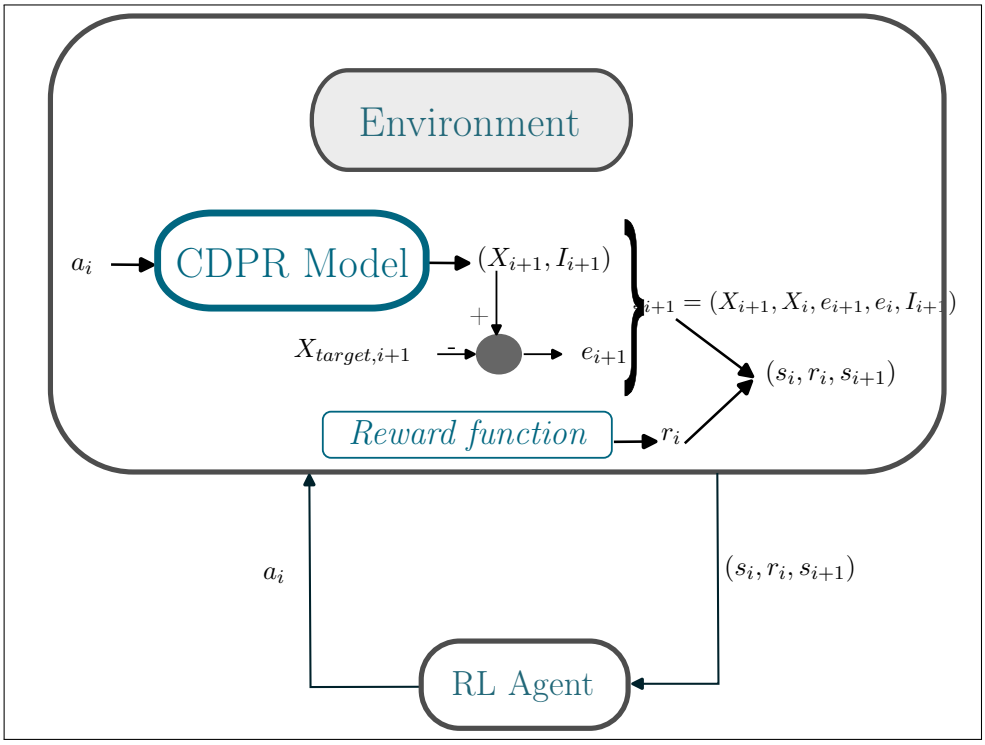


Figure 5.1: Interaction between the agent and the reinforcement learning environment, the agent takes the tuple (s_i, a_i, r_i, s_{i+1}) as input and outputs the action a_i to be applied to the motors.

approach with the conventional reinforcement learning, we will retrain the controller using the same environment and the same hyperparameters, the only difference is that the configuration of the robot will be changed, and we will compare the performance of the two controllers on the new configuration.

5.3 Actuator level policies: Multi agent reinforcement learning controller

The main idea of the actuator level policy method is to learn a policy that maps the state of each actuator to the torque to be applied to the motor, instead of learning a policy that maps the state of the robot to the torques to be applied to the motors. Since each actuator is individually controlled by a single agent, we refer to this approach as multi-agent reinforcement learning. However, it is important to note that the agent learns a single policy, which is then applied uniformly across all actuators The main advantage of this approach is that the learned policy can generalize to environments with different number of actuators, as the policy is learned to control the actuator and not the robot. The agent-environment interaction for the multi agent

reinforcement learning is shown in figure 5.2. The state of the actuator is defined as follows:

$$s_i^j = (L_{i-3}^j, L_{i-2}^j, L_{i-1}^j, L_i^j, L_{i,target}^j, I_i^j) \tag{5.2}$$

where L_i^j is the length of the cable j at step i, $L_{i,target}^j$ is the target length of the cable j at step i, and I_i^j is the current of the motor j at step i. We include the length of the cable at step $i-3$ in the state to give the agent more information about the dynamics of the cable, the tension applied in the other end of the cable, and the speed of the cable. We believe that this information is important for the agent to learn a good policy. The observations and the actions are normalized to be in the range $[-1, 1]$. As we have n actuators on the robot, at each step i, the agent will do n different interactions with the environment, but learn only one policy, so we only have one neural network that maps the state of the actuator to the torque to be applied to the motor and one agent that learns this policy from the different samples collected from the n actuators.

5.4 Proof of concept

To demonstrate the feasibility of our approach, we will use the dynamic model of one cable.

Assumptions

The following assumptions are made in deriving the dynamic model:

- The cable is inextensible.
- The cable is wound around the winch without slipping.
- There is no dry friction or viscous friction acting on the system
- The winch apply a torque $\tau(t)$ which is converted to a tension $T(t)$ applied to one end of the cable which is proportional to the current $I(t)$ flowing through the motor.
- To the other end of the cable, the mobile platform apply a force $F_{\text{ext}}(t)$.
- The length of the cable between the winch and the end effector is denoted as $L(t)$.
- The mass of the cable between the winch and the end effector is denoted as $m(t)$.

Dynamic model of the cable

Applying Newton's second law to the cable, we have:

$$m(t)\ddot{L}(t) = F_{\text{ext}}(t) - T(t) \tag{5.3}$$

$F_{\text{ext}}(t)$ and $m(t)$ estimation

The cable is supposed to be inextensible and uniform in mass, so an estimation of $m(t)$ could be computed using the length of the cable and the mass per unit length of the cable . We can estimate $F_{\text{ext}}(t)$ based on the equation 5.3 using the estimated value of $m(t)$ and the measured value of $T(t)$ as it is proportional to the current $I(t)$ flowing through the motor and the measured values of $L(t)$ during the last time steps.

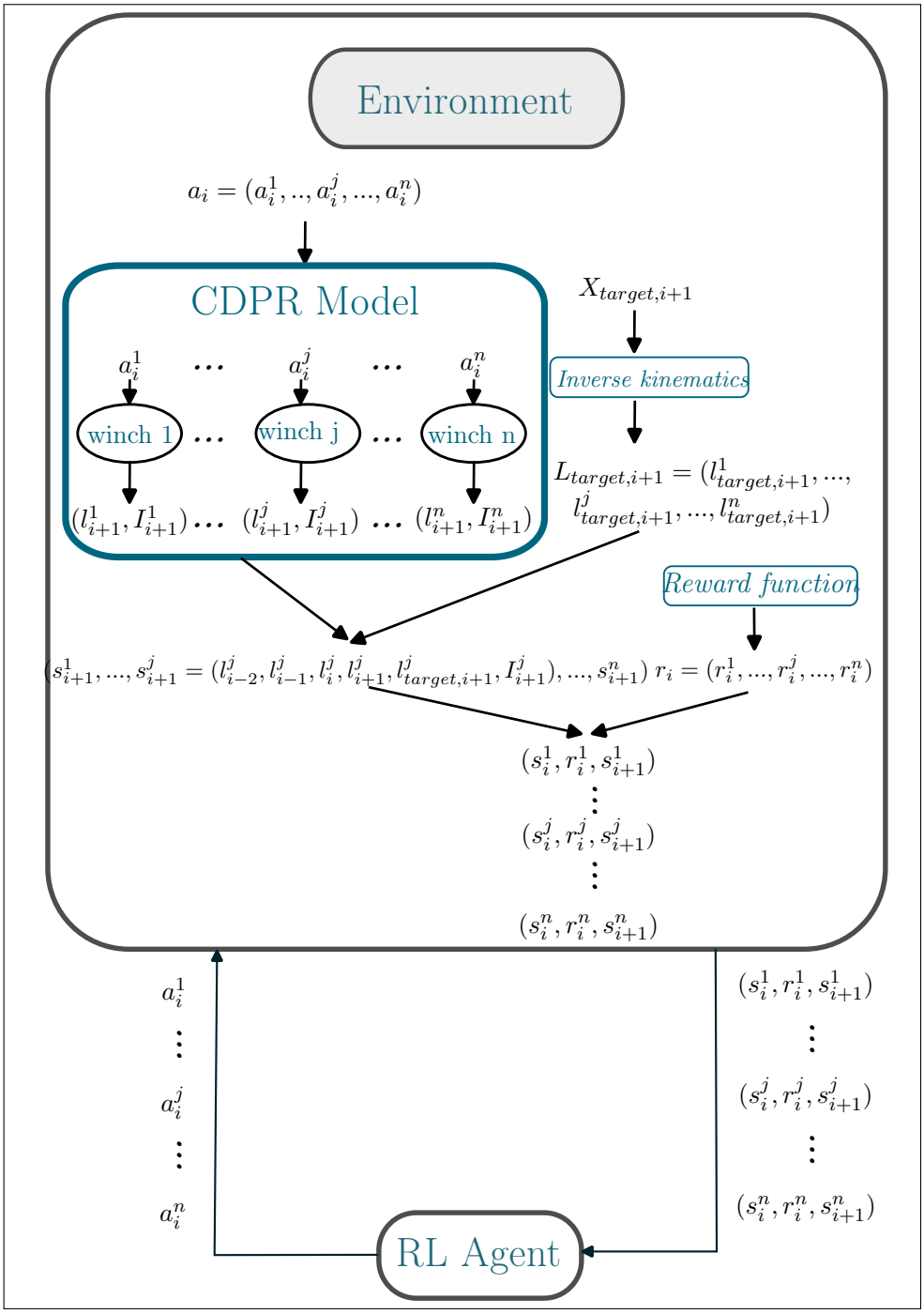


Figure 5.2: Interaction between the agent and the environment for the multi agent reinforcement learning, at each step i, the agent interacts with the environment n times, one for each actuator, from each interaction the agent collects the tuple $(s_i^j, a_i^j, r_i^j, s_{i+1}^j)$, and output the action a_i^j to be applied to the motor j., the agent learns one policy that maps the state of an actuator to the torques to be applied to the motor.

Existence of a control law

We know an estimate of F_{ext} denoted by $\hat{F}_{\text{ext}}$ and of m denoted by $\hat{m}$, with:

$$\begin{aligned} \hat{F}_{\text{ext}}(t) &= F_{\text{ext}}(t) - \epsilon_{\text{ext}}(t) \quad \text{bounded by a small term} \\ \hat{m}(t) &= m(t) - \epsilon_m(t) \quad \text{bounded by a small term} \end{aligned} \tag{5.4}$$

The objective is to perform trajectory tracking with $L_d(t)$ as the desired length, i.e., how to choose $T(t)$ to ensure:

$$\begin{aligned} &\|L_d(t) - L(t)\| = \|e(t)\| \leq \epsilon_d \\ &\text{(small or zero when } \epsilon_{\text{ext}}(t) \text{ and } \epsilon_m \text{ are zero)} \end{aligned} \tag{5.5}$$

$L(t)$ is assumed to be bounded (as well as its derivatives), which is verified for the cable robot.

If we choose $T(t)$ as follows:

$$T(t) = \hat{F}_{\text{ext}} - \hat{m}\left(\ddot{L}_d(t) + k_v\dot{e}(t) + k_p e(t)\right) \tag{5.6}$$

Equation 5.3 becomes:

$$F_{\text{ext}} - \hat{F}_{\text{ext}} + \hat{m}\left(\ddot{L}_d(t) + k_v\dot{e}(t) + k_p e(t)\right) = m\ddot{L}(t) \tag{5.7}$$

$$\text{So: } \epsilon_{\text{ext}}(t) + \hat{m}\left(\ddot{e}(t) + k_v\dot{e}(t) + k_p e(t)\right) = \epsilon_m\ddot{L}(t) \tag{5.8}$$

$$\text{And: } \ddot{e}(t) + k_v\dot{e}(t) + k_p e(t) = \epsilon(t) \tag{5.9}$$

$$\text{With: } \epsilon(t) = \frac{\epsilon_m\ddot{L}(t) - \epsilon_{\text{ext}}(t)}{\hat{m}} \tag{5.10}$$

We thus obtain a second-order equation with an arbitrary choice of $k_v > 0$ and $k_p > 0$ to ensure condition 5.5.

The choice of $k_v > 0$ and $k_p > 0$ can also be formulated as a convex problem solvable by numerical techniques.

So this control law could be learned by a neural network using the reinforcement learning agent, the only condition is that the state of the actuator should contain the length of the cable at the previous time steps to estimate the speed of the cable and the tension applied in the other end of the cable, and the current of the motor to estimate the force applied to the cable.

5.5 Reward function

There is two possibilities to design the reward function for the multi agent reinforcement learning, the first one is to design a reward function that is based on the error in the position of the end effector, in this case the reward function will be similar to the one presented in the chapter 3.5.1. The only difference is that the same error is given to the n actuators at each step i. In this case it has a collaborative aspect as all the actuators gets the same reward, so an interesting error between the actual cable length and the target cable length on one actuator will be penalized by all the actuators as the reward is based on the position error. We will refer to this reward function in the results part as the reward 1 "r_1". we can also use a reward function that is based on the error of the actuator only, in this case the reward function will be:

$$\begin{aligned} r_i^j = &-\alpha_1\left|e_i^j\right| - \alpha_2(\left|e_i^j\right| - \left|e_{i-1}^j\right|) - \alpha_3\left|I_i^j\right|^2 \\ &- \alpha_4\left|a_i^j - u_{sat,i}^j\right| \end{aligned} \tag{5.11}$$

where e_i^j is the error between the current length of the cable j and the target length of the cable j at step i, I_i^j is the current of the motor j at step i, a_i^j is the action of the agent j at step i, $u_{sat,i}^j$ is the action after applying the saturation function to the action of the agent j at step i. In the training phase, we kept the same values for the coefficients $\alpha_1, \alpha_2, \alpha_3, \alpha_4$. We will refer to this reward function in the results part as the reward 2 "r_2".

5.6 Results

5.6.1 validation trajectories

To validate the actuator level policy, we used the similar trajectories as the one used in the last chapter, the difference is that the trajectories are now used in 3D space. So, the insect trajectory is not projected on the plane. And the third trajectory, the sinusoidal one, the Y coordinates for the target position is generated using sinusoidal function with low frequency, so the robot will have to move in 3D space to track the target position.

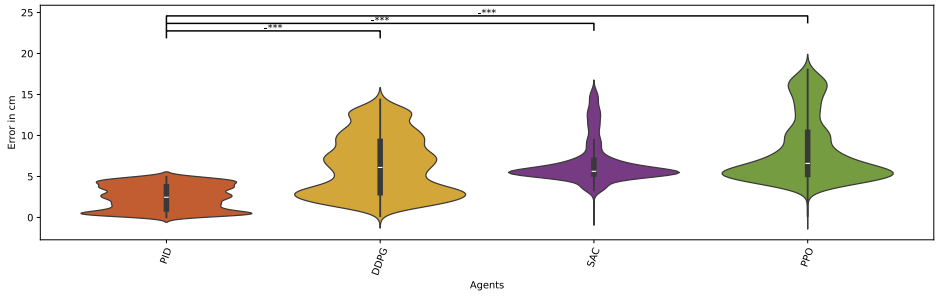
5.6.2 Traditional reinforcement learning controller

To train the controller using the conventional reinforcement learning, we used the same environment and the same hyperparameters as the one used in the last chapter, the only difference is that the configuration of the robot has changed, so as the state space and the action space of the agent. We trained for 5 experiments for each configuration and we used the same reward function with the same coefficients. The results are presented in figure 5.3, the three agents converge and succeed to achieve working policy, however the current is so noisy, so the policies could not be used on the real robot, the tracking error is also higher than the one obtained in the last chapter, this could be explained by the fact that the state of the robot is different, so the tuning of the hyperparameters and reward function coefficients should be done again to achieve the same performance as the one obtained in the last chapter.

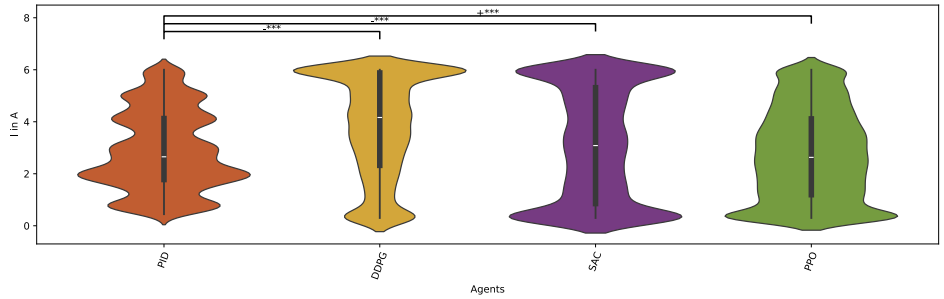
5.6.3 Multi agent reinforcement learning controller on the same configuration as training

Comparision between RL agents

For the training, we used the hyperparameters as in the last chapter we only increased the buffer size by 4 times as at each step we get 4 samples instead of one. Two versions of the reward functions mentioned in the last section were used. The PPO failed to find an optimal policy, so only results for SAC and DDPG are represented. In figure 5.4, we can see that the error tracking using the reward 2: the non-collaborative reward is higher than the one obtained using the reward 1: the collaborative reward, also we noticed that the agent suffer from oscillatory behavior using the reward 2, this could be explained by the fact that the reward 2 structure is diffrent than the reward one structure used in the last chapter so maybe a new adaptation of the reward coefficients is required, for the sake of simplicity and the clarity of the results, in next section we will use the reward 1 with the DDPG policy found using the optimal hyperparameters used in the last chapter.



(a) Tracking error comparison

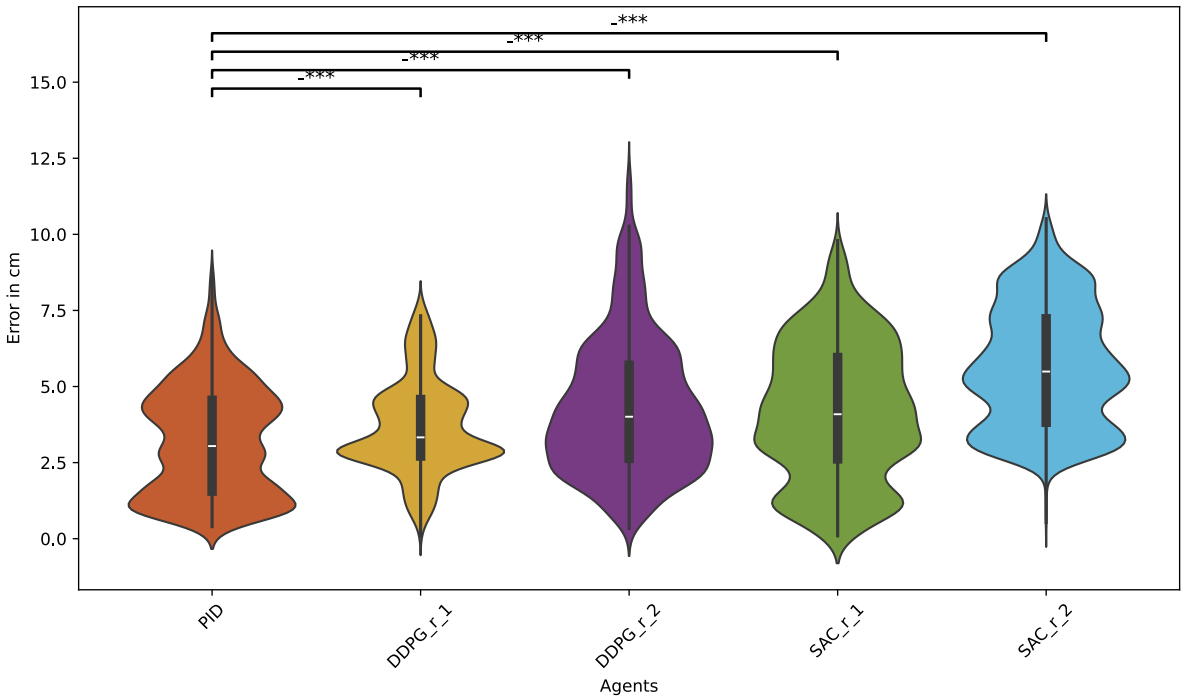


(b) Current comparison

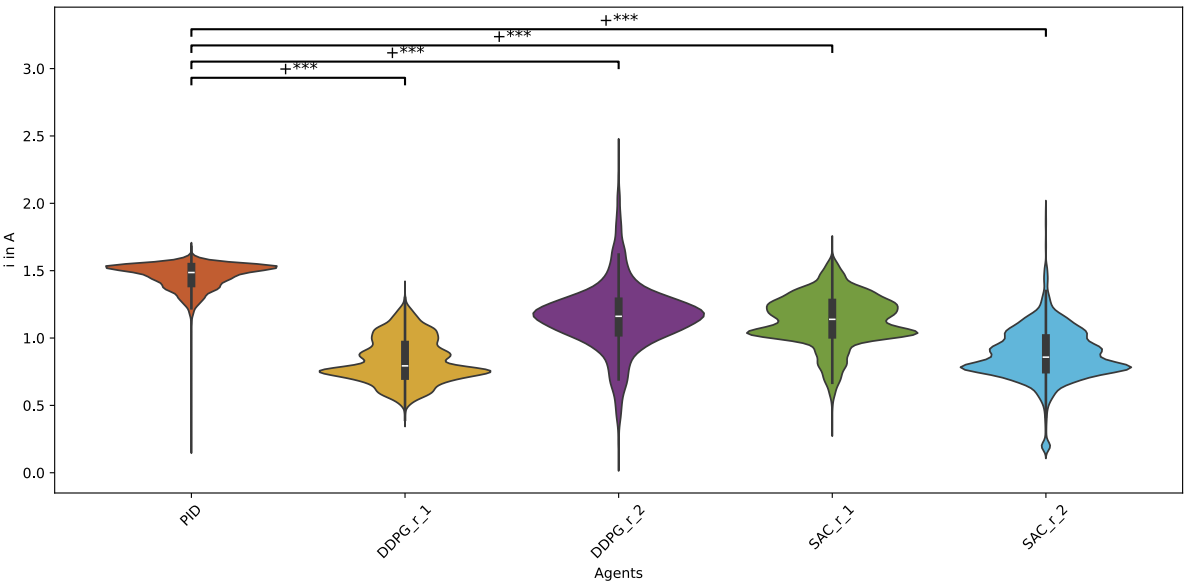
Figure 5.3: Simulation results: Comparison of the tracking error and current of the agent for 8 cables configuration on trajectory 1 for the three algorithms: DDPG, PPO, and SAC. The DDPG, SAC, and PPO are the agents learned using $\alpha_4 = 0.5$ for the current limits respect part of the reward function.

Comparision between single DDPG, multi DDPG and PID controllers

In figure 5.5, we can see that the new actuator level policy keep similar performance as the policy learned using the conventional reinforcement learning, the error on the first and third trajectory is less than the one obtained using the PID controller, and the error on the second trajectory is close to the PID, there is difference of $2cm$ basically due to the static error that could be improved by hyperparameters tuning like it was mentioned in the last chapter, the variation in the current is less than the one obtained using the conventional reinforcement learning, this could be explained by the fact that the agent is learning to control the actuator and not the robot, so the agent is more focused on the actuator dynamics and the force applied to the cable.
We also conducted the robustness test used in the last chapter, the actuator level policy kept the same performance as the one obtained using the conventional reinforcement learning, It was robust to mass change and noise.
In the next section, the transferability of the actuator level policy is tested on the real robot, and only comparison between the actuator level policy and the PID controller is presented as conventional reinforcement learning can not be used directly for different configuration of the robot.

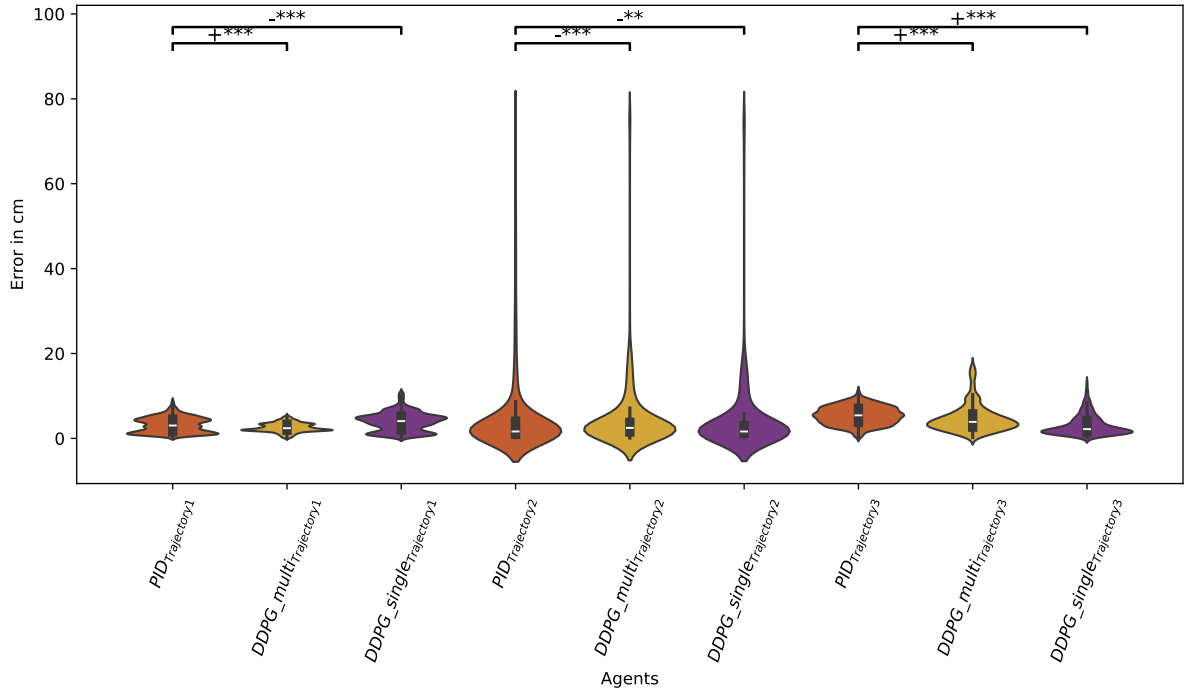


(a) Tracking error comparison

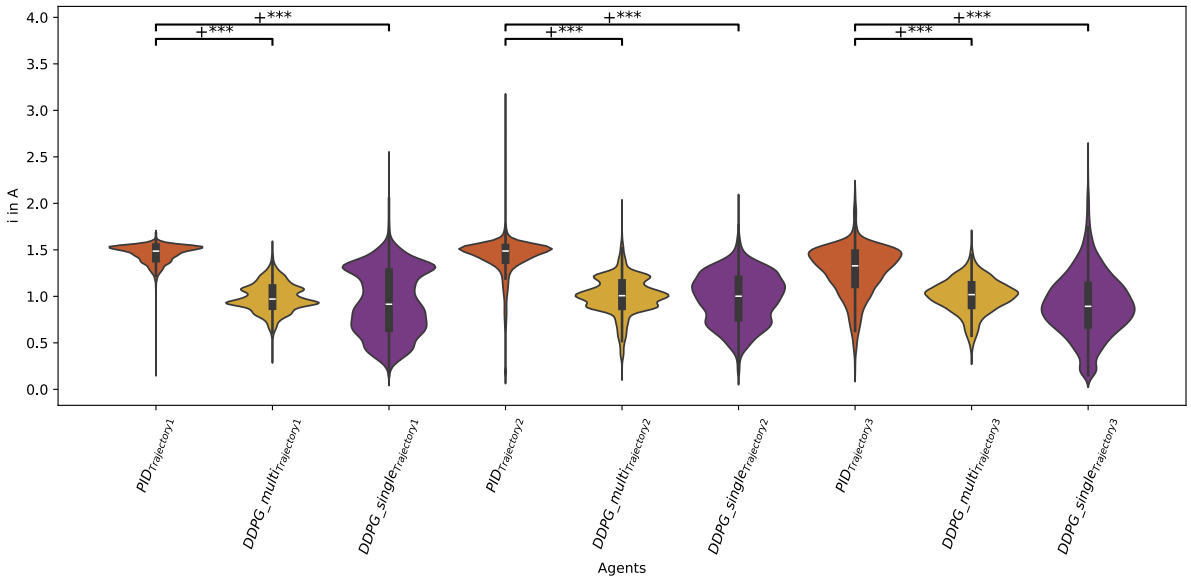


(b) Current comparison

Figure 5.4: Experimental results: Comparison of the tracking error and current of the agent for 4 cables configuration on trajectory 1 for the algorithms: DDPG and SAC using reward 1 and reward 2.



(a) Tracking error comparison



(b) Current comparison

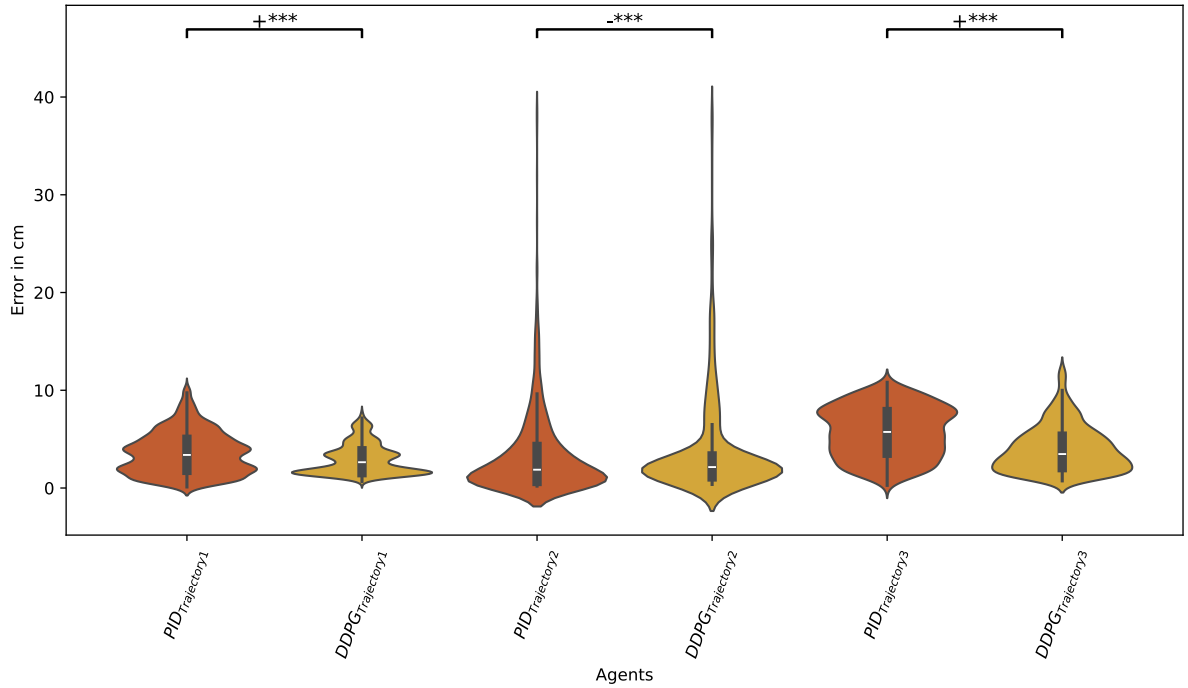
Figure 5.5: Experimental results: Comparison of the tracking error and current of the agent for the multi agent policy vs the conventional single agent policy ns the PID controller on the testing trajectories

5.6.4 Multi agent reinforcement learning controller on the diffrent configuration than training

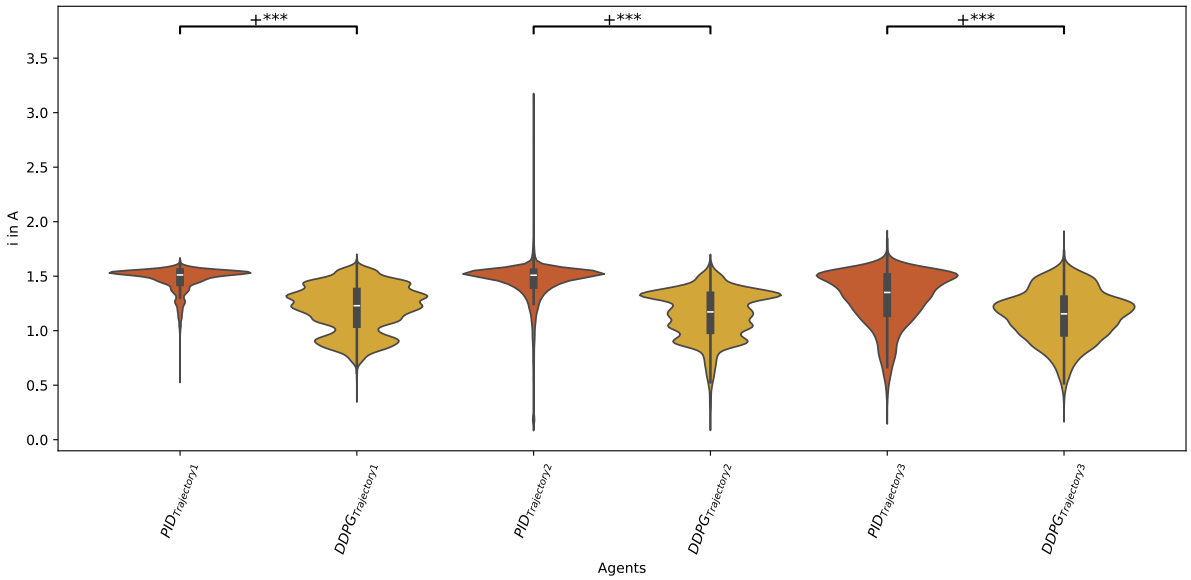
In this section, we will test the transferability of the actuator level policy from 4 cables configuration to 8 cables configuration, the policy is the same one learned using the 4 cables configuration. In figure 5.6, we can see that the actuator level policy kept the same performance as the one obtained using the 4 cables configuration. The tracking error on the second trajectory is still close to the PID, the performance on this trajectory is compromised only by the static error. And the DDPG still outperforms the PID on the first and third trajectories. The current mean is higher than the one obtained using the 4 cables configuration, contrary to the PID that kept the same current mean, however the current mean in still lower than the one obtained using the PID controller.
In table 5.1, we summarize the performance of the different controllers. The actuator level policy outperforms the PID controller. While it keeps the benefits of the conventional reinforcement learning: robustness to noise on fixed position target, it could also be used to control the robot with different number of actuators without any retraining like the PID controller and other classical control methods. Moreover, the actuator level policy is more robust to the error in the position estimation, as the agent control the cable length directly, so the position estimation is not used during the control process.

Table 5.1: A comparison of the performance of the different controllers

Configuration \ Controller designed for 4 cables configuration	PID	RL	multiRL
4 cables	✓	✓	✓
Different mass than training ($m = 0.5, ..., 2kg$)	✓	✓	✓
Fixed target position with uniform noise	×	✓	✓
Configuration with different number of cables	✓	×	✓
Error in position estimation	×	×	✓



(a) Tracking error comparison



(b) Current comparison

Figure 5.6: Experimental results: Comparison of the tracking error and current of the agent for the multi agent policy vs the conventional single agent policy ns the PID controller on the testing trajectories

5.7 Conclusion

In this chapter, a new reinforcement learning approach to control CDPRs with different number of actuators has been proposed, the main idea of the multi agent reinforcement learning is to learn a policy that maps the state of each actuator to the torque to be applied to the motor, instead of learning a policy that maps the state of the robot to the torques to be applied to the motors. The main advantage of this approach is that the learned policy can generalize to environments with different number of actuators, as the policy is learned to control the actuator and not the robot. The results showed that the actuator level policy learned using 4 cables configuration outperforms the PID controller on the 8 cables configuration and kept the same performance as the one obtained using the 4 cables configuration. Moreover, the absence of position estimation in the actuator state enhances the agent's robustness to errors in inverse kinematics, making this approach a promising candidate for the development of fault-tolerant controllers for CDPRs, a concept that has yet to be fully explored.

Conclusion and Perspectives

1 Conclusion

In this thesis, a sim-to-real approach for the control of CDPRs using reinforcement learning was proposed. The main objective was to improve the tracking performance of the robot at high speed and high acceleration, so that the tracking error is minimized during the insect tracking task on the Lab-On-cables project. Each stage, from simulation to transferability and generalization, has been detailed comprehensively. First, the simulation of the robot at the laboratory is developed using Matlab Simulink, then it was used as the basis of the reinforcement learning environment. Other parts like the target trajectory generation or the current limitation were added to have comprehensive RL framework for the tracking task. This framework is then used with the most known RL algorithms for continuous space to learn the optimal policy. The hyperparameters tuning with the reward design are also discussed. The learned policy was then transferred to the real robot and tested. The results showed that the learned policy is able to track the target trajectory at high speed and high acceleration. The actuator level approach was then proposed to solve the generalization problem of the learned policy across different robot configurations. The results showed that the learned policy is able to track the target trajectory with same performance on different configurations without any further training or tuning of the hyperparameters.

2 Major Contributions

The main contributions of this thesis on the use of reinforcement learning (RL) for controlling Cable-Driven Parallel Robots (CDPRs) are as follows:

- Development of a detailed RL framework for the tracking control at high speed and high acceleration of CDPRs. All the other works that have been done using RL for CDPRs have focused on the static control of the robot of the control at low speed. This work is the first to establish a framework for the task of tracking at high speed. Policies learned using this framework have shown to be robust and efficient in tracking tasks. It also outperforms the PID controller designed specifically for this task at high speed.

- Validation of the transferability of the learned policies from simulation to real-world for CDPRs. This is a crucial step in the development of RL-based controllers for real-world applications. The policies learned in simulation were successfully transferred to the real robot keeping the same performance, the only condition is the smoothness of the motor commands. This is the first study to validate the sim-to-real transfer of end-to-end RL policies for CDPRs, which is a significant step towards the use of RL for the control of CDPR, as learning in the real world is practically impossible.

- Proposal of a novel approach for learning at the actuator level. This approach solves the main drawback of RL-based controllers which is the lack of generalization across different robot configurations. As the time of writing this thesis, this is the first study to propose such an approach for CDPRs. The traditional RL approach is time consuming and requires retraining from scratch and retuning of the hyperparameters for each new configuration. This is big limitation for the use of RL as it could take months (launching many experiments of training and testing) to have a controller for a new configuration. Using the actuator-level approach, the learned policy can be used directly on any configuration without any further training or tuning of the hyperparameters. This is a major breakthrough in the field of CDPRs control as the policy (the neural network) is now associated with the motor, so once an individual user determines the most effective policy, they are able to share it directly with other users, supporting collective access to improved control policies for CDPRs.

3 Perspectives

The work presented in this thesis opens up several perspectives for future research. The use auto RL to learn the optimal hyperparameters for the training of the neural network and inverse reinforcement learning to learn the reward function could be investigated. This could help finding better performing policies and reduce the static tracking error encountered in the current work. Another perspective is to use the actuator level policy to control the robot to replicate the human movements like in the field of rehabilitation for example as the position of the cable is directly controlled in such applications. Finally the use of the actuator level policy could be also investigated for other types of robots like serial robots as the idea is not limited to CDPRs.

Résumé étendu

Introduction

Dans le cadre du projet Lab-on-Cables (LoC), le robot parallèle à câbles est utilisé pour suivre un insecte en vol libre à l'aide des caméras montées sur un effecteur cubique. Une telle application serait impossible avec un autre type de robot en raison des exigences élevées en termes de vitesse et d'accélération pour le suivi d'insecte, de l'ampleur de l'espace de travail nécessaire pour recréer un environnement naturel, et du déplacement silencieux du robot, évitant d'effrayer l'insecte.
Ce dernier est suivi à l'aide d'une caméra de détection de profondeur 3D qui calcule sa position dans le repère mobile, puis un algorithme de contrôle est utilisé pour déplacer le robot afin de maintenir l'insecte au centre de l'effecteur. Il est basé sur un contrôleur PID associé au modèle géométrique du robot pour calculer la vitesse souhaitée de l'effecteur, combiné à un solveur d'optimisation QP pour ajuster la tension dans les câbles.
L'objectif principal du projet est de capturer le mouvement de l'insecte à l'aide de caméras haute vitesse afin d'étudier son comportement. La séquence d'images collectées lors de cette expérience correspond essentiellement à une vidéo de 4 secondes enregistrée en mode ralenti. Ainsi, perdre l'insecte pendant quelques millisecondes hors du champ de la caméra pourrait entraîner une perte d'informations importantes. C'est pourquoi il est essentiel de disposer d'un algorithme de contrôle capable de gérer efficacement des trajectoires rapides tout en anticipant les déplacements de l'insecte.
C'est ici que l'utilisation de l'apprentissage par renforcement (RL) entre en jeu. Cette discipline a montré un grand potentiel ces dernières années en permettant aux robots d'apprendre directement à partir de données visuelles et d'exécuter des tâches complexes difficiles à modéliser analytiquement. L'utilisation de l'apprentissage par renforcement (RL) dans le contrôle des robots parallèles à câbles reste encore peu développée. Les rares études menées jusqu'à présent se concentrent principalement sur des trajectoires à basse vitesse ou sur des tâches de prise et de déplacement. Cette thèse a pour principal objectif d'explorer l'application du RL au contrôle des robots parallèles à câbles, en particulier pour le suivi de trajectoires à grande vitesse. Cette recherche s'inscrit dans le cadre du projet Lab-On-Cables.

Apprentissage profond par renforcement : Des concepts aux algorithmes

Ces dernières années, l'intelligence artificielle (IA) a pris une place essentielle en robotique. Définie par John McCarthy comme « la science et l'ingénierie de la fabrication de machines intelligentes », elle regroupe de nombreuses techniques, dont l'apprentissage automatique, qui révolutionne le domaine. Parmi ses approches, l'apprentissage par renforcement se distingue par

son fonctionnement basé sur l'interaction entre un agent et son environnement. Contrairement aux méthodes supervisées et non supervisées, il ne repose pas sur des données préexistantes, mais sur un processus d'essai-erreur visant à maximiser une fonction de récompense, ce qui en fait une solution privilégiée pour le contrôle des robots.
Avec les avancées récentes en apprentissage profond, l'apprentissage par renforcement profond (DRL) s'est imposé comme une solution prometteuse, notamment grâce aux travaux de DeepMind et à l'élaboration de l'algorithme Deep Q-Network (DQN). Ces progrès ont ouvert de nouvelles perspectives pour le contrôle des robots parallèles à câbles, offrant une approche plus flexible et adaptative du suivi de trajectoire. Toutefois, son implémentation demeure un défi en raison de la complexité des algorithmes, du réglage des hyperparamètres et des nombreuses décisions de conception qu'elle implique.
Afin d'implementer efficacement le DRL dans le contrôle des CDPRs, un cadre de travail complet doit être établi, incluant la modélisation du robot, la simulation de l'environnement, le choix de l'algorithme d'apprentissage et la définition de la fonction de récompense. Ce processus, bien que fastidieux, est essentiel pour garantir la réussite de l'apprentissage et la transférabilité de la politique apprise vers un robot réel.

Robots Parallèles à Câbles : Modélisation et Simulation

Les robots parallèles à câbles (CDPRs) sont une catégorie de robots parallèles où la transmission des forces et des couples s'effectue via des câbles. Leur rapidité et leur forte capacité d'accélération leur permettent d'être utilisés dans des domaines variés tels que la rééducation, l'industrie, la construction et la logistique. Cependant, leur contrôle et leur modélisation restent particulièrement complexes en raison des non-linéarités.
Dans le cadre du projet Lab-on-Cables (LoC), un CDPR est utilisé pour suivre des insectes volants. L'effecteur, une structure cubique équipée de caméras et d'une source lumineuse, capture des images de l'insecte, tandis qu'une caméra de profondeur estime sa position. Ce petit laboratoire mobile, suspendu par des câbles, suit ainsi l'insecte en temps réel dans son environnement. Capable d'atteindre une vitesse de 3,6 m/s et une accélération de 17 m/s^2, ce CDPR met en lumière les défis de commande spécifiques à ce type de robot.
Face à ces défis, l'intelligence artificielle représente une alternative prometteuse pour la commande des CDPRs. Une première étape essentielle consiste en la modélisation dynamique du robot et le développement d'un outil de simulation sous Matlab/Simulink. Ce modèle dynamique, validé à l'aide de données réelles issues d'expériences de suivi de trajectoire sous commande PID, offre une base fiable pour l'exploration de nouvelles stratégies de commande.
Les éléments fondamentaux établis, incluant l'analyse cinématique, la modélisation dynamique et la simulation, ouvrent la voie à l'intégration d'un algorithme d'apprentissage par renforcement pour la commande des CDPRs. La flexibilité de cette simulation est un atout majeur, permettant d'adapter le modèle à différentes configurations de CDPRs, indépendamment du nombre et de la disposition des câbles. Cette approche facilitera notamment le développement et l'évaluation d'un algorithme de commande pour une configuration à 12 câbles, une perspective envisagée en l'absence du système physique correspondant dans le laboratoire à ce jour.

Une approche d'apprentissage par renforcement profond pour le contrôle des robots parallèles à câbles

Dans ce travail, on s'intéresse à l'apprentissage par renforcement de bout en bout, qui met pleinement en valeur le potentiel du DRL en contrôle. Son principal avantage réside dans sa capacité à apprendre une politique de commande à partir de zéro, sans nécessiter de connaissances préalables sur le système à contrôler. En traitant le système dans sa globalité plutôt qu'en sous-systèmes, elle permet de surmonter certaines limites des techniques de commande classiques, notamment la gestion des non-linéarités, la prise en compte des perturbations et l'incertitude du modèle.
L'élaboration d'un cadre de travail adapté est une étape essentielle. Celui-ci définit l'environnement où s'inscrit le problème de contrôle, les contraintes à respecter, ainsi que la fonction de récompense, qui guide l'apprentissage de l'agent tout en garantissant la sécurité du système. Ces éléments sont cruciaux pour assurer le transfert efficace de la politique apprise vers un robot réel.
La première étape pour définir l'environnement consiste à déterminer l'état de l'agent, qui doit inclure toutes les informations essentielles à la prise de décision. Dans notre cas, cet état est défini par la position et la vitesse de l'effecteur, la position de l'insecte ainsi que les courants des moteurs. L'action de l'agent correspond à la commande en vitesse envoyée aux moteurs.
La génération de trajectoires cibles joue un rôle fondamental dans l'apprentissage, car elle permet d'explorer différentes configurations afin d'optimiser la politique. Ce processus est conçu de manière à produire des positions aléatoires tout en restant dans l'espace de travail et en garantissant la continuité des trajectoires.
Le robot est contrôlé en vitesse. De plus, une fonction de limitation du courant a été intégrée avant l'envoi des commandes afin d'éviter tout dépassement indésirable et d'assurer que les câbles restent toujours en tension.
La fonction de récompense, élément clé de l'apprentissage par renforcement, guide l'agent dans l'optimisation de sa politique en évaluant la pertinence de ses actions. Dans notre cas, elle est définie en fonction de la distance entre la position de l'insecte et celle de l'effecteur. Une pénalisation est également appliquée pour limiter les courants moteurs excessifs ainsi que les commandes susceptibles de générer des courants hors des limites autorisées.
L'interaction entre l'environnement et les agents d'apprentissage est explorée à travers différents algorithmes. Une fois l'environnement établi, il sert de base à l'apprentissage d'une politique optimale pour le suivi de trajectoire des CDPRs.

Processus d'entraînement et résultats expérimentaux

Le réglage des hyperparamètres est une étape essentielle mais chronophage, car le processus d'entraînement peut s'étendre sur plusieurs heures, voire plusieurs jours. Afin d'optimiser les ressources de calcul, l'ajustement des hyperparamètres a débuté en utilisant ceux du modèle de pendule de l'environnement OpenAI Gym, avant d'être progressivement affinés pour améliorer les performances et accélérer la convergence. Une fois la meilleure politique apprise pour chaque agent, une phase de validation a été menée pour évaluer leur efficacité sur différentes trajectoires sélectionnées. L'un des défis majeurs de l'apprentissage par renforcement en robotique réside dans l'écart entre la simulation et le monde réel. Une politique optimale en simulation ne garantit pas nécessairement les mêmes performances sur un robot physique, où divers facteurs peuvent altérer son efficacité. Pour assurer la transférabilité des agents RL, des expérimentations sur un

robot réel ont été réalisées, comparant les performances des agents d'apprentissage par renforcement et du contrôleur PID en termes de précision de suivi, de respect des limites de courant et de robustesse face aux perturbations.
Les tests ont confirmé la transférabilité des politiques apprises par les agents DDPG, PPO et SAC, à condition que le signal de commande reste suffisamment lisse. Bien que prometteurs, les agents RL ont initialement souffert d'une erreur statique impactant leur précision en phase stationnaire. Une optimisation supplémentaire du DDPG a permis de réduire cette erreur à 2 cm, contre 5 cm auparavant, et d'atteindre des performances similaires, voire supérieures, à celles du contrôleur PID. Notamment, sur certaines trajectoires, l'agent DDPG s'est révélé plus performant que le PID tout en consommant moins de courant. Il a également démontré une meilleure robustesse face aux variations de masse et aux perturbations du signal cible, là où le PID a échoué.
L'objectif final étant d'élaborer une politique de contrôle adaptée au suivi d'insectes en vol, les résultats obtenus avec l'agent DDPG sont encourageants. La trajectoire de l'insecte projetée sur le plan x-z a pu être validée avec succès sur le robot réel. Les prochaines étapes consisteront à étudier la généralisation de cette approche d'apprentissage par renforcement à différentes configurations du robot.

Vers les robots parallèles à n câbles : une approche généralisée

Pour répondre au problème de généralisation des politiques apprises sur différentes configurations de robots parallèles à câbles, une nouvelle approche d'apprentissage par renforcement (RL) est proposée. Cette méthode permet à la politique apprise de s'adapter à des environnements avec un nombre variable d'actionneurs, sans nécessiter de réentraînement. Elle vise ainsi à surmonter l'une des principales limitations des approches de RL de bout en bout : le manque de généralisation, qui rend souvent les politiques spécifiques à une seule configuration du robot.
Contrairement aux méthodes classiques qui associent l'état global du robot aux couples appliqués aux moteurs, l'approche proposée repose sur un apprentissage par renforcement multi-agent. L'idée principale est d'apprendre une politique qui relie l'état de chaque actionneur au couple moteur à appliquer, plutôt que de chercher directement à contrôler l'ensemble du robot. Cette approche présente un avantage clé : en se concentrant sur les actionneurs individuels, elle permet d'adapter la politique apprise à des robots ayant un nombre d'actionneurs différent, sans modification majeure.
Les résultats expérimentaux ont démontré la robustesse de cette méthode. Une politique apprise sur une configuration à 4 câbles a surpassé un contrôleur PID sur une configuration à 8 câbles, tout en maintenant des performances équivalentes à celles obtenues avec 4 câbles. Cette flexibilité illustre la capacité de la politique à s'adapter à diverses configurations sans nécessiter un nouvel entraînement à chaque changement.
En outre, cette approche renforce la robustesse du système en éliminant le besoin d'estimer la position dans l'état de l'actionneur. Cela améliore la fiabilité de l'agent face aux erreurs de calcul de la cinématique inverse, un problème fréquent dans les CDPRs aux configurations complexes. Cette caractéristique ouvre la voie au développement de contrôleurs tolérants aux pannes, un domaine encore peu exploré pour ce type de robots.
En conclusion, cette approche d'apprentissage par renforcement multi-agent, qui apprend à contrôler chaque actionneur indépendamment, constitue une solution efficace et robuste pour le contrôle des CDPRs dans des configurations variées. Elle représente une avancée significative vers des contrôleurs autonomes et adaptatifs pour les robots complexes.

Conclusion

Dans cette thèse, une approche sim-to-real a été proposée pour le contrôle des robots parallèles à câbles (CDPRs) à l'aide de l'apprentissage par renforcement. L'objectif principal était d'améliorer la performance du suivi du robot à grande vitesse et forte accélération, afin de minimiser l'erreur lors de la tâche de suivi d'insectes dans le cadre du projet Lab-On-Cables.
Chaque étape, de la simulation à la généralisation en passant par le transfert, a été explorée en détail. Tout d'abord, une simulation du robot en laboratoire a été développée sous Matlab Simulink et utilisée comme base pour l'environnement d'apprentissage par renforcement. Des éléments clés, tels que la génération de trajectoires cibles et la limitation du courant, ont été intégrés afin de créer un cadre complet pour la tâche de suivi. Ce cadre a ensuite été exploité avec les algorithmes de renforcement les plus adaptés aux espaces continus afin d'apprendre une politique optimale. Le réglage des hyperparamètres ainsi que la conception de la fonction de récompense ont également été étudiés. Une fois la politique apprise, elle a été transférée sur le robot réel et testée. Les résultats ont démontré sa capacité à suivre la trajectoire cible avec précision, même à grande vitesse et forte accélération.
Afin d'améliorer la généralisation de la politique apprise sur différentes configurations de robots, une approche centrée sur les actionneurs a été développée. Les expérimentations ont montré que cette politique pouvait être appliquée à diverses configurations sans nécessiter un nouvel apprentissage ni un ajustement des hyperparamètres, garantissant ainsi une robustesse accrue.
Le travail réalisé ouvre plusieurs perspectives pour les recherches futures. L'intégration de l'auto-RL pour l'optimisation automatique des hyperparamètres du réseau neuronal, ainsi que l'apprentissage par renforcement inverse pour la définition de la fonction de récompense, pourraient être explorés afin d'améliorer encore la performance et de réduire l'erreur statique de suivi. Une autre piste prometteuse serait l'utilisation de la politique au niveau des actionneurs pour contrôler le robot dans le but de reproduire des mouvements humains, notamment dans le domaine de la rééducation, où la position du câble serait directement contrôlée. Enfin, cette approche pourrait également être étendue à d'autres types de robots, tels que les robots séries, démontrant ainsi que son application ne se limite pas aux CDPRs.

Bibliography

[1] Rémi Pannequin, Mélanie Jouaiti, Mohamed Boutayeb, Philippe Lucas, and Dominique Martinez. Automatic tracking of free-flying insects using a cable-driven robot. *Science Robotics*, 5(43):eabb2890, June 2020.

[2] Lawrence L. Cone. Skycam-an aerial robotic camera system. *Byte*, 10(10):122, 1985.

[3] D. Surdilovic and R. Bernhardt. STRING-MAN: a new wire robot for gait rehabilitation. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, pages 2031–2036 Vol.2, New Orleans, LA, USA, 2004. IEEE.

[4] Xinyu Geng, Meng Li, Yufei Liu, Yuanyuan Li, Wei Zheng, and Zhibin Li. Analytical tension-distribution computation for cable-driven parallel robots using hypersphere mapping algorithm. *Mechanism and Machine Theory*, 145:103692, March 2020.

[5] Tuong Phuoc Tho and Nguyen Truong Thinh. An Overview of Cable-Driven Parallel Robots: Workspace, Tension Distribution, and Cable Sagging. *Mathematical Problems in Engineering*, 2022:1–15, July 2022.

[6] L. Barbazza, F. Oscari, S. Minto, and G. Rosati. Trajectory planning of a suspended cable driven parallel robot with reconfigurable end effector. *Robotics and Computer-Integrated Manufacturing*, 48:1–11, December 2017.

[7] Thomas Rousseau, Christine Chevallereau, and Stéphane Caro. Human-cable collision detection with a cable-driven parallel robot. *Mechatronics*, 86:102850, October 2022.

[8] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-End Training of Deep Visuomotor Policies. 2016.

[9] John McCarthy et al. What is artificial intelligence?, 2024. Accessed: October 10, 2024.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[11] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[12] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. 2013.

[15] Alessandro Montenegro, Marco Mussi, Alberto Maria Metelli, and Matteo Papini. Learning optimal deterministic policies with stochastic policy gradients. *arXiv preprint arXiv:2405.02235*, 2024. Accepted at ICML 2024.

[16] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, July 2019. arXiv:1509.02971 [cs, stat].

[17] Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed Distributional Deterministic Policy Gradients, April 2018. arXiv:1804.08617 [cs, stat].

[18] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods, October 2018. arXiv:1802.09477 [cs, stat].

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017. arXiv:1707.06347 [cs].

[20] John Schulman, Sergey Levine, Philipp Moritz, Michael I Jordan, and Pieter Abbeel. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 1889–1897. PMLR, 2015.

[21] Hsiao-Ru Pan, Nico Gürtler, Alexander Neitz, and Bernhard Schölkopf. Direct advantage estimation. 2022.

[22] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

[23] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic Algorithms and Applications, January 2019. arXiv:1812.05905 [cs, stat].

[24] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[25] Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning quadrupedal locomotion over challenging terrain. *Science Robotics*, 5(47):eabc5986, October 2020.

[26] Fereshteh Sadeghi and Sergey Levine. CAD2RL: Real Single-Image Flight without a Single Real Image, June 2017. arXiv:1611.04201 [cs].

[27] Athanasios S. Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.

[28] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.

[29] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World, March 2017. arXiv:1703.06907 [cs].

[30] Shikun Liu, Edward Johns, and Andrew J Davison. Domain adaptation in reinforcement learning via latent unified state representation. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 2021. Accepted at AAAI 2021.

[31] Zhuangdi Zhu, Kaixiang Lin, and Jiayu Zhou. Transfer learning in deep reinforcement learning: A survey. *CoRR*, abs/2009.07888, 2020.

[32] Shangding Gu, Long Yang, Yali Du, Guang Chen, Florian Walter, Jun Wang, Yaodong Yang, and Alois Knoll. A review of safe reinforcement learning: Methods, theory and applications. 05 2022.

[33] Julian Ibarz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine. How to Train Your Robot with Deep Reinforcement Learning; Lessons We've Learned. *The International Journal of Robotics Research*, 40(4-5):698–721, April 2021. arXiv:2102.02915 [cs].

[34] Mohammad M. Aref and Jouni Mattila. Automated Calibration of Planar Cable-Driven Parallel Manipulators by Reinforcement Learning in Joint-Space. In *2018 6th RSI International Conference on Robotics and Mechatronics (IcRoM)*, pages 172–177, Tehran, Iran, October 2018. IEEE.

[35] Alex Grimshaw and John Oyekan. Applying Deep Reinforcement Learning to Cable Driven Parallel Robots for Balancing Unstable Loads: A Ball Case Study. *Frontiers in Robotics and AI*, 7:611203, February 2021.

[36] Renyu Yang, Jianlin Zheng, and Rong Song. Continuous mode adaptation for cable-driven rehabilitation robot using reinforcement learning. *Frontiers in Neurorobotics*, 16:1068706, December 2022.

[37] Yuming Liu, Zhihao Cao, Hao Xiong, Junfeng Du, Huanhui Cao, and Lin Zhang. Dynamic Obstacle Avoidance for Cable-Driven Parallel Robots With Mobile Bases via Sim-to-Real Reinforcement Learning. *IEEE Robotics and Automation Letters*, 8(3):1683–1690, March 2023.

[38] Chenglin Xie, Jie Zhou, Rong Song, and Ting Xu. Deep Reinforcement Learning Based Cable Tension Distribution Optimization for Cable-driven Rehabilitation Robot. In *2021 6th IEEE International Conference on Advanced Robotics and Mechatronics (ICARM)*, pages 318–322, Chongqing, China, July 2021. IEEE.

[39] Yanqi Lu, Chengwei Wu, Weiran Yao, Guanghui Sun, Jianxing Liu, and Ligang Wu. Deep Reinforcement Learning Control of Fully-Constrained Cable-Driven Parallel Robots. *IEEE Transactions on Industrial Electronics*, 70(7):7194–7204, July 2023.

[40] Hao Xiong, Tianqi Ma, Lin Zhang, and Xiumin Diao. Comparison of end-to-end and hybrid deep reinforcement learning strategies for controlling cable-driven parallel robots. *Neurocomputing*, 377:73–84, February 2020.

[41] Dinh-Son Vu and Ahmad Alsmadi. Trajectory Planning of a CableBased Parallel Robot using Reinforcement Learning and Soft Actor-Critic. *WSEAS TRANSACTIONS ON APPLIED AND THEORETICAL MECHANICS*, 15:165–172, December 2020.

[42] Caner Sancak, Fatma Yamac, and Mehmet Itik. Position control of a planar cable-driven parallel robot using reinforcement learning. *Robotica*, pages 1–18, March 2022. Publisher: Cambridge University Press.

[43] Adhiti Raman, Ameya Salvi, Matthias Schmid, and Venkat Krovi. Reinforcement Learning Control of a Reconfigurable Planar Cable Driven Parallel Manipulator. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9644–9650, London, United Kingdom, May 2023. IEEE.

[44] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep Reinforcement Learning That Matters. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), April 2018.

[45] Andreas Pott, Christian Meyer, and Alexander Verl. Large-Scale Assembly of Solar Power Plants with Parallel Cable Robots.

[46] Robert L. Williams, Ming Xin, and Paul Bosscher. Contour-Crafting-Cartesian-Cable Robot System: Dynamics and Controller Design. In *Volume 2: 32nd Mechanisms and Robotics Conference, Parts A and B*, pages 39–45, Brooklyn, New York, USA, January 2008. ASMEDC.

[47] Johann Lamaury and Marc Gouttefarde. Control of a large redundantly actuated cable-suspended parallel robot. In *2013 IEEE International Conference on Robotics and Automation*, pages 4659–4664, Karlsruhe, Germany, May 2013. IEEE.

[48] Andreas Pott. *Cable-Driven Parallel Robots: Theory and Application*. Springer Tracts in Advanced Robotics. Springer International Publishing, Cham, 2018. ISSN: 1610-7438, 1610-742X.

[49] M.M. Aref and H.D. Taghirad. Geometrical workspace analysis of a cable-driven redundant parallel manipulator: KNTU CDRPM. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nice, September 2008. IEEE.

[50] Sana Baklouti. *Vibration Analysis and Reduction of Cable-Driven Parallel Robots*. PhD thesis, Mechanical engineering [physics.class-ph]. INSA de Rennes, 2018. English. NNT : 2018ISAR0034. tel-02163227.

[51] Andreas Pott. Cable-driven parllel robots theory and application. In *Springer Tracts in Advanced Robotics*. Springer, 2018.

[52] Theresa Eimer, Marius Lindauer, and Roberta Raileanu. Hyperparameters in Reinforcement Learning and How To Tune Them, June 2023. arXiv:2306.01324 [cs].

[53] James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. 2012.

[54] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks, November 2017. arXiv:1711.09846 [cs].

[55] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study, June 2020. arXiv:2006.05990 [cs, stat].

[56] Hemant Singh. Implementing ddpg algorithm on the inverted pendulum problem. `https://keras.io/examples/rl/ddpg_pendulum/`.

[57] Ilias Chrysovergis. Implementation of a proximal policy optimization agent for the cartpole-v1 environment. `https://keras.io/examples/rl/ppo_cartpole/`.

[58] Grid'5000 Team. Getting Started with Grid'5000, 2024. Accessed: 2024-10-29.

[59] Ariel Hart. Mann-whitney test is not just a test of medians: differences in spread can be important. *BMJ*, 323(7309):391–393, Aug 18 2001.

[60] SciPy Developers. *scipy.stats.mannwhitneyu — SciPy v1.10.0 Manual*, 2023. Accessed: 2024-10-27.

[61] Andrei A. Rusu, Mel Vecerik, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-Real Robot Learning from Pixels with Progressive Nets, May 2018. arXiv:1610.04286 [cs].

[62] Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. A Generalist Agent, November 2022. arXiv:2205.06175 [cs].