Master Systèmes Dynamiques et Signaux

Master thesis

Investigation of Deep Reinforcement Learning Methods for Control in Underwater Robotics

Author :
M. Romain Bornier

Jury :
Pr. L. Hardouin
Pr. L. Jaulin

Version of August 25, 2025

Acknowledgments

I express my deepest gratitude to Gilles Le Chenadec and Benoit Clément, who guided me throughout the duration of this internship. Their explanations, support, and valuable advice have been truly essential for my work.

I also sincerely thank my friend and fellow intern, Antoine Morvan, for his constant support and collaboration during this period. My thanks also go to Katell Legattu, whose expertise in Reinforcement Learning applied to underwater robotics, as well as her insights into the research conducted within the laboratory, have been particularly enlightening.

Finally, I wish to thank all the members of the laboratory with whom I had the opportunity to exchange ideas. Their advice has not only contributed to the successful completion of this internship, but also provided me with valuable guidance for my future career.

Contents

Introduction

In recent years, autonomous underwater vehicles have become essential tools for ocean exploration, environmental monitoring, and resource management. However, the complexity and unpredictability of the underwater environment pose significant challenges to their control and autonomy.

1 Context

The underwater environment presents unique challenges for robotic control. Many methods developed for surface or terrestrial robotics cannot be applied directly underwater due to the specific physical constraints and uncertainties of the marine domain. Strong and unpredictable currents, limited visibility, communication difficulties, and the lack of accurate GPS signals make precise control and navigation particularly difficult. Moreover, underwater vehicles are often over-actuated and receive delayed, noisy sensor feedback, which complicates traditional model-based control approaches.

Artificial intelligence (AI), especially **reinforcement learning** (RL), offers a promising alternative by allowing autonomous systems to learn control policies directly from interaction with their environment. Instead of relying solely on predefined models, RL allows agents to adapt their behavior based on experience, making it well suited to handle uncertainty, partial observability, and non-linear dynamics. Such adaptive learning methods could allow underwater robots to cope robustly with disturbances, optimize trajectories in real time, and recover from failures such as actuator malfunctions or sensor drift.

Recent advances in deep learning have improved RL by integrating **deep neural network** (DNN), **leading to deep reinforcement learning** (DRL). Thanks to the powerful function approximation of DNN, DRL algorithms can scale to high-dimensional state and action spaces, essential for complex tasks in underwater robotics.

During the past decade, the DRL field has grown rapidly, with a variety of algorithms that address challenges such as sample efficiency, stability, exploration, and generalization. From value-based methods such as DQN to policy gradient approaches like **Proximal Policy Optimization** (PPO) and **Soft Actor-Critic**, each offers advantages and limitations depending on the task and environment.

Given this diversity, it is crucial to perform a systematic analysis of existing DRL approaches in underwater robotics. Understanding the strengths, weaknesses, and assumptions behind these methods helps identify suitable algorithms for real-world deployment. This analysis also highlights gaps in the literature and guides future research to improve the robustness, adaptability, and real-time performance of autonomous underwater systems.

2 Objectives

The primary objective of this internship is to advance the development and evaluation of RL approaches for underwater robotics. This work follows two complementary directions: (1) revisiting and reproducing previous PhD research, and (2) establishing a comprehensive benchmark of modern deep RL (DRL) algorithms through simulation.

A key challenge of this project arises from the rapid evolution of AI tools and robotic simulation platforms in recent years. In this context, my work focuses on designing and implementing a complete simulation pipeline: from environment modeling in **Gazebo Harmonic**, to interaction via the **ROS2** middleware, followed by reinforcement learning integration using a **Gym-based Python environment** and the **Stable Baselines3** framework, and visualization and analysis through PyQt5 and Matplotlib. The core objective is to ensure that these heterogeneous components work seamlessly together, providing a modern and coherent framework for reinforcement learning experiments in robotics.

In parallel, a thorough benchmark of contemporary DRL algorithms is performed, specifically SAC and PPO. This involves a critical review of the relevant literature, consolidation of theoretical foundations, practical implementation within underwater robot control contexts, and in-depth analysis of the strengths and limitations of these algorithms with respect to the unique challenges of the underwater domain.

Importantly, the internship aims not simply to compare algorithmic performance, but to deepen the understanding of RL methods by systematically investigating their efficiency, robustness, and limitations. By integrating theoretical analysis with simulation-based experiments, this work highlights the potential pitfalls of popular RL techniques, fostering informed discussions on their true capabilities and practical considerations in realistic marine environments. Rather than seeking a single 'best' algorithm, the focus is on identifying the nuanced trade-offs and factors that influence successful deployment.

Through the identification of key parameters that affect RL training, the development of rigorous evaluation frameworks, and the creation of analytical tools to interpret the results, this internship aspires to provide actionable insights. These contributions are intended to guide future research and practical applications, supporting the careful and effective use of RL techniques for autonomous underwater vehicle control in complex, nonlinear domains.

3 Planning

At the beginning of the internship, the initial goals differed somewhat from those defined later. The research initially aimed to explore the integration of RL into classical control strategies such as PID or LQR. However, these objectives were not fixed and, in close collaboration with my supervisors, the scope of the internship evolved based on my findings and questions during the work.

Given my previous experience with the PPO algorithm during my last internship, and the lab's ongoing experiments with the SAC algorithm, it quickly became clear that a comparative study of different RL methods in our underwater robotic environment would be both relevant and valuable. Consequently, we decided to focus exclusively on benchmarking algorithms rather than investigating hybrid AI-classical control methods, an avenue that could be explored in future PhD work. This strategic refocusing was considered more appropriate to ensure the delivery of concrete and meaningful results within the limited time frame of the internship.

Another important factor shaping the planning was the technical challenge of adapting previous PhD work to current tools and simulation platforms. Because the feasibility of this adaptation was uncertain at the beginning, it was difficult to establish a precise schedule. As a result, the internship followed a non-linear progression, with objectives refined iteratively.

The Gantt chart in Figure 1 illustrates the approximate distribution of time among the main tasks during the internship.

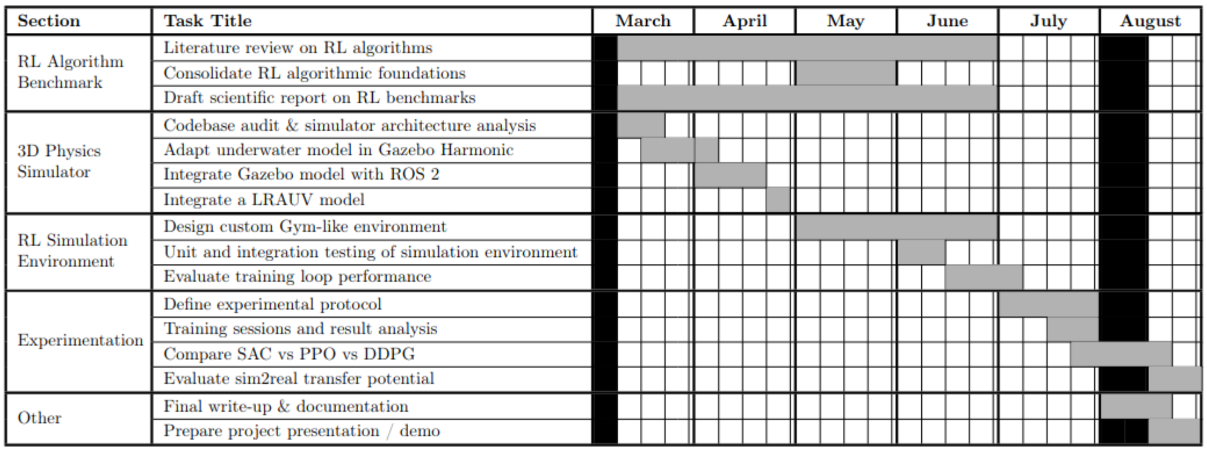


Figure 1: Gantt chart showing the planned time allocation for the major internship tasks.

The remainder of this report is organized into three main chapters. Chapter 1 introduces the fundamental concepts of reinforcement learning and presents the algorithms relevant to this work, with a particular focus on SAC and PPO. Chapter 2 describes the implementation of the simulation environment, including the integration of Gazebo Harmonic, ROS2, and the reinforcement learning pipeline. Finally, Chapter 3 is dedicated to experimentation, where the influence of key parameters is systematically analyzed, and the performance of different algorithms is compared.

Chapter 1

Reinforcement learning concepts and algorithms

1 Core principles of reinforcement learning

In RL, agents, such as robots, interact with an environment and learn through trial and error. Unlike supervised learning, which relies on labeled data, RL depends on the agent's experience gained by exploring the environment, performing actions, and receiving feedback in the form of rewards. This feedback encourages the agent to reinforce actions that produce positive outcomes and avoid those that lead to negative consequences, enabling it to learn behaviors that achieve specific goals.

1.1 Markov Decision Processes: formal framework

Among the various mathematical frameworks developed to model reinforcement learning, the **Markov Decision Process** (MDP) stands out as the standard formalism to describe the interaction between an agent and its environment. An MDP is a discrete-time stochastic control process defined by the tuple (S, A, T, R) [1], where:

- S: the set of all possible states,
- A: the set of actions available to the agent,
- T: the transition function, i.e., the probability of reaching state s_{t+1} after taking action a_t in state s_t, $P(s_{t+1} \mid s_t, a_t)$,
- R: the reward function, representing the immediate (or expected) reward received for a transition from s_t to s_{t+1} following action a_t, $R(s_{t+}|s_t, a_t)$.

When both S and A are finite, the MDP is said to be *discrete.* For example, an action set such as $\{$*up*, *down*, *left*, *right*$\}$ is discrete, commonly found in video games. Conversely, *continuous* MDPs feature states and actions with continuous values, such as controlling an **Autonomous Underwater Vehicle** (AUV) where actions such as thruster forces or positions are really valued.

The choice between discrete and continuous spaces significantly influences algorithm design: discrete problems are often simpler and computationally less demanding, while continuous problems offer greater expressiveness but require more sophisticated and resource-intensive algorithms.

Figure 1.1 [2] illustrates a simple discrete MDP with three states $S = \{s_0, s_1, s_2\}$ and two actions $A = \{a_0, a_1\}$. Transition probabilities are displayed next to the arrows; for example, $P(s_2 \mid s_0, a_0) = 0.5$. The reward function is represented by orange arrows and is defined as $R(s_0|s_2, a_1) = -1$ and $R(s_0|s_1, a_0) = +5$.

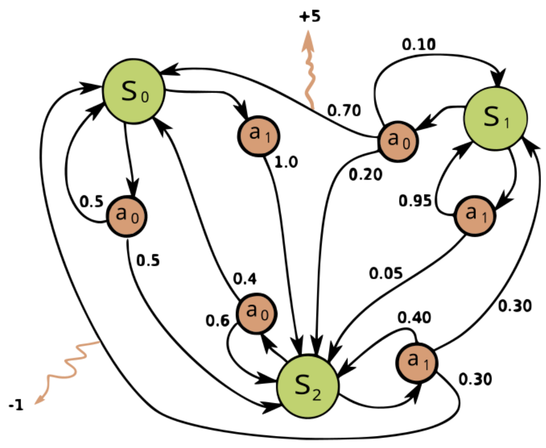


Figure 1.1: Example of a simple MDP

A fundamental property of MDPs is the **Markov property**, which asserts that the future state depends only on the current state and action, not on the sequence of past states and actions:

$$P(s_{t+1} \mid s_t, a_t) = P(s_{t+1} \mid s_1, a_1, \ldots, s_t, a_t) \tag{1.1}$$

This property simplifies the modeling of decision-making problems by focusing only on the current state.

For simplicity of notation, and as illustrated in Figure 1.1, the reward function will henceforth be written as $R(s_{t+1}|s_t, a_t) = R_{t+1} = r_{t+1}$.

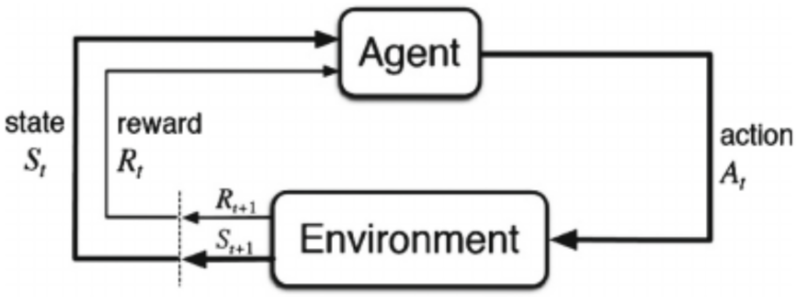


Figure 1.2: Illustration of the agent environment interaction in a Markov Decision Process.

Within this framework, the agent aims to learn the most effective behavior to perform the given tasks. As illustrated in Figure 1.2 [3], in RL the agent perceives the current state s_t through observations o_t and selects an action a_t according to a **policy** π. Executing this action causes a transition to the next state s_{t+1}, while the environment provides a reward r_{t+1}, which quantifies the immediate quality of the pair of action states chosen. This loop of interaction continues, allowing the agent to learn which actions are most beneficial over time.

The policy π defines a mapping from states to actions and can be either *deterministic*, where the same action is always selected for a given state ($\pi(s_t) = a_t$), or *stochastic*, where a probability distribution over possible actions is specified given the state ($\pi(a_t \mid s_t) = P(a_t = a \mid s_t = s)$).

The objective in RL is to learn an **optimal policy** π^* that maximizes the **expected cumulative reward**, representing the average total reward achievable over all possible episodes starting from the initial state and weighted by their likelihood under π. An **episode** is a complete interaction with the environment, from initial state to termination, either upon reaching a **terminal state** or after a fixed horizon. Within an episode, each action influences not only the immediate reward, but also the future states and rewards. The **discount factor** $\gamma \in [0, 1]$ controls the trade-off between short-term and long-term objectives, with higher values favoring distant rewards. Formally, the performance of a policy π is:

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \right]. \tag{1.2}$$

1.2 Learning paradigms and policy evaluation

In RL, agents learn from experience using **Monte Carlo**(MC) or **Temporal Difference**(TD) methods [4]. MC provides **unbiased** estimates from complete episode returns, ideal for **episodic tasks** but prone to **high variance** and **delayed updates** [5, 6]. TD updates **incrementally** via **bootstrapping**, blending immediate rewards with future estimates for greater **sample efficiency** and real-time stability, at the cost of some **bias**.

Policy evaluation can be **on-policy** or **off-policy** [7]. On-policy updates from the *current policy* ensure stable convergence but waste past data, reducing the efficiency of **sample**. Off-policy reuses episodes from other policies (e.g., via replay buffers) to boost **data efficiency** and exploration, but may suffer from **bias**, **variance**, and instability, issues mitigated by importance sampling, trust region constraints, or entropy regularization. The choice reflects trade-offs between **bias/variance** and **stability/efficiency**.

1.3 Balancing exploration and exploitation

In RL, agents face the **exploration–exploitation trade-off**: they must **explore** unknown sequence of actions to gain information while **exploit** known rewarding actions to maximize cumulative reward [4, 8]. During training, exploration dominates to gather diverse experiences, shifting toward exploitation as knowledge improves. Common strategies include ϵ**-greedy**, which selects a random action with probability ϵ and the best known action otherwise, typically decreasing ϵ over time, and **entropy regularization**, which encourages randomness of the policy in gradient-based methods to prevent premature convergence to suboptimal solutions [4, 9]. Properly balancing these mechanisms improves both learning efficiency and policy quality in complex environments.

1.4 RL Strategies: policy-based vs. value-based

In RL, two main strategies guide the search for the optimal policy π^*. **Policy-based** methods learn a direct mapping from states to actions, while **value-based** methods estimate a **value function** for each state and choose actions that lead to a higher estimated value.

1.4.1 Value-based methods

Value functions In value-based RL, the objective is to assign a numerical value to states or state–action pairs, reflecting the **expected cumulative return** when starting from a given state (or state–action pair) and following a policy π. The **return** G_t denotes the discounted sum of future rewards from the timestep t onward $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$.

The **state value function** $V^\pi(s)$ is the expected return from state s under policy π:

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t \,|\, s_t = s \right]. \tag{1.3}$$

The **action value function** $Q^\pi(s, a)$ extends this definition to state–action pairs.

These functions are related through: $V^{\pi}(s) = \mathbb{E}_{a\sim\pi(\cdot|s)}\left[Q^{\pi}(s,a)\right]$, where the notation $a \sim \pi$ means that action a is chosen according to policy π.

Finding the **optimal policy** π^* is often done using the optimal value functions: $V^*(s) = \max_\pi V^\pi(s)$ or $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ [10].

Bellman equations

In MDP, the Bellman equations relate the value of a state (or a state–action pair) to the value of its successors, reducing computational redundancy [11, 1].

Therefore, for a policy π, the value functions can be defined as:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\big[r_{t+1} + \gamma V^{\pi}(s_{t+1}) \,\big|\, s_t = s\big], \tag{1.4}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\Big[r_{t+1} + \gamma\, \mathbb{E}_{a_{t+1}\sim\pi}\big[Q^{\pi}(s_{t+1}, a_{t+1})\big] \,\Big|\, s_t = s,\, a_t = a\Big]. \tag{1.5}$$

Value function estimation

When the dynamics of the environment are unknown, the value functions are estimated from the sampled data using MC or TD methods.

MC methods estimate the state values using the total discounted return G_t of complete episodes:

$$V(s_t) \leftarrow V(s_t) + \alpha\left(G_t - V(s_t)\right), \tag{1.6}$$

TD methods incrementally update the value function using the TD-error (δ_t), which is based on the Bellman equations (1.5). The value function is then updated as:

$$V(s_t) \leftarrow V(s_t) + \alpha\, \delta_t \qquad \text{with} \qquad \delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{1.7}$$

The updates are controlled by a **learning rate** α, which balances the speed of learning and the stability of convergence.

Example of value-based algorithms

Two common TD value-based algorithms are **SARSA** (State-Action-Reward-State-Action) and **Q-learning**. SARSA is **on-policy** and updates the state action value using the TD-error along the experienced trajectory:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big]. \tag{1.8}$$

Q-learning is **off-policy** and updates towards the maximal next-state value:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big]. \tag{1.9}$$

In practice, both algorithms typically select actions according to an ϵ-greedy policy: with probability $1-\epsilon$ the agent exploits by choosing the action with the highest estimated Q-value, and with probability ϵ it explores by sampling a random action, thus maintaining a balance between exploration and exploitation. The key distinction is that SARSA updates its value function using the action actually taken by the agent (on-policy) 1.8, while Q-learning updates as if the greedy action had been selected (off-policy), regardless of the agent's exploratory choice 1.9.

1.4.2 Policy-based methods: direct policy optimization

Policy-based methods aim to learn the optimal policy π^* **directly**, without relying on an intermediate value function. They optimize a parameterized policy π_θ to maximize the expected cumulative reward:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\sum_{t=0}^{\infty} \gamma^t r_{t+1}\Big]. \tag{1.10}$$

These methods are divided into two families: **gradient-based**, which update policy parameters using the gradient $\nabla_\theta J(\theta)$ and are effective in high-dimensional or continuous action spaces, and **gradient-free**, which rely on heuristics such as random search or evolutionary strategies when gradient information is noisy or unavailable [7].

Policy gradient methods

The evolution of the policy is based on the objective $J(\theta)$, which represents the expected return over a trajectory τ, the sequence of state-action pairs, $\tau = ((s_t, a_t), (s_{t+1}, a_{t+1}), \ldots, (s_T, a_T))$, generated by the policy π_θ:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[G_t], \quad G_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}. \tag{1.11}$$

Equivalently, summing over all trajectories weighted by their probability:

$$J(\theta) = \sum_{\tau} P(\tau;\theta)\, G_t, \tag{1.12}$$

$$P(\tau;\theta) = \prod_{t=0}^{T} P(s_{t+1} \mid s_t, a_t)\, \pi_\theta(a_t \mid s_t), \tag{1.13}$$

where $P(s_{t+1} \mid s_t, a_t)$ is the transition probability of the environment and $\pi_\theta(a_t \mid s_t)$ is the probability that the agent selects the action a_t from state s_t given our policy [8].

To maximize $J(\theta)$ Policy gradient methods perform gradient ascent with a learning rate α:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta), \tag{1.14}$$

Using the policy gradient theorem, the gradient can be estimated as

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t\right]. \tag{1.15}$$

Example of a policy-gradient algorithm

REINFORCE [12] is a straightforward MC policy-gradient algorithm that samples complete episodes and updates the policy parameters based on the observed returns G_t. The algorithm 1 presents its pseudocode.

Algorithm 1: REINFORCE Algorithm

```
  Input: Stochastic policy $\pi_\theta(a \mid s)$, number of episodes $N$, learning rate $\alpha$
  Output: Optimized policy parameters $\theta$
1 Initialize $\theta$;
2 for $i = 1$ to $N$ do
3     Generate a complete episode $s_0 \to a_0 \to r_1 \to \cdots \to s_T$ following $\pi_\theta$;
4     for $t = 0$ to $T - 1$ do
5         Compute return: $G_t = \sum_{k=t+1}^{T} \gamma^{k-t} r_k$;
6         Update parameters: $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t$;
7 return $\theta$
```

1.4.3 Actor-Critic methods

Actor–Critic methods [13] combine policy-based and value-based approaches to exploit the strengths of both: the Actor selects actions according to a parameterized policy, while the Critic evaluates these actions with a value function, providing a low-variance learning signal to improve the policy efficiently.

Example of an actor–critic algorithm with Q-value estimation: At each step, the actor selects an action according to the policy π_θ, while the critic evaluates it using the action value function Q_w. The Critic is updated incrementally via the TD error, and the Actor is updated using the policy gradient. This interaction enables learning from partial episodes, enhancing sample efficiency and stability compared to pure Monte Carlo methods.

Algorithm 2: Actor-Critic with Q-value Estimation

```
  Input: Initial $\theta, w$; learning rates $\alpha_\theta, \alpha_w$; discount $\gamma$; episodes $N$
  Output: Optimized $\theta, w$
1 for $i = 1$ to $N$ do
2     Initialize $s_0$;
3     repeat terminal until
4         Sample $a_t \sim \pi_\theta(\cdot \mid s_t)$ and observe $r_{t+1}, s_{t+1}$;
5         Sample $a_{t+1} \sim \pi_\theta(\cdot \mid s_{t+1})$;
6         $\delta_t \leftarrow r_{t+1} + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t)$;
7         $w \leftarrow w + \alpha_w \, \delta_t \, \nabla_w Q_w(s_t, a_t)$;
8         $\theta \leftarrow \theta + \alpha_\theta \, \nabla_\theta \log \pi_\theta(a_t \mid s_t) \, Q_w(s_t, a_t)$;
9 return $\theta, w$
```

1.5 Deep reinforcement learning

Classical RL methods, such as tabular Q-learning, store value functions $V(s)$ or action value functions $Q(s, a)$ in lookup tables, associating each state or state–action pair with a numerical value. This is feasible only for small, discrete spaces and fails in most real-world problems, including robotics, complex video games, and autonomous driving, where states are continuous, high-dimensional, and often derived from unstructured sources such as images or sensor data.

DRL uses deep networks $f_\theta(x)$ with weights θ to approximate state-value functions $V^\pi(s) \approx V_\theta(s)$, action-value functions $Q^\pi(s, a) \approx Q_\theta(s, a)$, or policies $\pi(a \mid s) \approx \pi_\theta(a \mid s)$. By the universal approximation theorems [14, 15], a network with at least one hidden layer and sufficient neurons can approximate any continuous function in a compact domain, allowing agents to learn complex and generalizable behaviors in rich state spaces.

A key example is the *Deep Q-Network (DQN)* [16], which uses convolutional networks to estimate $Q(s, a)$ directly from raw image frames in Atari games, automatically extracting relevant features.

In summary, the "deep" in DRL shifts RL from symbolic representations to distributed representations, allowing scalable, generalizable learning in complex domains of the real world.

2 Detailed study of selected algorithms

This chapter presents a brief overview of the reinforcement learning algorithms used in the internship. All selected algorithms belong to the class of model-free methods and are capable of handling continuous state and action spaces. Their implementations are based on the Stable Baselines library [17], an improved and actively maintained version of the original OpenAI Baselines repository [18].

2.1 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [19] is a model-free, on-policy reinforcement learning algorithm that balances performance and stability. It uses an actor-critic framework with an **advantage function** for critics and with a **clipped objective** to limit large policy updates, ensuring stable learning. PPO performs particularly well in continuous action spaces and is valued for its simplicity, reliability, and strong empirical results.

The network architecture uses a shared backbone with separate heads for the policy (actor) and value function (critic). Sharing layers enables efficient state representation, reduces overfitting, and stabilizes training.

The PPO objective function or loss combines three components: the clipped surrogate policy loss L^{CLIP}, the loss of the value function L^{VF}, and an entropy bonus $S[\pi_\theta]$ for exploration. These elements thus then the advantage function will be defined later. The total loss is written as:

$$L^{\text{total}}(\theta) = \mathbb{E}_t\Big[L^{\text{CLIP}}(\theta) - c_1 L^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\Big], \tag{1.16}$$

where c_1 and c_2 are scalar coefficients controlling the contributions of the value and entropy terms [19].

2.1.1 Clipped surrogate policy loss

Large policy updates can destabilize training. PPO addresses this problem with a probability ratio between the current and the old policy $r_t(\theta)$ (1.17) and a clipped surrogate loss: $L^{CLIP}(\theta)$ (1.18).

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \tag{1.17}$$

$$L^{CLIP}(\theta) = \mathbb{E}_t\Big[\min\big(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\big)\Big] \tag{1.18}$$

where $\hat{A}_t$ an estimator of the Advantage Function and where ϵ defines the trust region to ensure safe policy updates. This mechanism limits too large policy changes and stabilizes training.

Advantage function

The advantage function A_t quantifies how much better taking an action a_t in state s_t is compared to the expected value of that state: $A_t = Q(s_t, a_t) - V(s_t)$

In practice, PPO uses *Generalized Advantage Estimation* (GAE) [19] to calculate A_t efficiently while balancing bias and variance, using TD-error $\delta_t = r_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t)$

$$A_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \tag{1.19}$$

where γ is the discount factor and $\lambda \in [0, 1]$ controls the bias-variance trade-off.

In practice, advantages are computed on a batch of trajectories to give an estimator $\hat{A}_t = \frac{A_t - \text{mean}(A)}{\text{std}(A)+\epsilon}$ of the advantage function which ensures consistent scaling for gradient updates.

Clipping mechanism and stability

The clipping in $L^{CLIP}(\theta)$ constrains policy updates to a safe region, preventing excessively large changes that could destabilize learning. Table 1.1 [20] summarizes how the clipped term is applied depending on the advantage A_t and the probability ratio $r_t(\theta)$. This mechanism ensures smooth, incremental policy improvements and guards against destructive updates.

Advantage A_t	**Ratio $r_t(\theta)$**	**Clipped Term Used**
$A_t > 0$	$r_t(\theta) > 1 + \epsilon$	$(1 + \epsilon)A_t$
$A_t < 0$	$r_t(\theta) < 1 - \epsilon$	$(1 - \epsilon)A_t$
Else	Any	$r_t(\theta)A_t$

Table 1.1: Behavior of the clipped objective under different advantage and ratio conditions

2.1.2 Value loss :

The value loss $L^{VF}(\theta)$ measures the mean-squared error between predicted state values and observed returns:

$$L^{VF}(\theta) = (V_\theta(s_t) - G_t)^2 \tag{1.20}$$

where G_t is the empirical return starting from s_t. A gradient descent is applied to train the Critic, minimizing the difference between predicted and actual returns to provide reliable value estimates that stabilize policy updates.

2.1.3 Entropy bonus:

The entropy term $S[\pi_\theta](s_t)$ encourages exploration by promoting uncertainty in the policy's action selection. Maximizing this term discourages the policy from becoming overly deterministic too early, thus improving exploration and preventing premature convergence to suboptimal strategies. The coefficient c_2 in the total loss controls the strength of this regularization.

The pseudocode of the PPO algorithm is summarized in Algorithm 3.

Algorithm 3: Proximal Policy Optimization (Actor-Critic) [19]

```
for iteration = $1, 2, \ldots$ do
    for actor = $1, \ldots, N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in the environment for $T$ timesteps;
        Store $(s_t, a_t, r_{t+1}, s_{t+1})$ in buffer;
        Compute advantages $\hat{A}_1, \ldots, \hat{A}_T$ using GAE;
    for epoch = $1, \ldots, K$ do
        Shuffle buffer and split into minibatches of size $M$;
        foreach minibatch do
            Compute policy ratio $r_t(\theta)$;
            Evaluate clipped policy loss $L^{\text{CLIP}}(\theta)$;
            Evaluate value loss $L^{VF}(\theta)$;
            Evaluate entropy bonus $S[\pi_\theta]$;
            Update $\theta$ via gradient ascent on $L^{\text{total}}(\theta)$;
    $\theta_{\text{old}} \leftarrow \theta$;
```

2.2 Soft Actor-Critic: detailed derivation and equations

The **Soft Actor-Critic** (SAC) algorithm [21] is an **off-policy**, **actor–critic** method specifically designed for continuous action spaces. It demonstrates high sample efficiency and is particularly effective in tasks that require adaptability, enhancing exploration and performance in complex, dynamic environments.

The pseudo-code of the SAC algorithm is summarized in algorithm 4.

2.2.1 Maximum entropy reinforcement learning framework

SAC is built upon the **maximum entropy reinforcement learning** framework, which augments the standard RL objective with an entropy term. This encourages exploration

Algorithm 4: Soft Actor-Critic (SAC) [21, 22]

```
1  Initialize policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, and empty replay buffer $\mathcal{D}$;
2  Set target network parameters: $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$;
3  for each iteration do
4      for each environment step do
5          Sample action $a_t \sim \pi_\theta(\cdot \mid s_t)$;
6          Execute $a_t$ and observe reward and next state $(r_{t+1}, s_{t+1}, d_t)$;
7          Store transition $(s_t, a_t, r_{t+1}, s_{t+1}, d_t)$ in $\mathcal{D}$;
8      if it's time to update then
9          for each gradient step do
10             Sample a batch $B$ of transitions from $\mathcal{D}$;
11             Sample $\tilde{a}_{t+1} \sim \pi_\theta(\cdot \mid s_{t+1})$;
12             Compute the target value $y(r_{t+1}, s_{t+1}, d_t)$;
13             Update critics $\phi_1, \phi_2$ with (1.28) via gradient descent;
14             Sample $\tilde{a}_t \sim \pi_\theta(\cdot \mid s_t)$;
15             Update policy $\theta$ with (1.29) via gradient ascent;
16             Update target networks $\phi_{\text{targ},1}, \phi_{\text{targ},2}$ with (1.27);
```

while learning a high-performance policy. The objective function is therefore defined as:

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \big(r_{t+1} + \alpha\, \mathcal{H}(\pi(\cdot \mid s_t)) \big) \right], \tag{1.21}$$

where the entropy $\mathcal{H}$ is given by:

$$\mathcal{H}(\pi(\cdot|s)) = -\mathbb{E}_{a\sim\pi}[\log \pi(\cdot|s)], \tag{1.22}$$

and $\alpha > 0$ is the temperature parameter that balances exploration and exploitation [21, 22].

The entropy-regularized state value function is defined as:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi}\left[Q^\pi(s,a) - \alpha \log \pi(a|s)\right], \tag{1.23}$$

and the corresponding entropy-augmented action value function, derived from the Bellman equation, is:

$$Q^\pi(s,a) = \mathbb{E}_\pi \Big[r_{t+1} + \gamma\, \mathbb{E}_{a_{t+1}\sim\pi} \big[Q^\pi(s_{t+1}, a_{t+1}) - \alpha \log \pi(a_{t+1}|s_{t+1}) \big] \bigm| s_t = s, a_t = a \Big]. \tag{1.24}$$

2.2.2 Action sampling via the reparameterization trick

Actions are generated via a differentiable transformation of Gaussian noise $\xi \sim \mathcal{N}(0, I)$.

$$\tilde{a}_t = a_\theta(s,\xi) = \tanh\left(u_\theta(s,\xi)\right), \quad \text{with} \quad u_\theta(s,\xi) = \mu_\theta(s) + \sigma_\theta(s) \odot \xi, \tag{1.25}$$

where $\mu_\theta(s)$ and $\sigma_\theta(s)$ are the outputs of the policy network. Thus, $\tilde{a}_t$ denotes an action **sampled from the current policy** π_θ at the state s_t. The tanh function ensures that actions remain within the valid bounds of the environment while preserving differentiability for backpropagation.

2.2.3 Critics and actor updates

SAC maintains two Q-function approximators, Q_{ϕ_1} and Q_{ϕ_2}, to mitigate overestimation bias [21]. The target value is based on the TD-error and uses the target networks $\phi_{\text{targ},1}$ and $\phi_{\text{targ},2}$:

$$y(r_{t+1}, s_{t+1}, d_t) = r_{t+1} + \gamma(1-d_t)\cdot\left[\min_{j=1,2} Q_{\phi_{\text{targ},j}}(s_{t+1}, \tilde{a}_{t+1}) - \alpha \log \pi_\theta(\tilde{a}_{t+1}|s_{t+1})\right], \tag{1.26}$$

where d_t is the terminal flag and $\tilde{a}_{t+1} \sim \pi_\theta(\cdot|s_{t+1})$ is the action sampled via the reparameterization trick.

The parameters $\phi_{\text{targ},i}$ correspond to slowly updated copies of the critic networks, known as *target networks* and ensure that the regression target changes gradually between updates.

To achieve this, SAC applies *Polyak averaging*:

$$\phi_{\text{targ},i} \leftarrow \rho\,\phi_{\text{targ},i} + (1-\rho)\,\phi_i, \tag{1.27}$$

The **critic loss** for each $i \in \{1,2\}$ is:

$$L(\phi_i) = \mathbb{E}_{(s_t,a_t,r_{t+1},s_{t+1},d_t)\sim\mathcal{D}}\left[\left(Q_{\phi_i}(s_t,a_t) - y(r_{t+1},s_{t+1},d_t)\right)^2\right], \tag{1.28}$$

where $\mathcal{D}$ is the replay buffer storing past transitions. The critic parameters ϕ_i are updated to minimize the difference between the predicted and target Q-values.

The **policy (actor) loss** is:

$$L_\pi(\theta) = \mathbb{E}_{s_t\sim\mathcal{D},\tilde{a}_t\sim\pi_\theta(\cdot|s_t)}\left[\min_{j=1,2} Q_{\phi_j}(s_t,\tilde{a}_t) - \alpha\log\pi_\theta(\tilde{a}_t|s_t)\right]. \tag{1.29}$$

The policy parameters θ are updated to maximize this entropy-augmented expected Q-value.

Chapter 2

Simulator implementation

This chapter presents the development of a high-fidelity physics simulation framework for training autonomous underwater vehicles using reinforcement learning. It covers simulator selection and adaptation, integration with modern robotics software, and the design of the BlueROV2 RL environment, including state representation, rewards, and training protocols. Together, these elements provide a robust foundation for effective simulation-based underwater vehicle control.

1 High-fidelity physics-based simulation environments

This section first reviews the scientific rationale for simulation-based reinforcement learning (RL) in underwater robotics and summarizes prior laboratory research. Then, it details the adaptations required to update legacy simulation frameworks for modern software environments. Subsequently, a comparative evaluation of leading robotics simulators is presented, followed by a description of the BlueROV2 simulation model in development and an overview of system integration alongside encountered technical challenges.

1.1 Review of prior laboratory research

One of the initial tasks during this project was to analyze previous work conducted by laboratory members on RL applied to underwater vehicles, encompassing both simulation and real-world experiments. The research of T. Chaffre [1], K. Lagattu [23], and Y. Sola [24] laid the foundation employing a modular simulation framework based on ROS middleware, the Gazebo physics simulator extended with the `uuv_simulator` package, and Python-based control and experimentation layers, illustrated in Figure 2.1. ROS facilitates communication through independent nodes that exchange data through publish/subscribe topics and synchronous services. This modularity ensures that control algorithms developed in simulation are seamlessly transferred to physical platforms.

The Gazebo simulator manages hydrodynamic effects and rigid-body dynamics consistent with Fossen's 6-DoF marine vehicle models, with *uuv_simulator* providing underwater-specific extensions such as configurable thruster layouts, ocean current modeling, realistic sensor emulations, and fault injection. Python scripts orchestrate high-level experiment control and interact readily with machine learning libraries for RL training loops.

This hardware-in-the-loop capability architecture allowed Chaffre to transfer SAC-based controllers from simulation to a RexROV2-like platform, and Lagattu to port DRL-based fault-tolerant control policies from simulated BlueROV2 heavy models to real underwater tank tests.

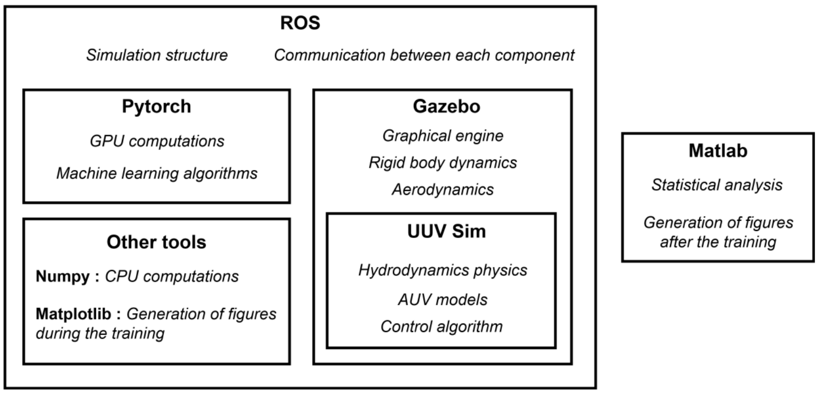


Figure 2.1: Simulation architecture adapted from Y. Sola [24].

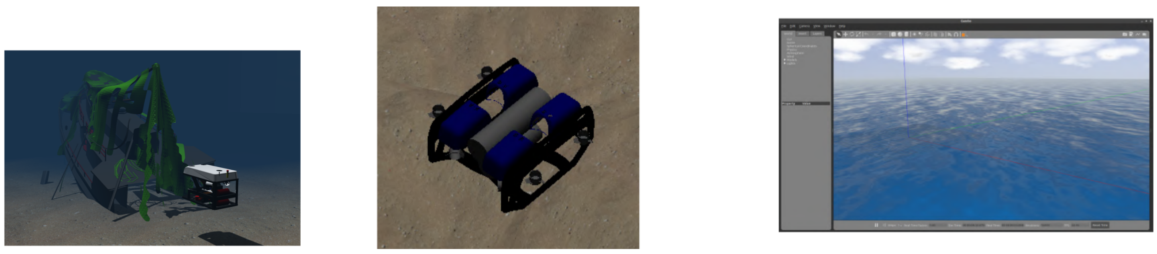


Figure 2.2: Gazebo with UUV Simulator environments: RexRov by T. Chaffre (left) and BlueROV Heavy by K. Lagattu (right).

1.2 Technological advances and framework adaptation

Given the four-year gap since these previous studies, significant changes in the software ecosystem have occurred, including the advent of ROS 2, which is incompatible with Ubuntu versions older than 22.04. Contemporary systems predominantly employ Ubuntu 24.04 or later, necessitating updates to simulation architectures.

An initial approach using Docker containers to preserve legacy ROS and Ubuntu environments encountered practical difficulties due to complex dependency resolution and limited access to external resources. Consequently, this project prioritized adapting the simulation framework to the modern Ubuntu and ROS 2 versions for sustainable development.

1.3 Comparative evaluation of contemporary robotics simulators

To select an optimal simulator compatible with my hardware and research objectives, training an analogous AUV to the BlueROV2 with RL, I reviewed the leading robotic simulators (Table 2.1). Since BlueROV2 real-world control is based on ROS topics, maintaining ROS 2 integration was crucial.

Industry consultations identified Isaac Sim as the prevalent RL simulator for its NVIDIA GPU acceleration, but its transition to the RL-focused Genesis platform lacks underwater simulation capabilities. MarineGym offers underwater RL environments but is not yet available.

Therefore, I selected Gazebo Harmonic, compatible with Ubuntu 24.04 and ROS 2 Jazzy, as the best compromise, despite its lack of native underwater modules, relying instead on plugin-based extensions.

Simulator	Hydrodynamics / Underwater Capabilities	ROS 2 Compatibility	Notes
MarineGym / Genesis MarineGym (active 2023–2025), Genesis v0.1+ (2023)	MarineGym offers dedicated marine and underwater RL simulation with explicit hydrodynamics and thruster modeling. Genesis is related research software supporting marine robotics but lacks clear native underwater hydrodynamics simulation features.	Partial; evolving via custom ROS 2 bridges	RL-focused underwater simulators with small but growing communities; ecosystems are less mature than Gazebo.
Unity (Unity Robotics) Unity 2024.1 LTS	No native hydrodynamics; can be added via third-party plugins or custom physics; underwater scenes possible with extended modelling	Yes, via Unity Robotics Hub	High visual realism and flexible environment creation; ideal for vision-based robotics; underwater use requires external assets.
Isaac Sim / Isaac Gym Isaac Sim 2024.1	PhysX 5 supports rigid-body dynamics and basic fluids; lacks specialized underwater hydrodynamic models	Possible via ROS 2 bridge / Isaac ROS	GPU-accelerated for manipulation and navigation; marine scenarios require heavy customization.
Gazebo (Ignition & Harmonic) Harmonic (2025 LTS), Garden (2024), Fortress (2023)	Supports underwater vehicle simulation via external plugins such as `uuv_simulator`. Compatibility matures on Fortress/Garden; Harmonic requires plugin adaptation.	Fully supported via the `ros_gz` bridge	ROS 2-native successor to Gazebo Classic; modular architecture; LTS stability in Harmonic. The Underwater features are plugin-based.
Webots R2024a (2024)	Basic hydrodynamics; limited underwater support without extensions	Yes, via dedicated ROS 2 interface	User-friendly with fast setup; suited for education and rapid prototyping; limited marine physics fidelity.

Table 2.1: Comparison of widely used robotic simulators for scientific research (2025), focusing on hydrodynamics/underwater capabilities and ROS 2 integration.

1.4 Development of a blueROV2 underwater simulation model

Given that Gazebo Harmonic lacks a native BlueROV2 model, I adapted an open-source ROS 2-integrated BlueROV2 environment [25] developed by Centrale Nantes. Although this implementation employs an earlier ROS 2 architecture and models the BlueROV2 Classic, our laboratory's BlueROV2 Heavy differs by its eight-thrust configuration, increased stability, greater payload capacity and reinforced frame, affecting dynamics and control.

However, this repository provided a functional simulation environment (Figure 2.3) with comprehensive actuator control, forming a suitable base for extension to the Heavy platform.

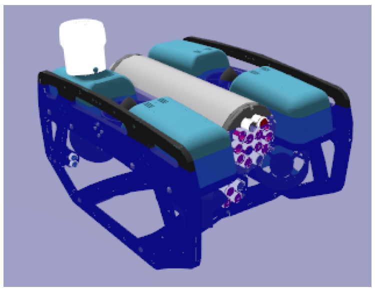


Figure 2.3: Gazebo Harmonic BlueROV2 underwater simulation environment.

1.5 System integration and technical challenges

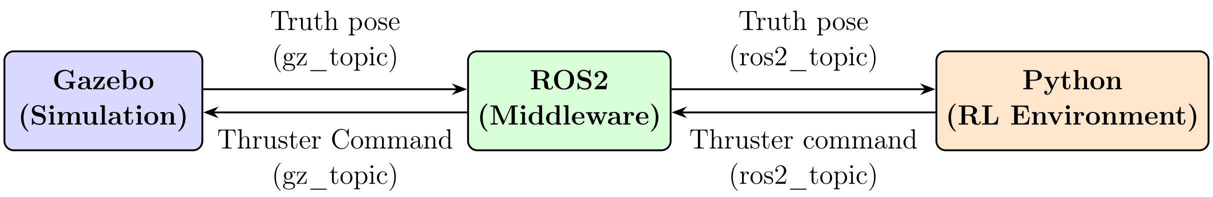


Figure 2.4: Architecture linking Gazebo Harmonic, ROS 2, and RL control.

Integrating Gazebo Harmonic, ROS 2 middle ware, and Python-based RL frameworks was a primary challenge (Figure 2.4). The `ros_gz_bridge` package enables bidirectional communication between Gazebo and ROS 2 topics, while Python algorithms interact with ROS 2 via publishers and subscribers. Ground-truth pose data derives from a ROS 2 node adapted from the Centrale Nantes repository [25]; Location in the real world is based on inertial sensors.

Two implementation challenges arose. Initially lacking a robust reset service, I implemented a PID controller to return the robot to its starting position after each RL episode, which proved inefficient due to RL's computational demands. Subsequently, I developed a `ros_gz_bridge` extension for Gazebo services, enabling a reliable and efficient reset function.

Furthermore, integrating the asynchronous ROS 2 architecture with the synchronous RL training loop (e.g., with Stable Baselines 3) required abandoning the conventional continuous ROS control node. Instead, ROS 2 publishers and subscribers are instantiated and invoked on demand from Python scripts, which ignore the multithread benefit of ROS, but provides a practical solution for robot RL control in Gazebo Harmonic.

2 Reinforcement learning simulation framework

Building on the foundations of high-fidelity physics simulation and system integration, this section details the design and configuration of the reinforcement learning (RL) environment specifically developed for autonomous underwater vehicle control.

2.1 Development and implementation of the blueROV2 RL environment

2.1.1 Core environment design

The RL environment simulates the underwater context in which BlueROV2 operates and interacts. It defines the state space, including the robot's position, orientation, and sensor feedback, the continuous action space consisting of thruster command inputs, and the reward function guiding the learning process. Through iterative interactions comprising observation, action execution, and reward feedback, the agent progressively refines its control policy. Precise modeling of underwater dynamics and sensor characteristics within this environment is essential to develop robust controllers transferable to real vehicles.

The environment is implemented as a Gym compatible interface by subclassing `Gym.Env`, following the requirements of the Stable Baselines3 framework. This implementation involves specifying the observation and action spaces, the `step` function for environment transition, the `reset` function for episode initialization, and the reward calculation mechanism. The continuous action space corresponds to six thruster commands:

$$A = \{u_i \quad \text{for} \quad i \in [1, 6]\}$$

The observation space is derived from Y. Sola's work [24] optimized for waypoint tracking tasks:

$$S_t = \{\psi_e, \mathbf{x}_e, u_{t-1}\}$$

where ψ_e denotes the tracking error of the yaw angle, $\mathbf{x}_e$ is the positional error vector between the current and target positions, and u_{t-1} represents the action vector of the previous step.

The `step` function applies thruster commands, allows the robot dynamics to act, then obtains updated states from Gazebo's truth pose node, outputting the new observations, reward, termination status, and diagnostic information. The `reset` function re-initializes the robot pose at the start of the episode. Reward and termination criteria are tailored to mission specifications.

2.1.2 Mission-specific environment setup

The mission replicates the waypoint tracking scenario described by Y. Sola [24]. Although AUVs serve multiple purposes, such as path planning, obstacles avoidance, and station keeping, this study focuses on tracking waypoint within simulation due to testing opportunities in a limited real world

Each episode requires the vehicle to reach a randomly assigned 3D waypoint within a bounded region. Episodes are limited to 1000 time steps to ensure practical training duration.

Initialization sets the AUV position to $\mathbf{x} = [0, 0, -20]$ meters in the Gazebo frame, with the yaw randomly sampled from $[0, 360°]$, and roll and pitch set to zero. The target waypoint is uniformly sampled within a 3D box centered on the initial position:

- Vertical bounds: $[-60, -1]$ meters.
- Horizontal bounds (X, Y): $[-20, 20]$ meters.

Episodes end successfully when the AUV reaches within 3 meters of the waypoint without breaching vertical limits. Violations of vertical boundaries or exceedance of timestep limits count as failures (collision or timeout).

To mimic real-world noise, sensor readings are perturbed by uniform noise in the $[0.05, 0.1]$ range, while thruster commands are similarly affected by noise in $[0.01, 0.05]$.

The reward function r_t, inspired by Y. Sola [24] and [26], incorporates position-based criteria:

$$r_t = \begin{cases} r_{\text{waypoint}} & \text{if } d_t < \epsilon \quad \text{(success)} \\ r_{\text{collision}} & \text{if } z \notin [z_{\min}, z_{\max}] \quad \text{(collision)} \\ r_{\text{toward}} & \text{if } d_t < d_{t-1} \\ r_{\text{backward}} & \text{if } d_t \geq d_{t-1} \end{cases} \tag{2.1}$$

where d_t is the distance to the waypoint and the vertical limits and threshold are $z_{\min} = -60\,m$, $z_{\max} = -1\,m$, $\epsilon = 3\,m$.

Components of the reward function include:

- $r_{\text{waypoint}} = 500$: Positive reward for reaching the waypoint and ending the episode successfully.
- $r_{\text{collision}} = -550$: Negative reward for breaching vertical bounds, which ends the episode with failure.
- $r_{\text{toward}} = K_r \times \exp(-d_t/20)$: Variable positive reward for decreasing distance, encouraging progress.
- $r_{\text{backward}} = -10$: Negative reward for stagnation or moving away from the waypoint.

This reward structure incentivizes efficient and safe navigation toward the target while penalizing undesirable behaviors such as boundary violations or regression.

Figure 2.5 illustrates the detailed flow of the reinforcement learning cycle used for BlueROV2, highlighting key steps such as environment reset, action selection, state updates, reward calculation, and episode termination.

2.2 Reinforcement learning simulation phases

Training phase The RL agent is trained on episodes that replicate the waypoint tracking task (Section 2.1.2), each with a randomly located target waypoint. Training continues until one million time steps are completed and is managed via Stable Baselines3 interface with the custom environment. To ensure reproducibility in the stochastic nature of RL, a fixed random seed of 42 is used, standardizing task sequences between runs.

Testing phase Following training, models are evaluated in a 500 episodes testing phase, where the neural network parameters remain fixed. The testing loop is custom-implemented to allow fine control over the evaluation flow. A different fixed seed of 123 ensures varied yet reproducible scenarios. This phase assesses the agent's ability to generalize the policies learned beyond the training distribution.

Training utilizes automated library routines, whereas testing employs a manually defined control loop. Due to inertial and thruster dynamics of the underwater field, computational timing directly affects robot responsiveness; timing differences between phases may influence agent performance. Furthermore, distinct seeds for each phase balance reproducibility with diverse environmental exposure.

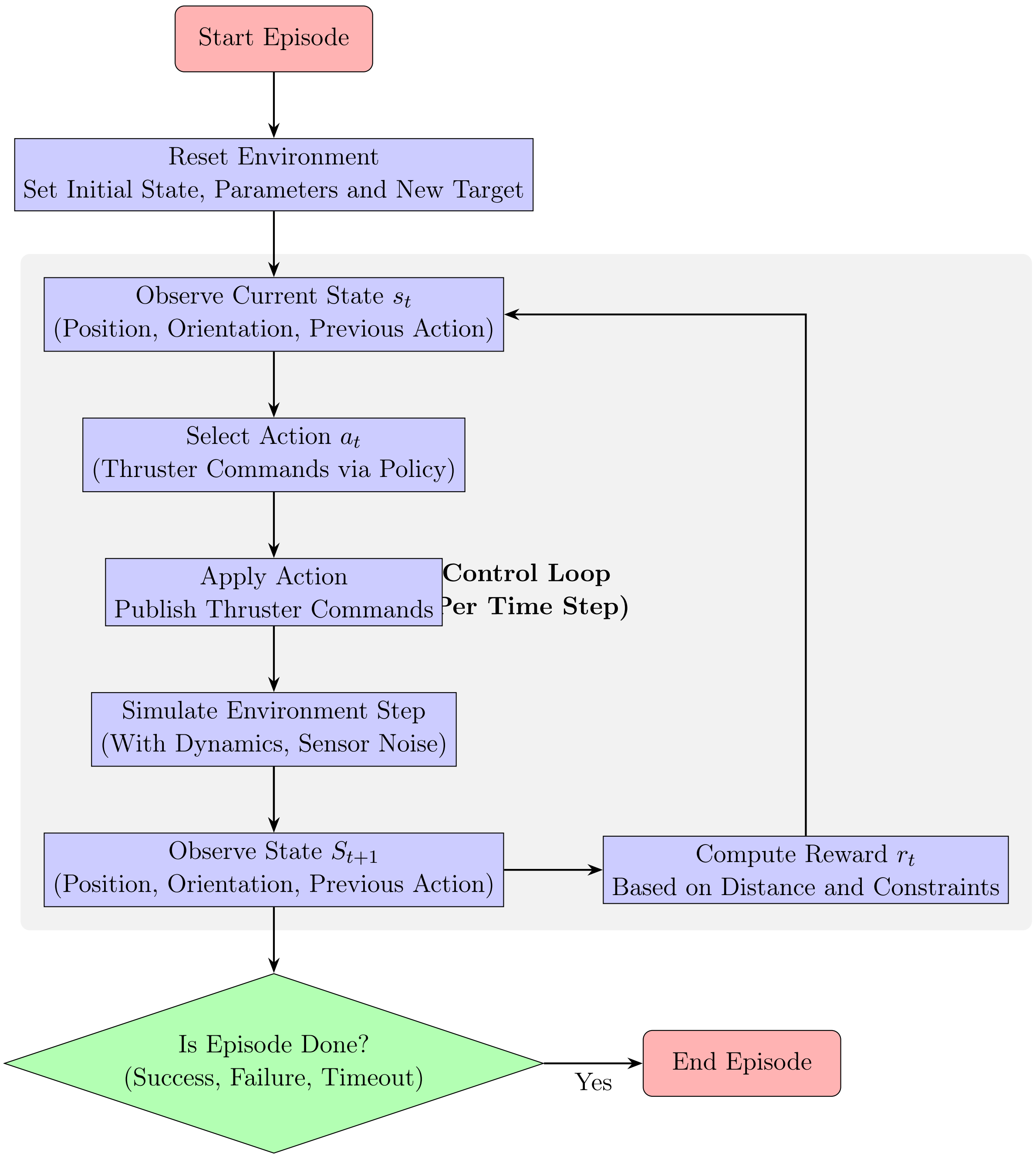
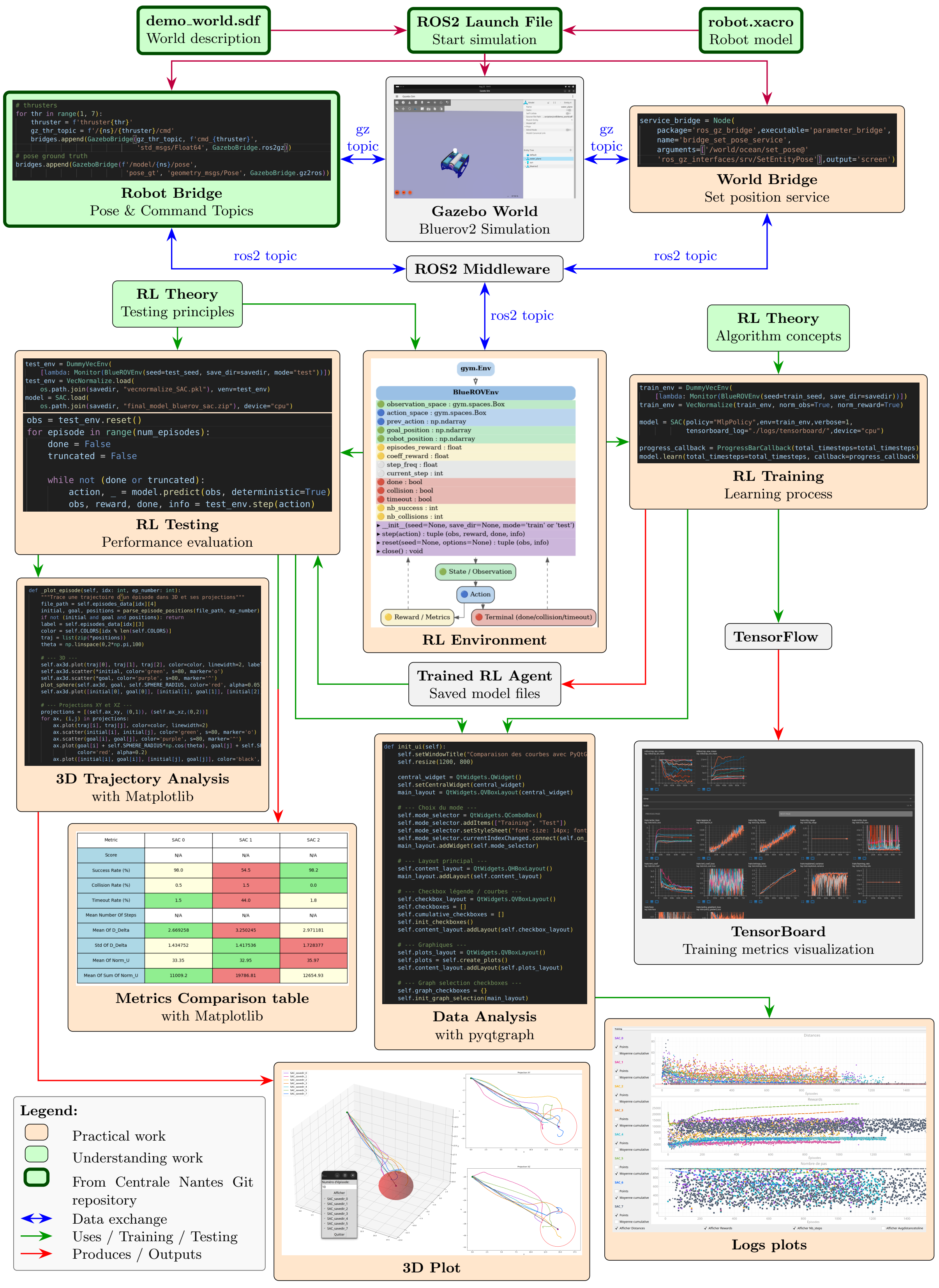

Start Episode
Reset Environment
Set Initial State, Parameters and New Target
Observe Current State s_t
(Position, Orientation, Previous Action)
Select Action a_t
(Thruster Commands via Policy)
Apply Action
Publish Thruster Commands
Control Loop
Per Time Step)
Simulate Environment Step
(With Dynamics, Sensor Noise)
Observe State S_{t+1}
(Position, Orientation, Previous Action)
Compute Reward r_t
Based on Distance and Constraints
Is Episode Done?
(Success, Failure, Timeout)
Yes
End Episode

Figure 2.5: Enhanced Reinforcement Learning Task Flow Diagram for BlueROV2

During both training and testing phases, observation and reward normalization are consistently applied to improve learning stability and performance. Normalizing observations standardizes diverse sensory inputs to a uniform scale, enhancing numerical stability and enabling faster, more reliable convergence. Reward normalization maintains a consistent feedback magnitude, preventing extreme values from destabilizing training and improving sample efficiency. Together, these normalizations mitigate environmental noise and scaling discrepancies, ensuring robust policy learning and better generalization across varied underwater scenarios.

Integrated Pipeline: Gazebo Simulation, ROS 2 Bridging, and RL Training for BlueROV2

The diagram 2.6 on the following page presents the complete simulation chain developed during this project. It brings together both the conceptual understanding and the practical implementation work carried out, while clearly illustrating the links between the different components of the chain. To enhance readability, the diagram is enriched with annotated screenshots, showing key code segments, the simulation rendering, and representative plots obtained from the experiments.



```python
# thrusters
for thr in range(1, 7):
    thruster = f'thruster{thr}'
    gz_thr_topic = f'/{ns}/{thruster}/cmd'
    bridges.append(GazeboBridge(gz_thr_topic, f'cmd_{thruster}',
                                'std_msgs/Float64', GazeboBridge.ros2gz))
# pose ground truth
bridges.append(GazeboBridge(f'/model/{ns}/pose',
                            'pose_gt', 'geometry_msgs/Pose', GazeboBridge.gz2ros))
```

```python
service_bridge = Node(
    package='ros_gz_bridge',executable='parameter_bridge',
    name='bridge_set_pose_service',
    arguments=['/world/ocean/set_pose@'
    'ros_gz_interfaces/srv/SetEntityPose'],output='screen')
```

```python
test_env = DummyVecEnv(
    [lambda: Monitor(BlueROVEnv(seed=test_seed, save_dir=savedir, mode="test"))])
test_env = VecNormalize.load(
    os.path.join(savedir, "vecnormalize_SAC.pkl"), venv=test_env)
model = SAC.load(
    os.path.join(savedir, "final_model_bluerov_sac.zip"), device="cpu")
```

```python
obs = test_env.reset()
for episode in range(num_episodes):
    done = False
    truncated = False

    while not (done or truncated):
        action, _ = model.predict(obs, deterministic=True)
        obs, reward, done, info = test_env.step(action)
```

```
BlueROVEnv
● observation_space : gym.spaces.Box
● action_space : gym.spaces.Box
● prev_action : np.ndarray
● goal_position : np.ndarray
● robot_position : np.ndarray
● episodes_reward : float
● coeff_reward : float
● step_freq : float
● current_step : int
● done : bool
● collision : bool
● timeout : bool
● nb_success : int
● nb_collisions : int
▸ __init__(seed=None, save_dir=None, mode='train' or 'test')
▸ step(action) : tuple (obs, reward, done, info)
▸ reset(seed=None, options=None) : tuple (obs, info)
▸ close() : void
```

```python
train_env = DummyVecEnv(
    [lambda: Monitor(BlueROVEnv(seed=train_seed, save_dir=savedir))])
train_env = VecNormalize(train_env, norm_obs=True, norm_reward=True)

model = SAC(policy="MlpPolicy",env=train_env,verbose=1,
            tensorboard_log="./logs/tensorboard/",device="cpu")

progress_callback = ProgressBarCallback(total_timesteps=total_timesteps)
model.learn(total_timesteps=total_timesteps, callback=progress_callback)
```

```python
def _plot_episode(self, idx: int, ep_number: int):
    """Trace une trajectoire d'un épisode dans 3D et ses projections"""
    file_path = self.episodes_data[idx][4]
    initial, goal, positions = parse_episode_positions(file_path, ep_number)
    if not (initial and goal and positions): return
    label = self.episodes_data[idx][3]
    color = self.COLORS[idx % len(self.COLORS)]
    traj = list(zip(*positions))
    theta = np.linspace(0,2*np.pi,100)

    # --- 3D ---
    self.ax3d.plot(traj[0], traj[1], traj[2], color=color, linewidth=2, labe
    self.ax3d.scatter(*initial, color='green', s=80, marker='o')
    self.ax3d.scatter(*goal, color='purple', s=80, marker='^')
    plot_sphere(self.ax3d, goal, self.SPHERE_RADIUS, color='red', alpha=0.05
    self.ax3d.plot([initial[0], goal[0]], [initial[1], goal[1]], [initial[2]

    # --- Projections XY et XZ ---
    projections = [(self.ax_xy, (0,1)), (self.ax_xz,(0,2))]
    for ax, (i,j) in projections:
        ax.plot(traj[i], traj[j], color=color, linewidth=2)
        ax.scatter(initial[i], initial[j], color='green', s=80, marker='o')
        ax.scatter(goal[i], goal[j], color='purple', s=80, marker='^')
        ax.plot(goal[i] + self.SPHERE_RADIUS*np.cos(theta), goal[j] + self.S
                color='red', alpha=0.2)
        ax.plot([initial[i], goal[i]], [initial[j], goal[j]], color='black',
```

Metric	SAC 0	SAC 1	SAC 2
Score	N/A	N/A	N/A
Success Rate (%)	98.0	54.5	98.2
Collision Rate (%)	0.5	1.5	0.0
Timeout Rate (%)	1.5	44.0	1.8
Mean Number Of Steps	N/A	N/A	N/A
Mean Of D_Delta	2.669258	3.250245	2.971181
Std Of D_Delta	1.434752	1.417536	1.728377
Mean Of Norm_U	33.35	32.95	35.97
Mean Of Sum Of Norm_U	11009.2	19786.81	12654.93

```python
def init_ui(self):
    self.setWindowTitle("Comparaison des courbes avec PyQtG
    self.resize(1200, 800)

    central_widget = QtWidgets.QWidget()
    self.setCentralWidget(central_widget)
    main_layout = QtWidgets.QVBoxLayout(central_widget)

    # --- Choix du mode ---
    self.mode_selector = QtWidgets.QComboBox()
    self.mode_selector.addItems(["Training", "Test"])
    self.mode_selector.setStyleSheet("font-size: 14px; font
    self.mode_selector.currentIndexChanged.connect(self.on_
    main_layout.addWidget(self.mode_selector)

    # --- Layout principal ---
    self.content_layout = QtWidgets.QHBoxLayout()
    main_layout.addLayout(self.content_layout)

    # --- Checkbox légende / courbes ---
    self.checkbox_layout = QtWidgets.QVBoxLayout()
    self.checkboxes = []
    self.cumulative_checkboxes = []
    self.init_checkboxes()
    self.content_layout.addLayout(self.checkbox_layout)

    # --- Graphiques ---
    self.plots_layout = QtWidgets.QVBoxLayout()
    self.plots = self.create_plots()
    self.content_layout.addLayout(self.plots_layout)

    # --- Graph selection checkboxes ---
    self.graph_checkboxes = {}
    self.init_graph_selection(main_layout)
```

Figure 2.6: End-to-End Framework for Training and Evaluating RL Agents on BlueROV2

Chapter 3

Experimentation

This chapter focuses on deepening the understanding of reinforcement learning (RL) algorithms beyond the implementation of the RL environment by investigating how various theoretical and simulation-related factors impact training efficiency and performance, particularly in underwater robotics. A key objective is to foster critical discussions on RL efficiency and to highlight important considerations that must be approached with caution. Although much of the literature introduces novel RL algorithms with advanced mathematical frameworks evaluated in predefined environments, practical applications frequently rely on common algorithms without a clear understanding of their limitations.

By linking theoretical insights with simulation experiments in an underwater robot context, this work identifies key parameters influencing training, establishes a systematic evaluation framework, and develops analysis tools to visualize and interpret results. Such efforts aim to provide deeper insights into RL dynamics in complex domains and guide future algorithmic and practical improvements.

1 Design of experimental evaluation framework

1.1 Selection and hypothesized impact of key influencing factors

As outlined in the planning section of the introduction, several factors that could influence reinforcement learning (RL) performance were identified through ongoing discussions with my tutors and consideration of practical challenges encountered during the internship. The differences in RL algorithms used by the laboratory compared to those with which I was familiar led us to hypothesize that distinct behaviors and performances might emerge when these algorithms are applied in our underwater simulation environment.

Algorithm selection: potential challenges in practical settings While previous comparative studies such as [19] and [21] provide theoretical and empirical insights on RL algorithms, these are often based on idealized settings. In contrast, the complexities of realistic underwater simulations may introduce nuances that alter algorithm performance, suggesting that the choice of RL algorithm could present unique challenges or unexpected outcomes in this context. Consequently, this work explores the benchmarking of two commonly used algorithms in robotics: PPO and SAC, with the aim of uncovering potential differences and pitfalls specific to the underwater domain.

Normalization and scaling effects in reward function design The adopted reward function (Equation 2.1)—originally implemented without modification—was later used with reward normalization. Despite this, it is hypothesized that the proportional coefficient K_r in the r_{toward} term could still strongly affect learning dynamics. Specifically, if the robot fails to reach the waypoint, the cumulative progress reward might reduce the relative impact of actually reaching the waypoint (r_{waypoint}), even after normalization. These considerations highlight that, while normalization helps address scale disparities, a careful balance between reward components (and particularly the choice of K_r) could be crucial for effective learning. This remains an open question warranting systematic exploration in the context of underwater RL.

1.2 Parameter exploration and comparative methodology

To analyze the identified factors, the experimental approach was structured in two main parts. The first part focuses on investigating the influence of specific parameters—namely, step duration and reward coefficient—within the context of the SAC algorithm. The second part extends the analysis to a comparative study between algorithms. All possible combinations of step duration and reward coefficient were tested using SAC, and the results were ranked and visualized according to metrics such as reward coefficient and step duration. Subsequently, the best-performing SAC and the initial configuration was compared against the PPO algorithm using the same parameter settings.

Although this methodology is not entirely optimal—since the best configuration for SAC may not correspond to the optimal settings for PPO—it facilitates a clear comparison of behavioral differences between the two algorithms. This approach is also pragmatic, substantially reducing computational demands, which was necessary due to internship time constraints. Given that individual training runs could last up to 30 hours and each evaluation phase approximately 15 hours, only a limited number of tests could be conducted.

Time limitations meant that it was not possible to fully complete the experimental framework or to obtain comprehensive results. Nevertheless, this process significantly enhanced understanding of RL simulation and highlighted several valuable observations that merit further discussion and exploration.

Parameter selection for the initial configuration was inspired by the work of Y. Sola, with the baseline configuration $SAC_0 = (K_r = 40, f_{RL} = 25\text{Hz})$. Here, $f_{RL} = 25\text{Hz}$ corresponds to the average computation frequency achievable on the utilized PC. When I dedicated solely to simulation tasks, without graphical rendering frequencies up to 55Hz could be achieved, which also aligns with typical computational rates for PPO. The value 10Hz was chosen to encapsulate the lower end of average operational frequencies. Similarly, coefficients for the reward parameter (K_r) were selected to explore a range around the initial value of 40, hence the values 1 and 100 were also tested.

This experimentation matrix resulted in a total of nine unique simulations, each combining different step durations and reward coefficients (see Table 3.1). This design enables the isolation of the effect of each parameter by holding others constant. The notation SAC_i references the specific SAC agent corresponding to the i-th configuration within the experimental framework.

f_{RL} \ K_r	1	40	100
55	SAC_1	SAC_2	SAC_3
25	SAC_4	SAC_0	SAC_5
10	SAC_6	SAC_7	SAC_8

Table 3.1: Parameters Experimentation Framework

2 Development and application of analysis tools

In order to systematically assess the influence of different parameters and to perform comparative analyses among various SAC agent configurations as well as between SAC and PPO algorithms, it was necessary to establish robust criteria and dedicated tools for evaluating algorithm efficiency. To this end, a suite of analysis tools was developed, enabling comprehensive examination of both the training and testing phases for each agent. Some of these tools were custom-developed in Python, while others leverage established logging utilities such as Tensorboard, integrated via the Stable Baselines3 framework. These tools facilitate the collection of both statistical metrics and quantitative data essential for effective evaluation.

2.1 Training phase: diagnostic logging and visualization strategies

During the training phase, logging was primarily managed using Tensorboard through the Stable Baselines3 library. Tensorboard generates a variety of plots indexed by training time steps, as illustrated in diagram page 29. For each algorithm, metrics such as the mean episode length and mean episode reward are visualized as functions of the training step count. In addition, algorithm-specific metrics are provided to better align with the internals of each method; for example, SAC logs include the evolution of actor and critic losses as well as the entropy coefficient, while PPO is accompanied by its own method-relevant diagnostic plots.

2.2 Testing phase: quantitative evaluation metrics

2.2.1 Establishing quantitative performance indicators

To quantitatively evaluate agent performance, a set of metrics inspired by the work of Y. Sola [24] was employed. These metrics encompass both training performance and testing evaluation. For assessment, each controller (PID and SAC) was subjected to 500 test episodes, with the following metrics calculated:

- **Success rate:** The proportion of episodes in which the waypoint is reached within predefined boundaries and time limits.
- **Collision rate:** The percentage of episodes ending in collision (specifically, AUV crossing vertical bounds $[-60, -1]$ on the Z-axis).
- **Timeout failure rate:** The fraction of episodes terminated by exceeding a 1,000-step limit.
- **Mean and standard deviation of $d\delta$:** Measures of deviation from the ideal straight-line trajectory, with mean indicating accuracy and standard deviation indicating trajectory stability.
- **Mean of $\|u\|$:** Average thruster effort, computed as

  $$\|u\| = \sqrt{u_1^2 + u_2^2 + u_3^2 + u_4^2 + u_5^2 + u_6^2},$$

  providing an estimate of control intensity.
- **Mean number of steps:** The average duration of episodes, contributing to the interpretation of trajectory following and efficiency.
- **Mean of $\sum \|u\|$:** The average total thruster usage per episode, reflecting the overall energy consumption.

Taken together, these quantitative metrics enable meaningful comparison between controllers with respect to reliability, tracking accuracy, and energy efficiency.

2.2.2 Three-dimensional trajectory visualization for behavioral assessment

To qualitatively observe agent behavior, three-dimensional trajectory plots were generated for individual episodes, as shown in the diagram page 29. These plots facilitate direct visual comparison of the spatial behavior of different agent configurations within the same environmental sequence.

2.3 Supplementary visualization and comparative tools

Additional logging utilities were implemented to complement the standard Tensorboard output. These custom logs present the evolution of final episode distance, episode rewards, and episode length throughout both training and testing phases, with all metrics plotted against episodes rather than time steps. Such representations provide a more intuitive view of the agent's learning dynamics and allow for side-by-side visualization of multiple agents with differing configurations, as illustrated in diagram page 29. This comprehensive set of tools thus supports robust analysis and transparent presentation of experimental results.

3 Results and Analysis

3.1 Investigating the Influence of Specific Parameters on the SAC Algorithm

In RL, evaluating both training and testing phases is crucial: training assesses whether the agent learns a satisfactory policy, while testing examines its generalization ability.

3.1.1 Training Phase

During training, performance convergence indicates that the agent has reached optimal or near-optimal behavior. First, we monitor the mean episode length and the mean episode reward as functions of training timesteps (Figure 3.1).

The episode length curve is a direct indicator of task success: decreasing trends reflect growing ability to reach the goal, while values fixed at 1000 timesteps indicate persistent failure. In Figure 3.1, the agents SAC1, SAC2, and SAC3 remain close to this maximum, showing no convergence. In contrast, SAC6 and SAC7 converge to shorter episodes (500 steps), and other agents also exhibit decreasing trajectories.

Reward curves confirm this observation: except for SAC1–3, most agents follow an inverted exponential trend with a transient increase before stabilizing. SAC1–3 correspond to an update frequency of 55 Hz, supporting the hypothesis that excessively high update

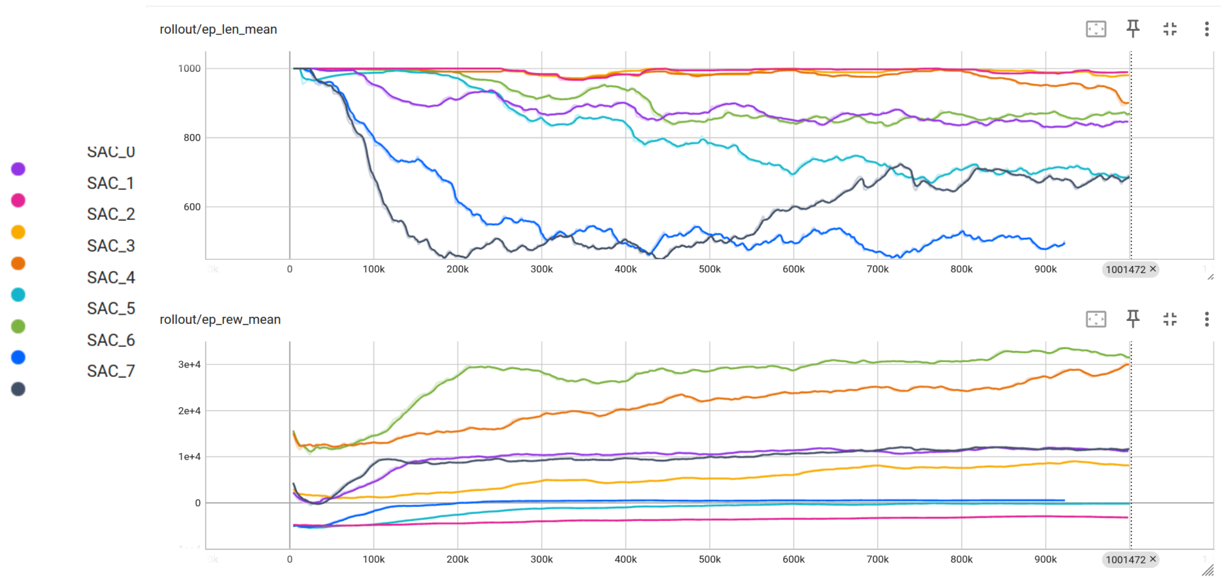


Figure 3.1: Tensorboard logs: mean episode length and mean episode reward over timesteps.

frequencies hinder adaptation by restricting the agent's interaction time with the environment.

Comparisons between agents with identical reward coefficients but different update frequencies (e.g., SAC0 vs. SAC7, SAC4 vs. SAC6) show that lower update frequencies promote faster convergence, as longer interaction windows enhance exploration and goal-reaching ability.

The role of the reward coefficient is more nuanced. Although larger coefficients yield higher episode rewards, the effect on training is not straightforward: for example, despite the SAC5 coefficient of 100, it does not outperform lower-coefficient configurations. Among agents with the same update frequency (SAC0, SAC4, SAC5), SAC4 manage to reach the goal faster, suggesting that very high trajectory rewards may bias the agent towards maintaining its trajectory rather than reaching the goal.

Reward curves also suggest that higher coefficients can accelerate convergence (e.g., SAC5 vs. SAC4, SAC7 vs. SAC6), but excessively large values (SAC5, coefficient 100) may instead slow learning compared to moderate values (SAC0, coefficient 40). In particular, comparisons with agents that do not converge (SAC1–3) are not conclusive for analyzing reward effects.

Figure 3.2 provides further insight into reward coefficients by showing reward distributions per episode and success proportions over 100-episode windows. For small coefficients, maximum rewards coincide with successful episodes, whereas for higher coefficients, maximum rewards increasingly correspond to failures. This supports the hypothesis that exces-

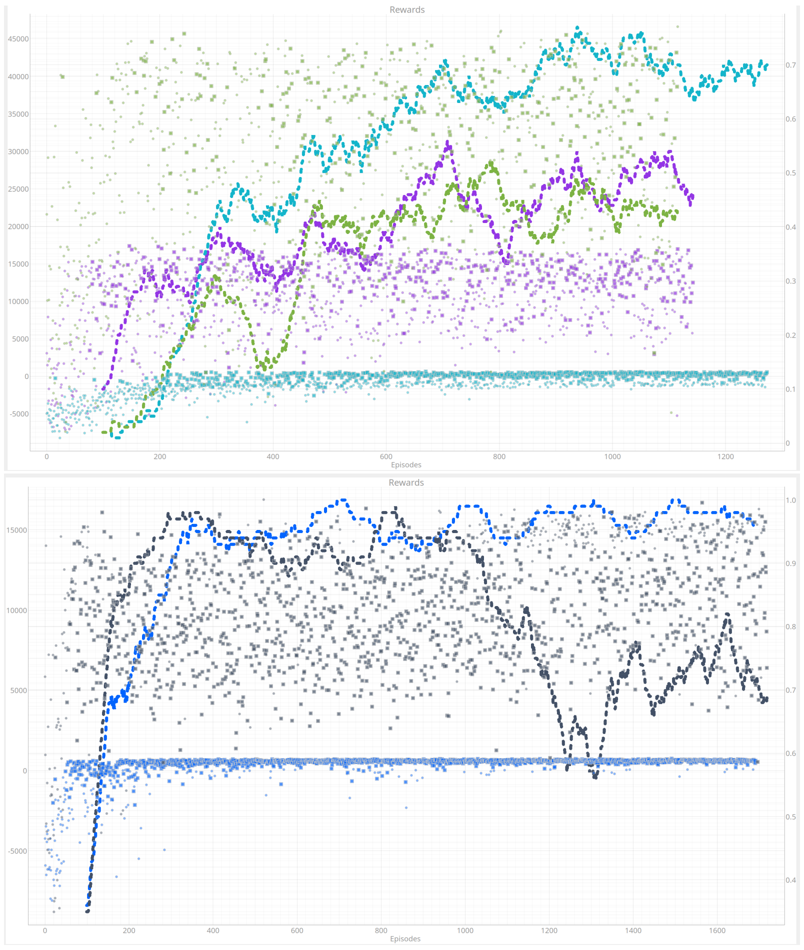


Figure 3.2: Episode reward evolution: SAC0 (violet), SAC4 (cyan), SAC5 (green), SAC6 (blue), SAC7 (gray).

sive reward scaling causes the agent to maximize reward without achieving the intended task.

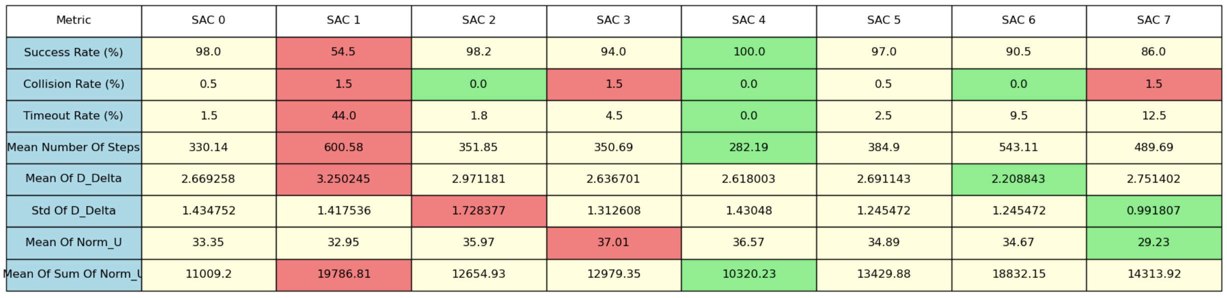
For example, SAC4 achieves higher success rates than SAC0 or SAC5 despite its smaller coefficient. Similarly, SAC6 outperforms SAC7, whose success rate decreases during training even as its mean reward increases. This decoupling of reward maximization from task success confirms that excessive reward values can produce misleading learning signals, reducing real performance.

During training, certain configurations clearly outperform others. First, it is crucial to avoid excessively high update frequencies, as they prevent the agent from converging toward stable values. Second, careful tuning of the reward coefficient is necessary to ensure that reward maximization aligns with task success, rather than merely producing high terminal rewards without achieving the goal.

Based on these observations, SAC4 ($K_r = 1, f_{RL} = 25$ Hz) and SAC6 ($K_r = 1, f_{RL} = 10$ Hz) emerge as the most effective configurations according to the training metrics. However, computational efficiency must also be considered: SAC4 completes training in approximately 11h44, whereas SAC6 requires about 20h. Consequently, from a practical standpoint, the SAC4 configuration is preferred.

3.1.2 Test Phase

After analyzing the training phase, we now evaluate the agents' performance during testing, using a new dataset and fixing the update frequency at 10 Hz to allow the robot to evolve naturally in the environment. Table 3.3 summarizes the evaluation metrics across 200 test episodes.



Metric	SAC 0	SAC 1	SAC 2	SAC 3	SAC 4	SAC 5	SAC 6	SAC 7
Success Rate (%)	98.0	54.5	98.2	94.0	100.0	97.0	90.5	86.0
Collision Rate (%)	0.5	1.5	0.0	1.5	0.0	0.5	0.0	1.5
Timeout Rate (%)	1.5	44.0	1.8	4.5	0.0	2.5	9.5	12.5
Mean Number Of Steps	330.14	600.58	351.85	350.69	282.19	384.9	543.11	489.69
Mean Of D_Delta	2.669258	3.250245	2.971181	2.636701	2.618003	2.691143	2.208843	2.751402
Std Of D_Delta	1.434752	1.417536	1.728377	1.312608	1.43048	1.245472	1.245472	0.991807
Mean Of Norm_U	33.35	32.95	35.97	37.01	36.57	34.89	34.67	29.23
Mean Of Sum Of Norm_U	11009.2	19786.81	12654.93	12979.35	10320.23	13429.88	18832.15	14313.92

Figure 3.3: Comparison of test metrics after 200 episodes

This table highlights several key results. SAC1 obtained the lowest success rate, which can be explained by its lack of convergence during training. Surprisingly, SAC2 and SAC3, despite not converging in training, achieved high success rates in testing. Conversely, SAC6 and SAC7—promising during training—performed worse in testing, which indicates a gap between training convergence and generalization. Moreover, the tendency observed during training persists: lower reward coefficients generally correspond to higher success rates.

Figure 3.4 presents the evolution of cumulative mean values for final distance, episode reward, and number of steps. Interestingly, the number of steps shows that SAC6 and SAC7, although efficient during training, are not the fastest in testing. Instead, SAC2 and SAC3 significantly reduce episode length, outperforming expectations based on their training results.

To deepen the analysis, Figure 3.5 compares the number of steps with the average deviation from the ideal straight-line trajectory (Initial–Goal). Results show no direct correlation between trajectory straightness and step count: SAC4 achieves the lowest step number, while SAC6 follows the most trajectory-aligned path. This implies that SAC6 takes more steps despite staying closer to the ideal trajectory, whereas SAC4 is more efficient overall.

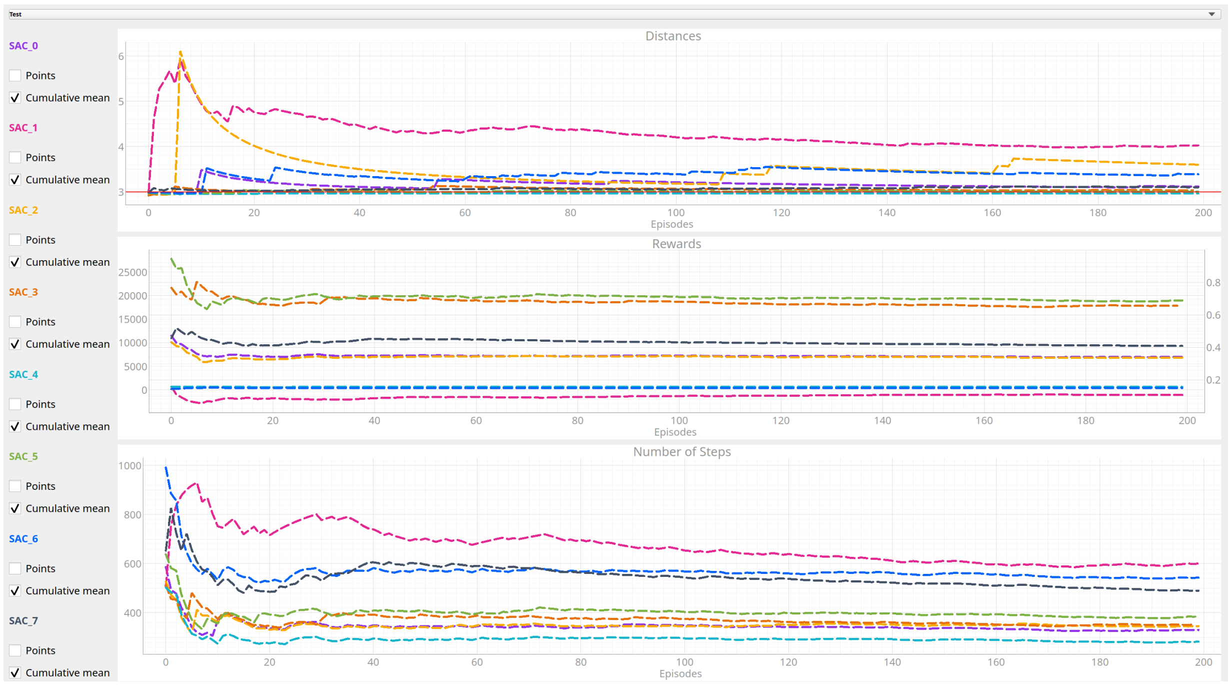


Figure 3.4: Test logs: cumulative mean of final distance (top), episode reward (middle), and number of steps (bottom) across episodes

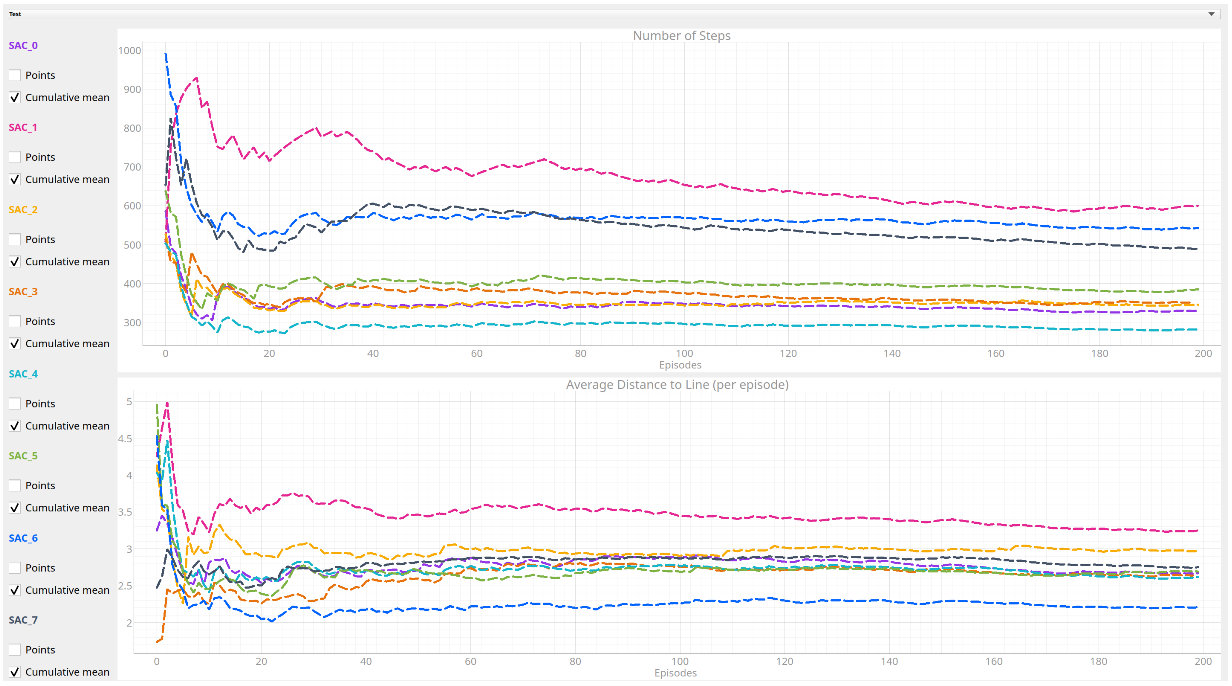


Figure 3.5: Test logs: cumulative mean of steps (top) and mean distance to straight-line trajectory (bottom)

Figure 3.6 illustrates representative test trajectories (episode 2). While subtle differences make it difficult to declare an optimal trajectory visually, interesting trends emerge regarding the influence of the reward coefficient. For instance, SAC5 (highest $K_r = 100$) tends to hover around the goal boundary, maximizing trajectory-related rewards rather than prioritizing success. SAC7 shows a similar pattern. On the contrary, SAC1's trajectory indicates awareness of the goal location but without identifying an efficient approach, leading to circular movements around the goal until the episode reward is reached. These examples illustrate how excessively large reward coefficients can bias behavior, while moderate coefficients encourage more goal-directed trajectories.

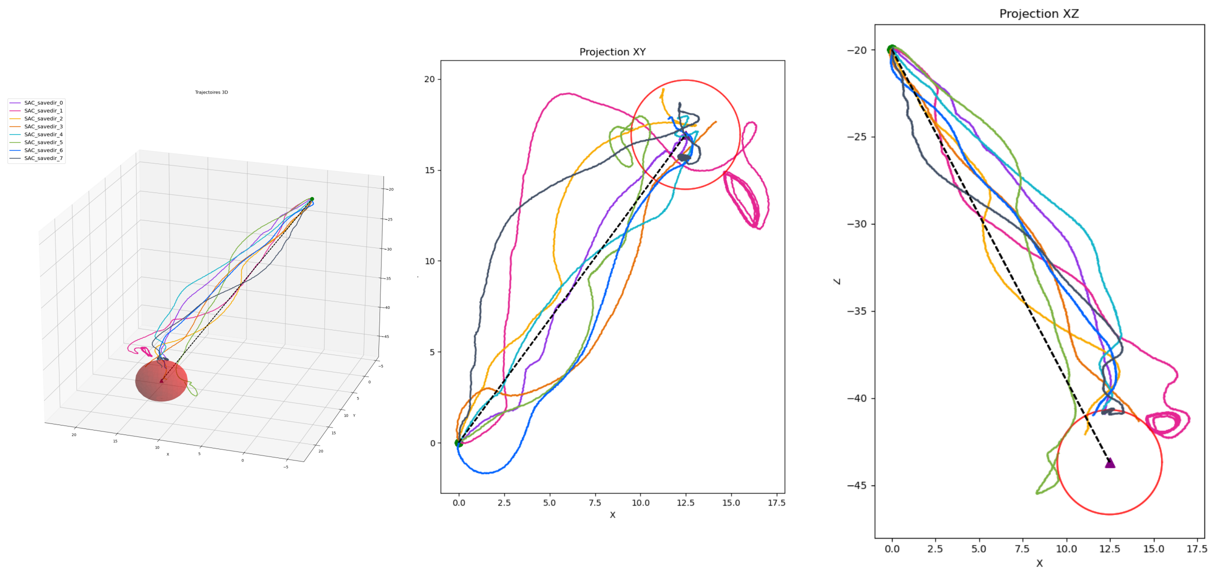


Figure 3.6: Episode 2 trajectories of SAC agents: 3D view (left), XY projection (middle), XZ projection (right)

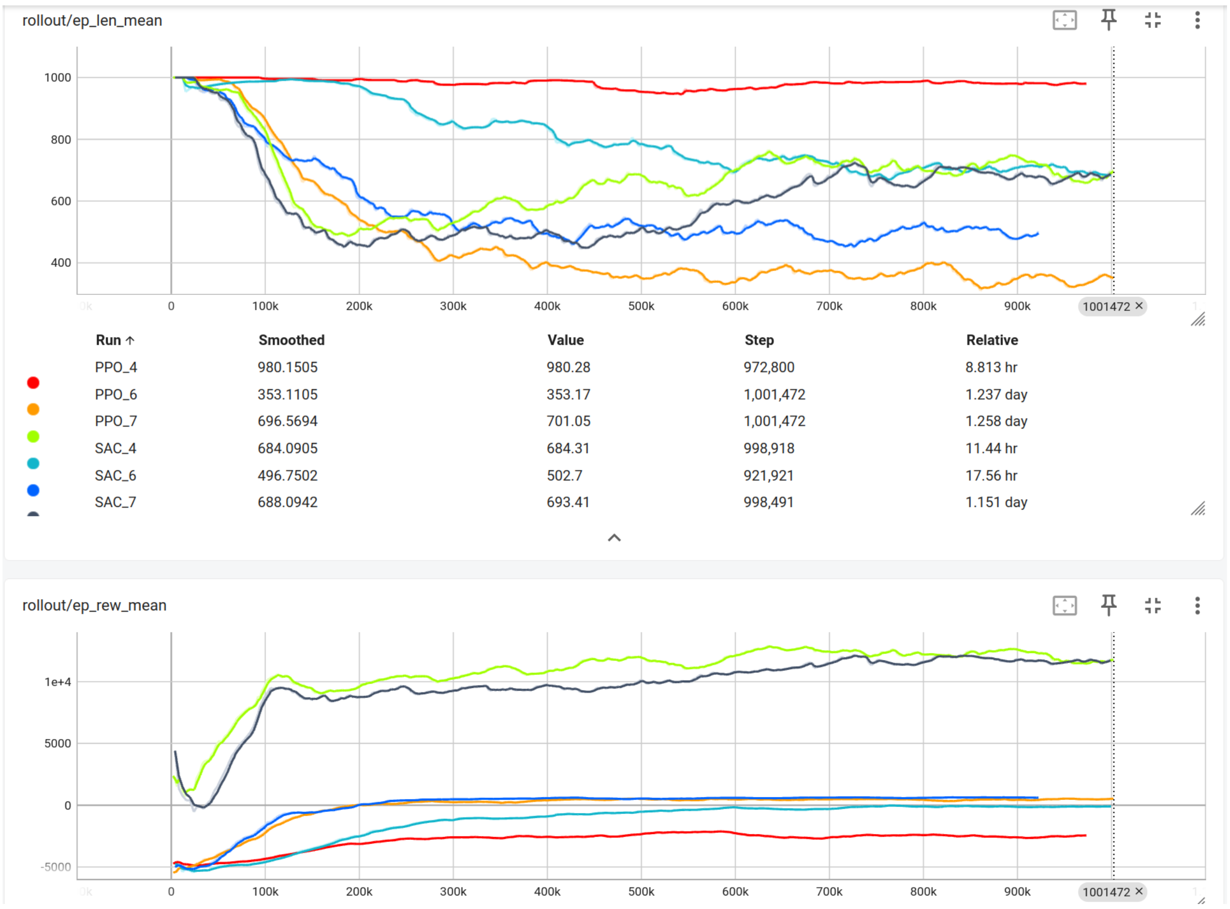
Conclusion on Test Phase

The test results confirm that both the reward coefficient K_r and update frequency f_{RL} significantly influence generalization. Overall, SAC4 ($K_r = 1$, $f_{RL} = 25$ Hz) performs best, consistent with the training-phase hypothesis and offering the most balanced behavior. Unexpectedly, SAC2 ($K_r = 40$, $f_{RL} = 55$ Hz) achieved the second-highest efficiency, despite poor convergence during training. This suggests that an agent does not need to fully solve the task during training to generalize effectively in testing, and that certain parameter combinations may allow better adaptation during deployment.

While SAC4 represents the most interpretable and stable choice, SAC2's surprising performance highlights an interesting direction for further research, particularly regarding the discrepancy between training convergence and real-world generalization.

3.2 Comparing the Influence of RL Algorithm Choice: SAC vs PPO

This section investigates the impact of the reinforcement learning algorithm by comparing the performance of Soft Actor-Critic (SAC) and Proximal Policy Optimization (PPO). To ensure comparability, equivalent configurations are analyzed using the same numbering convention as before. Due to time constraints, this analysis is limited to configurations 4, 6, and 7, and some plots are omitted.



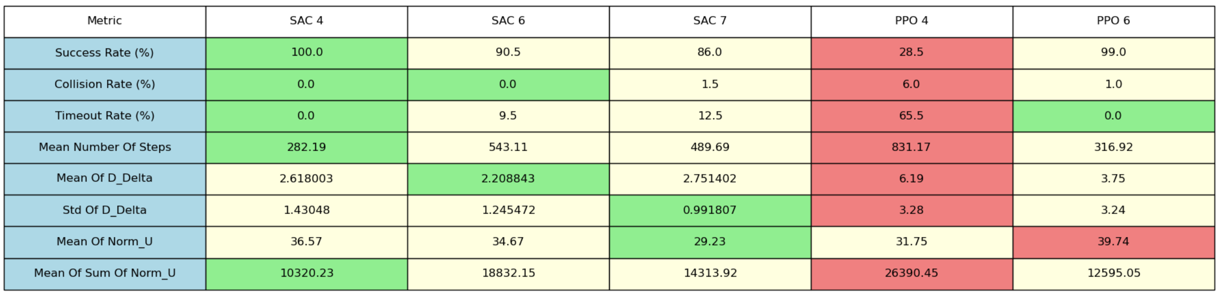
	Run ↑	Smoothed	Value	Step	Relative
🟥	PPO_4	980.1505	980.28	972,800	8.813 hr
🟧	PPO_6	353.1105	353.17	1,001,472	1.237 day
🟩	PPO_7	696.5694	701.05	1,001,472	1.258 day
🟦	SAC_4	684.0905	684.31	998,918	11.44 hr
🔵	SAC_6	496.7502	502.7	921,921	17.56 hr
●	SAC_7	688.0942	693.41	998,491	1.151 day

Figure 3.7: Tensorboard logs: mean episode length and mean episode reward over timesteps for SAC and PPO agents

Figure 3.7 compares the training behaviors of SAC and PPO. Except for configuration 4, both algorithms exhibit similar dynamics with respect to the reward coefficient. For example, SAC7 and PPO7 as well as SAC6 and PPO6 display nearly identical trends. However, PPO agents appear more reactive: PPO6 shows a sharper reduction in episode length than SAC6, and PPO7 achieves faster reward maximization than SAC7. Conversely, PPO4 fails to converge, unlike its SAC counterpart.

This difference aligns with known theoretical properties: SAC, being an off-policy method based on maximum entropy reinforcement learning [21], benefits from sample reuse and tends to exhibit greater stability even under suboptimal configurations. PPO, by contrast, is an on-policy algorithm [19] that often requires more carefully tuned hyper-parameters but can adapt quickly in training when the configuration is appropriate.

Overall, PPO6 appears the most efficient during training, although prior observations suggest that test-phase results can diverge significantly from training performance.



Metric	SAC 4	SAC 6	SAC 7	PPO 4	PPO 6
Success Rate (%)	100.0	90.5	86.0	28.5	99.0
Collision Rate (%)	0.0	0.0	1.5	6.0	1.0
Timeout Rate (%)	0.0	9.5	12.5	65.5	0.0
Mean Number Of Steps	282.19	543.11	489.69	831.17	316.92
Mean Of D_Delta	2.618003	2.208843	2.751402	6.19	3.75
Std Of D_Delta	1.43048	1.245472	0.991807	3.28	3.24
Mean Of Norm_U	36.57	34.67	29.23	31.75	39.74
Mean Of Sum Of Norm_U	10320.23	18832.15	14313.92	26390.45	12595.05

Figure 3.8: Comparison of SAC and PPO test metrics after 200 episodes

Table 3.8 summarizes test metrics. As expected, PPO4 confirms its inefficiency, as already suggested by its training curves. SAC4, in contrast, demonstrates high reliability. For configuration 6, PPO6 shows superior performance compared to SAC6, suggesting that PPO benefits strongly when granted sufficient interaction time for exploration.

This observation is consistent with empirical findings in the literature: PPO typically performs best with moderate update frequencies and larger batch sizes to stabilize learning, whereas SAC can tolerate higher update frequencies thanks to its off-policy nature and replay buffer. Indeed, PPO6 and PPO7 achieve success rates of 99% and 95%, respectively, compared to 90.5% and 86.0% for SAC6 and SAC7. As with SAC, PPO also shows sensitivity to the reward coefficient, as indicated by the 6% performance drop between PPO6 and PPO7.

Conclusion on SAC vs PPO

The comparison highlights both similarities and differences. SAC and PPO follow comparable trends in how configuration parameters (update frequency, reward coefficient) shape behavior. However, their optimal configurations diverge: SAC is more robust to high update frequencies due to its off-policy formulation and entropy regularization, which enforce stability and sustained exploration. PPO, on the other hand, is more efficient when exploration time is sufficient, but requires careful tuning of learning rate and update frequency to avoid premature convergence.

In summary, SAC provides greater robustness across a wider range of configurations, while PPO, when properly tuned, can achieve higher success rates in fewer episodes. This trade-off reflects the underlying algorithmic differences: entropy-maximizing off-policy updates versus policy-regularized on-policy updates. These results reinforce the idea, well-supported in the RL literature, that algorithm choice and hyperparameter tuning are strongly interdependent and problem-specific. Further experiments would be needed for a more comprehensive comparison, but the results already confirm that algorithm choice and parameter tuning are strongly interdependent.

Chapter 4

Conclusion et perspectives

This thesis has explored the application and critical evaluation of deep reinforcement learning (DRL) algorithms—specifically Soft Actor-Critic (SAC) and Proximal Policy Optimization (PPO)—within the challenging domain of autonomous underwater vehicle (AUV) control. The core objective was to bridge theoretical insights and practical implementations by developing a high-fidelity simulation framework and conducting systematic experimentation designed to elucidate the impact of key algorithmic and environmental parameters on learning performance, robustness, and generalization.

A key contribution is the design and implementation of a fully integrated simulation pipeline combining Gazebo Harmonic, ROS2 middleware, and a customized RL environment interfaced via Stable Baselines3. This architecture supports reproducible training and testing of DRL algorithms under realistic underwater dynamics and sensor noise conditions, thus addressing a critical gap between algorithmic advancement and practical robotics deployment.

The extensive experimental analysis revealed several important findings. Firstly, update frequency and reward function scaling critically influence training convergence and test-phase generalization. Excessively high update rates restrict the agent's ability to sufficiently explore the environment, while overly large reward coefficients can decouple reward maximization from true task success, leading to suboptimal policies. Moderately tuned configurations—especially with lower update frequencies and well-balanced reward parameters—yield the most robust and efficient controllers.

Secondly, comparative benchmarking of SAC and PPO highlighted their complementary strengths. SAC's off-policy maximum entropy framework confers greater stability and robustness across a wider range of hyperparameters, making it more tolerant to changes in update frequency and facilitating sustained exploration. PPO, as an on-policy method, requires more careful tuning but can deliver superior performance when allotted sufficient exploration time, achieving higher success rates within fewer episodes. These observations underscore the necessity of algorithm-specific parameter optimization rather than one-size-fits-all solutions.

Additionally, the thesis uncovered intriguing phenomena such as cases where agents lacking full convergence during training nevertheless generalized well during testing, suggesting that factors beyond nominal convergence influence real-world applicability. These findings invite further investigation into the complex dynamics of RL policy learning under uncertain and nonlinear environmental conditions.

In conclusion, this work advances understanding of how state-of-the-art DRL methods can be effectively applied to autonomous underwater vehicle control. By integrating theoretical foundations, modern simulation tools, and rigorous experiments, it provides actionable insights into algorithm selection, configuration, and evaluation within realistic robotic contexts. The developed framework and analysis tools lay a solid foundation for future research aiming to enhance the adaptability, safety, and efficiency of underwater autonomous systems leveraging reinforcement learning.

Future directions include extending the current study to incorporate model-based methods, hybrid AI-classical control architectures, real-world hardware validation, and multi-agent coordination scenarios. Moreover, deeper exploration into reward function engineering and adaptive hyperparameter tuning will be crucial to further close the gap between simulation and deployment in highly dynamic and uncertain underwater environments.

Bibliography

[1] Thomas Chaffre. *Reinforcement Learning and Sim-to-Real Transfer for Adaptive Control of AUV*. PhD thesis, Flinders University, College of Science and Engineering, December 2022. Defended at Flinders University on December 13, 2022.

[2] Wikipedia contributors. Markov decision process. `https://en.wikipedia.org/wiki/Markov_decision_process`, 2025. Accessed: 2025-08-12.

[3] Dasong Wang and Roland Snooks. *Artificial Intuitions of Generative Design: An Approach Based on Reinforcement Learning*, pages 189–198. Springer, 2021.

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[5] David Silver. Monte carlo and temporal difference methods. *Reinforcement Learning Course*, 2020.

[6] Restack.io. Comparison of monte carlo and temporal difference learning in reinforcement learning. `https://www.restack.io/p/reinforcement-learning-answer-monte-carlo-vs-td-cat-ai`, 2023. Accessed: 2025-05-19.

[7] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[8] Hugging Face. Welcome to the deep reinforcement learning course - hugging face deep rl course. `https://huggingface.co/learn/deep-rl-course/unit0/introduction`, 2025. Accessed: May 14, 2025.

[9] Volodymyr Mnih et al. Asynchronous methods for deep reinforcement learning. *International Conference on Machine Learning*, 2016.

[10] Felix Helfenstein. Benchmarking deep reinforcement learning algorithms. *ETH Zurich Bachelor Thesis Repository*, 2021(1), May 2021. Submitted on May 18, 2021.

[11] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716–719, 1952.

[12] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.

[13] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 12, 2000.

[14] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, 1989.

[15] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[16] 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.

[17] Stable Baselines3 Developers. Stable baselines3 documentation. `https://stable-baselines3.readthedocs.io/en/master/index.html`. Accessed: 2025-08-14.

[18] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

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

[20] Daniel Bick. Towards delivering a coherent self-contained explanation of proximal policy optimization, 2023. Unpublished manuscript.

[21] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[22] OpenAI Spinning Up. Soft actor-critic (sac) - spinning up in deep rl. `https://spinningup.openai.com/en/latest/algorithms/sac.html#pseudocode`. Accessed: 2025-08-13.

[23] Katell Lagattu. *Actuator Fault Recovery Control Strategy with Deep Reinforcement Learning - Application to Unmanned Underwater Vehicles.* PhD thesis, Université de Bretagne Occidentale, 2023. Mid-candidature review, supervised by Benoit Clement, Eva Artusi, Gilles Le Chenadec, Karl Sammut, and Paulo Santos.

[24] Yoann Sola. *Contributions to the development of Deep Reinforcement Learning-based controllers for AUV.* Phd thesis, École nationale supérieure de techniques avancées Bretagne, Brest, France, September 2021.

[25] Centrale Nantes ROV Team. Bluerov2 simulation and control framework. `https://github.com/CentraleNantesROV/bluerov2`, 2023. Accessed: 2025-08-14.

[26] Ignacio Carlucho, Mariano De Paula, Sen Wang, Yvan Petillot, and Gerardo G. Acosta. Adaptive low-level control of autonomous underwater vehicles using deep reinforcement learning. *Robotics and Autonomous Systems*, 107:71–86, 2018.

Résumé —

This thesis investigates the application of deep reinforcement learning (DRL) algorithms, Soft Actor-Critic (SAC) and Proximal Policy Optimization (PPO), for autonomous control of the BlueROV2 underwater vehicle. A high-fidelity simulation framework combining Gazebo Harmonic, ROS 2, and Stable Baselines3 supports reproducible training and evaluation under realistic underwater conditions.

Experimental analysis examines the influence of update frequency and reward scaling on training convergence and policy robustness. The results show that moderate update rates and balanced reward parameters improve performance, with SAC displaying greater stability and tolerance to hyperparameter variations, while PPO achieves high success rates when carefully tuned.

Fortunately, some agents with incomplete training convergence still generalized well during testing, highlighting complexities in the development of reinforcement learning policy. This work bridges the gap between DRL theory and practical deployment in underwater robotics by providing an integrated pipeline and systematic evaluation.

The thesis contributes valuable insights into algorithm selection, tuning, and evaluation for underwater autonomous vehicles and lays groundwork for future research in model-based control, hybrid architectures, real-world validation, and multi-agent systems. Further studies on reward engineering and adaptive hyperparameter tuning will enhance the autonomy of AUV and safety.

Mots clés : Artificial intelligence, Reinforcement learning, Autonomous Underwater Vehicle, Proximal Policy Optimization algorithm, Soft Actor-Critic algorithm

Polytech Angers
62, avenue Notre Dame du Lac
49000 Angers