Deep Reinforcement Learning Control Law for Target Tracking for Unmanned Surface Vehicles

Lars Skogen Johnsen[1]

359056@oslomet.no

Preface

This report is written in accordance with the final project course requirement in ACIT4830 - Special Robotics and Control Subject for the spring semester 2025.

All project files and contributions are highlighted under Otter-USV/Otter API/Otter_dl.py and Otter-USV/Otter API/Otter_simulator_DRL.py in `https://github.com/LSJohnsen/Otter-USV`.

Abstract

This project investigates the application of Deep Reinforcement Learning (DRL) for target tracking control of Unmanned Surface Vehicles (USVs), using a simulator for the Otter USV as the test platform. A pure DRL controller, implemented with the Proximal Policy Optimization (PPO) algorithm, was trained and evaluated in a Python Vehicle Simulator environment. Two scenarios were examined: straight-line and circular target tracking under ideal conditions. The DRL controller's performance was compared against a conventional PI/PID controller using quantitative error indices, specifically the Integral of Absolute Error (IAE) for target distance and heading. Results indicate that the DRL controller significantly outperforms PID control in reducing target distance error, while maintaining comparable heading accuracy. However, the DRL model exhibits less smooth control signals, suggesting a need for improved reward shaping and control allocation. The study highlights the advantages and challenges of DRL in marine control applications and suggests future improvements including better hyperparameter tuning, environment normalization, and alternative actor-critic algorithms. The findings demonstrate the feasibility of DRL for USV control, offering an alternative to classical control strategies in dynamic and uncertain environments.

Contents

List of Tables

List of Figures

1 Introduction

With the continuous increase in computational power of modern computer systems, the use of machine learning for control applications has seen significant advances in both academia and industry in recent years. Applying machine learning methods has become a widely adopted approach in Unmanned Surface Vessel (USV) path following and target tracking missions, where it can uncover complex patterns in water dynamics that traditional control methods often overlook (Zhang, Ren, Cui, Fu, & Cong, 2024; Zhao et al., 2020). As the use of USVs is expanding across domains like hydrography, environmental monitoring, and meteorological surveying, there is a growing need for control strategies capable of completing missions autonomously and reliably in uncertain marine environments.

Traditional control strategies often rely on accurate control system modeling and extensive manual tuning, which can be difficult to achieve in complex and variable marine environments. Among the techniques that address these limitations, Deep Reinforcement Learning (DRL) has emerged as a promising solution, enabling autonomous agents to learn effective control policies through trial-and-error interaction with dynamic environments. In addition to being used as a standalone control method, reinforcement learning has also been applied to assist in tuning the parameters of conventional control laws such as Proportional-Integral-Derivative (PID) and Model Predictive Control (MPC), where manual tuning is typically time-consuming and system-specific (Cui, Peng, & Li, 2022). training DRL agents in real-world scenarios is generally impractical due to long training times and safety concerns. However, this challenge is mitigated by the advances in accurate hydrodynamic modeling, such as those presented in Fossen (2021), which allow simulations to accurately replicate the real-world vessel behavior, making them suitable for training DRL agents. This enables the development of robust control strategies in a safe and controlled virtual setting before deployment on physical USVs.

1.1 Otter USV System

The Otter USV, displayed in Figure 1, is the smallest unmanned surface vehicle produced by Maritime Robotics, with a total length of two meters (Maritime Robotics, 2025). Its main frame is mounted on two pontoons in a catamaran configuration, providing stability and low drag. The vessel is equipped with two fixed stern thrusters that enable propulsion and maneuverability through differential drive, allowing for both forward motion and rotation. Due to its small size, long battery life, and high maneuverability, the vessel is an effective option for zero-emission data acquisition for cases like environmental monitoring in coastal and other shallow areas. Additionally the USV comes configured with a high-precision Global Positioning Systems (GPS) and an Inertial Measurement Unit (IMU), enabling centimeter-level accuracy in deployment. In the simulation environment, the Otter is modeled with six degrees of freedom (DOF), accounting for translational and rotational motion in all three spatial dimensions, allowing for an accurate approximation of the vessel's expected real-world behavior under dynamic marine conditions.

1.2 Python Vehicle Simulator Environment

The Python Vehicle Simulator is an open-source tool that includes models for a variety of autonomous underwater vehicles (AUVs), ships, and unmanned surface vehicles (USVs) (Fossen, 2023). Featuring accurate hydrodynamic modeling and integrating modern guidance and control methods based on Fossen (2021), the simulator and its associated modeling techniques have been widely adopted in academic research and are considered a standard framework for marine vehicle simulation. An example of this is the Robot Operating System (ROS) Gazebo simulation framework for the WAM-V vehicle, presented in Bingham et al. (2019), which builds on Fossen's modeling approach. This framework has supported research an areas like dynamic obstacle avoidance and DRL-based motion control under environmental disturbances (Li, Chavez-Galaviz, Azizzadenesheli, & Mahmoudian, 2023; Yuan & Rui, 2023), providing an accurate representation of the expected real-world performance.

The DRL-based control law applied in this work builds on the modeling principles used in the simulator. It incorporates relevant modifications and includes a PID control law as presented in Gresberg, Johnsen, and Stensrud (2023), which serves as a baseline for performance comparisons.

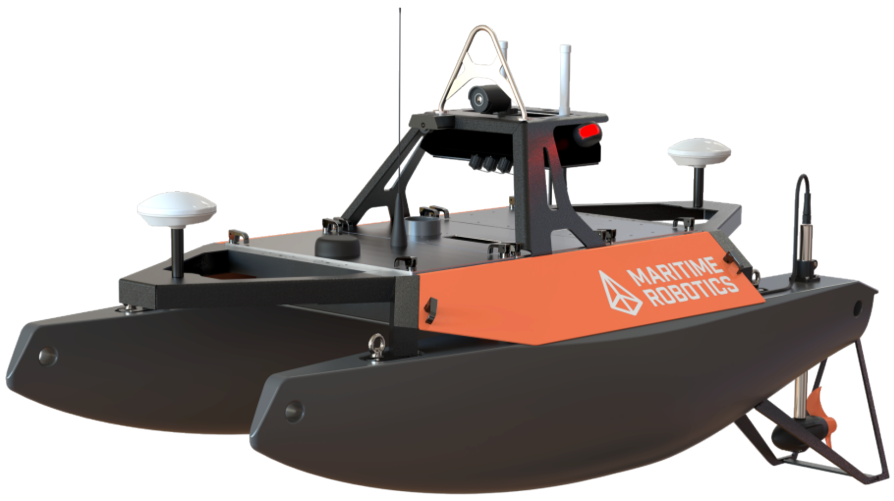

Figure 1: Otter Unmanned Surface Vessel.
Maritime Robotics (2025)

1.3 Target Tracking Control Laws

Target tracking is one of the most common control operations for USVs, alongside path following, path tracking, and path maneuvering (Breivik, Hovstein, & Fossen, 2008). The objective of target tracking is to control the USV's motion to follow either a stationary or moving target, where the target's current position is known in near real-time, but no information about its future movement is available.

While target tracking is not a new concept, with examples of autonomous USV straight-line target tracking being demonstrated as early as Caccia, Bibuli, Bono, and Bruzzone (2008) and (Breivik et al., 2008) by employing either one PI and a PD, or two PI controllers, regulating the surge and heading velocities, modern control methods are significantly more advanced. These approaches offer improved robustness and adaptability, particularly in the presence of external disturbances such as waves, wind, and ocean currents. Cui et al. (2022) highlight this through their Filtered Probabilistic Model Predictive Control approach, which combines MPC for action planning with probabilistic prediction models of future states. Their study demonstrates a USV capable of effectively maintaining position and reaching targets in both seen and unseen environments, even under varying levels of wind and current disturbances. The method achieved significantly better disturbance rejection compared to the baseline PID control approach.

While PID control offer benefits in its simplicity, with easy implementation and tuning, and requiring limited computational power, and potentially performing better than NMPC and DRL control in some simulated scenarios (Frafjord, Saksvik, Kjerstad, & Coates, 2024), it generally displays aggressive control actions which could make precise USV control difficult with complex dynamic conditions. On the other hand MPC or NMPC, while possibly managing optimal control if the model is accurate, would require an accurate model of the dynamics affecting the vessel, such as inertia, damping, and thruster responses. Consequently NMPC control is computationally demanding when compared to linear control methods.

1.4 Learning-Based Control

Considering the potential difficulties in dealing with the nonlinear problem of hydrodynamics in PID control, and the requirements of accurate modeling for MPC control implementation, reinforcement learning based control methods are becoming continuously more popular due to their ability to capture the dynamics of advanced systems and environments through iterative learning. Cui et al. (2022) presented benefits in applying reinforcement learning methods in MPC, where its application made it possible to iteratively learn the USV system dynamics, which were then applied to control systems. Similarly, Hu, Wan, and Lei (2022) has proposed DRL as an effective method to determine the necessary velocities in USV collision avoidance, applying Proximal Policy Optimization (PPO) to solve the USV action space, where the MPC control aids in reducing the complexity in learning the model through a preexisting control law. While its comparison with pure a pure PPO DRL control law displayed both methods effectively avoiding the object, applying the MPC-aided DRL approach resulted in smoother and safer obstacle avoidance, albeit at slightly longer learning times and path lengths.

Pure DRL control has also seen recent use in both multi-USV task planning and target tracking (Zhang et al., 2024; Wang, Hu, Wang, Liu, & Xie, 2025). In Wang et al. (2025), a DRL framework based on the Soft Actor-Critic (SAC) algorithm was introduced to perform straight-line target tracking without relying on separate guidance or motion control methods, and the results were compared to pure pursuit methods with a PID feedback controller. Using SAC, which receives a cubic-spline forecast of the target's future position, the DRL control achieved up to 20% faster target pursuit. In multi-USV task planning, which separates the problem into task allocation and collision avoidance, the use of DRL control was also shown to increase efficiency while ensuring safe navigation when following set formations (Zhang et al., 2024).

1.5 Objectives

This project aims to implement a pure DRL control law to the simulation environment presented in Fossen (2023) and Gresberg et al. (2023) for target tracking purposes. The DRL-based controller will be compared against the PI surge and PID yaw control law in Gresberg et al. (2023) under two scenarios:

- *Straight-line target tracking with no external disturbances.*
- *Circular target tracking with no external disturbances.*

The results will compare the performance of the two control strategies under realistic USV dynamics, highlighting their relative strengths in tracking accuracy, stability, and adaptability across the two scenarios using quantitative error metrics.

1.6 Outline

The report is divided into seven chapters. This introductory chapter has provided background on the project topic, the selected USV, and the simulation environment, along with a brief review of recent research on the application of DRL control laws in surface vessels. Chapter 2 presents the theoretical framework for the project, including key concepts related to USV frame transformations, deep learning, and reinforcement learning. The materials used throughout the project are highlighted under chapter 3. Chapter 4 details the implementation methods and relevant code structure, including the applied test procedures. The results are presented in Chapter 5. Finally, Chapters 6 and 7 discuss the findings and conclude the report, addressing limitations and recommending future directions for DRL implementation in USV control.

2 Theory

This chapter briefly presents the most relevant theory related to the modeling of the Otter USV. Based on Fossen (2021) the full framework for the simulators dynamics and kinematic equations is found in Gresberg et al. (2023) and Fossen (2023). Subsequently, the chapter highlights the mathematical framework for the implementation of DRL control.

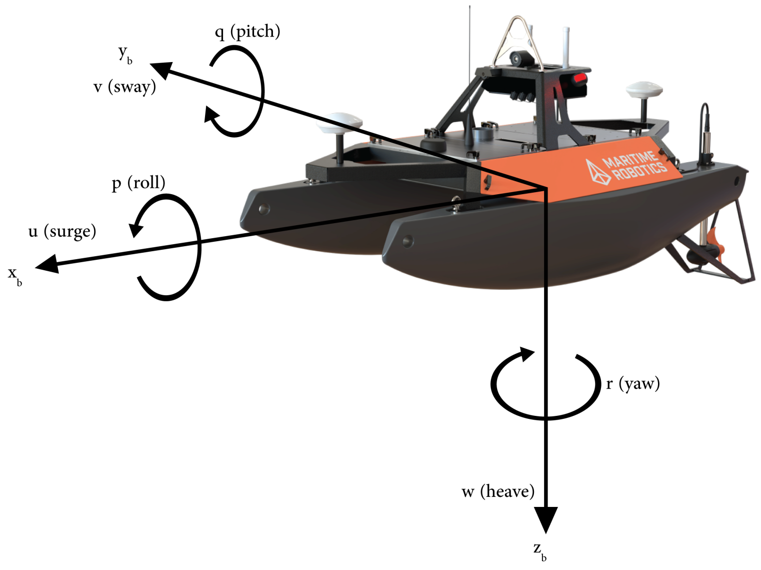

Figure 2: The Otter USV body-frame $\{b\}$ with 6 degrees of freedom.

2.1 Otter USV model

the Otter USV is modeled using the SNAME six DOF convention in Figure 2 and Table 1, with the translational motions—surge, sway, and heave—given in the North-East-Down (NED) in the body-frame, and rotational motions—roll, pitch, and yaw—about the corresponding axes.

Table 1: SNAME notation.

(SNAME, 1950)

DOF	Motion	Force & Moment	Linear & Angular Velocity	Position & Euler Angle
1	Surge	X	u	x
2	Sway	Y	v	y
3	Heave	Z	w	z
4	Roll	K	p	ϕ
5	Pitch	M	q	θ
6	Yaw	N	r	ψ

The simulation environment (Fossen, 2023; Gresberg et al., 2023), is modeled after Fossen's equation (Fossen, 2021):

$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{\eta}) = \boldsymbol{\tau} \tag{1}$$

Where the total sum of forces acting on the vessel $\boldsymbol{\tau}$ is determined through the inertial $\boldsymbol{M}$, Coriolis $\boldsymbol{C}(\boldsymbol{\nu})$, and Damping matrices $\boldsymbol{D}(\boldsymbol{\nu})$. $\boldsymbol{M}$ is a model for how the vehicle resists changes in motion due to mass or inertia, as the sum of the rigid-body inertia matrix $\boldsymbol{M_{RB}}$ and the added mass matrix $\boldsymbol{M_A}$. Similarly, the Coriolis-centripetal matrix is found by the sum of the rigid-body $\boldsymbol{C_{RB}}(\boldsymbol{\nu})$ and added mass $\boldsymbol{M_A}(\boldsymbol{\nu})$ matrix. $\boldsymbol{D}(\boldsymbol{\nu})$ calculates the dissipative forces from the resistance the surrounding water produces, such as viscous drag. Finally, $\boldsymbol{g}(\boldsymbol{\eta})$ are the restoring forces due to gravity and buoyancy.

The USV position and orientation in the inertial-frame $\{n\}$ are given by:

$$\boldsymbol{\eta} = \begin{bmatrix} x & y & z & \phi & \theta & \psi \end{bmatrix}^{\mathrm{T}} \tag{2}$$

and the vessel's linear and angular velocities in the body-frame $\{b\}$ are expressed as:

$$\boldsymbol{\nu} = \begin{bmatrix} u & v & w & p & q & r \end{bmatrix}^{\mathrm{T}} \tag{3}$$

By applying a third orthogonal rotation matrix about the z-axis, the relationship between the relationship between the frames $\{b\}$ and $\{n\}$ is determined:

$$\boldsymbol{R_b^n}(\psi) = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \in SO(3) \tag{4}$$

where:

$$\dot{\boldsymbol{\eta}} = \boldsymbol{R_b^n}(\psi)\boldsymbol{\nu} \tag{5}$$

2.2 Target Localization

When applying a pure pursuit localization, the error as distance to the target is simply found by the relative distance between the target and the vehicle as the euclidean distance:

$$d_t = \sqrt{(\Delta x)^2 + (\Delta y)^2} \qquad \text{where:} \quad \Delta x = x_{\text{target}} - x_{\text{usv}}, \quad \Delta y = y_{\text{target}} - y_{\text{usv}} \tag{6}$$

likewise determining the azimuthal angle $\theta \in (-\pi, \pi)$ between the USV and target can be done by the arc tangent:

$$\theta = atan2(\Delta y, \Delta x) \tag{7}$$

2.3 Proximal Policy Optimization

Schulman, Wolski, Dhariwal, Radford, and Klimov (2017) proposed Proximal Policy Optimization (PPO) as a new group of policy gradient methods in policy-based reinforcement learning, built on the methods of trust region policy optimization (TRPO). The policy refers to the course of action an agent takes in a given scenario, based on a set of parameters, while the gradient indicates how changes to these parameters affect performance. A policy gradient algorithm uses this information to adjust the parameters in order to maximize a given reward function by following the expected return. The PPO algorithm in Schulman et al. (2017), combining the policy gradient loss $L_t^{\text{CLIP}}(\theta)$, value loss function $L_t^{\text{VF}}(\theta)$, and the entropy bonus $L_t^{\text{S}}(\theta)$ is presented as:

$$L_t^{\text{CLIP+VF+}S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{8}$$

The coefficients c_1 and c_2 are used to weigh the importance of each term, generally set to 1 and 0.01 respectively. The value loss L_t^{VF} is a squared-error loss function $(V_\theta(s_t) - V_t^{\text{target}})^2$, where $V_\theta(s_t)$ is the predicted total future reward from state s_t under the current policy. This prediction is generated by a neural network that uses the parameters θ and the input state to estimate the return (Schulman, Moritz, Levine, Jordan, & Abbeel, 2015). V_t^{target} represents an estimate of the actual return, computed from observed rewards during training. The final term S is the entropy bonus, which can encourage exploration by penalizing deterministic policies and promoting more stochastic action selection while training.

2.3.1 Clipped Surrogate Objective

As TRPO maximizes a surrogate objective, an alternative approximation of the main objective which is unconstrained, often leading to excessively large updates to the policy. the PPO clipped surrogate objective was proposed as a method of determining the policy gradient loss to avoid new policies from changing too much from older ones (Schulman et al., 2017). The clipped surrogate objective function is given as:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{9}$$

Where $\hat{\mathbb{E}}_t$ denotes the empirical average over a finite batch of samples, which is the average computed from a set of data collected during one training cycle. Each sample in the batch includes the agent's interaction with the environment, such as states, actions, and rewards. The $r_t(\theta)$ term is the probability ratio $\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\right)$, which indicates how the probability of taking an action (a_t) in a state (s_t) has changed under the new policy compared to the old policy by reflecting how much the policy updates for that specific action in the given state. Where the epsilon is a hyperparameter, the following term will clip the surrogate objective to keep it within a desired bound [1-ϵ, 1+ϵ] to keep the training stable. The final $\hat{A}_t$ term is an estimator of the advantage at timestep t:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \quad \text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{10}$$

Following Schulman et al. (2017), δ_t is the difference between the predicted value of a state to the actual reward as an indicator of the agent doing better or worse than expected. The discount factor (γ) determines how much future rewards matter relative to those in the immediate future. λ refers to the Generalized Advantage Estimator (GAE) smoothing parameter, reducing the variance in training to improve stability.

2.4 Multilayer Perceptron

The Multilayer Perceptron (MLP) is one of the most commonly used types of NNs in deep learning (Popescu, Balas, Perescu-Popescu, & Mastorakis, 2009). It uses a feedforward architecture, as illustrated in Figure 3, consisting of an input layer that receives the raw data, one or more hidden layers that process the inputs using activation functions, and an output layer that produces the final result as a prediction, classification, or control action, depending on the application. Training is performed by backpropagation, which alternates a forward pass which computes the activations, loss evaluation, and a backward pass which computes the gradients and update weights until convergence, where further training would return negligible improvement.

Each unit in the hidden layer is an artificial neuron that receives multiple inputs $x = [x_1, x_2, ..., x_I]$, either from the previous layer of neurons or directly from the input layer (Costa et al., 2023). In the forward pass, the neuron computes a weighted sum of its inputs:

$$u_j = \sum_{i=0}^{I} w_{ji} x_i \tag{11}$$

where I denotes the input dimension and w_{ji} represents the weights associated with each input. These weights are updated during training via backpropagation and gradient descent. The result u_j, is then passed through a nonlinear activation function to compute the neuron's output:

$$\hat{y}_j = f(u_j) \tag{12}$$

MLPs typically use nonlinear activation functions to enable the network to model complex relationships. Common examples of include the sigmoid function, hyperbolic tangent (tanh), and Rectified Linear Unit (ReLU) (Costa et al., 2023; Stable Baselines Revision, 2024):

$$\textbf{Sigmoid:} \quad \sigma(x) = \frac{1}{1+e^{-x}} \tag{13}$$

$$\textbf{tanh:} \quad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{14}$$

$$\textbf{ReLU:} \quad \text{ReLU}(x) = \max(0, x) \tag{15}$$

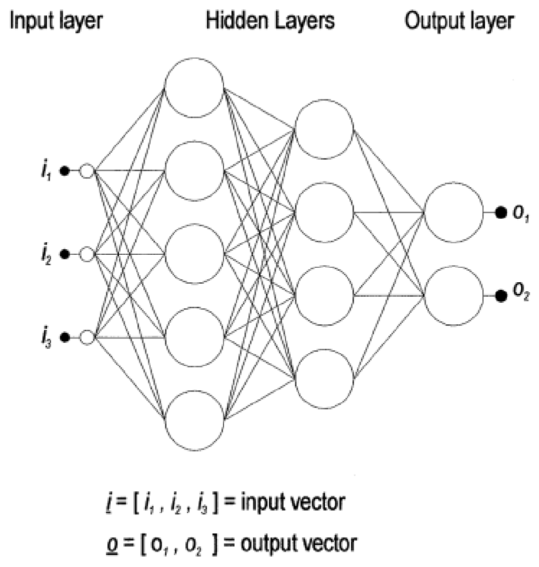
Input layer
Hidden Layers
Output layer
i_1
i_2
i_3
o_1
o_2
$\underline{i} = [i_1, i_2, i_3]$ = input vector
$\underline{o} = [o_1, o_2]$ = output vector

Figure 3: Multilayer perceptron with two hidden layers, three input layers, and two output layers. (Gardner & Dorling, 1998)

2.4.1 Loss, Backpropogation & Optimization

Once the network's forward pass (Eq. 11-12) and activation function (Eq. 13-15) are defined, training proceeds by choosing a suitable loss function to minimize loss, which in this case is the clipped surrogate loss (Eq. 9). This is done through a backwards pass, also known as backpropagation, where the gradient of the loss is calculated by determining the error signal at the network's output, propagating it backward through each layer to compute the partial derivatives of the loss with respect to every weight, and then using an optimizer to adjust the weights (wji) that most rapidly decreases the loss.

The Adaptive Moment Estimation (Adam) optimizer in Algorithm 1 (Diederik, 2014), is a Stochastic Gradient Descent (SGD) method, which is an iterative method optimizing the object function that automatically adjusts each parameter's learning rate by keeping simple running averages of both the gradients and their squares. It adds only a small amount of extra memory for these averages, often converging faster than base SGD. Following Diederik (2014), the Adam optimizer follows the main steps:

Algorithm 1 Simplified Adam Optimizer (Diederik, 2014)

```
Require: learning rate $\alpha$, decay rates $\beta_1, \beta_2 \in [0, 1)$
1: Initialize $m \leftarrow 0,\ v \leftarrow 0,\ t \leftarrow 0$
2: while not converged do
3:     $t \leftarrow t + 1$
4:     $g \leftarrow \nabla_\theta f(\theta_{t-1})$                          ▷ compute current gradient
5:     $m \leftarrow \beta_1 m + (1 - \beta_1) g$                        ▷ update first moment
6:     $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$                       ▷ update second moment
7:     $\hat{m} \leftarrow m/(1 - \beta_1^t), \quad \hat{v} \leftarrow v/(1 - \beta_2^t)$     ▷ apply bias correction
8:     $\theta \leftarrow \theta - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \varepsilon}$                        ▷ parameter update
9: end while
```

three hyperparameters are set to determine the step-size, decay rates, and a stability constant. the learning-rate determines the scale of each policy update, which increases leading to faster learning speed against the risk of overshooting the minima if updates are too large. The first exponential decay rate β_1 controls the moving average, which is how many past gradients the optimizer remembers, with smaller values leading to react more to recent changes. β_2 is similarly changed for the moving average of past squared gradients as a variance estimate, with smaller values yielding a more noisier but more responsive variance estimate that adapts quickly to changes in gradient size, and larger values producing a smoother, more stable estimate that prevents drastic changes per step.

After each step increment, where $f_t(\theta)$ is a stochastic scalar function differentiable with the parameter θ, the current gradient in step 4 of Algorithm 1 is calculated by the vector of the partial derivatives of f_t at each timestep t. In the following steps the gradient exponential moving averages (m_t) and squared gradient (v_t) are estimated. When decay rates are small $\beta \approx 1$ and during the initial timesteps these are biased towards zero, $\hat{m}_t$ and $\hat{v}_t$ are calculated to correct for these biases. Finally the last steps computes the policy update through the predetermined parameters, where ϵ is a simple hyperparameter to avoid zero division.

2.5 Reinforcement-Learning Environment

Applying gymnasium, a vectorized environment can be created as an object-oriented class. In any complete or custom environment the following functions must be defined (Towers et al., 2024):

- *Initial function*: generation of the environment's initial state.
- *Observation function*: returns given states of the environment.
- *Transition function*: computes environment's next state from an action.
- *Reward function*: computes rewards from the changes in the states given actions.
- *Terminal function*: computes if the episode should end.

Following Towers et al. the observation function for the Otter USV must be contained within the possible actions and observations for the vehicle, given as the observation space (S) and action space (A). Following the SNAME notation in Table 1 and Eq. (6)-(7), the observation space can be defined as:

$$\boldsymbol{S} = \begin{bmatrix} d_t & \theta_t & u & v \end{bmatrix}^{\mathrm{T}} \tag{16}$$

The distance d_t and heading error θ_t to the target, as well as the surge and sway velocities are applied as the environments observation space, while the action space simply represents the vessels possible control actions:

$$\boldsymbol{A} = \begin{bmatrix} \tau_x & \tau_\psi \end{bmatrix}^{\mathrm{T}} \tag{17}$$

Where the action space denotes the possible forces the vessel can generate in surge and yaw.

3 Materials

The implementation is written entirely in Python, using the Stable Baselines3 (SB3) library, a framework for reinforcement learning algorithms built on PyTorch (Stable Baselines Revision, 2024). All simulations were conducted on an Acer Nitro 16 laptop. Details about the hardware specifications and software environment are provided in Appendix A to ensure reproducibility.

The GitHub repository containing all source code and contributions is listed in Appendix B.

4 Methods

This chapter presents the project implementations, such as creation of the simulator environment, reward handling, and the test procedures followed in control method comparisons.

4.1 Gymnasium environment

Gymnasium is an Application Programming Interface (API) for reinforcement learning in diverse environments (Towers et al., 2024). As a maintained fork of OpenAI's Gym library it provides an easy interface for working on pythonic RL cases, also providing several built-in environments which can be directly applied to RL. Following Farama Foundation (2023), the simulation environment in Fossen (2023) can be applied to create a custom gymnasium environment by applying the states and actions in Eq. (16) and (17).

In the original Python Vehicle Simulator, the full system dynamics are rolled out over a specified time horizon using a fixed sample time. To integrate with SB3's PPO implementation, the simulator was modified to include a single-timestep interface that conforms to Gymnasium's *step()* function. This allows each call to *step()* to represent one environment transition in the reinforcement learning, while internally rolling out the dynamics in full before returning the next state, reward, and termination flag. This structure enables proper gradient updates and policy optimization during training.

4.2 PPO Implementation

Following the NN definition from the second chapter, and based on the observation and action spaces, the PPO agent using the SB3 MlpPolicy applies a neural network with an input layer corresponding to the observation space, two hidden layers with 64 nodes each, and an output layer corresponding to the action space, consisting of two actions (Stable Baselines Revision, 2024). Following the SB3 implementation, a standard PPO architecture was created as displayed in Listing 1:

Listing 1: PPO architecture

```
class OtterENV(gym.env):
    def __init__(self, simulator, otter)
        self.action_space = ...
        self.observation_space = ...

    def step(self, action)
        action = ...
        reward = ...
        return obs, reward, done, info

    def render(self, mode="human")

    def seed(self, seed=None)
```

The *OtterEnv* class is created to allow the PPO agent to interact with the environment through actions and observations. An object from the Otter API is also instantiated and integrated into the simulation environment to retrieve the correct state information at each timestep.

due to the USV having two outputs, the policy is trained to select actions based on observations relevant to both outputs. Increasing the number of observations does provide more information about the simulation states, however it can also make training more difficult as it may require more episodes or lead to convergence issues. Similarly, it can lead to overfitting, where an increase in the observation space may cause the model to converge to unreliable local minima by identifying patterns that do not accurately reflect the true dynamics of the USV. Therefore, certain variables such as the yaw angle are excluded from the observation space. In this case, the yaw angle is represented through the azimuth angle error, which provides equivalent directional information about the USV.

As Otter is capable of producing up to 200N of force, split between surge and yaw at any given time, the USV action space has been normalized in the range $(-1, 1)$. This is similarly done to target distance observations as this increases consistency, reducing the difficulty for the model to learn as it does not need to understand the scale of observations in different scenarios.

4.2.1 Step function & Reward handling

Reward signals in deep reinforcement learning provide the feedback necessary for an agent to learn which actions maximize long-term performance. At every step increment in the training, rewards determine if the current observation is an improvement or not compared to previous observation, given an effective reward structure. In pure pursuit target tracking, the most important reward component is the change in distance to the target. Rewarding reductions in this distance encourages the agent to minimize it consistently, thereby improving tracking performance. The pseudo-code for the full reward function handling is presented in Algorithm 2:

Algorithm 2 Reward Function

```
Require: current distance $d_t$, previous distance $d_{t-1}$, heading error $\theta_t$, sway velocity $v$, yaw rate $r$
 1: if first call (no $d_{t-1}$ stored) then                  ▷ initialize previous distance
 2:     $d_{t-1} \leftarrow d_t$
 3: end if
 4: reward $\leftarrow 0$
 5: reward $\leftarrow$ reward $+ 3 \cdot (d_{t-1} - d_t)$                  ▷ progress bonus
 6: $d_{t-1} \leftarrow d_t$                                    ▷ update stored distance
 7: if circular_tracking then
 8:     reward $\leftarrow$ reward $- 0.1 \cdot |r|$                  ▷ penalize yaw rate
 9: else
10:     reward $\leftarrow$ reward $- 0.1 \cdot d_t$                  ▷ penalize distance
11:     reward $\leftarrow$ reward $+ 0.5 \cdot \cos(\theta_t)$       ▷ reward heading alignment
12:     reward $\leftarrow$ reward $- 0.2 \cdot |v|$                  ▷ penalize sway velocity
13:     reward $\leftarrow$ reward $- 0.1 \cdot |r|$                  ▷ penalize yaw rate
14: end if
15: if $d_t <$ target position radius then
16:     reward $\leftarrow$ reward $+ 1.0$                    ▷ bonus for staying close to target
17: end if
18: return reward
```

The largest changes in the reward are driven by the change in target distance at each step for both straight-line and circular target tracking. A slight penalty is applied to large yaw rates to discourage oscillation and encourage smooth turning, but the penalty is kept small to ensure that minimizing the target distance remains the main priority. In straight-line tracking, additional rewards are given for heading alignment and low sway velocities, as these promote correct USV orientation and straight movement. These rewards are not applied in circular tracking, where rewarding heading alignment can lead the agent to settle into a local maximum by circling in another location where it can more easily maintain the correct heading. Similarly, penalizing sway velocity is in circular movement is counterproductive since this motion on water naturally causes lateral drift.

In the straight line target tracking there is a mild penalty based on the current distance to target. While this was active during model training, after the addition of the progress bonus function this variable is ultimately redundant and should be removed from future iterations.

4.2.2 MLP Policy

Following Popescu et al. (2009), one or two hidden layers are generally sufficient, where too few neurons can result in underfitting, where it fails to accurately capture the patterns in the training data, or too many neurons leading to overfitting. As such the SB3 MlpPolicy using hidden two layers of 64 neurons each should be adequate, however a comparison can not be made without creating a custom actor-critic architecture. In the context of PPO, the MLP is used to approximate both the policy and the value functions, which act as the actor and critic.

Whereas the SB3 PPO method typically applies clipping to the output actions within the range $-1 < u < 1$, a tanh activation function was instead applied in the same range to encourage smoother gradients and more predictable changes in network output. In the context of an USV system, this can result in more stable control actions and reduce the likelihood of continually applying maximum thrust. In a real world scenario this could lead to smoother turning, gentler acceleration, and higher energy efficiency.

4.2.3 Hyperparameter selection

Following the documentation in Raffin, Antonin and Hill, Ashley and Gleave, Adam and Kanervisto, Anssi and Ernestus, Noah (n.d.), the chosen PPO parameters are presented in Table. 2:

Table 2: PPO Hyperparameters Used in Training

Parameter	**Value / Description**
`policy`	`Multilayer perceptron`
`verbose`	1
`device`	`cpu`
`normalize_advantage`	`True`
`gae_lambda`	0.98
`learning_rate`	0.0001
`clip_range`	0.2
`n_steps`	4096 (default is 2048)
`ent_coef`	0.01
`policy_kwargs`	`activation_fn=nn.Tanh`

The GAE parameter was increased from the default value of 0.95 to 0.98 to place greater emphasis on long-term reward dependencies when estimating the advantage function. Similarly, the entropy coefficient was set to 0.01 to encourage more exploration by injecting randomness into the policy, which can slow convergence but improve policy robustness across episodes. While partly a consequence of previous reward shaping being suboptimal, these adjustments were motivated by observations of the model converging to local minima, prioritizing heading accuracy at the expense of minimizing distance to the target. Various n steps values were tested where larger values generally lead to better estimates of advantages and returns, however leading to longer training times. The training rate could possibly be slightly increased, where a larger value could improve training times, and too small learning rates increases the risk of getting stuck in local minima, Increasing the learning rate too much could however lead to model instability.

4.2.4 PyTorch Training Loop

SB3 uses PyTorch as its deep learning framework to implement and train neural networks. PyTorch handles the forward and backward passes during each epoch, enabling the policy and value networks to improve through gradient-based optimization. Following Table 2, after a training phase involving 2048 to 4096 steps of collected observations, actions, and rewards, PPO performs training updates over several epochs. This phase follows the standard PyTorch training procedure: accumulated gradients are first cleared, a forward pass is executed to produce predicted values and action probabilities, and the loss is computed using the clipped surrogate loss described in Equation (9). Gradients of the loss are then calculated via backpropagation, and the network weights are updated using the Adam optimizer.

4.2.5 Training & Evaluation

Applying vectorized environments it is possible to run multiple independent environments in parallel, allowing for several active environments during training (Raffin, Hill, Gleave, Kanervisto, & Ernestus, 2025b). By applying *SubprocVecEnv*, a multiprocess vectorized wrapper is created, which can significantly reduce training length in computationally complex environments, given that the number of environments does not exceed the CPU's number of logical cores. As such, eight environments were employed during model training to reduce training time. Furthermore, the trained model is saved by applying the *VecNormalize* wrapper, which normalizes the agent observations across the vectorized environments, with benefits in improving training stability and performance.

Once the model is trained, it is evaluated over 10,000 timesteps. A running average of the IAE is calculated to assess the agent's average performance, while minimizing the influence of outlier episodes.

4.3 Test Procedure

The DRL controller training and simulations were evaluated in two scenarios:

- *1*: Straight-line target tracking
- *2*: Circular target tracking

where the model training and testing followed the parameters presented in Table 3:

Table 3: Model training & testing parameters

Trajectory Type	**Sampling time**	**Training time steps**	**Episode Termination**
Straight-line tracking (DRL)	0.1	50 000 000	60m
Straight-line tracking (PI & PID)	0.02	n/a	60m
Circular tracking (DRL)	0.1	90 000 000	$2\pi \cdot 20\,\mathrm{m}$
Circular tracking (PI & PID)	0.02	n/a	$2\pi \cdot 20\,\mathrm{m}$

Due to the significantly increased computational cost associated with smaller sampling times, the DRL simulations used a sampling time five times larger than that of the PID control simulations. While this may slightly reduce the accuracy of the simulated dynamics, it provides a practical benchmark for evaluating DRL performance before committing to longer and more resource-intensive training runs.

4.3.1 Target tracking procedure

In the straight-line tracking scenario, the USV starts 10 meters east of the target's initial position and is oriented with a northward heading. The target moves north at a velocity of 1.5 m/s for a distance of 60 meters, after which the episode ends. In the circular tracking scenario, the USV begins 10 meters southeast of the target's initial position, while the target moves along a clockwise circular path with a radius of 20 meters. The episode ends once the target completes a full rotation. All simulations assume ideal conditions, with no disturbances from wind, waves, or ocean currents.

4.3.2 Error Indices

The Integral of Absolute Error (IAE) was used to evaluate the DRL control performance improvements over the training period:

$$\mathrm{IAE}_{d_t}(t) = \int_0^t \|d_t\| \, dt \tag{18}$$

$$\mathrm{IAE}_{\theta}(t) = \int_0^t \|\theta_t\| \, dt \tag{19}$$

where d_t denotes the distance to the target and θ_t the heading error at time t. While heading error is less critical in straight-line pursuit, IAE_θ becomes more relevant in circular tracking, where frequent heading adjustments are required. Monitoring both indices provides insight into tracking quality across different trajectory demands.

5 Results

the following section briefly presents the results of the DRL and PID control target tracking, including the IAE error indices, the reinforcement learning model improvements over its training period, and finally a visualization of the USV path and control allocations.

5.1 Target Tracking Error Indices

Table 4 presents the IAE error indices for the distance to the target across straight-line and circular trajectories. The corresponding IAE values for the heading error during circular tracking are shown in Table 5.

Table 4: Comparison of IAE for distance to target between DRL and PID Control.

Trajectory Type	IAE (DRL)	IAE (PID)
Straight-line tracking	91.47	472.30
Circular tracking	284.67	1076.53

Table 5: Comparison of IAE for heading error between DRL and PID Control

Trajectory Type	IAE (DRL)	IAE (PID)
Circular tracking	55.61	53.05

These indices serve as quantitative performance measures for comparison between the control strategies. The PID simulations are deterministic, producing consistent results for a given set of parameters. For the DRL controller, the reported IAE values represent the running average over multiple episodes. In both trajectory types, the DRL controller achieved lower IAE values for distance to the target compared to the PID controller, with the heading error being close to identical in circular pursuit.

5.2 Straight-line Target Tracking

5.2.1 DRL control law

Figure 4 highlights the model's performance improvements over the training period of 50,000 episodes for straight-line target tracking. The model rapidly improves until around 8000 episodes, where a knee point appears. Beyond this point the rate of improvement slows with diminishing returns. However, gradual gains continue throughout the training, with the IAE values more than halving after the knee point, indicating sustained but slower learning.

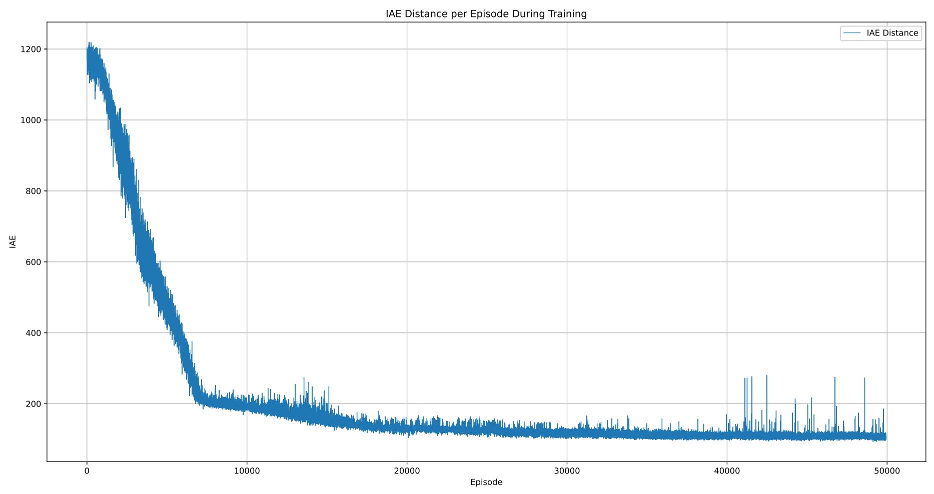

Figure 4: Straight-line: Integral of absolute error target distance trend under training.

The final evaluated model in Figure 5 shows the USV effectively tracking the target over a 60-meter distance, where the corresponding colors of the target and USV indicate their positions at each timestep. After an initial correction to align with the target path, the vehicle is able to catch up and remain within one meter of the target throughout the simulation.

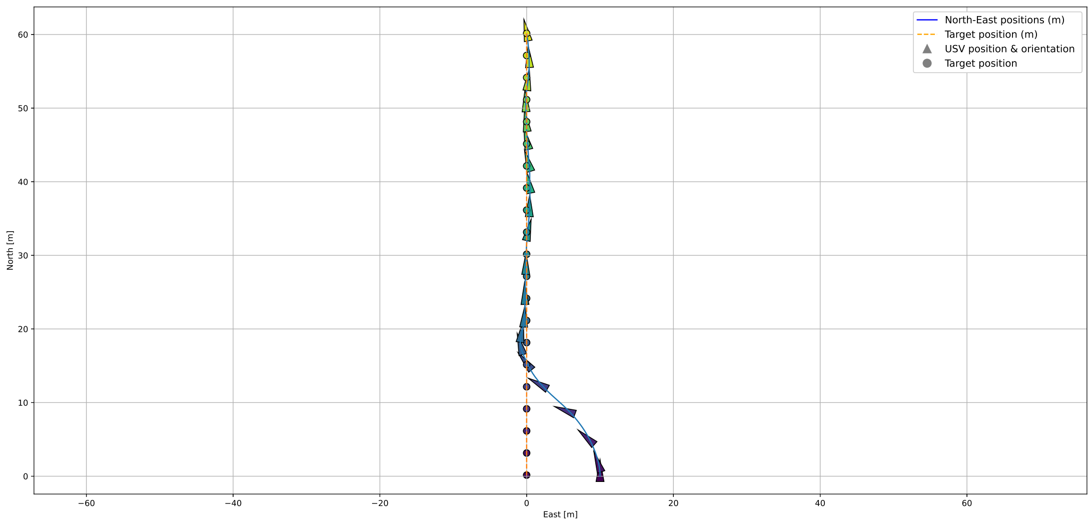

Figure 5: Straight-line: Final DRL model target pursuit.

The DRL control actions shown in Figures 6–7 are passed through the simulator's control allocation highlighted in Figure 8 as the normalized control signals (blue graph), which accounts for the physical actuator time constants of the vessel. As the control allocation favors yaw adjustments, the model applies near-maximal surge force throughout most of the simulation, with continuous heading corrections. Although these adjustments become smaller after the initial alignment with the target path, the results suggest that the model struggles to reach a steady state.

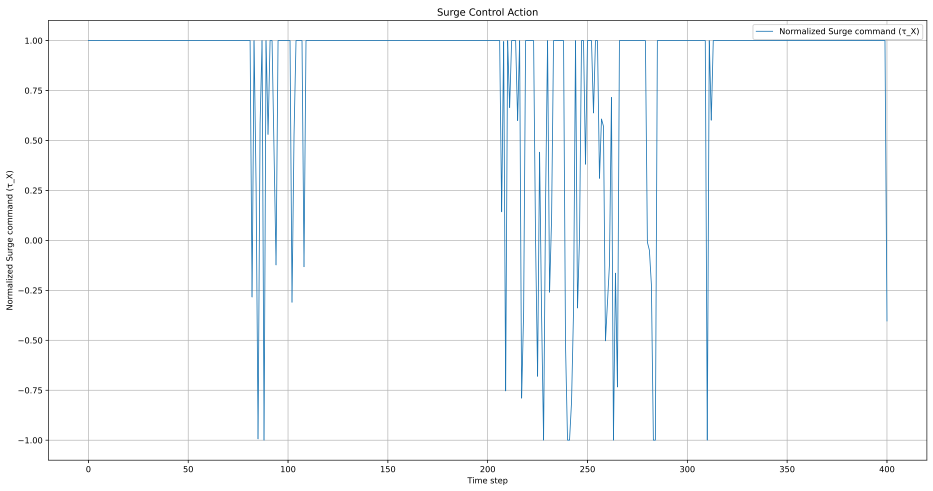

Figure 6: Straight-line: Final DRL Model surge control action.

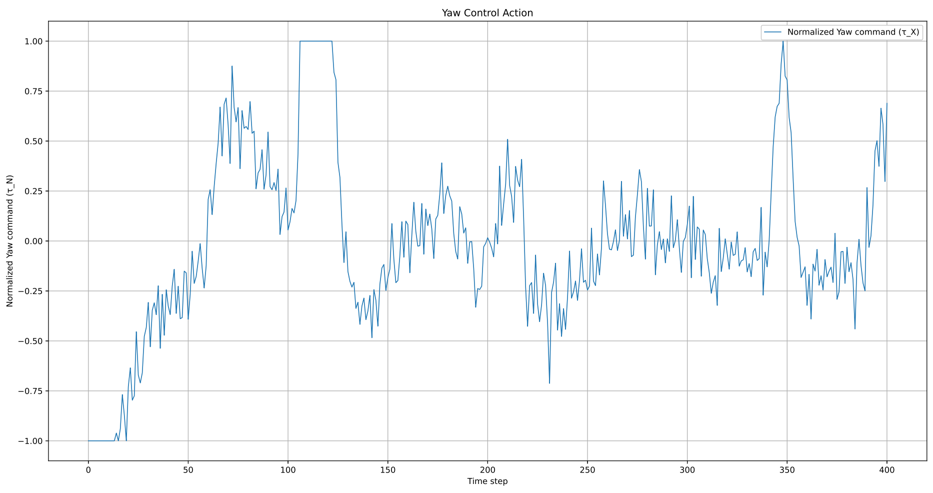

Figure 7: Straight-line: Final DRL Model yaw control action.

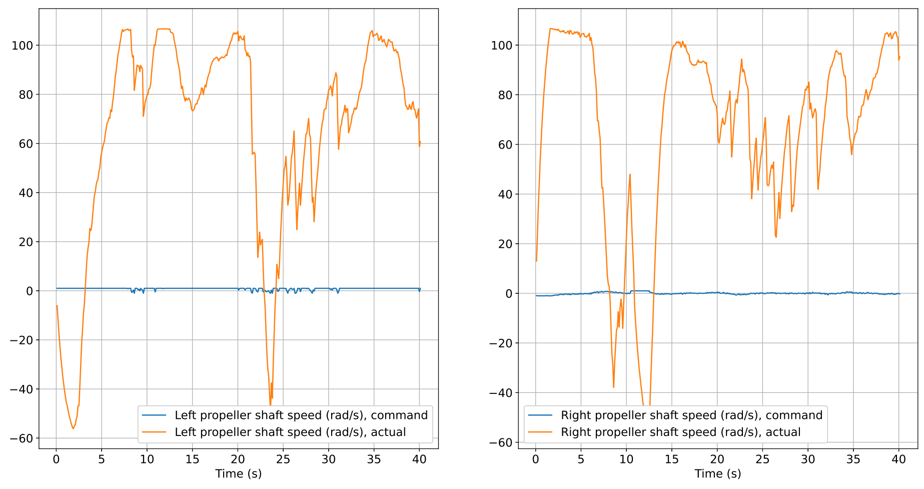

Figure 8: Straight-line: Final DRL model Control forces.

5.2.2 PI & PID control law

Figure 9 displays the performance of the PID control law in the straight-line target tracking scenario. The controller manages to align the USV with the target path after an initial delay. While the USV eventually approaches the target trajectory, it struggles to fully close the distance, maintaining a consistent offset throughout the simulation. This behavior is reflected in the IAE values, which remain significantly higher compared to the DRL controller.

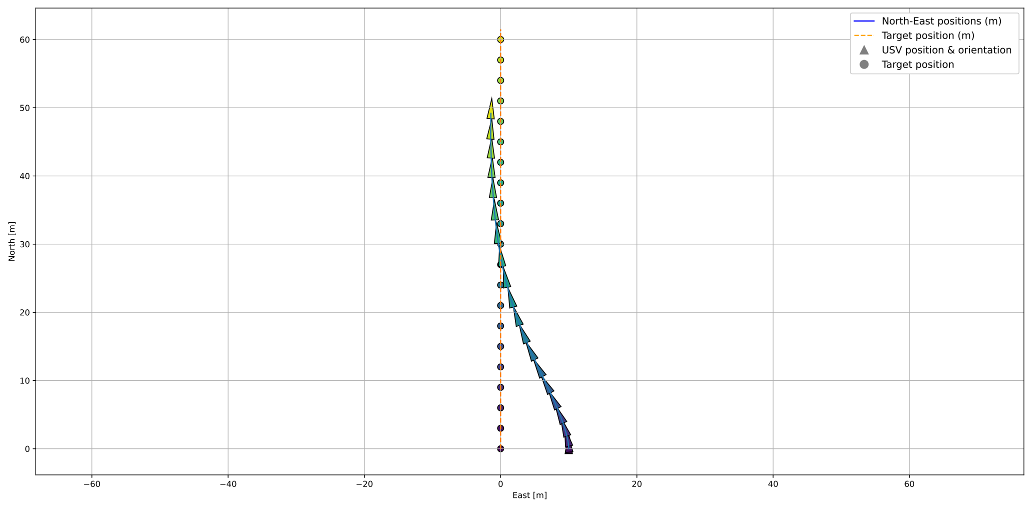

Figure 9: Straight-line: Target tracking PID.

The PID control actions, with PI surge and PID yaw controls displayed in Figure 10, are smoother compared to the DRL controller, with moderate initial adjustments followed by smaller, more consistent corrections. As the simulation progresses, the control signals gradually stabilize, indicating a response that converges toward steady-state behavior.

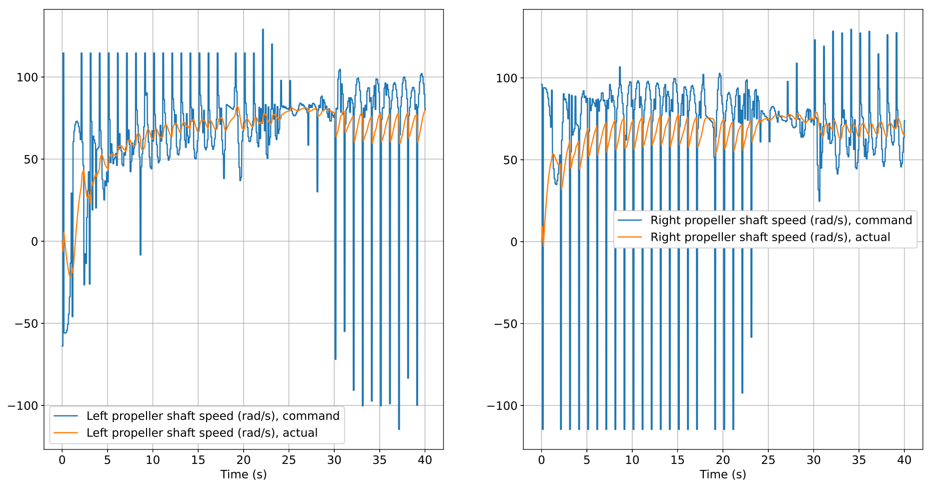

Figure 10: Straight-line: PID control forces

5.3 Circular Target Tracking

5.3.1 DRL control law

The circular target tracking model was trained over 100,000 episodes, with each episode ending after the target completed a full circular trajectory of $2\pi \cdot 20\,\mathrm{m}$. Since each episode is more than twice as long as those in the straight-line tracking case, the IAE knee point, visible in Figure 11, occurs earlier at around episode 4,000. Beyond this point, the rate of improvement slows significantly. Although training continued for more than three times the duration shown in the plot, the performance gains beyond this point were minimal.

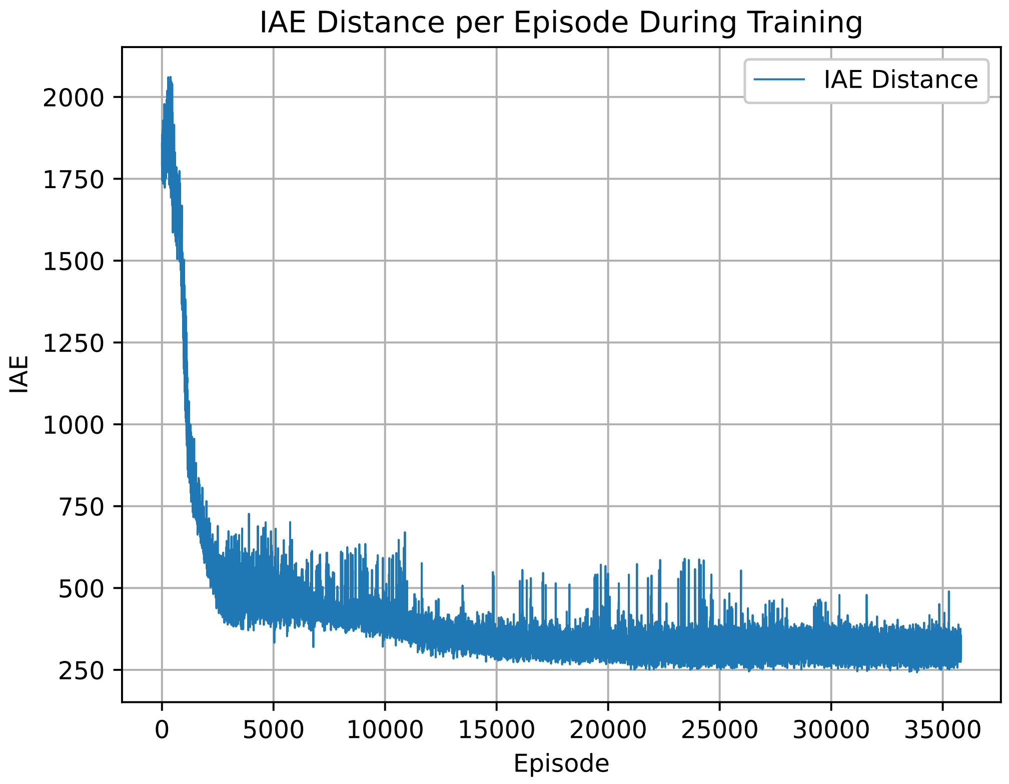

Figure 11: Circular: Integral of absolute error distance target distance trend under training.

As shown in Figure 12, the heading IAE decreases rapidly during the initial training episodes and then stabilizes. Beyond this point, only minor fluctuations are observed, with limited further improvement, which is an expected outcome based on the reward structure.

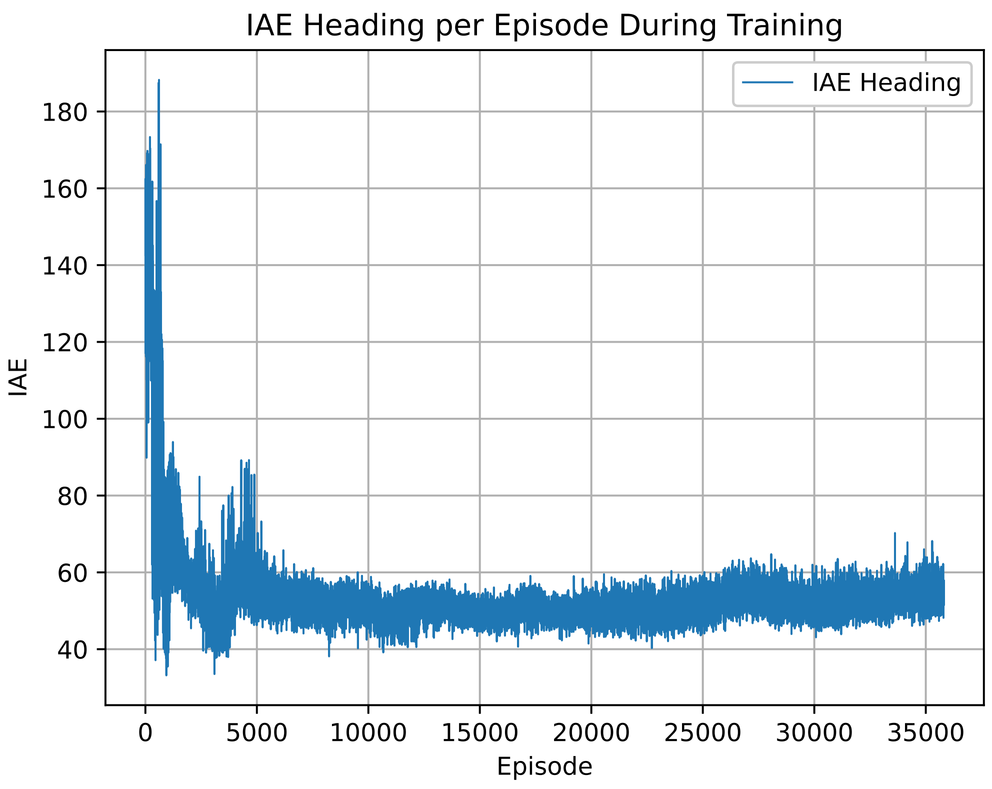

Figure 12: Circular: Integral of absolute error distance heading trend under training.

The final evaluated model presented in Figure 13 shows the USV tracking the target along a circular path. After an initial phase of catching up to the target, the USV briefly overshoots, causing a misalignment with the desired path. As a result, the vessel must readjust its heading and regain proximity to the target, staying close to the target path in the final stages of the evaluation.

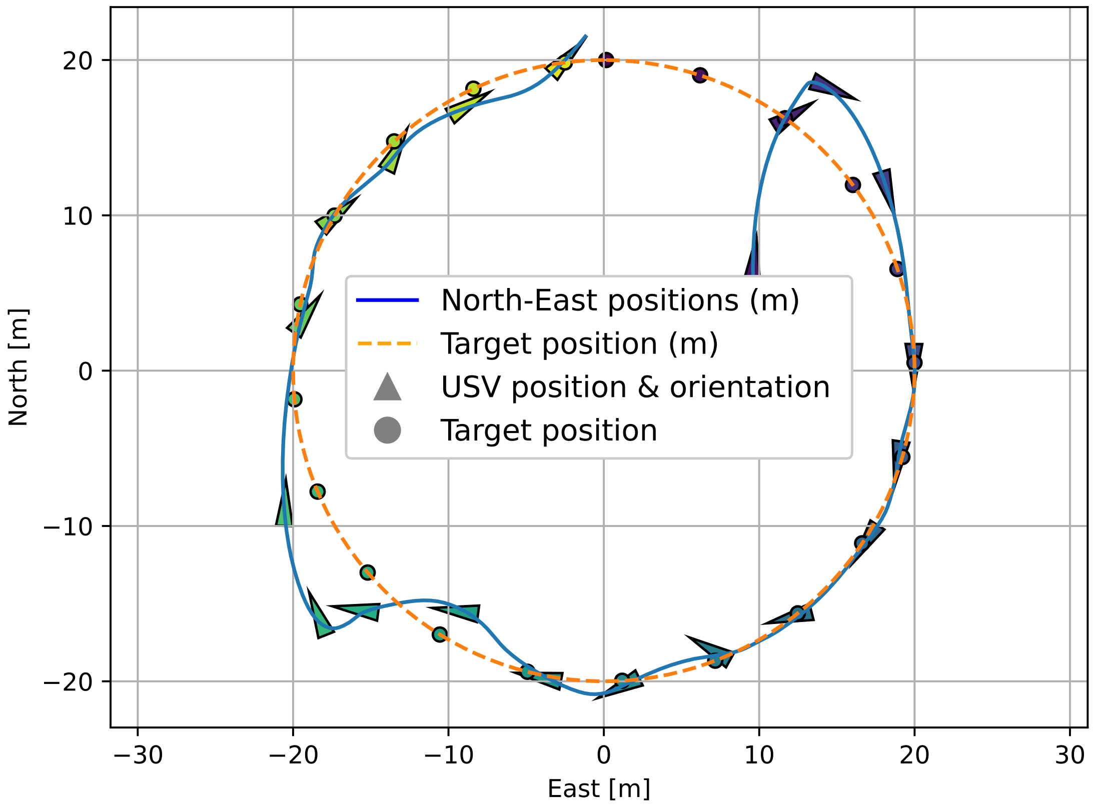

Figure 13: Circular: Final DRL model target pursuit.

Similar to the straight-line case, Figures 14-16 display an initial increase in surge with heading adjustments to align with the target path, however struggling to reach a steady state, corresponding with the alignment issues seen in Figure 13.

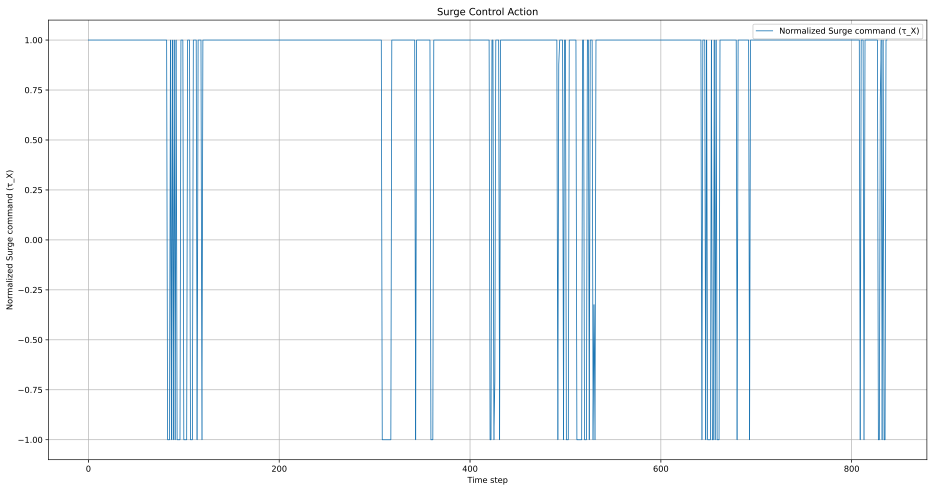

Figure 14: Circular: Final DRL Model surge control action.

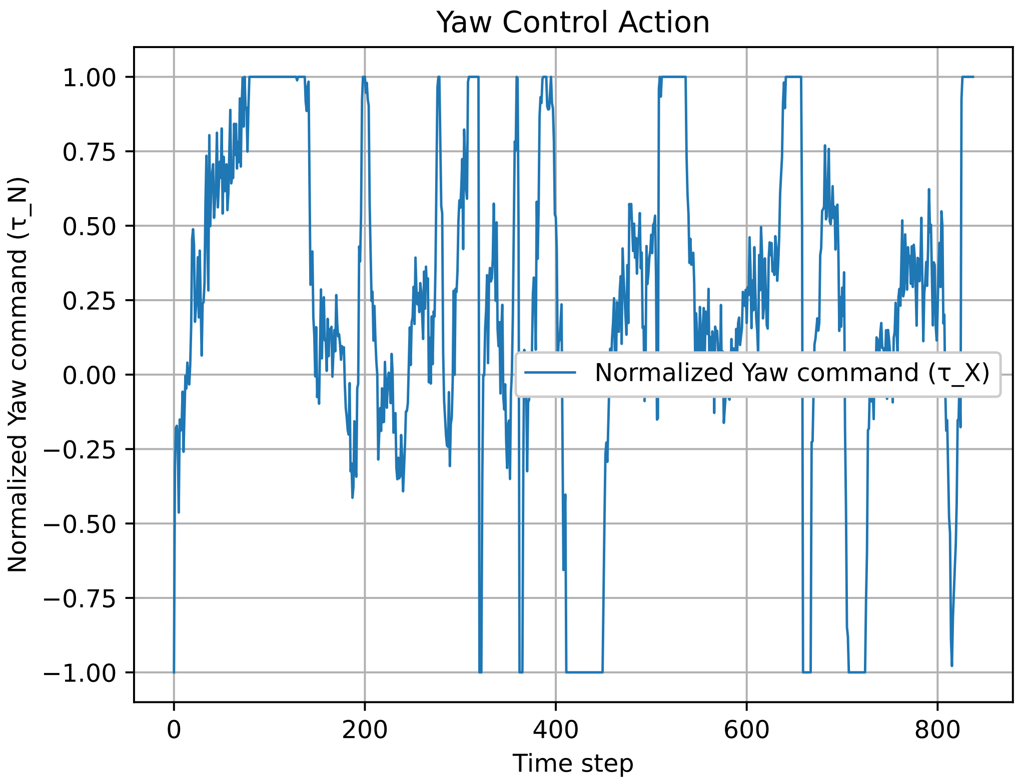

Figure 15: Circular: Final DRL Model yaw control action.

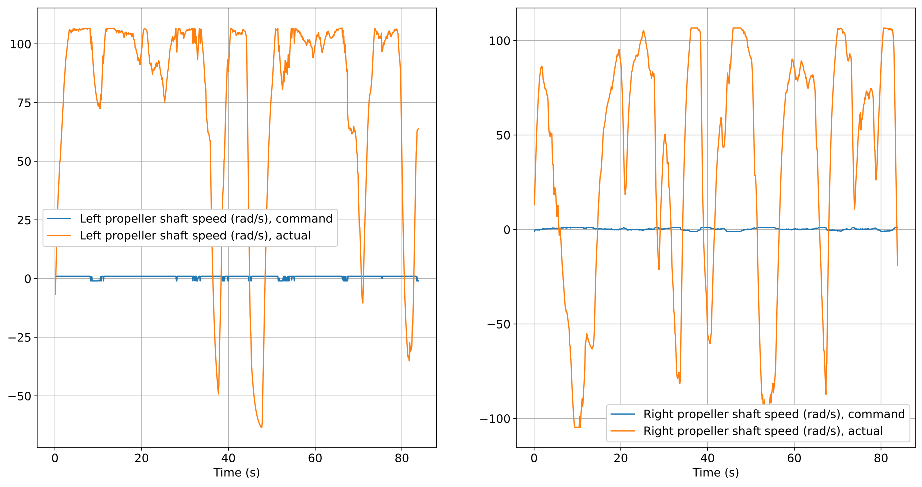

Figure 16: Circular: Final DRL model Control forces.

5.3.2 PI & PID control law

Figure 18 displays the PID control law performance in the circular tracking scenario. The USV follows a smooth trajectory but consistently lags behind the target along the circular path. Although the tracking remains stable, the controller is unable to close the positional gap, resulting in a persistent offset throughout the simulation, and a large IAE error.

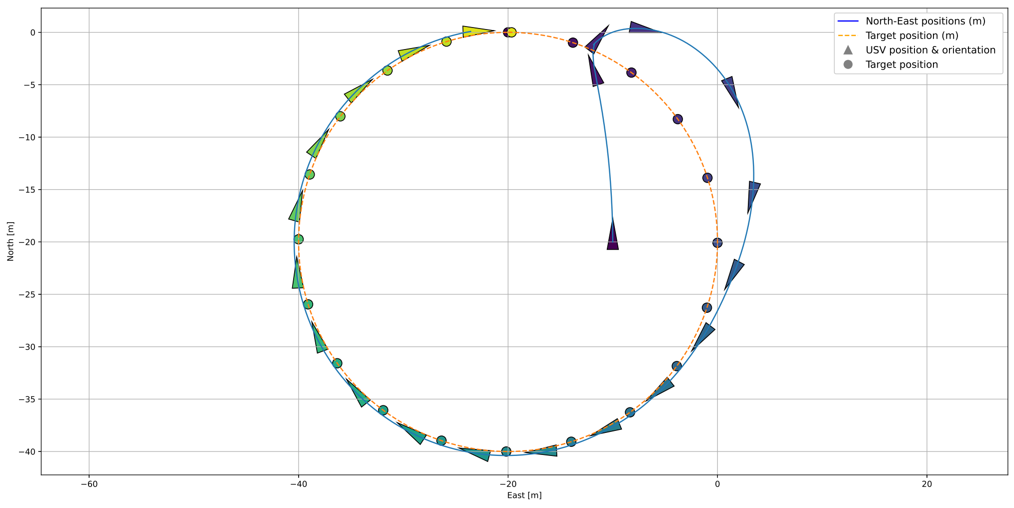

Figure 17: Straight-line target tracking PID.

The PI and PID control forces shown in Figure 18 exhibit much smoother inputs compared to those of the DRL controller, with an initial increase in surge and heading adjustments to reach the target path, followed by a steady-state response.

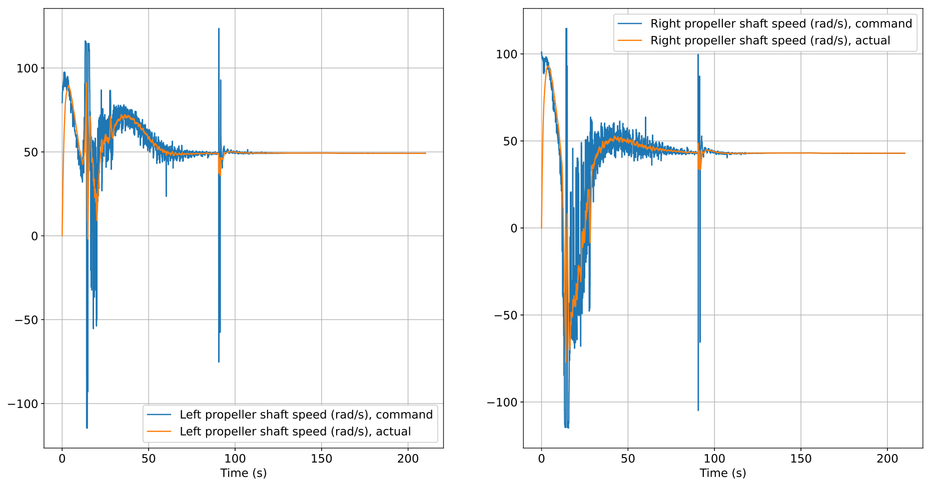

Figure 18: Circular: PID control forces.

6 Discussion

The project main findings are discussed in the following chapter. The different capabilities of the DRL and PID control laws are highlighted, and potential improvements to the final DRL agent are discussed.

6.1 Straight-line target tracking

In the first case of straight-line pursuit, the DRL model shows significant improvements compared to the PID control law, highlighted by the difference in the error indices. The agent is able to intercept the target path faster while maintaining close proximity to the target throughout the remainder of the evaluation. The controller does, however, struggle to reach a steady-state response, which is likely a result of attempting to remain close to the target at all times, leading to frequent small control adjustments. A possible solution to improve this would be to decouple the control signals from the simulator's control allocation, allowing the model to directly output desired forces rather than normalized action parameters. This would enable the implementation of a reward function that explicitly promotes smooth control actions. While the actuator time constants are applied in the simulation environment, resulting in realistic simulator outputs, constant changes to actuator outputs can lead to increased wear in practical real-world employment of the agent.

In contrast, the PID controller produces significantly smoother control signals, resulting in more fluid trajectories despite its limited ability to close the gap to the target. This smoothness leads to less frequent actuator adjustments and more stable motion, which is advantageous for practical deployment. The sharp and continuous heading corrections seen in the DRL model could be mitigated by incorporating penalties on yaw rate or control effort in the reward function, encouraging the agent to prioritize both accuracy and control efficiency. While increasing the gains on the surge PI controller potentially could improve the USV's ability to intercept the target more quickly, this often leads to overshooting, which worsens subsequent tracking performance. Due to the underactuated nature of the USV, recovering from such overshoot is challenging and can cause instability in the control response.

6.2 Circular target tracking

Due to the computational cost of longer episodes, their durations were set specifically to match the distances used in the test procedures during training. While this approach is appropriate for straight-line pursuit, slightly increasing the episode length for circular target tracking could lead to improved results. In the circular case, the USV must intercept the target path at a near-perpendicular angle, spending much of the episode duration aligning its position and heading. Additionally, the higher velocities increase the likelihood of overshooting the target position. Doubling the episode duration for the target to complete two full laps in this case would give the agent more time to reach a steady-state position with only minor adjustments.

Additionally, when training the DRL agent for circular target tracking, it would likely yield better results to start the episode with the USV positioned close to and aligned with the target's position and heading. This setup would allow the model to focus on learning to track the target, rather than spending much of the training duration on interception. A model trained in this manner could later be fine-tuned with varied starting positions to improve its ability to handle target interception scenarios.

Similarly to straight-line pursuit, the DRL agent struggles to employ smooth control signals, signaling a need for a more refined reward function or control allocation method.

6.3 Improving the DRL agent

As *VecNormalize* was introduced in the later stages of the implementation, actions and observations were initially normalized manually to improve performance. These manual normalizations were not removed upon integrating *VecNormalize*, which may have led to degraded results due to redundant or conflicting normalization, where this at best introduces unnecessary duplication.

Tuning the PPO hyperparameters could further improve training stability and control performance. A lower learning rate may help reduce oscillatory behavior in policy updates, while a smaller clipping range can prevent overly aggressive changes that destabilize training. Increasing the entropy coefficient slightly encourages exploration and avoids premature convergence to overly reactive policies. Additionally, using a larger number of steps per update and a smaller batch size can improve the stability and efficiency of each training iteration. However, changes to hyperparameters require retraining the agent, which is time-consuming and demands a substantial number of episodes before reliable conclusions can be drawn.

Applying another policy based RL algorithm, such as Soft Actor-Critic (SAC) could be an alternative to PPO for this type of control task. SAC is more sample-efficient and can learn effectively from data collected by previous versions of the policy or from replay buffers, rather than requiring fresh on-policy data from each policy update (Raffin, Hill, Gleave, Kanervisto, & Ernestus, 2025a). This allows SAC to make better use of each environment interaction, reducing the overall training time compared to on-policy methods like PPO. This can lead to more stable actions and fewer abrupt actuator changes, which is desirable in real-world applications. SAC's ability to learn effective policies with fewer interactions makes it well suited for scenarios with long episodes or high computational cost.

Due to the computational intensity associated with higher sampling rates, the PID controller was simulated at 50 Hz, while the DRL controller operated at a reduced sampling rate of 10 Hz. It is likely that lowering the DRL sampling interval would have improved performance. However, this would require significantly longer training times, which was beyond the scope of this project.

7 Conclusion & Future Work

7.1 Conclusion

This project presents a comparative analysis of learning-based and classical control approaches for USV target tracking, highlighting their respective strengths and implementation considerations. The DRL agent demonstrates significantly better performance at close ranges, converging more quickly to the target and maintaining close proximity during pursuit. However, its control signals are less consistent compared to the smoother and more stable outputs of the PID controller. The PID-controlled trajectories are visibly smoother in both scenarios, likely due to conservative control parameters. Overall, the results reveal a trade-off between the high tracking accuracy of the DRL agent and the stable, low-variation behavior of the PID controller.

7.2 Future Work

While the DRL controller performs well under the defined test conditions, further evaluation is needed to assess its robustness to changes in target speed, trajectory variability, and environmental disturbances. Future work could explore adjustments to the reward function to penalize excessive control activity, along with improvements to the action space and normalization strategy. Hyperparameter tuning and the use of alternative policy-based algorithms, such as SAC, may also enhance training stability and control smoothness. As DRL-based tuning of controller parameters in established methods such as PID or MPC has proven effective in reducing computational demands, this approach could be implemented to determine optimal control parameters more efficiently, potentially addressing issues related to unstable control allocations.

Given the accurate hydrodynamic modeling of the simulator environment, the results are expected to reflect the real-world performance of the DRL model on the Otter USV under ideal conditions. However, practical testing is required to validate the model's effectiveness in real-world deployment.

References

Bingham, B., Aguero, C., McCarrin, M., Klamo, J., Malia, J., Allen, K., ... Waqar, R. (2019, October). Toward maritime robotic simulation in gazebo. In *Proceedings of mts/ieee oceans conference.* Seattle, WA.

Breivik, M., Hovstein, V. E., & Fossen, T. I. (2008). Straight-line target tracking for unmanned surface vehicles.

Caccia, M., Bibuli, M., Bono, R., & Bruzzone, G. (2008). Basic navigation, guidance and control of an unmanned surface vehicle. *Autonomous Robots*, *25*(4), 349–365.

Costa, L., Guerreiro, M., Puchta, E., de Souza Tadano, Y., Alves, T. A., Kaster, M., & Siqueira, H. V. (2023). Multilayer perceptron. *Introduction to Computational Intelligence*, *105*.

Cui, Y., Peng, L., & Li, H. (2022). Filtered probabilistic model predictive control-based reinforcement learning for unmanned surface vehicles. *IEEE Transactions on Industrial Informatics*, *18*(10), 6950–6961.

Diederik, K. (2014). Adam: A method for stochastic optimization. *(International Conference on Learning Representations).*

Farama Foundation. (2023). *Creating custom environments — gymnasium documentation.* Retrieved from `https://gymnasium.farama.org/introduction/create_custom_env/` (Accessed: 2025-04-10)

Fossen, T. I. (2021). *Handbook of marine craft hydrodynamics and motion control* (2nd ed.). John Wiley & Sons.

Fossen, T. I. (2023). *Python vehicle simulator.* Retrieved from `https://github.com/cybergalactic/PythonVehicleSimulator` (GitHub repository, accessed December 18, 2023)

Frafjord, A. T., Saksvik, I. B., Kjerstad, Ø. K., & Coates, E. M. (2024). A comparative study of control laws for a maritime surface vessel tracking an underwater vehicle. *IFAC-PapersOnLine*, *58*(20), 273–280.

Gardner, M. W., & Dorling, S. R. (1998). Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment*, *32*(14-15), 2627–2636.

Gresberg, S. A. S., Johnsen, L. S., & Stensrud, S. (2023, November). *Target tracking for unmanned surface vehicles.* Retrieved from `https://github.com/SigurdGresbe/Otter_Thesis` (Bachelor thesis in mechatroics, Oslo Metropolitan University (OsloMet))

Hu, B., Wan, Y., & Lei, Y. (2022). Collision avoidance of usv by model predictive control-aided deep reinforcement learning. In *2022 ieee international conference on industrial technology (icit)* (pp. 1–6).

Li, J., Chavez-Galaviz, J., Azizzadenesheli, K., & Mahmoudian, N. (2023). Dynamic obstacle avoidance for usvs using cross-domain deep reinforcement learning and neural network model predictive controller. *Sensors*, *23*(7), 3572.

Maritime Robotics. (2025). *Otter unmanned surface vehicle (usv).* Retrieved from `https://www.maritimerobotics.com/otter` (Accessed: 04-Mar-2025)

Popescu, M.-C., Balas, V. E., Perescu-Popescu, L., & Mastorakis, N. (2009). Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, *8*(7), 579–588.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., & Ernestus, N. (2025a). *Stable-baselines3: Soft actor-critic (sac).* `https://stable-baselines3.readthedocs.io/en/master/modules/sac.html`. (Accessed: 2025-05-20)

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., & Ernestus, N. (2025b). *Vectorized environments — stable-baselines3 documentation.* Retrieved from `https://stable-baselines.readthedocs.io/en/master/guide/vec_envs.html` (Accessed: 2025-05-07)

Raffin, Antonin and Hill, Ashley and Gleave, Adam and Kanervisto, Anssi and Ernestus, Noah, y. . . u. . h. n. . A., title = PPO — Stable-Baselines3 Documentation. (n.d.).

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

SNAME. (1950). *Nomenclature for treating the motion of a submerged body through a fluid* (Technical and Research Bulletin No. 1-5). The Society of Naval Architects and Marine Engineers.

Stable Baselines Revision. (2024). Stable baselines documentation: Policies module [Computer software manual]. `https://stable-baselines.readthedocs.io`. (`https://stable-baselines.readthedocs.io/en/master/modules/policies.html` (accessed May 5, 2025))

Towers, M., Kwiatkowski, A., Terry, J., Balis, J. U., De Cola, G., Deleu, T., ... others (2024). Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*.

Wang, Z., Hu, Q., Wang, C., Liu, Y., & Xie, W. (2025). Target tracking control for unmanned surface vehicles: An end-to-end deep reinforcement learning approach. *Ocean Engineering*, *317*, 120059.

Yuan, W., & Rui, X. (2023). Deep reinforcement learning-based controller for dynamic positioning of an unmanned surface vehicle. *Computers and Electrical Engineering*, *110*, 108858.

Zhang, J., Ren, J., Cui, Y., Fu, D., & Cong, J. (2024). Multi-usv task planning method based on improved deep reinforcement learning. *IEEE Internet of Things Journal*, *11*(10), 18549–18567.

Zhao, Y., Qi, X., Ma, Y., Li, Z., Malekian, R., & Sotelo, M. A. (2020). Path following optimization for an underactuated usv using smoothly-convergent deep reinforcement learning. *IEEE Transactions on Intelligent Transportation Systems*, *22*(10), 6208–6220.

A Software & Hardware

All project simulations were ran on the following software and versions:

Table 6: Software list

Software	Version
Windows	11
Python	3.11.9
Numpy	1.26.4
MatPlotLib	3.10.0
Stable-baselines3	2.6.0
Torch	2.6.0+cu126
Gymnasium	1.1.0

The project simulations were done on an ACER Nitro 16 Laptop with the following hardware components:

Table 7: Hardware list

Hardware	Type
CPU	AMD Ryzen 9 8945HS
GPU	NVIDIA Geforce RTX 4070 Laptop
RAM	32 GB DDR5 @ 5600 MHz

B Github Repository

All project files and contributions are highlighted under Otter-USV/Otter API/Otter_dl.py and Otter-USV/Otter API/Otter_simulator_DRL.py in `https://github.com/LSJohnsen/Otter-USV`.