POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Learning Trajectory Tracking For An Autonomous Surface Vehicle In Urban Waterways

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

Author: **Toma Sikora**

Student ID: 963884
Advisor: Prof. Riccardo Scattolini
Academic Year: 2021-22

Abstract

Roboat is an autonomous surface vessel (ASV) for urban waterways, developed as a research project by the AMS Institute and the MIT. The platform can provide numerous functions to a city, such as dynamic infrastructure or autonomous garbage collection.
The goal of this thesis project is to develop a learning-based controller for the Roboat platform with the goal of improving robustness and generalization. When subject to uncertainty in the model or external disturbances, the proposed controller should be able to track set trajectories with less tracking error than the current Nonlinear Model Predictive Controller (NMPC) used on the ASV.
To achieve this, a simulation of the system dynamics was developed based on research done on the platform and previous literature. The simulation also includes the modelling of the necessary uncertainties and disturbances. In this simulation, a trajectory tracking agent was trained using the Proximal Policy Optimization algorithm which was then validated and compared to the current control strategy both in simulation and in the real world.

Keywords: reinforcement learning, trajectory tracking, autonomous surface vessel, urban waterways, Model Predictive Control.

Abstract in lingua italiana

Roboat è un natante autonomo di superficie, o "Autonomous Surface Vessel" (ASV) per corsi d'acqua urbani, sviluppato come progetto di ricerca dall'istituto AMS e dal MIT. La piattaforma può fornire numerose funzioni a una città con corsi d'acqua, per esempio utilizzandolo come infrastruttura dinamica o per la raccolta dei rifiuti.

Lo scopo di questa tesi è sviluppare un controllore di tipo learning-based per Roboat con l'obiettivo di migliorare la robustezza del sistema a fronte di incertezza di modello o di disturbi esterni. L'obiettivo del controllore è quello di seguire traiettorie date con un errore di inseguimento inferiore rispetto a quello garantito da un algoritmo di Nonlinear Model Predictive Controller (NMPC) già implementato sul sistema. Per raggiungere questo scopo, è stato sviluppato un simulatore dinamico basato sull'analisi del funzionamento di Roboat e facendo riferimento alla letteratura nel settore. Il simulatore realizzato include anche un modello dell'incertezza e dei principali disturbi che agiscono sul sistema.

Successivamente, usando il simulatore, è stato creato un agente con l'algoritmo Proximal Policy Optimization. L'agente è stato quindi convalidato e confrontato con l'attuale strategia di controllo sia in simulazione che in esperimenti reali.

Parole chiave: veicoli autonomi, inseguimento di traiettorie, corsi d'acqua urbani, Reinforcement Learning, Model Predictive Control.

Contents

Introduction

Water transport presents a highly efficient method of movement that has been used by humans for thousands of years. Today it accounts for around 80% of international trade. This setting, together with others, such as environmental monitoring, and search and rescue, seeks reliable control systems. Recent advancements in control theory strive to achieve that and reduce the difficulty of the human's task in vessel control significantly. The final objective in this process is making the system autonomous.

Autonomous vehicles are systems able to operate on their own in their designated environment by sensing it through their sensors and acting on it with their actuators. The last couple of decades saw their dramatic rise in forms vehicles in road, aerial, maritime, and space settings.
Making a system autonomous presents a specific set of challenges that can roughly be divided into these steps: localization, perception, planning, and control. Localization is the process of determining ones state with respect to the environment. Perception entails developing an understanding of ones environment. Planning is the process of determining a sequence of valid future states to reach the designated goal state. The output of a planner can be a path or a trajectory, depending on whether the sequence of states includes timing. And lastly, control translates the high-level plan into actions by determining the low-level actuators behavior to track it.

The focus of this study is trajectory tracking for autonomous surface vehicles (ASVs). Up until recently, most research in the field concerned large ships and tankers travelling in open waters. This setting is characterized by massive, heavy vessels, slow dynamics from low frequency waves, wind, and current, and a propeller and rudder actuation, as explained in detail in [10].
However, recent technological advancements have given rise to another sub-field in the area. The novel maritime solutions using small autonomous vessels, such as the Roboat platform [37]. When moving in urban waterways, these smaller vessels encounter a different set of difficulties and objectives. Now, situations like docking, maneuvering in tight spaces, or performing evasive maneuvers upon encountering other boats, shift the focus from energy consumption to high precision movement.

To achieve that, several significant challenges have to be overcome. To begin with, while errors in control for large ships come mainly from low frequency waves, wind, and current, ASVs in urban waterways are, on the other hand, affected mostly by varying payload, wind, current, and high frequency waves. Firstly, due to the ASV's own low weight, varying payload changes the underwater hull shape, which in turn affects the thruster dynamics. Secondly, the impact of wind on the system is more pronounced because a significant portion of the vessel is above the waterline. Therefore, depending on the wind parameters like the angle of attack, the wind acts as a force. Thirdly, the current acts as a constant drift translating the system. And lastly, depending on the setting, high frequency waves can act as weak pulses of force on the boat. Together, these effects render control of an ASV in urban waterways a complex and specific problem.

State of the art

Similar vessel control problems, like dynamic positioning, way-point tracking, and autopilots for course-keeping and trajectory tracking have been tackled in the past. Methods used in solving those problems range from classical control, such as PID, LQG, and NMPC controllers, Kalman filtering, underactuated control, feedback linearization, and many more. An overview of the results of their implementation is provided in [12]. Let us consider a few of them here.
To begin with, the use of PID controllers and Kalman filters is widespread with one of the most basic problems in vessel control, dynamic positioning. It is defined as the act of to keeping a floating vessel on a specified position by proper action of the propulsion system of the vessel. One example of such an approach is [3], where the Kalman filter and optimal control are used to position a ship. Furthermore, when considering tasks like way-point tracking or autopilots for course-keeping and trajectory tracking, the use of techniques such as LQG, NMPC and feedback linearization has been proposed. Using an LQG for high-precision track control of ships was attempted successfully in [17]. NMPC, another well established approach in literature, is chosen for many autopilot systems, including the aforementioned Roboat platform. Detailed implementation of the controller based on the static estimated model of the small scale of the system can be found in [36]. Feedback linearization has also been used extensively. For example in [11], it is applied to automatic ship steering with a flexible design to allow easy optimization.

Although still in its early stages, research on small ASVs exists, for example [36] takes on the problem of station keeping when exposed to current and wind. As for the urban waterways setting, apart from the Roboat project [15, 37], there has been little to no

research.
With the increasing popularity of learning methods in control, some research has also been done in vessel control. In [40], deep reinforcement learning with an actor critic architecture was used to train a model to track a predefined trajectory for an autonomous underwater vehicle (AUV). Moreover, in [7] an on-line selective reinforcement learning approach combined with Gaussian Process (GP) regression for trajectory tracking is presented. In [21] deep reinforcement learning for trajectory tracking is used to control an unmanned aerial vehicle. Although relatively recent, published around 5 to 10 years ago, they are still ancient given the volatility with which this field is advancing.
Traditional control approaches, such as the nonlinear model predictive control (NMPC), can have trouble with precise trajectory tracking when encountering mentioned difficulties. To overcome the difficulties, reliable measurements of model parameters is needed. However, filtering parameters with uncertainty is infeasible without data from proper sensor. This is the case with the current control system for Roboat. Given its limited knowledge of the dynamics of the system, the planned actuation does not yield expected motion in the real world. This can lead to dangerous situations in some of the envisioned use cases, like the situations mentioned before.

Problem statement

This thesis project presents an attempt to solve precise and robust trajectory tracking for an autonomous surface vessel (ASV), specifically the Roboat platform, to sail in urban waterways by using a novel architecture based on reinforcement learning.

For this purpose, the entire system, with the aforementioned disturbances and uncertainties has to be modelled and considered in a simulator acting as its digital twin. This can be done based on information about their behavior according to [10, 15]. However, these disturbances cannot be measured directly in real time due to a lack of sensors and approximating them reliably is unfeasible. Furthermore, modelling the system itself is difficult because of varying thruster dynamics and unknown payload size and distribution. This is because, the force which the thrusters output, depends on the vessel's wake described by the wake fraction number. This parameter changes significantly, as can be seen in [10, 26]. The fact that the payload cannot be measured directly means we do not have exact real time information about the systems mass.
The presented challenges mean that architectures presented in the past, like [1, 12], would have limited success in precise trajectory tracking. They require a precise vessel model, including the thrusters, and the spectrum of both disturbances and uncertainties, which

can only be roughly approximated in real time [10, 12].

Approach and contributions

Given the above-mentioned problems, we hypothesize that a learning-based controller can be trained to perform precise trajectory tracking for the setting. Inspiration for such an approach stems from recent breakthroughs in RL for robot locomotion by the Robotic Systems Lab, presented in [20]. The paper presents a novel architecture and learning approach to tackle the problem of robust locomotion of a quadrupedal robot. The main problems it faces are environments with highly irregular profiles, deformable terrain, slippery surfaces, and overground obstructions. These cannot be precisely modelled, nor measured with available sensors. Therefore, their approach uses only proprioceptive sensors as input, such as the inertial measurement unit and joint encoders. The success of the approach, as well as the parallels to the Roboat setting (unmeasurable environment factors: leg slip factor and wake fraction number of a boat's hull, current strategies overly complex yet not robust or general enough), give hope and motivate the work on a similar controller.

To develop it, two main challenges will be considered. Firstly, a more advanced simulator will be built based on the research done, including the approximation of aforementioned factors (payload, wind, current, thrusters, and waves). Specifically for the thrusters, their behavior will be approximated from odometry and localization data recorded from the Roboat platform. Additionally, some of their parameters which cannot be calculated directly (such as the wake fraction number), will be randomized in the simulator.

Secondly, a controller will be trained in the developed simulator based on an RL paradigm to track a given trajectory and to adjust to the disturbances and uncertainties within a reasonable range. The learning process can then, if needed, be improved through an adaptive curriculum approach. Inspired by the automatic curriculum learning for RL agents called Paired Open-Ended Trailblazer [35], it consists of gradually increasing the level of difficulty of the environment that the agent faces based on the agent's performance. The more successful the agent is, the harder the challenges it faces get. This game like process makes learning more natural and robust.

The document is divided as follows:

- Chapter 1 contains the theoretical background of vessel control containing. Firstly, a general description of the kinematics and dynamics of a vessel is given, followed by the thruster dynamics and disturbance and uncertainty effects on the system. Secondly, specific details for the Roboat platform, for which the thesis is aimed for, are also presented.
- Chapter 2 focuses on deep reinforcement learning methods and specific algorithms used here. An introduction into the reinforcement learning field is given, with the most important algorithms presented. Following this, methodologies used in literature to solve the same or similar problems are presented as inspiration for this study.
- Chapter 3 presents the implementation of a simulator of the Roboat platform from the ground up. It is capable of simulating all desired effects, such as the disturbances, uncertainties, and thruster dynamics. Furthermore, it can also be used as an environment for training reinforcement learning agents. Here, the specific implementation details used to create the framework for learning trajectory tracking for the Roboat platform are presented.
  To prove its worth, it is compared to the current simulator used by Roboat in a number of metrics, followed by a demonstration of the disturbance and uncertainty modelling.
- In chapter 4, the performed experiments and obtained results are presented. Firstly, the process of the comparison in the simulator is explained in detail, followed by the presentation of the comparison in multiple metrics, namely the tracking precision and power consumption.
  After this, the procedure and the results of the trials on the real life platform are presented. Moreover, the problems encountered while performing the tests are explained.
- In the end, the conclusion of the work is presented, together with future possible improvements.

1 | ASV kinematics and dynamics

To begin to understand how to control a vessel, as with all control problems, one has to accurately model the system. Comprehensive work on the matter has been done by one of the most prolific and important scientists in the field, Thor I. Fossen, whose book, "Guidance and Control of Ocean Vehicles" [10] covers the process in length. Inspired by it, this chapter provides an introduction into the basics of kinematics and dynamics of the system.
The first part of the chapter presents a physical description of the system at hand, going over the rigid body assumptions, introducing also the hydrodynamic forces and moments. The system is finally described by its equations of motion. The second part of the chapter introduces the thruster dynamics of a vessel, through which the system is actuated, focusing on the ASV setting. The third part of the chapter revolves around the disturbances and uncertainties that affect such a system the most in real world, like varying payload, current, wind and waves. All of this is followed by mathematical equations describing the matter. In the end, the system is presented in its entirety, putting it all together.

1.1. General

The motion of a vessel in a fluid can be described by a rigid body moving in 6 independent degrees of freedom (DOF), necessary to determine both its position and orientation. In marine literature these 6 DOF are called surge, sway, and heave for position, and roll, pitch, and yaw for orientation. The notation that will be used in this study is presented in Table 1.1.

DOF		force and moment	linear and angular velocity	position and angle
1	**x direction - surge**	X	u	x
2	**y direction - sway**	Y	v	y
3	**z direction - heave**	Z	w	z
4	**rotation about x - roll**	K	p	ϕ
5	**rotation about y - pitch**	M	q	θ
6	**rotation around z - yaw**	N	r	ψ

Table 1.1: 6 DOF notation

To describe the movement of a rigid body in space, two different coordinate systems are defined. One connected to an inertial reference frame, and the other connected to the rigid body, relative to the first one. For this purpose, choosing a point on Earth's surface as the inertial reference frame is acceptable as the effects of Earth's motion on the system are negligible. Furthermore, the vessel's center of gravity (COG) is a convenient point to place the rigid body's point of origin, as it is more or less static and usually presents a point of symmetry. However, certain situations can render this assumption invalid. For example, when loaded unevenly the COG of a small ASV can move and the control will not have its expected effect. These types of situations will be explored later in the study. Logically, position and orientation $\boldsymbol{\eta}$ should be relative to the inertial reference frame, whereas linear and angular velocities $\boldsymbol{\nu}$ should be relative to the vessels's reference frame. The notation and relation between the two is defined as follows:

$$\boldsymbol{\eta} = [x, y, z, \phi, \theta, \psi] \tag{1.1}$$

$$\boldsymbol{\nu} = [u, v, w, p, q, r] \tag{1.2}$$

For ASVs in urban waterways, however, heave, roll, and pitch are close to zero due to low accelerations and speeds. Furthermore, the disturbances and uncertainties do not induce enough motion in those dimensions to significantly influence the vessel's behavior. Therefore, as is the case in literature, from now on, we will consider a 3 DOF system in x, y, ψ with their velocities u, v, and r. They are related through the following equation:

$$\dot{\boldsymbol{\eta}} = \boldsymbol{R}(\psi) \cdot \boldsymbol{\nu} \tag{1.3}$$

where $\boldsymbol{R}(\psi)$ is the transformation matrix from the rigid body's to the inertial frame.

The rigid body dynamics of the vessel can be derived using *Lagrangian* and *Newtonian*

mechanics. The system is described with the following nonlinear dynamic equations:

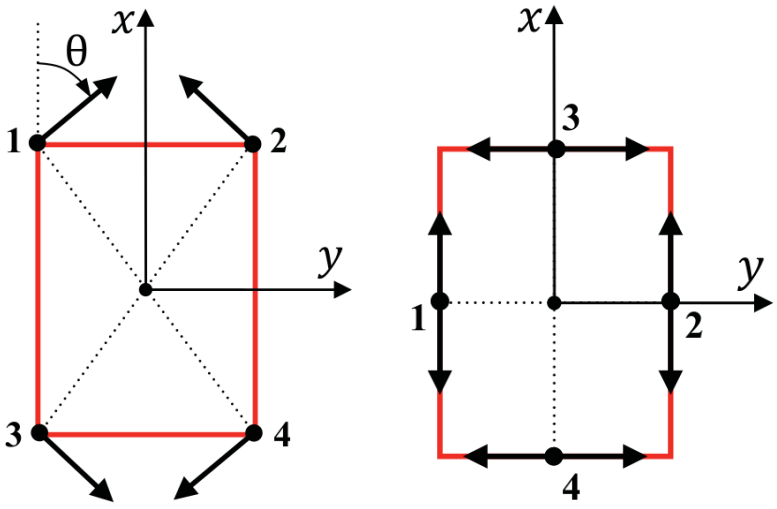
$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{\eta}) = \boldsymbol{\tau} \tag{1.4}$$

where:
$\boldsymbol{M}$ = inertia matrix,
$\boldsymbol{C}(\boldsymbol{\nu})$ = Coriolis and centripetal term matrix,
$\boldsymbol{D}(\boldsymbol{\nu})$ = damping matrix,
$\boldsymbol{g}(\boldsymbol{\eta})$ = gravitational force and moment vector,
$\boldsymbol{\tau}$ = vector of torques acting on the system such as the control inputs.

To complete the modelling of a vessel special care has to be given to hydrodynamic forces and moments acting on the system. They describe the effects of fluid particle velocity and acceleration on the immersed system and can be generally divided into two groups: radiation induced forces and Froude-Kirchoff and diffraction forces. In the scope of this study only the radiation induced forces will be considered as the Froude-Kirchoff and diffraction forces concern a body restrained from oscillating which is not the case here. The radiation induced forces are described as the sum of *added mass*, *radiation induced potential damping*, and *restoring forces*.
Firstly, added mass represents the inertia added to the system due to the volume of surrounding fluid deflected as a body moves through it. In reality, the whole fluid will be accelerated to a certain degree, however approximating the effect with a matrix of constants $\boldsymbol{M_A}$ is precise enough according to literature [10].
Secondly, hydrodynamic damping is mainly caused by: radiation-induced potential damping, linear skin friction, wave drift damping, and vortex shedding. These effects can be approximated with a damping matrices $\boldsymbol{D_L}$ an $\boldsymbol{D_Q}$ with linear and quadratic terms respectively.
Lastly, the restoring forces are equivalent to the spring forces in a mass-damper-spring system. However, they will not be considered here as their effect is negligible.

1.2. Thruster dynamics

By and large, marine vessels are actuated through a combination of main thrusters, bow thrusters, and a rudder system. Larger ships usually utilize one or two main thrusters with propellers to propel the system and a rudder system to steer the ship. However, smaller vessels, underwater systems, and ASVs are much more diverse in this regard. One of the main reasons for this is different objectives. Whereas reduction of energy consumption steers the design of large ships, smaller vessel design usually prioritizes maneuverability and precise motion. In case of both large and small vessels the addition of bow thrusters makes the system more maneuverable by actuating it perpendicularly to the ship's principle axis. Typically, ASVs and underwater systems use an "X" shaped configuration to achieve holonomic motion. However, the system studied in this work implements a cross-shaped actuator configuration. The reason being the significantly higher efficiency index, the value is 0.5 for the "X" shaped and 1.0 for the cross-shaped configuration as explained in chapter two of [36]. When considering a 3 DOF model of the system, this configuration renders the system overactuated. Comparison between the two configurations is shown in Figure 1.1.



(a) "X" thruster configuration (b) Cross-shaped thruster configuration

Figure 1.1: Comparison between thruster configurations that allow holonomic motion of the vessel in the horizontal plane.

Apart from some specific experimental examples, self propelled vessels generate thrust through propeller rotation. The physics of propulsion systems is complicated enough to

have a field of research dedicated to it, however, for the needs of this study, the following simplification will suffice.

The thrust with which the system is actuated can be described as a nonlinear function connecting the vehicle's velocity vector $\boldsymbol{\nu}$ and the number of propeller rotations per minute (the control variable $\mathbf{u}$), to the thrust output vector $\boldsymbol{\tau}$. By rotating, the propeller pushes the fluid away from it, creating a pressure difference in front versus behind it. As the fluid mass is accelerated in one direction, thrust is generated and the vessel travels in the opposite direction. The generated thrust can be approximated with equations (1.5) and (1.6), as described in the fourth chapter of [10]:

$$\boldsymbol{\tau} = \rho D^4 K_T(J_0)|\boldsymbol{u}|\boldsymbol{u} \tag{1.5}$$

$$J_0 = \frac{(1-\omega)\boldsymbol{\nu}}{\boldsymbol{u}D} \tag{1.6}$$

where:
ρ = the density of the fluid,
D = the propeller diameter,
K_T = the thrust coefficient,
J_0 = the advance number, defined with the equation (1.6),
ω = the wake fraction number.

Out of these parameters, the fluid density ρ and propeller diameter D are easily obtained and generally considered static. The thrust coefficient K_T is one of the most important characteristics of a propeller, specific for every different propeller. Usually, the manufacturer of a given propeller provides it in the form of a graph, such as Figure 1.2. If not provided by the manufacturer, it has to be approximated. And lastly, the wake fraction number ω describes the relationship between the actual vessel's speed and the speed of fluid coming at the propeller. It depends on the shape of the immersed portion of the vessel's hull and the propeller positioning, but usually has a value between 0.1-0.4 [10]. In effect, it is a measure of how effective the propeller is. For smaller vessels, such as is the case in this study, it is important to take the uncertainty of the wake fraction value into account, as it can change the effect of thrusters significantly.

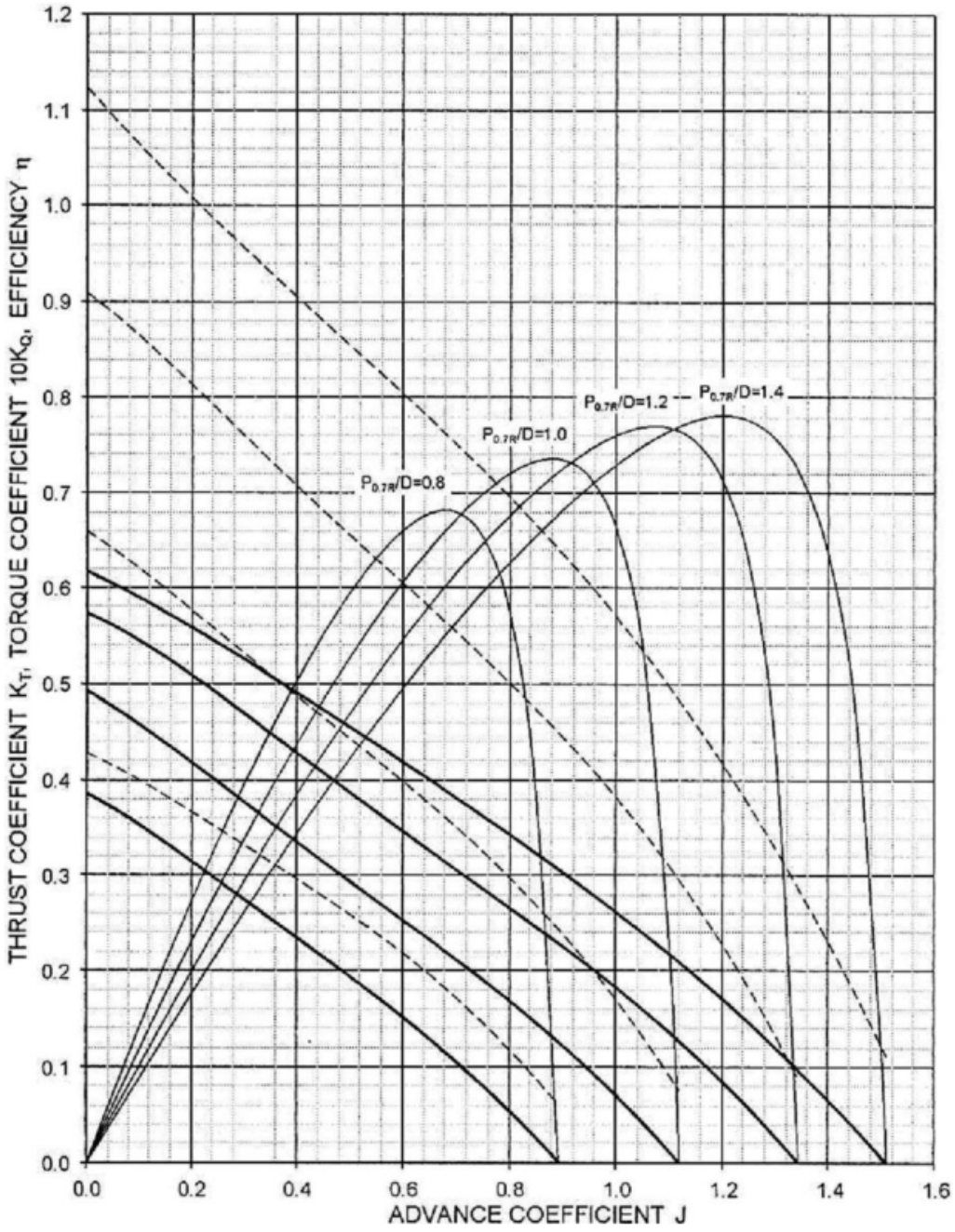


Figure 1.2: Example of a thrust coefficient graph for a variable pitch propeller from [26], provided by the propeller's manufacturer. The graph presents the values of the thrust coefficient (K_T), the torque coefficient (K_Q), and efficiency (η) in relation to the advance coefficient J, a value defined earlier in Figure 1.6. The full quasi linear downward lines present the thrust coefficient number for different propeller pitch settings.

1.3. Disturbances and uncertainties

As the vessel travels through a fluid, various disturbances and uncertainties act on it. On one hand, the disturbances such as wind, current, and waves disturb the action behavior of the system by generating forces and torques on it. On the other hand, uncertainties in the system parameters, that arise from either not being able to measure them precisely or because they change through time, make control of the system more difficult.

1.3.1. Varying payload

Firstly, the most pronounced uncertainty will be considered: varying payload. In case the vessel is used for transportation, as it is in this study, and the mass of the payload is significant with regards to the vessel's own mass, there will be visible effects on the system's behavior, making the entire system less prone to acceleration. Not only do the values of the mass matrix change, but also the thruster dynamics and the damping matrix, due to the changes in the shape of the immersed portion of the vessel's hull. This change will, therefore, also be reflected on the wake fraction number ω. It should be noted that both the magnitude and the placement of the payload can change the dynamical properties, as different placement can change the hull shape hitting the incoming water causing more or less drag.

1.3.2. Wind

Secondly, as a significant portion of any surface vessel is above water, the wind hitting it applies forces and torques $\boldsymbol{\tau_w}$ on the system. The effects of wind can be described with a mean direction and a slowly varying speed. Modelling wind has been explored in literature extensively and this specific study takes inspiration from [32]. They can be condensed in a representation such as the equation (1.7):

$$\boldsymbol{\tau_w} = \frac{1}{2}\rho_a V_{rw}^2 \begin{bmatrix} -c_x A_{FW} \cos(\gamma_{rw}) \\ c_y A_{LW} \sin(\gamma_{rw}) \\ c_z A_{LW} L_{OA} \sin(2\gamma_{rw}) \end{bmatrix} \tag{1.7}$$

where:
γ_{rw} = apparent wind angle,
ρ_a = the density of air,
V_{rw} = apparent wind speed,
A_{FW} = frontal projected windage area,

A_{LW} = lateral projected windage area,
L_{OA} = vessel's overall length,
ω = wind's apparent angle of attack.

1.3.3. Current

Thirdly, the current acts as a translation of the vessel's moving frame with a certain velocity. Therefore, the effect current has on the system can be easily modelled with two more parameters: the current speed V_c and heading β. This representation is well established in literature, for example in chapter three of [10], and higher complexity would only lead to increased computation time. The drifting effect is described by equations (1.8) and (1.9).

$$u_c = V_c \cos(\beta - \psi) \tag{1.8}$$

$$v_c = V_c \cos(\beta - \psi) \tag{1.9}$$

where u_c and v_c present the current's velocity in vessel's surge and sway directions respectively.

1.3.4. Waves

Lastly, the effect of wind generated waves on the vessel will be mentioned. Wind generated waves are usually divided into two groups: low frequency and high frequency waves. As the wind picks up, the drag on the surface creates the high frequency waves with low amplitude. If the wind blows for long enough, low frequency with high amplitude develop.

The effect of waves is significant both on large and small vessel's when the system operates in open waters. However, when dealing with ASVs in urban waterways, mentioned low frequency waves are not encountered. Moreover, the effect of high frequency waves on the system is observed not high enough to be considered.

1.4. The complete model

Finally, the equations of motion for the entire system can be written:

$$(\boldsymbol{M_{RB}} + \boldsymbol{M_A})\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D_L}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{\nu}\boldsymbol{D_Q}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{\eta}) = \boldsymbol{\tau} + \boldsymbol{\tau_E} \tag{1.10}$$

$$\dot{\boldsymbol{\eta}} = \boldsymbol{R}(\psi) \cdot \boldsymbol{\nu} \tag{1.11}$$

where the new terms are:
$\boldsymbol{M_{RB}} + \boldsymbol{M_A}$ = rigid body and added mass terms,
$\boldsymbol{D_L}(\boldsymbol{\nu})$ = linear damping matrix,
$\boldsymbol{D_Q}(\boldsymbol{\nu})$ = quadratic damping matrix,
$\boldsymbol{\tau_E}$ = vector of environmental torques.

Two things should be noted here. Firstly, the velocity vector $\boldsymbol{\nu}$ now presents the relative speed of the vessel with regards to the current. And secondly, the mass matrix is not static, as it changes with the varying payload. Having defined the model of the system, focus can be shifted to strategies with which to control it.

2 | Reinforcement Learning

Computer science often fascinates by its ability to yield quantum leaps in progress by taking inspiration from processes in the real world. One of the most glaring examples of this is translating the process of learning from nature into a set of powerful optimization algorithms that make up the subfield of machine learning called reinforcement learning. This chapter provides an introduction into it and an overview of its current state of art.

The first part of this chapter serves as a general introduction into the idea behind reinforcement learning, with its brief history and description. The second part of the chapter goes over the most popular reinforcement learning approaches, focusing control problems. The third part of the chapter covers available methodologies to improve the learning process and with it the resulting actor behavior.

2.1. General

Reinforcement learning has its roots in behavioral psychology of the early 20th century, exploring the use of rewards and penalties to encourage a desired behavior. However, it would not be until the second half of the century that great minds like Alan Turing and Richard Bellman lay down the foundations of its theory. The idea behind it is simple: a decision-making agent learns from a sequence of reward signals from the environment that provide some indication of the quality of its behavior. The goal is to optimize the sum of future rewards [31]. Figure 2.1 provides a general reinforcement learning environment architecture.

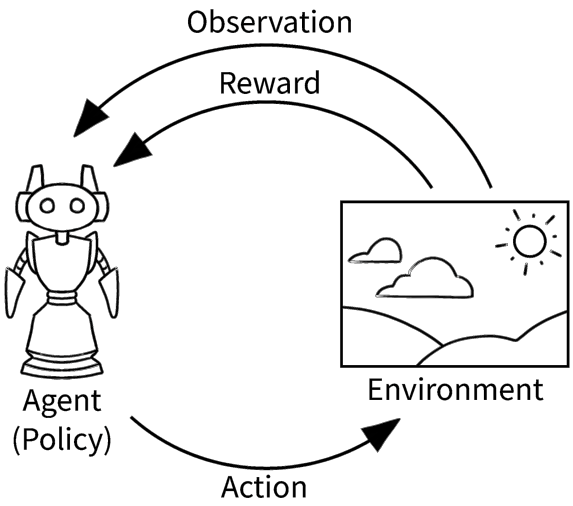


Figure 2.1: Basic high level abstraction of the architecture of the agent-environment loop. Image from OpenAI.

Basic reinforcement learning problems can be described through the notion of discrete-time stochastic control processes called Markov Decision Processes (MDPs). It is defined by a (S, A, P_a, R_a) tuple where:

- S is the set of possible states called the state space,
- A is the set of possible actions called the action space,
- P_a is the probability of action a taken in state s leading to state s'
- R_a is the immediate reward for taking action a in state s.

At each timestep, the process is in a state s. From that state, the agent chooses an action a from a set of actions available in state s, called $\Gamma(s)$. The process then changes to a new state s' based on the state transition function $P_a(s, s')$ and outputs a corresponding reward $R_a(s, s')$.
The agent's behavior, or more specifically the choice of action in any given state, is defined through the policy function $\pi(s, a)$. A policy that maximizes the reward function in the infinite horizon is called the optimal policy $\pi^*(s, a)$. The goal of reinforcement learning algorithms is to obtain this optimal policy for a given agent environment pair.

2.2. Algorithms

Having defined the problem, some of the most important algorithms for reinforcement learning in control can be explored. Firstly, Q-learning, deep Q-learning, and its variants will be introduced, as one of the first model-free algorithms. The problem will then be generalized to the continuous setting with policy gradient methods in which algorithms like the proximal policy optimization [33] have proven to be successful.

2.2.1. Q-learning, deep Q-learning, and variants

Q-learning, introduced in late 20th century works as follows. Facing an unknown MDP with discrete sets of possible states S and possible actions A, a Q-learning agent learns an action-utility mapping function $Q(s, a)$, giving the expected utility of taking a given action a in a given state s [31]. The values of the $Q(s, a)$ function are calculated according to the update rule (2.1):

$$Q_{new}(s_t, a_t) = (1 - \alpha)Q_{old}(s_t, a_t) + \alpha(r_t + \gamma max_{a \in \Gamma(s_t)} Q(s_{t+1}, a)) \tag{2.1}$$

where:
s_t, s_{t+1} = states at time t and t+1,
a_t = action taken at time t,
α = the learning rate of the algorithm,
r_t = the immediate reward for taking action a_t from state s_t,
γ = the discount factor of the algorithm, defining the future reach of the cumulative reward.

Moreover, in order not to get stuck in any local optima the agent should sometimes deviate from this update rule and take a random action. Defining the frequency of this happening is called "The Exploration Exploitation Dilemma". The most simple and efficient way to implement this is the "epsilon greedy" approach (2.2):

$$a_t = \begin{cases} \text{action with } max_{a \in \Gamma(s_t)} Q(s_{t+1}, a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{2.2}$$

By following the update rule (2.1) and exploring the problem for example with (2.2), as the agent operates in the MDP, the values of $Q(s, a)$ are proven to eventually lead to the optimal policy $\pi^*(s, a)$. The model-free nature of the algorithm comes from the fact that the values of P_a are not used in the calculations, but are inherently approximated.

Q-learning proved to be an algorithm capable for solving many simple problems, such as the game of Tic-Tac-Toe. However, since it can be applied only to discrete state and action spaces, it suffers greatly from the curse of dimensionality. That is, discretization of the state and action spaces for more complex problems in practice often leads to inefficient learning.

To solve the curse of dimensionality, instead of a function $Q(s, a)$ mapping predicted values for all possible state-action combinations, deep Q-learning approach utilizes a deep neural network. The input into the network is the current state of the environment and the predicted value for taking every possible action is generated as the output.
As for training the network, learning and updating is performed on the state-action pairs as they arrive from the running simulation, with the loss function defined as the mean squared error between the predicted and target Q-value. Unfortunately, since the target values are calculated from the same changing neural network, the learning process is often unstable. Furthermore, this training approach leads to considerable correlation between neighbour states as incoming training batches contain similar data.
Two methods are used to combat this. Firstly, a second neural network is used to approximate the target Q-value. Its parameters are kept frozen for n iterations and updated afterwards. The update consists of copying parameters from the prediction network, often called the actor network, to the target network, often called the critic network. Second method to solve correlation, called experience replay, includes storing past experience of the agent in a buffer as the training set. This buffer is then randomly sampled to create a training batch, reducing correlation greatly.

2.2.2. Policy Gradient Methods

A significant breakthrough in deep reinforcement learning performance came with the introduction of policy gradient methods. These methods, instead of trying to approximate the value of state-action pairs $Q(s, a)$ and extracting a policy $\pi(s, a)$ from that, aim to directly optimize the agent's policy.

The policy is presented as a probability distribution, parameterized with a set of parameters θ, and optimized through gradient ascent on the expected reward obtained by following that $\pi_\theta(s, a)$. The parameter vector θ can take various forms, such as a neural network or a genome in genetic algorithms, as the algorithm does not change with the choice, only the update method. Today, using a neural network is the most popular approach as it proved most powerful in solving complicated tasks, for example humanoid

actuation.

To perform gradient ascent, the most commonly used gradient estimator $\hat{g}$ is 2.3 obtained by differentiating the loss L^{PG} 2.4, and following the 2.5 update rule:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_t log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{2.3}$$

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{2.4}$$

$$\theta_{new} = \theta_{old} + \alpha \hat{g} \tag{2.5}$$

where:
$log\pi_\theta(a_t|s_t)$ = the log probability of the output of the policy π_θ in state s_t,
$\hat{A}_t$ = the estimator of the relative value taking the selected action at time t with regards to the old policy, called the advantage function,
α = the learning rate.

These functions describe the desired learning behavior: if the expected advantage of following the new policy with regard to the old policy is positive, the gradient will steer the policy in that direction. If there is no expected advantage, this algorithm steers the policy away. This approach is, however, very sensitive to the size of policy updates. Too large of an update and the learning process is too noisy, too small and the progress is extremely slow. This problem is caused by the advantage function $\hat{A}_t$ which, being a neural network itself, is a noisy estimator. Therefore, a wrong estimate with a large gradient could cause a significant setback.

2.2.3. Proximal Policy Optimization

Presented in 2017 in [33], the proximal policy optimization (PPO) is a policy gradient method for reinforcement learning. Its predecessors, the trust region policy optimization (TRPO) and actor-critic experience replay (ACER), although data efficient and with reliable performance, proved to be too complicated and rigid. PPO, on the other hand, is much simpler to implement, more general, and has better sample complexity.
The algorithm uses these two approaches: trust region methods and a clipped surrogate objective. Firstly, trust region methods optimize the policy by making small improvements on the policy chosen from a small region around the current policy called the trust region. It uses a surrogate objective presented in 2.6 constrained with the size of the policy

update.

$$\max_{\theta} \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t, s_t)}{\pi_{\theta_{old}}(a_t, s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right] \tag{2.6}$$

where:
$\pi_\theta(a_t, s_t), \pi_{\theta_{old}}(a_t, s_t)$ = the new and old policy functions,
$\hat{A}_t$ = the estimator of the relative value of selected action at time t, called the advantage function.

The new objective is to maximize the probability ratio between the policies multiplied with the advantage function. This leads to small and steady policy updates, improving the learning process.

Secondly, the clipped surrogate objective function is used to constrain the policy update by weighing the advantage of the update to the size of the update. This strikes a good balance between too large and too small policy updates using only first-order optimization, while retaining the positive aspects of TRPO and ACER. The proposed clipped surrogate objective function is presented in 2.7:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{2.7}$$

where:
ϵ = a hyperparameter, for example 0.2,
$clip(r_t(\theta), 1-\epsilon, 1+\epsilon)$ = modifies the surrogate objective by clipping the probability ratio,
$r_t(\theta)$ = immediate reward at time t.

The clipped surrogate objective function presents the main contribution of PPO because it disincentivizes large updates and makes the learning process more robust. Furthermore, it makes the implementation simpler while retaining the stability and reliability of trust-region methods. At the time of writing, PPO outperforms other similar algorithms and is used to learn a controller to track a trajectory in this study.

2.3. Methodologies

Having gone over the most important reinforcement learning algorithms, we can shift focus to the setting in which it is being used. The design of the learning environment depends in large part on the problem at hand. Recently, the mentioned algorithms have been applied to a number of settings. This includes vintage computer games in [25], traditional control problems like the inverted pendulum in [28], and even more complex robotics tasks such as humanoid robot actuation in [16]. Through the years, various methodologies formed to approach each one of them. This section gives more detail on how to set up an environment for learning ASV trajectory tracking.

2.3.1. Trajectory tracking

As mentioned in the introduction, a trajectory tracking algorithm implies taking a timed sequence of future states of the robot at hand as input, and calculating the necessary actuation to achieve the desired behavior of tracking the trajectory. To learn a controller to perform such a task, an environment needs to meet the following requirements.
Firstly, the agent needs to observe its current state and the desired trajectory. In the setting considered in this study, the current state of the ASV can be condensed in a 6-value vector of $q = [x, y, \psi, u, v, r]$. Furthermore, the future trajectory can be represented with an array of vectors of the same type, either containing also the time t or without it, if the time between neighbor states and the initial time is known and constant.

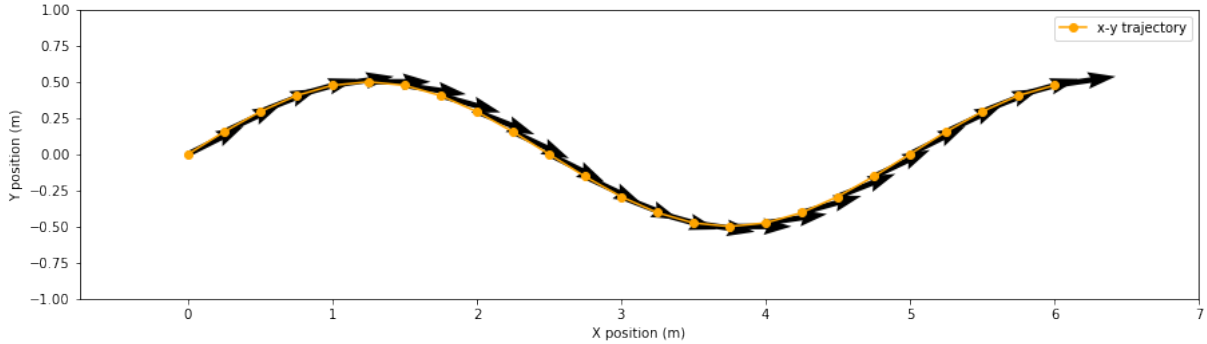


Figure 2.2: Visualization of a sinusoidal trajectory with timestep of 1s, each state defined with the x, y, and ψ values.

Secondly, the agent needs to be able to actuate the system. For the Roboat platform, with its four thrusters in a plus configuration, the actuation values can be condensed in a 4-value vector of $u = [f_1, f_2, f_3, f_4]$ presenting the forces applied by the thrusters.
And lastly, the agent needs to evaluate the consequences of its actions. This is tuned

through the design of the reward function. Considering the fact that its design can make or break the learning process, it is regarded as one of the main aspects of the environment. The reward function is in a lot of ways similar to any optimization function, for example the MPC optimization function over a finite horizon. However, whereas the MPC optimization function needs to closely define desired behavior, the reward function does not need to be as descriptive. Since the learning process goes on for thousands of episodes, depending on the complexity of the environment, the agent is usually able to extract good policies from simple reward functions. For example, in one of the staple reinforcement learning problems, the inverted pendulum (also known as the cart pole balancing), it is simply defined as $+1$ for every step in which the pole is upright ($+/-12$ degrees offset to 0 degrees).

In general, trajectory tracking for vessels is not an overly popular topic in the reinforcement learning community. Most work considering trajectory tracking takes on aerial or underwater vehicles. In [21] trajectory tracking for an unmanned aerial vehicles was proved to outperform a traditional PID controller. The reward function used during the training of a deep neural network controller was defined as 2.8:

$$R(t) = \frac{1}{2\pi} \exp\left(-\frac{(\Delta P_t + \Delta V_t + \Delta R_t)^2}{2}\right) \tag{2.8}$$

where:
ΔP_t = absolute value of the difference between desired and current position vector,
ΔV_t = absolute value of the difference between desired and current velocity vector,
ΔR_t = absolute value of the difference between desired and current rotation vector.

In [40] the same was attempted for of autonomous underwater vehicle. The reward function in this study penalizes squared state error and goes a step further including an action penalty 2.9:

$$r(s_t, a_t) = \left[-I\left(\hat{s_t} - s_t\right)^2 - \Lambda a_t^2\right] \tag{2.9}$$

where:
$\hat{s_t}, s_t$ = the desired and current state at time t,
a_t = the action vector at time t,
I, Λ = positive definite functions.

However, a small caveat to this study is that a new controller is trained for each type of trajectory. Therefore, the controller does not generalize for any trajectory but overfits to the one it trained on.

2.3.2. Vessel control with reinforcement learning

As for surface vessels, this section lays out previous work on using reinforcement learning for their actuation in various settings and goes over the suggested reward functions for inspiration.

The problems of course tracking and path following are very popular topics in this field. The first problem consists of steering the vessel to maintain the desired heading, while path following includes also the actuation of the thrusters to maintain the desired surge speed.
One of the first attempts of neural network based reinforcement learning control for path following was presented in [41], to actuate an underactuated ship. A simple Actor Critic reinforcement learning architecture was used to control the rudder angle. The reward function included an elaborate combination of penalizing tracking error and control efforts in a weighted exponential term.

In [39] the problem of path following for an unmanned surface vehicle (USV) is considered. The only difference in their definition of the task versus trajectory tracking is the lack of timing with way-points in the path. However, the reward function remains similar in the two cases. The definition of the reward function in this study is quite elaborate, made up of three summed parts 2.10, 2.11, and 2.12:

$$r_{\bar{\chi}} = \begin{cases} \exp -k_1 * |\bar{\chi}| & \text{if } i < 90deg \\ \exp -k_1 * (\bar{\chi} - 180) & \text{if } i \geq 90deg \\ \exp -k_1 * (\bar{\chi} + 180) & \text{if } i \leq -90deg \end{cases} \tag{2.10}$$

$$r_{e_y} = \exp -k_2 * |e_y| \tag{2.11}$$

$$r_{\sigma_\delta} = \exp -k_3 * \sigma_\delta \tag{2.12}$$

where:
$\bar{\chi}$ = the difference between the actual course angle and the desired course angle,
e_y = the cross track error,
σ_δ = the standard deviation of the recent 20 steering input commands.

These three values encapsulate three important ideas: position error, course error, and steady steering. However, the more complex the reward function, the harder it is to tune its hyperparameters to achieve satisfactory learning. As a general rule of thumb, one should keep the reward function as simple as possible while embedding the desired

behavior in it.

In [14] and [13] the path following problem for USVs was approached from a slightly different angle, using a combination of adaptive control and deep reinforcement learning algorithms. The idea is to develop a low level controller for surge speed and yaw angle through deep reinforcement learning and then use it with adaptive dynamic programming to perform the task. In both studies the Deep Deterministic Policy Gradient (DDPG) algorithm was used.
When it comes to reward function shaping, they utilize very similar architecture to [39]. The reward function in [14] consists of parts to reward small heading error, tracking the reference surge speed, and penalizing large standard deviation oscillations of the action values. In [13], instead of the surge speed reward term, a term penalizing cross track error is implemented. Moreover, in [14] the trained controller was deployed on a real USV. It is interesting that both studies encountered choppy and erratic control as a results of training in spite of the reward architecture. When deployed, this lead to reduced performance and undesirable behaviour.

In [23] and [22] a deep reinforcement learning solution was developed for path following of mariner vessels, a class of large ships, when faced with a current disturbance. In these works, the deep reinforcement learning is in charge of controlling the vessel's heading by actuating the rudder.
The reward functions are a simple combination of penalizing cross track error and fast rudder changes, similar to the examples above. Although reaching satisfying performance, the behaviour of the control is still at times erratic, hinting possible problems when deployed on the real system.
Another interesting remark in [22], is the introduction of transfer learning, used to avoid learning from scratch when moving on to different reference path shapes.

Similarly, in [38] the problem of course tracking was considered with the DDPG algorithm. The neural network was in charge of the rudder angle to keep the desired heading. In this article, however, the reward function was kept as simple as possible, containing only a linear term penalizing the heading error:

$$r_t = \begin{cases} 1 - 2 * |\psi_t^d - \psi_t| & |\psi_t^d - \psi_t| \leq 0.5 \\ -|\psi_t^d - \psi_t| & |\psi_t^d - \psi_t| > 0.5 \end{cases} \tag{2.13}$$

where:
ψ_t^d = is the desired heading angle in radians,
ψ_t = is the current heading angle in radians.

In [24], an advanced reinforcement learning based controller was developed for course tracking in varying operational conditions for USVs. The work focused on a fully actuated vessel and with elements inspired by reinforcement learning algorithms. The controller was designed in such a way to enable stability analysis through Lyapunov stability theory.

Another popular topic is the problem of dynamic positioning of a vessel. In [42] the implementation and performance testing of a reinforcement learning algorithm used for dynamic positioning of a marine surface vessel is considered. The marine vessel in question is actuated through a combination of two main rotating thrusters, and one bow thruster. Using the PPO algorithm, explained in 2.2, a neural network was trained to guide a vessel to a random position and keep that position. After training, the control scheme yielded good performance and energy efficiency. In the end, it was tested and evaluated on a model scale on the sea.
The reward function in this article was divided into four parts, described with Equations 2.14, 2.15, 2.16, 2.17, and 2.18:

$$d = \sqrt{\bar{x}^2 + \bar{y}^2} \tag{2.14}$$

$$R_{Gauss} = c_{Gauss} \exp -\frac{1}{2} * (d^2 + 0.2 * \bar{\psi}^2) \tag{2.15}$$

$$R_{AS} = max\left(0, \left(1 - 0.1 * \sqrt{d^2 + 0.2 * \bar{\psi}^2}\right)\right) + c_{const} \tag{2.16}$$

$$R_{vel} = -\sqrt{c_u * u + c_v * v + c_{\bar{\psi}} * \dot{\bar{\psi}}} \tag{2.17}$$

$$R_{act} = -\sum_{i}^{3} (c_{n,i} * |n_i| + c_{\dot{n},i} * \dot{n}_i + c_{\dot{\alpha},i} * \dot{\alpha}_i) \tag{2.18}$$

where:
$\bar{x}, \bar{y}, \bar{\psi}$ = the body frame errors,
$c_{Gauss}, c_{const}, c_u, c_v, c_{\bar{\psi}}, c_{n,i}, c_{\dot{n},i}, c_{\dot{\alpha},i}$ = constants for balancing contributions,
n_i, α_i = the control variables defining the propeller RPM and orientation for thrusters.

The reasoning behind the complex structure of the reward function is the following. The most significant part, R_{Gauss}, utilizes a multivariate Gaussian function to reward proximity to the goal state in the (d, ψ) space. To combat sparsity, a second term R_{AS} was added to guide the algorithm. Furthermore, to avoid overshooting the goal position, a small quadratic velocity cost R_{vel} was added, and finally, for energy efficiency R_{act} was added, penalizing thruster actuation and its derivative.
Defined this way, the reward function contains information about the position, orientation,

and velocity error, as well as encouraging low actuation values and changes to reduce energy consumption and thruster wear.

2.3.3. Real world system actuation through reinforcement learning

Once reinforcement learning algorithms became powerful enough, researchers started seriously experimenting with the transfer from simulation to the real world. These experiments led to numerous problems born from the discrepancy between the simulation and the real world. This section goes over the most significant findings in the field.

Since reinforcement learning algorithms learn to exploit the dynamics of the environment they are trained in, the learned actuation policy usually contains oscillatory and undesirable behavior, similar to the bang-bang controllers, for example explained in Example 10.0.1 of [34]. To avoid learning a controller of this nature, various approaches have been proposed.
In [6], a new architecture called Policy Inertia Controller is presented to avoid severe action oscillations that come from agents selecting very different actions within consecutive steps and similar states. This approach significantly reduces action oscillations without hampering the learning process.
Similarly, in [27] a regularization tool for the action policies called Conditioning for Action Policy Smoothness is introduced. By adding temporal and spatial smoothness terms to the loss function of the learning algorithm directly, this method shows consistent improvement in the smoothness of the learned policy, removing the high frequency elements all together.

Two articles in particular, [8] and [19], provide a good systematic overview of the problems faced in simulation to real transfer with reinforcement learning. In [8], nine unique challenges are presented, such as the limited amount of samples, satisfying safety constraints, and system delays. Each of these challenges, they argue, is not considered adequately in the current literature. In [19] the authors provide a number of successful transfer case studies and outline outstanding challenges in the field and mitigation strategies.

When it comes to these successful implementations of reinforcement learning on real systems, work from ETH Zurich on legged locomotion for quadrupeds presented in [20] particularly stands out. The work consists of actuating a highly complex dynamical system, a quadrupedal robot, to traverse a plethora of challenging terrains with zero-shot generalization. To this end, they developed a novel teacher-student training architecture, inspired by cutting edge findings such as the temporal convolutional networks from [2],

parametrized terrain generation from [35], and complex actuator modelling from [18]. This work provides a goldmine of useful information for future works on successful simulation to real transfer of reinforcement learning.

3 | Simulation of the Roboat and design of the RL controller

Given the sample hungry nature of reinforcement learning algorithms, training on real systems is rarely feasible. Therefore, it has become common practice to create a simulation of the systems in code, making it as realistic as possible while keeping the simulation complexity fairly low. The controller is trained first in the simulation, then transferred to the real world, where fine tuning can be performed if needed. The simulation of the true system is often called its "digital twin".
This chapter introduces various available simulators for the ASV setting, as well as the process of building the one used to train a controller in this study. Firstly, an overview of the simulators free to use for this setting is given. Secondly, the current simulator used by the Roboat team to simulate their platform is described. And lastly, the new simulator built for reinforcement learning is presented, with the rationale on why such a simulator is needed for the purposes of this study.

3.1. Available ASV simulator survey

Supporting many studies conducted on control of different vessel systems, a plethora of frameworks have been created. Understandably, each one of them is tailor made for the problem at hand, making it more or less suitable for this study. These simulators can be categorized based on the types of vessels available, the degree of complexity of the physics simulated (for example fluid dynamics or basic kinematics and dynamics), or the simulated disturbances (for example varying payload, wind, current, waves, thruster dynamics). To get a sense of the diversity of available platforms, Figure 3.1 provides some examples.

Figure 3.1: Examples of different ASV platforms.

Following a thorough research of available simulators, six promising candidates were taken in consideration for the purposes of this study. Table 3.1 presents a high level rundown of their capabilities, followed by a small overview of each and every one.

Simulator	**VRX**	**USV LSA**	**UWSim**	**RaiSim**	**Gazebo**	**ShipAI**
Vessel	WAM-V	4 basic models	Custom	Custom	Custom	Custom
Physics complexity	Intermediate	Advanced	Intermediate	Advanced	Basic	Basic
Payload	Yes	Yes	Yes	Yes	Yes	Yes
Wind	Yes	Yes	No	No	No	No
Current	No	Yes	No	No	No	No
Waves	Yes	Yes	Yes	No	No	No
Thruster dynamics	Yes	Yes	Yes	No	Yes	No

Table 3.1: Available ASV simulators considered

- **Gazebo**: classic.gazebosim.org

  General Gazebo is a multipurpose simulator applicable to a number of unmanned systems including aerial, ground, surface, and underwater vehicles. Its goal is to enable rapid robot design, testing, and training of AI algorithms in realistic

scenarios. It is written in C++ and can be used on top of ROS and upgraded to simulate new systems.

Strengths This simulator is a highly modular (large part of available ASV simulators have Gazebo at its core) and popular simulation tool in the robotics community. It offers basic ingredients to build complex physical systems quickly and freedom in the design of the simulated platform.

Weaknesses Most works using Gazebo, nonetheless, focus on aerial, ground, and underwater systems, with only a few concerning ASVs. Additionally, although it offers the tools needed to simulate the desired disturbances, the simulator by itself offers none of them out of the box, and the process is unnecessarily complex.

- **VRX simulator**: github.com/osrf/vrx

  General The simulator is designed with the RobotX competition in mind (eraurobotx.org/competitions), with the goal of finding innovative approaches to problems such as autonomous sensing and mission implementation. It is intended as a fist step toward solving challenges in the competition. The simulation is built in C++ around a wave adaptive modular platform (WAM-V) used in the competition, an example of which can be seen in 3.1, and includes aspects of the environment such as some of the disturbances and even obstacles.

  Strengths This simulator is built on top of the Gazebo package for the Robotic Operating System (ROS) and completely open sourced. As part of the simulator, parametrized models for wave and wind disturbances is implemented. The physical wave model is based on the Gerstner waves and the two-parameter Pierson Moskowitz spectrum and the wind model is represented with the mean and variance based on the Harris spectrum [4]. Moreover, the propulsion model is parametrized with the number and allocation of thrusters.

  Weaknesses However, being built for the WAM-V platform, changing the physical properties to suit the Roboat platform to a high degree is not possible. Furthermore, although parametrized, the propulsion model cannot be adjusted to resemble the plus configuration of the Roboat platform.

- **USV sim LSA**: github.com/disaster-robotics-proalertas/usv_sim_lsa

  General Built for the purpose of developing and testing control and trajectory strategies for USVs in disaster scenarios [29], this simulator offers four different boat models (rudder boat, differential boat, airboat, and sailboat) and a com-

plex disturbance simulation. It is written mostly in Python on top of Gazebo and uses multiple additional packages such as the OpenFoam computational fluid dynamics (CFD) simulator (openfoam.com) for wind currents.

Strengths Among all the simulators considered, USV sim LSA offers by far the most precise open source physics simulation. The four available boat models are divided in a multibody system of six parts with fixed joints to simulate realistic wave buoyancy effect. Moreover, three types of disturbances are modelled through CFD: wind currents, water currents, and waves. The way in which they affect the system can be "local", "global" or turned off completely.

Weaknesses However, all this comes at a cost of performance. Since the simulator was built with only testing in mind, the goal was to reach the highest level of realism possible. Unfortunately, although extremely powerful, this makes it unusable for reinforcement learning.

- **UWSim**: irs.uji.es/uwsim

  General Although originally built with underwater systems in mind, this simulator can also be used for surface vessels. Built as a ROS package in C++, it is modular, extensible, light, and offers basic wave simulation. However, it is mainly used as a visualization tool for other modules such as Gazebo, which is then responsible for dynamics and control.

  Strengths Being a native ROS package, UWSim is easily plugged into any ROS based framework such as the Roboat platform. Moreover, the simulator does include disturbance models to some degree and customizable vessel model.

  Weaknesses Sadly, UWSim lacks models of important disturbances studied in this work like current and wind. Furthermore, modeling more realistic boat behavior is either unfeasible or requires significant work.

- **RaiSim**: raisim.com

  General RaiSim is a multibody physics engine for simulating robotic systems and training AI algorithms on them. Written in C++ it is one of the fastest physics engines today, easy to use with almost no dependences. It gained popularity through a number of papers published recently, like [20] in which it was used to create a highly realistic quadrupedal robot and train a reinforcement learning agent for its locomotion.

  Strengths In RaiSim one can create a highly customizable system and define dis-

turbances and uncertainties through available its tools. Moreover, advanced reinforcement learning algorithms can be trained natively in it with proven speed and quality. It is, therefore, a great example of such architecture.

Weaknesses Unfortunately, simulating surface and underwater systems with disturbances and uncertainties would have to be done from scratch as it is still not available in the simulator. And since it is close sourced doing so might prove difficult, if not impossible.

- **ShipAI**: github.com/jmpf2018/ShipAI

  General A ship simulator wrapping basic ship physics and navigation in a reinforcement learning module created for the purposes of [9]. It comes with implementations of Deep Q-Network and Deep Deterministic Policy Gradient algorithms for ship navigation. The ship dynamics can be adapted for any platform and so can the reinforcement learning algorithm parameters such as the reward function and state design.

  Strengths ShipAI is an open sourced simulator already built on the OpenAI gym model making it simple, yet immensely modular, standardized, and capable. It is yet another successful example of wrapping a vessel simulation in a reinforcement learning module.

  Weaknesses The simulator considers only propeller rudder actuation systems meaning it cannot be used in this study. Additionally, it includes no disturbance or uncertainty models, nor offers considerable flexibility with the learning algorithms.

An overarching theme through the research done on available simulators is that using prebuilt ASV systems for reinforcement learning purposes introduces significant overhead in communication which would render the learning process painstakingly slow.
And in the same way, starting from simulators not originally built with ASV platforms in mind leads to tweaking complex frameworks to simulate the necessary dynamics instead of their initial usecase, creating multiple unsupervised dependencies in the process.
Therefore, for the purposes of this study and inspired by the listed frameworks, a new simulator of the Roboat platform's dynamics was built. Written in Python, following the OpenAI gym architecture, it is fast and modular with regards to both training new reinforcement learning algorithms and deployment as a ROS node controller. Moreover, the kinematics, dynamics, uncertainties and disturbances described in Chapter 1 have been included in code and available for both training and testing.

3.2. Roboat platform

Roboat is a novel ASV platform (roboat.org) with the mission of unlocking the potential of urban waterways through cutting-edge autonomous technologies and design. The four meter long vessel is designed to enable maximum payload and docking through six latching modules. Actuated through a combination of two main thrusters for cruising and two bow thrusters for precise movement (as described in Section 1.2), it can carry up to 6 people with a maximum payload of 1000kg. Furthermore, its rectangular shape makes generating infrastructure such as bridges possible, and its modular deck allows changing the boat functionality. The platform implements four main functionalities: transport, garbage collection, on-demand delivery of goods, and dynamic infrastructure.
The full scale vessel is fitted with an array of sensors for the tasks of perception and localization. These include a dual RTK GPS for heading and centimeter-level precise positioning, a 128-beam LiDAR for obstacle detection, a set of cameras for each latching module, and an IMU. Moreover, the latching system consists of 6 mechanical robotic arms guided by cameras. Based on ROS, computation is done on an Intel NUC computer, with a Raspberry Pi 4 used for image processing. Finally, the entire system is powered by a 12kWh battery yielding about 12 hours of autonomy.
The autonomy framework developed for navigation is implemented in C++ in the ROS environment with state estimation running at 100 Hz and all the other algorithms 10 Hz. An example of a Roboat vessel can be seen in 3.1.

3.3. Roboat simulator

Before deployment on the real life platform, new functionalities of the autonomous package can be tested on the system's so-called "digital twin". Built in Gazebo, it entails a physical simulation of the vessel, the sensors, the environment of Amsterdam canals with obstacles, the latching mechanism, and basic thruster dynamics. Since ROS packages work in a decoupled manner, switching between running the system in simulation or on the real platform is as easy as changing one parameter. Running the Roboat ROS framework virtually looks like 3.2.

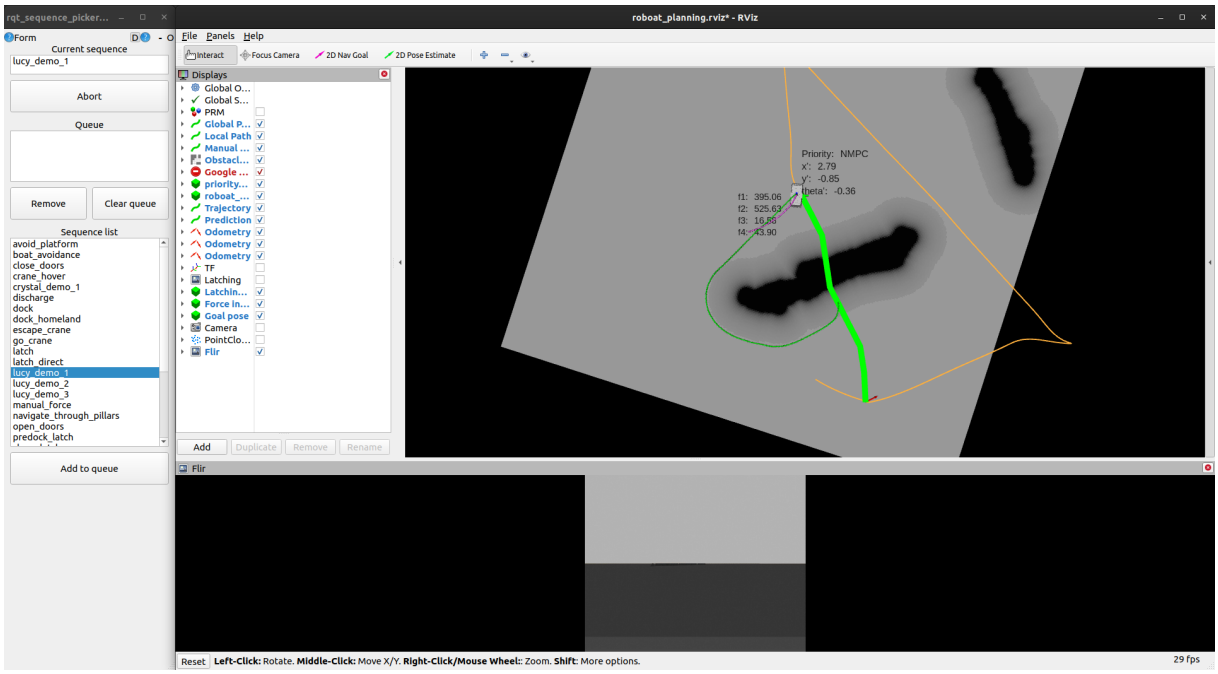


Figure 3.2: The Roboat simulation user interface.

The user interface of the simulation consists of the following parts. Firstly, two windows open up upon initialization, one presenting RVIZ visualization of the environment and the other an instruction stack dialogue defining the sequences to be run. The RVIZ windowis divided in typical fashion: the ROS topic selection list, the main 3D environment visualization, and the camera feed from the vessel. The grey 3D mesh presents the vessel itself, the thick green line is the global path, the darker, thin green line is the local path, the red arrow at the end of the global path is the goal state, the yellow line is the past traversed path, the purple line is the future path predicted by the NMPC. Furthermore, to the vessel's right are the u, v, r velocities, and to the right are the force values for the thrusters denoted in newtons.

3.3.1. Path and trajectory planning

The path planning module is divided in two parts: the global planner and the local planner. Given the desired goal state, the global planner calculates a high level path starting from the current state. It is formulated as a graph search problem on a sparse directed graph $G = (V, E)$ that represents the topological map of the Amsterdam canals, V being the canal intersections and E the canal segments. The global path is the shortest path from the current to the goal state, calculated through an A* search.

Starting from the global path, the local planner plans a sequence of states for the future 6 seconds, taking into account distance from detected obstacles and overall length. Finally, the trajectory is created by interpolating the local path and calculating the required speeds, adhering to canal speed limits and physical limitations of the system.
The framework architecture is presented in 3.3. Interesting parts regarding trajectory tracking in the architecture are: the path planner, the physical model, and the NMPC controller.

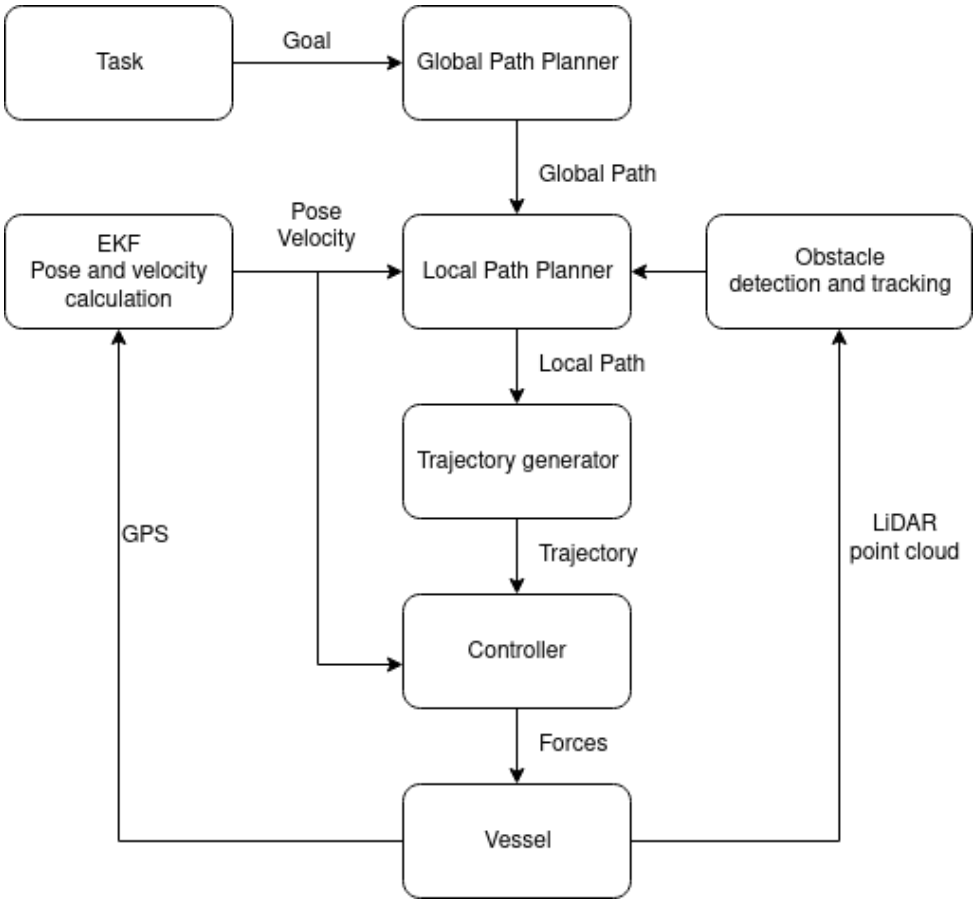


Figure 3.3: The Roboat navigation stack architecture.

3.3.2. Physical model

The platform is described as a 3 degree-of-freedom system with the 6 dimension state containing the global x, y, and ψ, and local surge, sway, and yaw velocities u, v, and r. As presented in Chapter 1 the physical model can be described with the nonlinear differential equation:

$$\boldsymbol{\eta} = [x, y, \psi] \tag{3.1}$$

$$\boldsymbol{\nu} = [u, v, r] \tag{3.2}$$

$$\dot{\boldsymbol{\eta}} = \boldsymbol{R}(\psi) \cdot \dot{\boldsymbol{\nu}} \tag{3.3}$$

$$\dot{\boldsymbol{\nu}} = \boldsymbol{M}^{-1}\boldsymbol{\tau} - \boldsymbol{M}^{-1}(\boldsymbol{C}(\boldsymbol{\nu}) + \boldsymbol{D}(\boldsymbol{\nu})) \tag{3.4}$$

$$\boldsymbol{\tau} = \boldsymbol{B}\boldsymbol{u} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ \frac{a}{2} & -\frac{a}{2} & \frac{b}{2} & -\frac{b}{2} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix} \tag{3.5}$$

In the equations, $\boldsymbol{M}$ presents the symmetric, positive-definite mass matrix, $\boldsymbol{C}$ presents the skew-symmetric matrix of Coriolis and centripetal forces, $\boldsymbol{D}$ the positive-semi-definite drag matrix, and $\boldsymbol{B}$ is the control matrix defining the thruster cross configuration. $\boldsymbol{R}(\psi)$ is the rotation matrix between the global and body reference frames. Moreover, $\boldsymbol{\tau}$ presents the actuation force vector from the thrusters, and $\boldsymbol{u}$ the control vector.

3.3.3. Nonlinear model predictive control

At the time of writing, the Roboat platform utilizes an NMPC for trajectory tracking with constant physical parameters of the system. The NMPC problem is formulated as an optimization of a least square function penalizing deviations of state $\boldsymbol{q_k}$ and control $\boldsymbol{u_k}$ from the reference trajectory over a given horizon N_c, presented in 3.6:

$$\min_{\boldsymbol{q_k}, \boldsymbol{u_k}} \frac{1}{2} \left[\sum_{k=j}^{j+N_c-1} \left(\left| \boldsymbol{q_k} - \boldsymbol{q_k^{ref}} \right|^2_{W_q} + \left| \boldsymbol{u_k} - \boldsymbol{u_k^{ref}} \right|^2_{W_u} + \left| \boldsymbol{q_{N_c}} - \boldsymbol{q_{N_c}^{ref}} \right|^2_{W_{N_c}} \right) \right] \tag{3.6}$$

where:
$\boldsymbol{q_j}$ = the current estimated state,
$\boldsymbol{q_k}, \boldsymbol{u_k}$ = the state and control vectors,
$\boldsymbol{q_k^{ref}}, \boldsymbol{u_k^{ref}}$ = the state and control references,
$\boldsymbol{q_{N_c}}, \boldsymbol{q_{N_c}^{ref}}$ = the terminal state vector and reference,
W_q, W_u, W_{N_c} = the weight matrices for deviation penalization.

The weights are set in such a way that they penalize position and orientation error approximately three times as much as velocity error. The same goes for using the bow thrusters, the use of which is discouraged through the cost seven times that of the main

thrusters, as they are less efficient at higher speed cruising.
This approach to control works perfectly well in controlled and static environments. However, as stated in the Introduction, with the increase in the mismatch between the model and the real world system the deviation from the planned trajectory increases too.

3.4. Reinforcement learning simulator

When it comes to more complex environments, control algorithms require elaborate models of the system which can prove too difficult to obtain. An example of such systems are legged robots in challenging environments, a setting taken on by in [20]. To overcome this, focus has somewhat shifted to learning approaches in control, enabled by the recent rise in computation power. Although partially sacrificing insight into the algorithm, when correctly set up, these algorithms find intrinsically organic and robust solutions. To train such a controller, it is common practice to build a "digital twin" of the real system. The following section presents the simulator built for the purposes of this study.

3.4.1. Environment setup

The recent rise and popularity of reinforcement learning has in a large part been possible due to its open source nature. Today, having a new environment in which to test reinforcement learning algorithms open for anyone to use is industry standard. One of the first to start this trend was a research and development company OpenAI (openai.com). In [5] they proposed a toolkit for reinforcement learning research called a "gym", which provides a common interface for any general task to be solved. The toolkit is continuously being developed and updated, with tens of available environments including games like Atari Breakout, bio inspired robot control problems such as bipedal and multi legged walkers, traditional robot control problems such as the inverted pendulum, and many more, as can be seen in Figure 3.4.

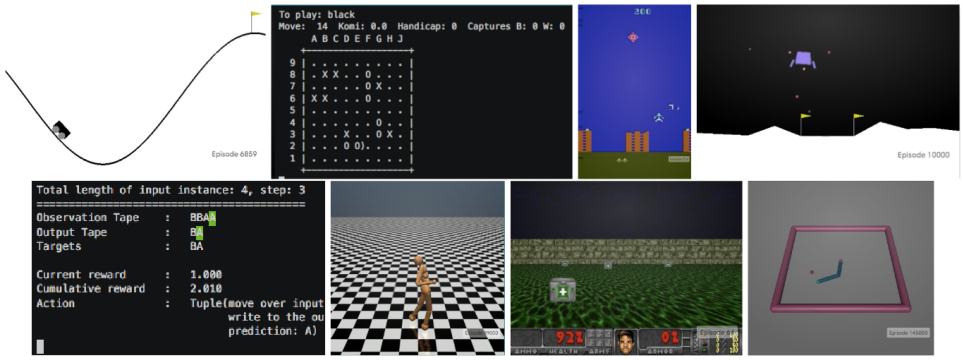


Figure 3.4: Examples of gym environments visualized from [5].

Furthermore, an API like this gives the developers an opportunity to quickly implement their own environments based on the same architecture and easily use learning algorithms on it. To use this interface the following requirements have to be met:

- Attributes:

  Observation space Defines the data structure that describes the manner in which the agent senses the environment. For example, the type *Box* encodes the agent's observation in a multidimensional array, with each dimension having a specified range.

  Action space Defines the data structure that describes the manner in which the agent operates in the environment. For example, the type *Discrete* encodes actions as a vector of discrete values from a specified range.

- Methods:

  Step A function that runs one timestep of the environment's dynamics. The input into the method is an action vector from the action space and it returns a tuple of $(observation, reward, done, info)$. The *observation* is the state of the environment after the timestep, *reward* is calculated from a reward function such as the ones presented in 2.3.1 as a result of the change, *done* is the episode termination signal, and $info$ as optional information in the form of a dictionary.

  Reset A function that resets the environment to initial conditions and returns the initial observation from the observation space.

To model the system at hand, the observation space is defined as an array of continuous values containing the values of the position difference between the current and the desired state in the vessel's reference system, the lookahead heading error, the current and desired

values surge, sway, and rotational velocities, and the values of actuation for the last step. This results in a vector of 14 values:

$$\boldsymbol{observation} = \begin{bmatrix} \Delta x \\ \Delta y \\ cos\Delta\psi \\ sin\Delta\psi \\ u \\ u_r \\ v \\ v_r \\ r \\ r_r \\ u_{1,t-1} \\ u_{2,t-1} \\ u_{3,t-1} \\ u_{4,t-1} \end{bmatrix} \tag{3.7}$$

The lookahead heading error ψ is divided into $cos\psi$ and $sin\psi$ because of discontinuity around 0 value and multiple possible definitions of the same angle, similar to [20]. Using the lookahead heading error instead of the current one was found to improve the learning process, inspired by trajectory tracking algorithms for cars.
Furthermore, the action space is defined as a four number vector with values between $[-1, 1]$. These are then mapped linearly to maximum and minimum main and bow thruster force values, taking into account a small loss of efficiency when operating in reverse.
The step and reset functions define the interaction between the simulated environment and the learning algorithm. As described, the step function takes as input an action vector, performs step in system's dynamics of one timestep with the desired actuation, calculates the reward according to the reward function, and outputs a $(observation, reward, done, info)$ tuple. And lastly, just like at initialization, the reset function restores the initial state for the system, generates a new trajectory to be tracked, and initiates uncertainties and disturbances.

3.4.2. Uncertainties and disturbances

These are implemented according to the research presented in 1.3. Firstly, the payload is varied by multiplying the mass matrix with a random value uniformly distributed from

[1, 2]. This results in an increase of up to 1169 kg. The specific terms in the 3 DOF are then modified separately with a small Gausian noise.
Secondly, the wind is described as a force and torque acting on the system, with the apparent wind speed V_{rw} and the wind's apparent angle of attack ω. The maximum value is defined as 10 m/s, being the maximum speed of the wind measured on the real system and the angle is taken at random.
And lastly, the current is described as a constant drift of the moving reference frame, with speed V_c and heading β. Maximum value of the current is defined as 0.5 m/s according to information available on the internet. All of these effects can be turned on or off in a given environment.

3.4.3. Reward function

Based on research presented in 2.3.1, the reward function is defined through the deviation from the desired state and the use of thrusters.

$$r_t = \begin{cases} \exp\left(-k_1 * \boldsymbol{\Delta q}_{\boldsymbol{xy,t}}^2\right) + \exp\left(-k_2 * \boldsymbol{\Delta \psi}_{\boldsymbol{t}}^2\right) - k_3 * \boldsymbol{u}_{\boldsymbol{cost,t}} - k_4 * \boldsymbol{\Delta u}_{\boldsymbol{cost,t}} & \text{if } \boldsymbol{\eta}_{\boldsymbol{error,t}} \leq 5 \\ -100 & \text{if } \boldsymbol{\eta}_{\boldsymbol{error,t}} > 5 \end{cases} \tag{3.8}$$

$$\boldsymbol{\Delta q}_{\boldsymbol{xy,t}} = \sqrt{(\hat{x}_t - x_t)^2 + (\hat{y}_t - y_t)^2} \tag{3.9}$$

$$\boldsymbol{u}_{\boldsymbol{cost,t}} = k_5 * (u_{1,t}^2 + u_{2,t}^2) + k_6 * (u_{3,t}^2 + u_{4,t}^2) \tag{3.10}$$

$$\boldsymbol{\Delta u}_{\boldsymbol{cost,t}} = \sum_{i=1}^{4} |(u_{i,t} - u_{i,t-1})| \tag{3.11}$$

where:
$\boldsymbol{\Delta q}_{\boldsymbol{xy,t}}$ = the Euclidian distance between the current and desired position at time t,
$\boldsymbol{\Delta \psi}_{\boldsymbol{t}}$ = the difference between the current and desired heading at time t,
$\boldsymbol{u}_{\boldsymbol{cost,t}}$ = the weighted sum of actuation values,
$\boldsymbol{\Delta u}_{\boldsymbol{cost,t}}$ = the derivative cost of actuation calculated as the sum of differences from state t-1 to t,
$k_i (i = 1, ..., 6)$ = constants balancing the effect of each component on the learning process.

The parameter values balancing the reward function are given in Table 3.2.

Parameter	k_1	k_2	k_3	k_4	k_5	k_6
Value	3	3	1	0.5	0.1	0.7

Table 3.2: Constant values in the reward function

Defined like this, the reward function guides the learning process towards precise state tracking through a high reward around the desired state and a Gaussian dropoff around it, and low actuation values both in magnitude and the rate of change. For better understanding, the Gaussian terms of the reward function for the pose are visualized in Figure 3.5.

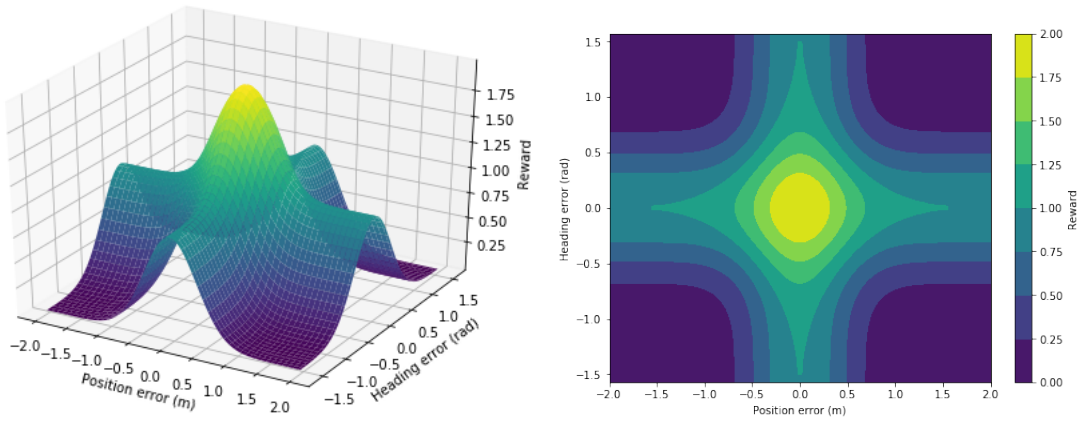


Figure 3.5: Plots of the Gaussian terms of the reward function for Euclidian position error and heading error.

3.4.4. Trajectory generation

To run the "data hungry" learning algorithm in the simulated environment, each episode needs a reference trajectory to track. Of course, in order for the algorithm to generalize, these reference trajectories should cover as many different situations as possible. Depending on the desired task, this problem can be approached in many ways. For example, for the task of dynamic positioning the desired final state in the current state's proximity can be picked at random and changed upon holding it for some time.

On the other hand, inspiration for trajectory tracking can be found in traditional vessel performance tests. The same reasoning of trajectory generalization mentioned before applies to testing the performance of traditional tracking controllers. Therefore, a trajectory

generator module was built to train the algorithm, based on a combination of the most prominent tests in literature, for example Chapter 5 of [10].

The trajectories generated by the generator cover trajectories with: straight line speed tracking, circles of different turning radii and speed, sine wave shaped trajectories with varying amplitude, period, and speed. These are stored as arrays of 600 states, presenting a 60 seconds long timeframe with 0.1s sampling frequency. Examples of the types of trajectories can be seen in 3.6. Once filled with states, the trajectory is randomly rotated to get a richer training dataset and better generalization.

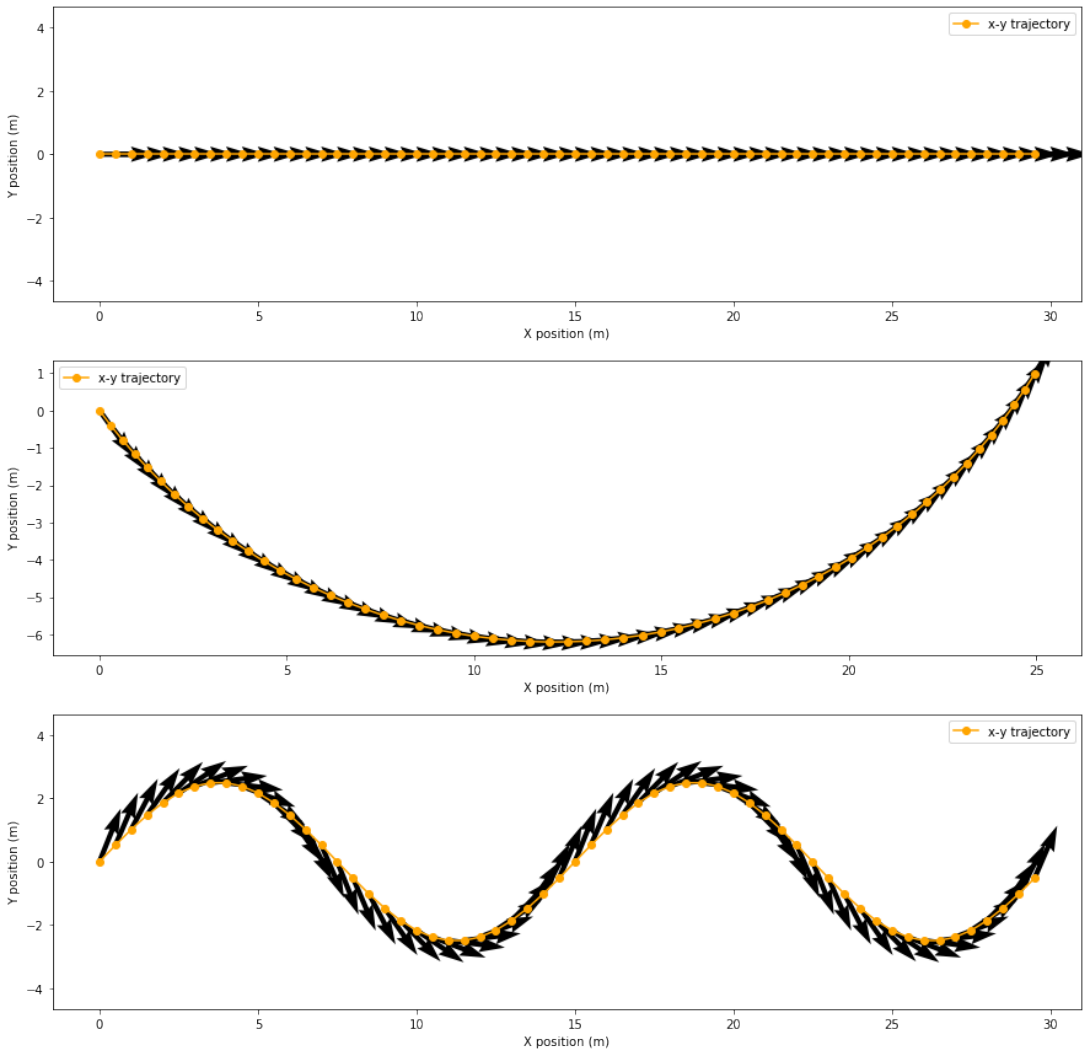


Figure 3.6: Examples of a straight, circular, and sine trajectories. The visualized states are sampled every 1s for clarity.

3.4.5. Algorithm setup

With the popularization of the reinforcement learning field in recent years, various toolkits offering implementations of complex algorithms for fast and reliable prototyping have been developed. One of the most prominent such toolkits is called the Stable Baselines3. Completely open sourced, Stable Baselines3 is a set of improved implementations of Reinforcement Learning (RL) algorithms based on OpenAI Baselines (github.com/openai/baselines). This toolset is its major structural refactoring. Unlike OpenAI Baselines, a unified structure for all algorithms, unified code style, thorough documentation, and additional algorithms, such as SAC and TD3 are available. More on the Stable Baselines3 framework can be seen in [30].
Table 3.3 presents the best candidates for training a trajectory tracking network, all of which are provided by the Stable Baselines3 framework. The choice of the PPO algorithm was made due to: availability of Box action spaces (not provided by the DQN), support for multiprocessing, and speed (PPO is an evolution of the slower TRPO, combining the ideas from TRPO and A2C as explained in 2).

Algorithm	Type	Box	Multiprocessing
DQN	Value	No	Yes
A2C	Policy	Yes	Yes
TRPO	Policy	Yes	Yes
PPO	Policy	Yes	Yes

Table 3.3: Comparison between considered RL algorithms.

Furthermore, Figure 3.7 from [33] provides a comparison of a number of reinforcement learning algorithms in the OpenAI Walker2d environment. The relevance of this lies in the fact that this environment closely resembles the problem at hand in the environment architecture: observation shape (reflecting robot's current physical state), problem generation (parametrized generation of terrain similar to trajectory generation), action space (continuous, controlling the forces and torques of actuators), reward shape (early termination and energy cost). From the Figure it is clear PPO outperforms other algorithms on a similar problem, further enforcing the choice of algorithm for the problem at hand.

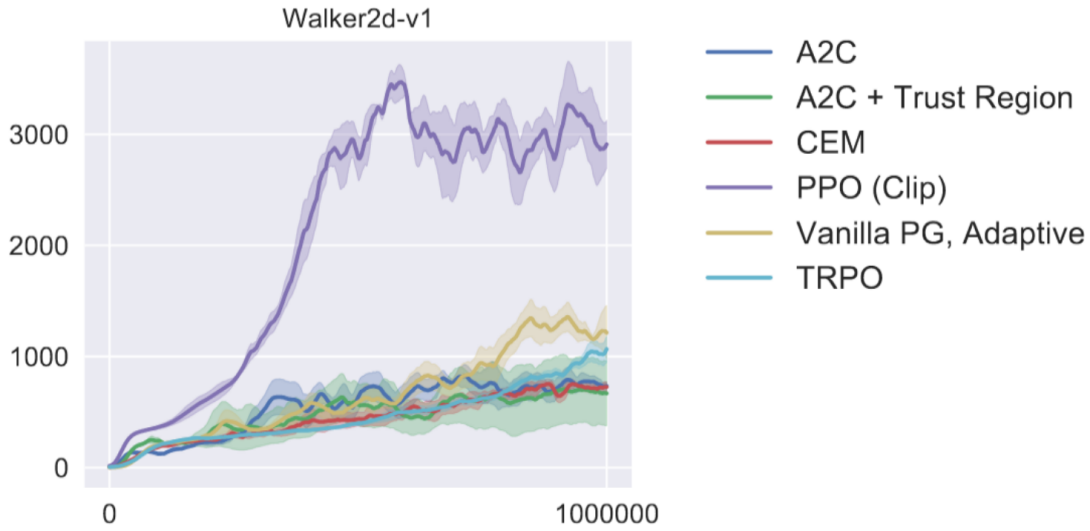


Figure 3.7: Comparison of performance between a number of reinforcement learning algorithms in the OpenAI Walker2d environment, a bipedal robot simulation.

Another useful component of the Stable Baselines3 framework is a function called `evaluate_policy(model, env, n_eval_episodes)`, which enables users to easily compare the quality of trained policies. It takes three parameters as input: the model network of the learned policy, the environment in which to run the agent, and the number of episodes to run. The output is the mean reward per episode and the standard deviation of reward per episode reward acquired in episodes run.

3.4.6. Learning process

After taking all the necessary steps to set up the algorithm and the environment, a reinforcement learning algorithm can start to learn. To track the improvements, the process is usually observed through a number of metrics. For environments that implement episode termination based on the performance the most telling are the average episodic reward and the average episode length. An increase in both of those metrics signals improved performance. Figure 3.7 is a typical example of the learning progress shown through the average episodic return.

To better convey how the algorithm improves over time, Figure 3.8 provides snippets of the model tracking performance immediately after initialization, after 100000, 200000, 500000, and finally one million steps of training. As the reference trajectory, a sine wave with a period of 15 m, an amplitude of 2 m, and a forward speed of 0.6 m/s is given, providing a challenging, yet fairly traversable task for the algorithm. The dashed orange

line presents the reference trajectory and the blue line the actual trajectory of the vessel for that episode. As explained before, the episode terminates when the current position goes beyond 5 m from the reference position.
The tracking obviously improves with more timesteps, starting from random weights and finding a crude policy already after 100000 steps of training. Slowly, the algorithm fine tunes the policy, exhibiting smooth behavior and high precision after one million steps.

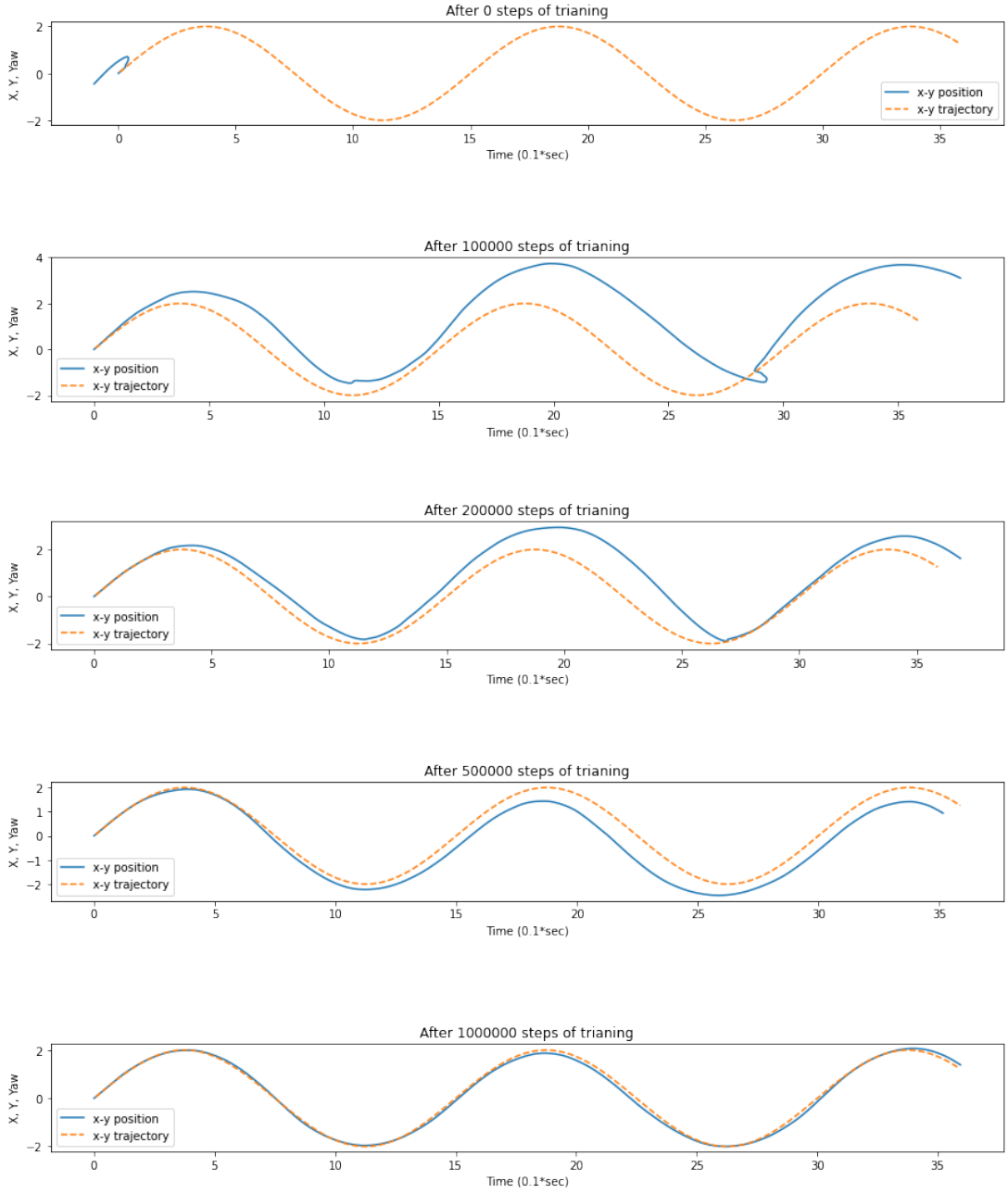


Figure 3.8: Visualization of intermediate trajectory tracking training results for the RL controller after initialization, 100000, 200000, 500000, and 1000000 training steps.

Figure 3.9 presents the plot of the episodic reward through the one million timesteps of training. From the Figure, it can be seen that in the first $\tilde{5}00000$ steps the algorithm learns the dynamics of the environment in large leaps, increasing the episodic reward drastically.Following this, the algorithm's improvement slows down, with the episodic reward plateauing at around 650. The training is stopped here to avoid overfitting and loss in generalization ability.

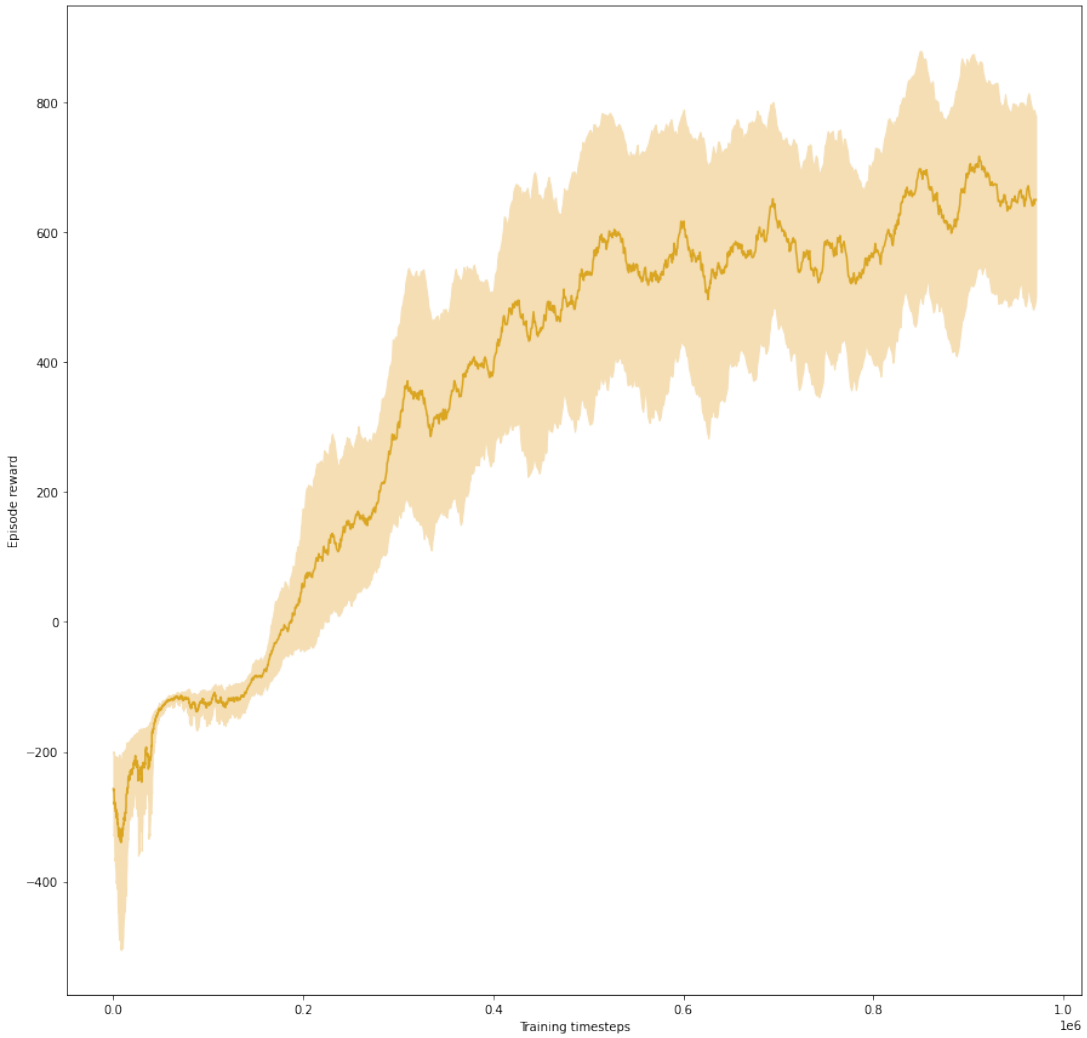


Figure 3.9: Plot of the episodic reward throughout one million steps of training. Orange line presents the moving average with a window size 50, light orange presents the standard deviation of the value.

Having obtained satisfactory performance from the algorithm, tests comparing it to the current NMPC controller can be performed. The results of the comparison are presented in Chapter 4.

3.5. Comparison

Before proceeding to train a control algorithm for the problem at hand, one must prove that the simulator in which the learning process is executed correctly approximates the real system. Furthermore, in order to compare the performance of the learned policy with the traditional NMPC algorithm, which can be run only in the Roboat simulation, one must validate that the new simulation closely resembles the simulation in ROS.
One way to approach this is to check the response of both the simulators when excited with the same actuation vectors. And in order to validate them against the real system the same can be done on the physical platform, be it in a more modest manner, as real world testing entails infrastructure complications such as launching and sail scheduling.
In this study, the system's behavior will firstly be tested without the disturbances to validate the reinforcement learning simulator against the Roboat simulator. Secondly, the behavior when facing varying payload, current, and wind will be studied. This will be done only in x axis as the same behavior applies to all 3 DOF.

3.5.1. Step response without disturbances or uncertainties

The experiments were done in the following manner: a step function climbing to the maximum available thruster RPM was given as a command to three different combinations of thrusters inducing motion in the system's 3 DOF. The response was recorded as an array of states and compared both in magnitude and difference between the two. Figures 3.10, 3.11, and 3.12 present the result visualizations.
Furthermore, to get a sense of the magnitude of the effect of disturbances and uncertainties the deffect induced by edge cases in varying payload, current, and wind is presented. These responses can be seen in Figures 3.13, 3.14, and 3.15.

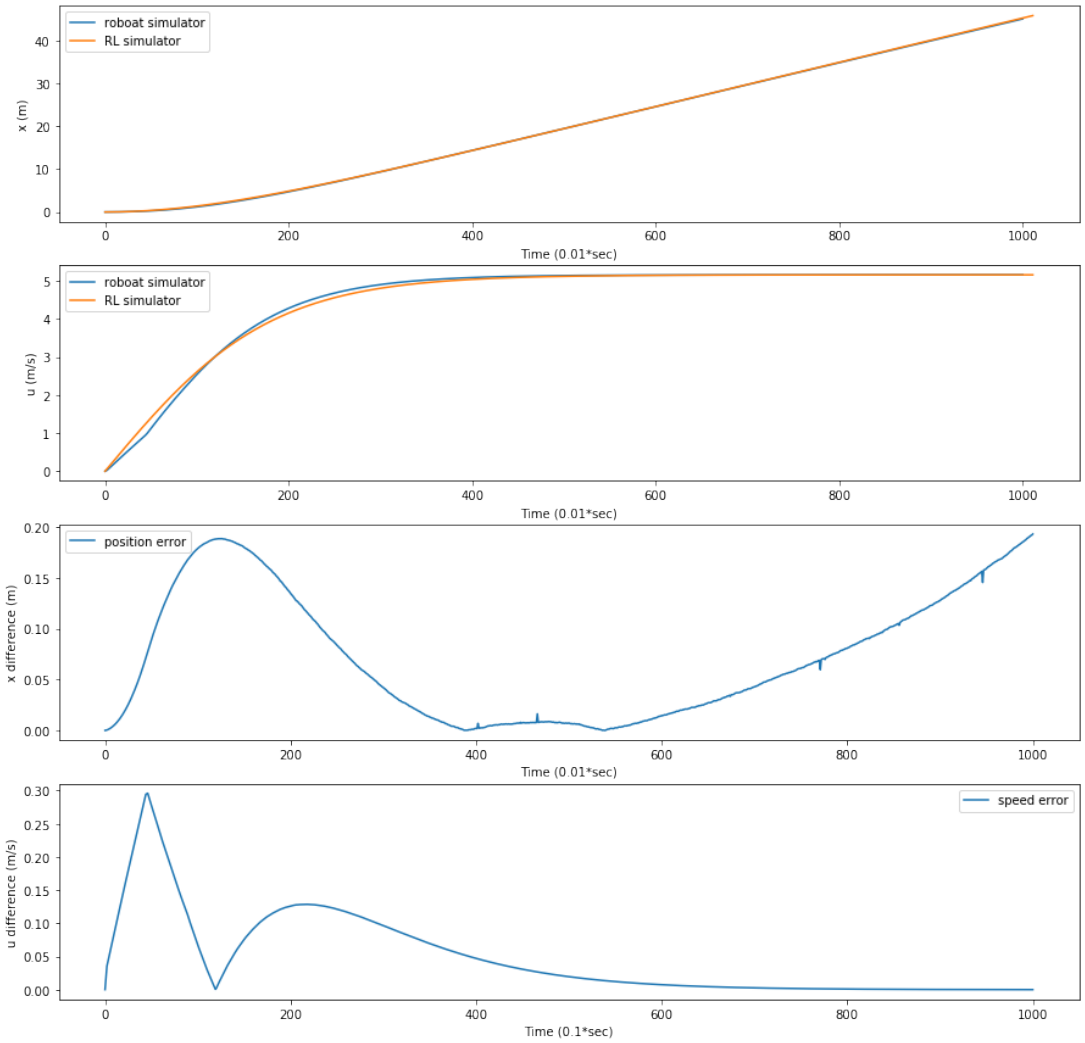


Figure 3.10: Simulator comparison between the Roboat simulator and the reinforcement learning simulator. X and u step response, followed by the difference between the two.

In Figure 3.10 the maximum thrust command was given to the two main thrusters, amounting to a straight, forward motion. The top two graphs contain the comparison between step responses in both position and velocity, and the bottom two contain the difference between the simulators in position and velocity. Comparing the difference between the responses, it is obvious the simulators work almost identically. The terminal velocity of the vessel is equal for both simulators and the rampup very similar.

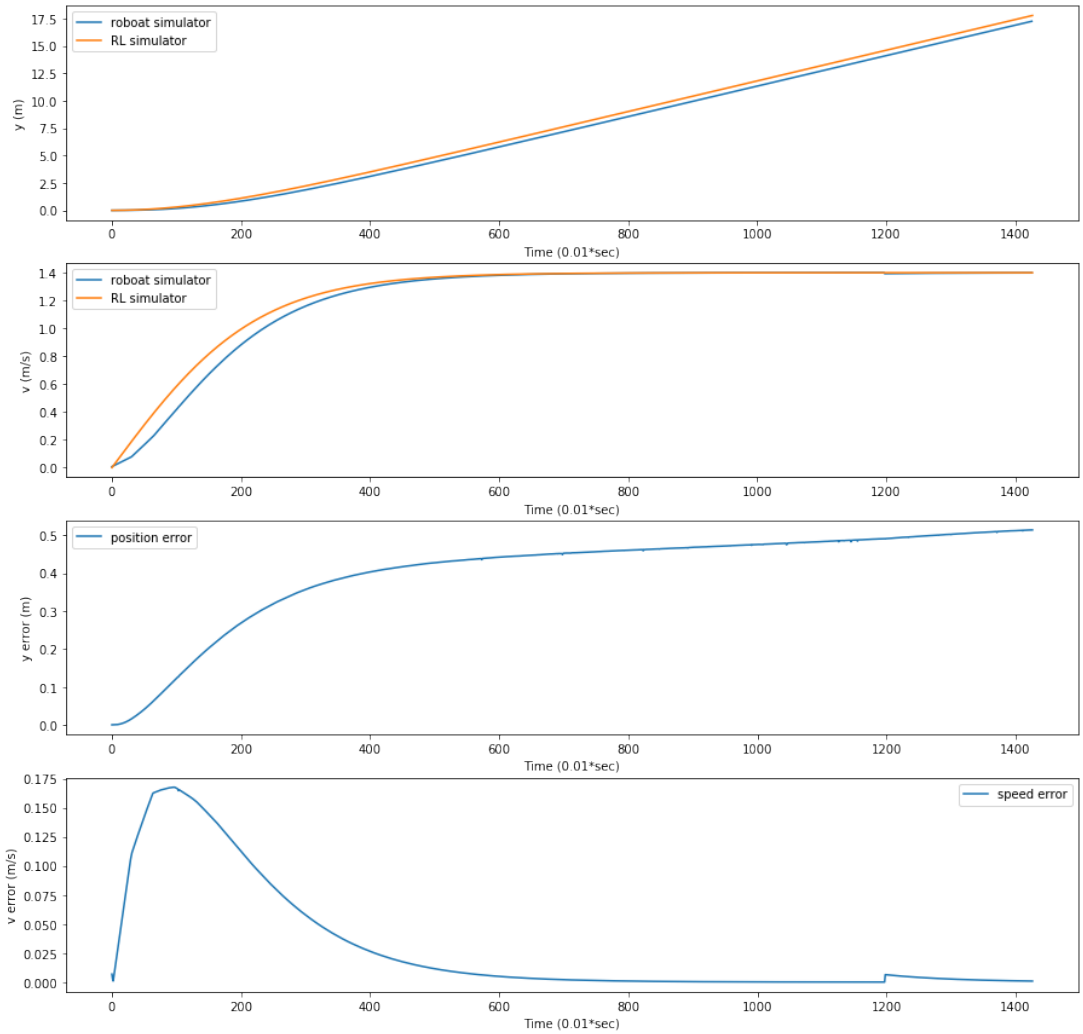


Figure 3.11: Simulator comparison between the Roboat simulator and the reinforcement learning simulator. Y and v step response, followed by the difference between the two.

In Figure 3.11 the maximum thrust command was given to the bow thrusters to induce sideways motion. The position difference settles at around 0.5m, which is not negligible, however, it can be attributed to initial thruster noise in the Roboat simulator leading to the accumulation. Other than that, the responses are once again very similar, with the terminal velocities equal, and the difference between the two is minimal.

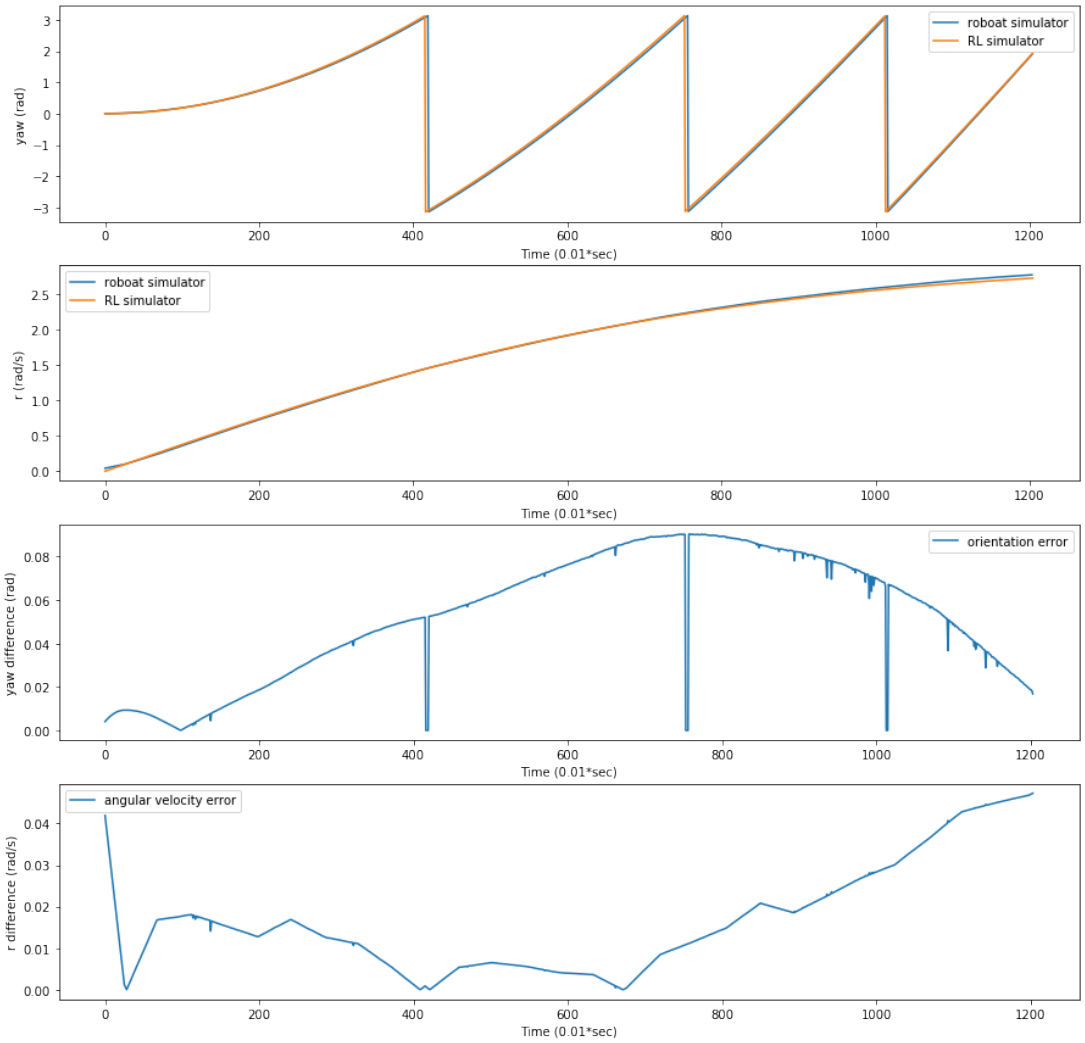


Figure 3.12: Simulator comparison between the Roboat simulator and the reinforcement learning simulator. ψ and r step response, followed by the difference between the two.

Lastly, Figure 3.12 contains the response to the maximum RPM given to the main thrusters, but in different directions. This induces a torque rotating the vessel. The saw-like appearance of yaw is thanks to the nature of angle notation, jumping between π and $-\pi$. The same reason causes the three spikes in the yaw difference graph, which have been canceled for more clear visualization. Still, the difference in both orientation and speed is minimal.

These three figures clearly show that the physics dynamics of the reinforcement learning simulator closely resemble those of the Roboat simulator. The small differences in position and velocity responses can be attributed to effects such as the added thruster noise and computation latency of the ROS communication frmework in the Roboat simulator. Therefore, the reinforcement learning simulator is a valid starting point for learning

actuation for the Roboat platform.

3.5.2. Step response with disturbances or uncertainties

As explained in the Introduction, varying payload, current, and wind significantly affect the dynamics of the system. To train an algorithm robust to these effects, an environment capable of simulating them realistically is needed. In the reinforcement learning simulator built for the purposes of this study, the effects of disturbances and uncertainties was implemented as described in 1.3. Figures 3.13, 3.14, and 3.15 provide visualization of the magnitude of their effect on the same step response experiment as in Figure 3.10.

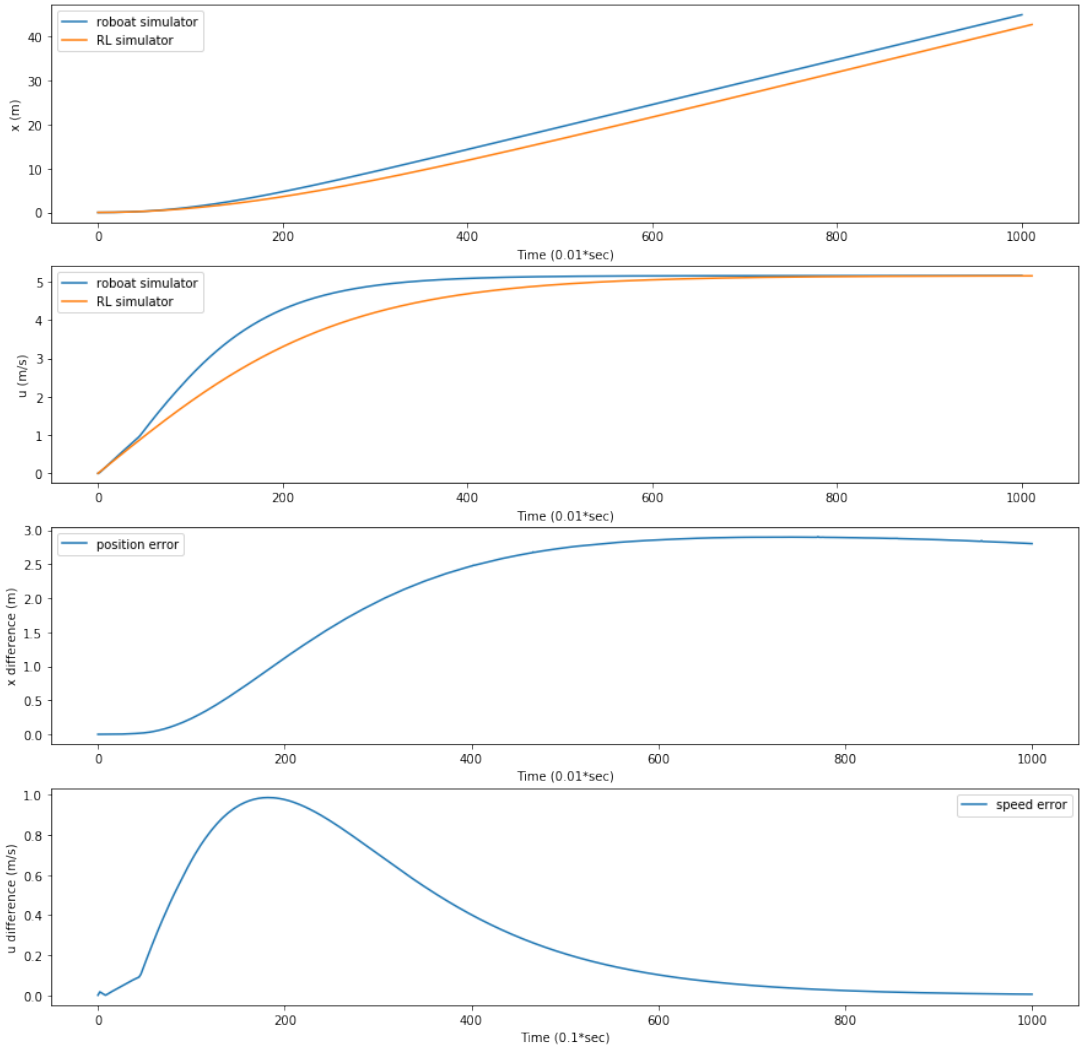


Figure 3.13: Visualization of the payload uncertainty effect. X and u step response, followed by the difference between the two.

In Figure 3.13, the vessels 3 DOF mass matrix was changed from the original values of 1169, 1450, and 3800, to 1600, 2000, and 5000 respectively. This is approximately the behavior of the system with six people on board. As can be observed from the graphs, this induces a significant error in the vessel's position, with the final value of around 3m. Moreover, the extra weight induces slower acceleration resulting in up to 1m/s speed error. Generally, increased payload changes the dynamics of the system affecting the acceleration resulting from the thrusters. The larger the payload, the lower the acceleration. A residual effect of the increased payload would also be the increased drag of the vessel and changed thruster dynamics, however, the magnitude of these effects is much lower.

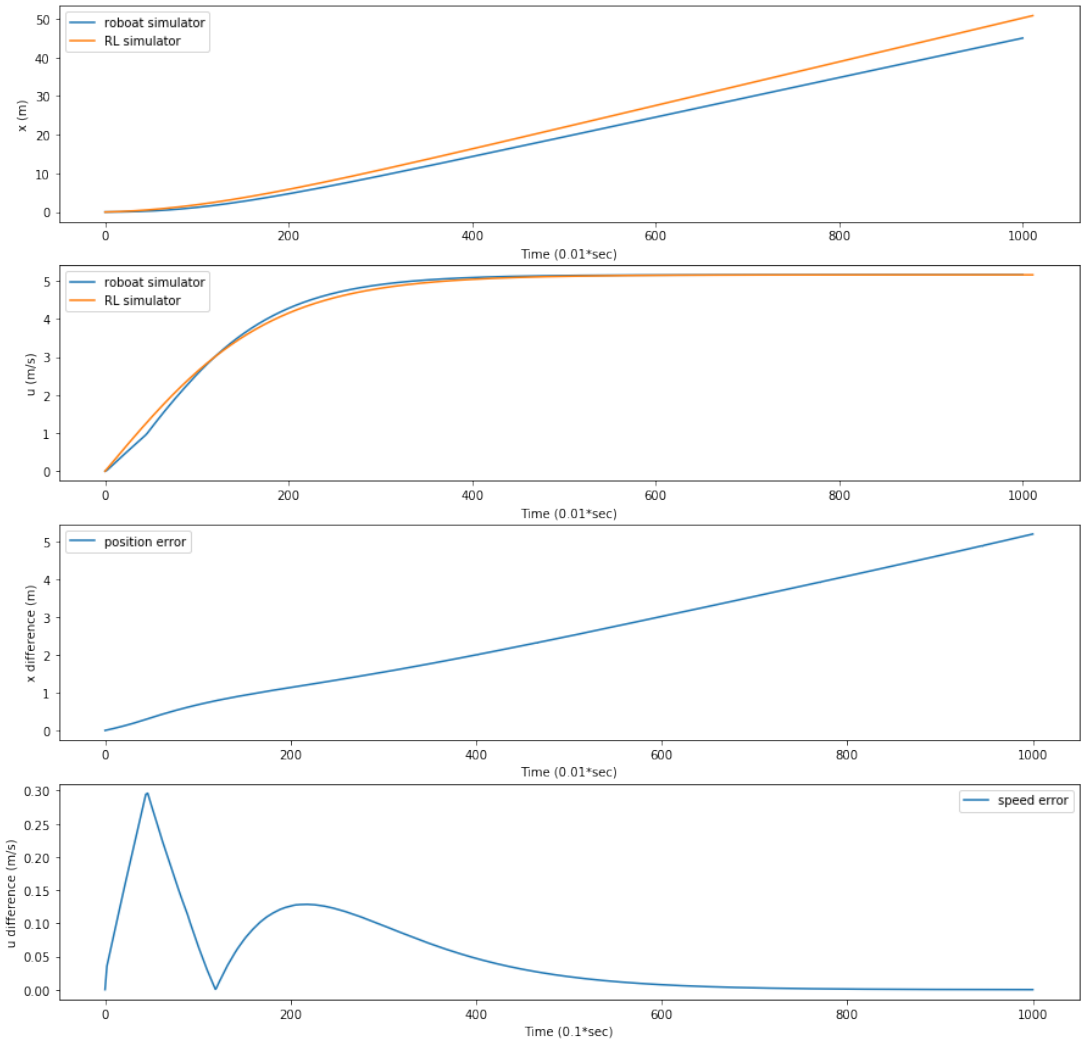


Figure 3.14: Visualization of the current disturbance effect. X and u step response, followed by the difference between the two.

In Figure 3.14 a current is acting on the vessel head on at a speed of 0.5 m/s. This induces a constant drift of the body of water resulting in a steadily increasing position error. It

should be noted that u represents the body frame velocity in which the current induced error is not visible, looking similar to behavior without the current disturbance. However, in plotting the global position x, it can clearly be seen.

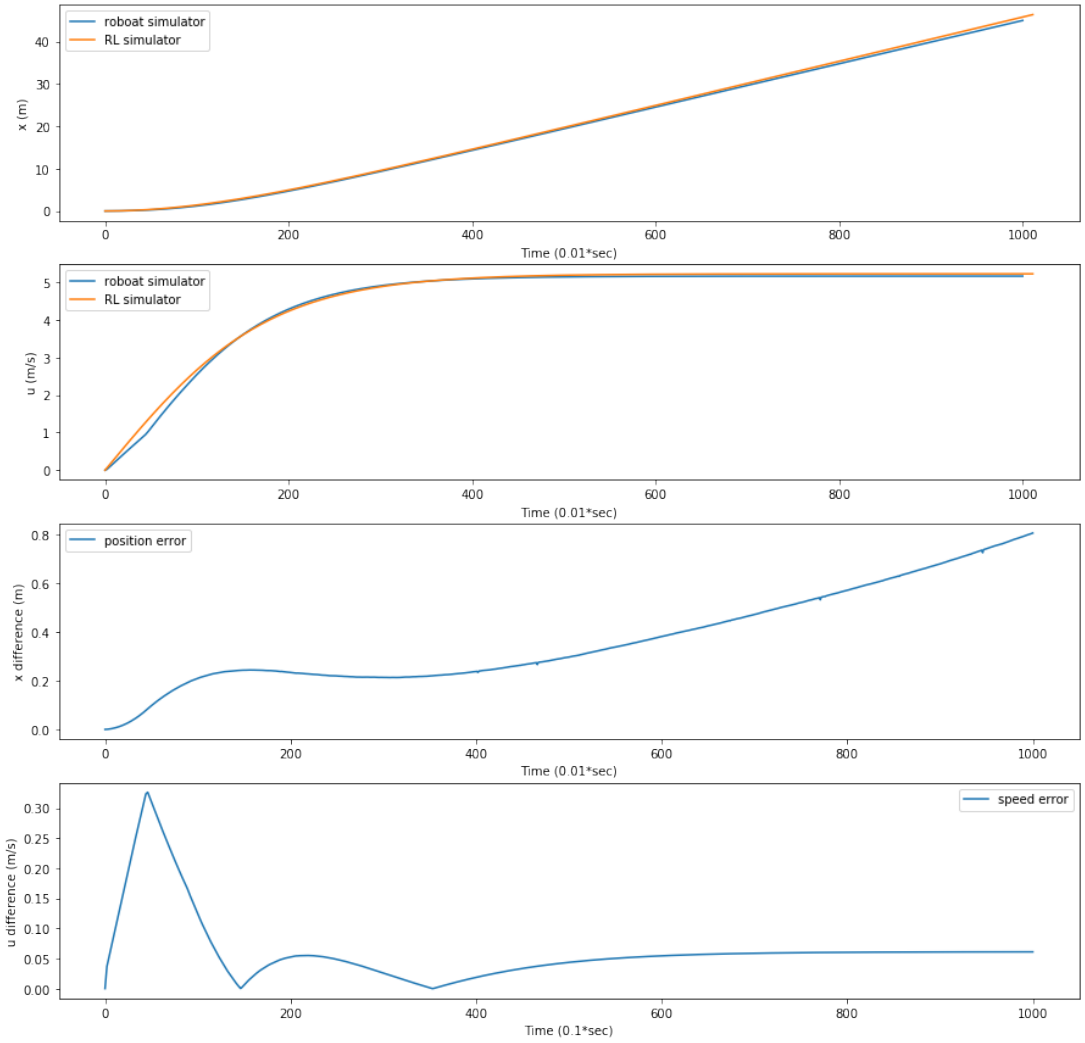


Figure 3.15: Visualization of the wind disturbance effect. X and u step response, followed by the difference between the two.

Finally, in Figure 3.15 the effect of wind can be observed, blowing head on at a speed of 10 m/s (approximately equal to the maximum wind speed of 5 on the Beaufort scale, measured on the real system, in the Amsterdam canals). The wind acts as a slowly varying force on the system, and its effect can be approximated as such with a randomly initialized constant strength. Depending on the principal wind direction, the resulting force acting on the system is changed because of the wind force. This changes the acceleration of the system and properties of both position and velocity plots. Upon reaching the terminal

state, a small speed error is visible in the plots, resulting in a slow increase of the position error. The effect of wind is relatively small compared to payload and current.

These Figures clearly demonstrate the limited ability of the current simulator. Furthermore, even though the newly developed simulator makes the inclusion of the most significant effects possible, due to the fact that direct measurement of their parameters is infeasible, a traditional controller cannot be easily upgraded to take these effects into account. Therefore, the hypothesis of this study is that a learning approach can be used to solve the problem.

4 | Results

This chapter presents the findings of the study through a comparison of the current and the novel approach to trajectory tracking. The comparison is done between two algorithms: the original NMPC and the reinforcement learning algorithm.

The results of the learning process will be shown in the following manner. Firstly, the performance of the NMPC and the reinforcement learning algorithm will be compared in simulation without the presence of uncertainties and disturbances. Following this, the same will be done with the inclusion of uncertainties and disturbances.

The comparison is done through the use of multiple metrics. Firstly, the tracking precision of the algorithms is compared with the Euclidian distance between the current position and the desired position on the reference trajectory. The general performance of the controllers is compared with the Root Mean Squared Error (RMSE) of that distance. Secondly, the average power usage of a given algorithm is measured with the Equation 4.1:

$$P_{average} = \sum_{i=1}^{N} \frac{(|f_1^i| + |f_2^i| + |f_3^i| + |f_4^i|) * |u^i|}{N} \tag{4.1}$$

where N is the number of data points, u^i is the surge speed of the vessel at time i and f_j^i is the force command for thruster j at time i.

All of the tests were carried out with sinusoidal trajectories as reference as they entail acceleration, deceleration, rotation, and coupled motion in surge, sway, and yaw degrees of freedom. This reference trajectory is calculated as follows:

$$y_r = A \sin(\frac{2\pi}{T} x_r) \tag{4.2}$$

where A is the amplitude in meters and T is the period in meters. The sine wave is then sampled to get motion of different speeds. The reference trajectory in the following tests has an amplitude of 2 meters, a period of 10 meters, and a speed of 0.6 m/s. A sine wave trajectory with these values provides a challenging tracking task for a controller given the

physical and dynamical properties of the vessel.

Having compared the algorithms in simulation, the same will be done on the real system. Since varying payload being the only disturbance that can be reliably measured, most focus will be given to the testing of its effects. For what concerns the current and wind, given the fact that it is infeasible to precisely measure the magnitude of current and wind, only approximations will be given.

4.1. Trajectory tracking comparison in simulation

As described, this section contains the comparison between the performance of the NMPC and the reinforcement learning algorithm in simulation. The first comparison is without the inclusion of the disturbances and uncertainties, followed by a comparison with their inclusion.

4.1.1. Comparison without uncertainties and disturbances

The tests were set up as follows. The vessel's state was initialized to the values of the first state in the reference trajectory: x and y position to 0, yaw to the initial heading of the sine, the surge, sway, and rotation speeds to 0. Afterwards, the system's behavior was simulated for 500 steps of 0.1 s or 50 s in total. The following Figures contain the results of these tests.

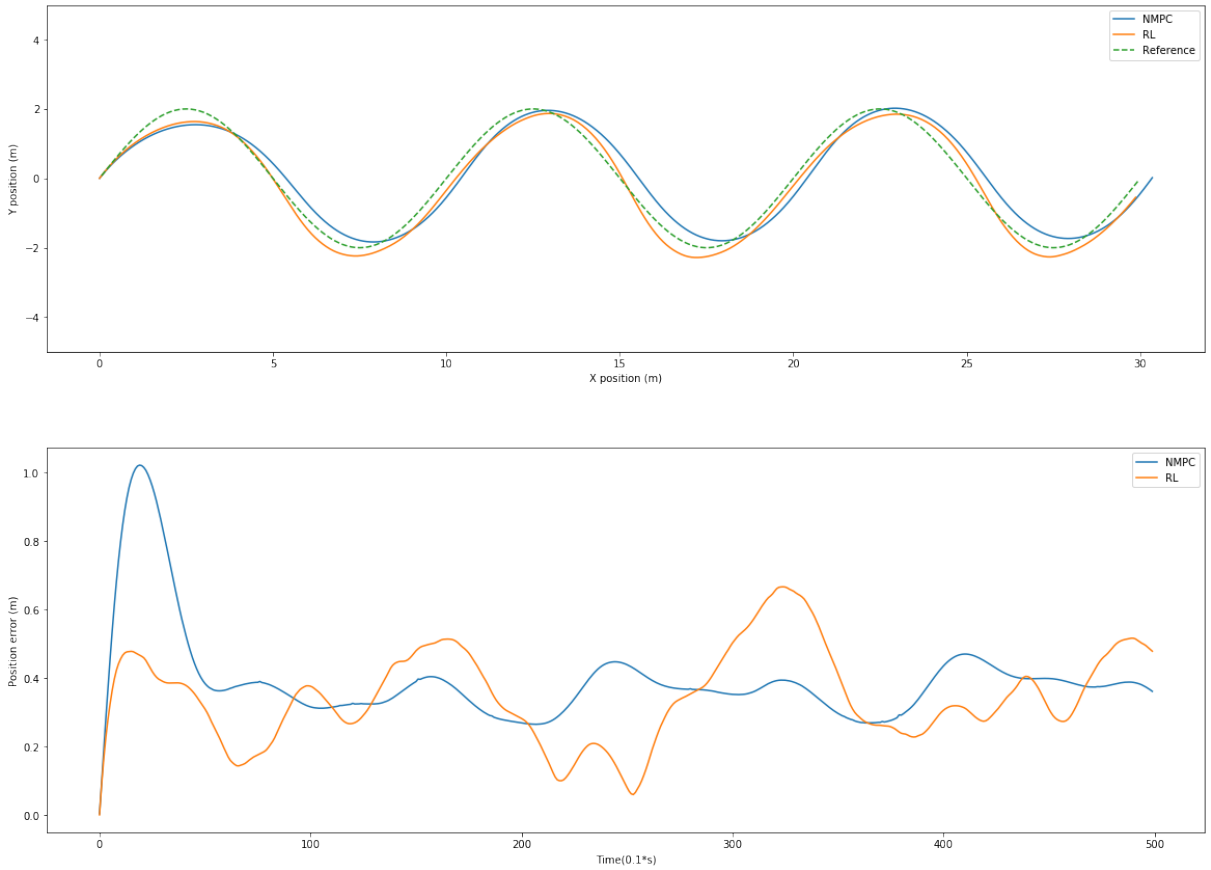


Figure 4.1: Comparison between the NMPC and RL algorithm trajectory tracking performance. Trajectories of NMPC, RL, and reference sine on the top graph. Tracking error of NMPC and RL on the bottom graph.

Figure 4.1 contains two graphs: the graph on the top provides the comparison between the performance in the x-y plane, and the graph on the bottom provides the comparison of the magnitude of the tracking error for each timestep. It is important to note that in the Roboat simulator, some noise with mean 0 N and standard deviation 1.5N is added to the force values upon initialization, however, the effect of this is negligible in the NMPC tests.

It is clear from the x-y graph that both controllers are capable of tracking the reference trajectory successfully. The average position error of the NMPC during the episode is 0.3018 m and the average position error of the RL algorithm is 0.2836 m. Comparing these values, the tracking error for the RL algorithm is 6.03% smaller compared to that of the NMPC.

Generally, it can be said that the RL algorithm behaves in a more reactive manner with regards to the increase of the tracking error, leading to a less sine-like trajectory at times almost perfectly aligned with the reference. On the other hand, the NMPC produces a

trajectory closer in shape to a sine wave with a more steady tracking error. This behavior can be explained with the shorter future horizon available to the RL algorithm.

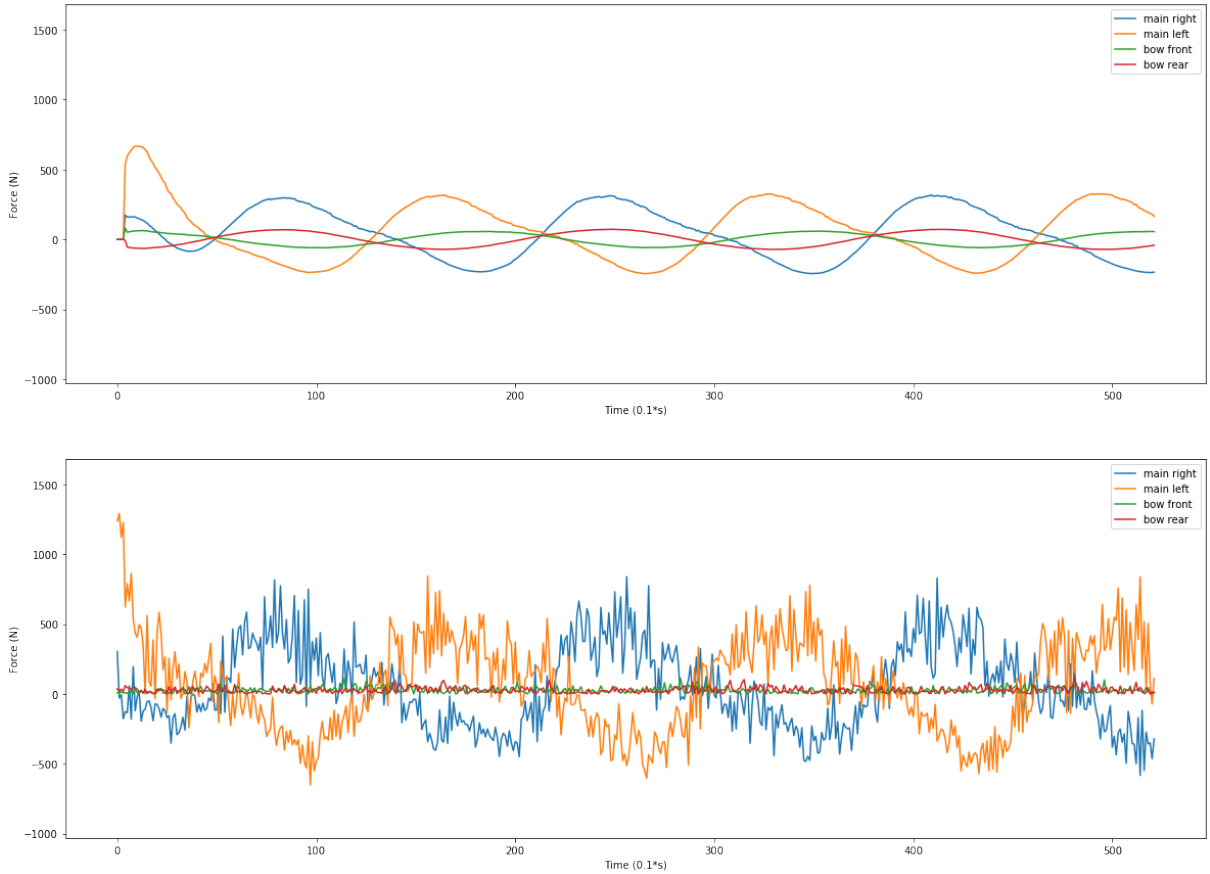


Figure 4.2: Comparison between the NMPC and RL algorithm trajectory tracking performance. Force allocation of NMPC (top) and RL (bottom) for tracking the sinewave trajectory.

The control forces of the algorithms that resulted in such behavior are presented in Figure 4.2. The graph on top contains the forces allocated by the NMPC, whereas the bottom graph contains the same for the RL algorithm.
The average power usage of the NMPC controller calculated with 4.1 is 323.7134 W and that of the RL algorithm is 491.3406 W. Therefore, on average the RL algorithm uses up 51.78% more power than the NMPC controller.
Although similar in the metrics, the controllers differ significantly in the shape of the actuation curves. While the NMPC exhibits smooth and continuous commands, those of the RL algorithm are very volatile and discontinuous. The noisy behavior is an artifact of the random initialization of weights in the training process and the availability of immediate effect of a force command on the system. This however, comes at a cost of higher

power usage. As can be seen later, this does not translate to the real world, where the response is slower and less reliable.
As expected, besides the volatile nature of RL, the actuation of the controllers is fairly similar with NMPC utilizing bow thrusters for rotation of the vessel to a higher degree.

4.1.2. Comparison with uncertainties and disturbances

Having compared the performance of the controllers without the effect of uncertainties and disturbances, the same can be done with their inclusion. Since varying payload produces the largest tracking error in the real world tests done on the platform, it will be considered first.

The comparison is done in exactly the same manner as before, the only significant difference being the addition of payload of around 50% of the vessel's own weight. This is simulated by changing the diagonal values of the mass matrix according to the vectors in 4.3, from the old values on the left to the new values on the right.

$$\begin{bmatrix} m_{11} \\ m_{22} \\ m_{33} \end{bmatrix} = \begin{bmatrix} 1169kg \\ 1450kg \\ 3800kg \end{bmatrix} \qquad \begin{bmatrix} m_{11} \\ m_{22} \\ m_{33} \end{bmatrix} = \begin{bmatrix} 1753.5kg \\ 2175kg \\ 5700kg \end{bmatrix} \tag{4.3}$$

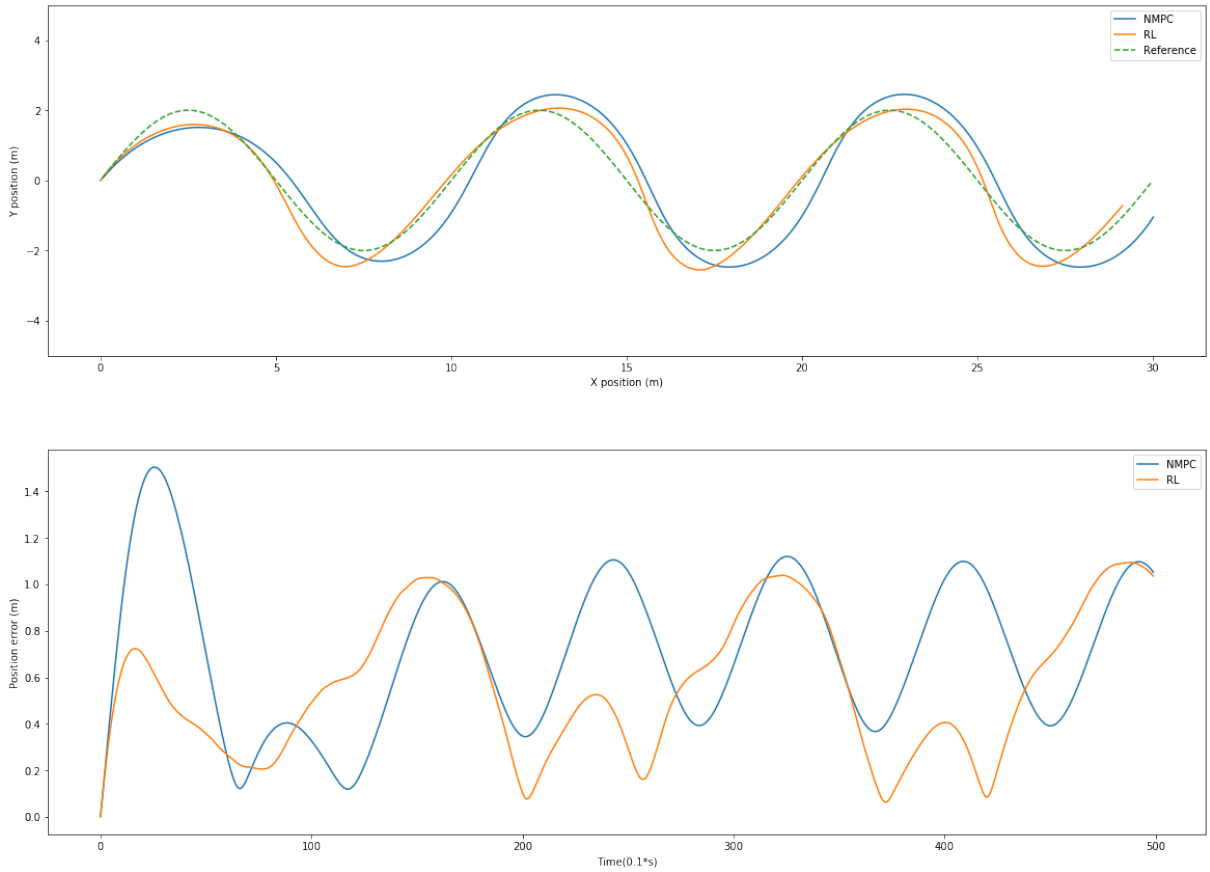


Figure 4.3: Comparison between the NMPC and RL algorithm trajectory tracking performance with payload equal to 50% of vessel's mass. Trajectories of NMPC, RL, and reference sine on the top graph. Tracking error of NMPC and RL on the bottom graph.

The graphs visualizing the explained tests are presented in Figure 4.3. Once again, both controllers exhibit relatively good performance in tracking the desired trajectory. However, the effect of the added payload is clearly visible as the tracking is less tight than in the previous case. This is most noticeable in the crests of the sines when achieving maximum amplitude of the sine. In these runs, the average position error of the NMPC during the episode increased to 0.6979 m, whereas the average position error of the RL algorithm increased to 0.5386 m. Comparing these values, the tracking error for the RL algorithm is now 22.83% smaller compared to that of the NMPC.
Just like before, the RL algorithm is more reactive and NMPC remains steady. However, for the NMPC the added weight induces lag and overshoot which it fails to cancel. On the other hand, RL algorithm's reactive behavior lets it cancel out the effects more successfully leading to a lower average tracking error.

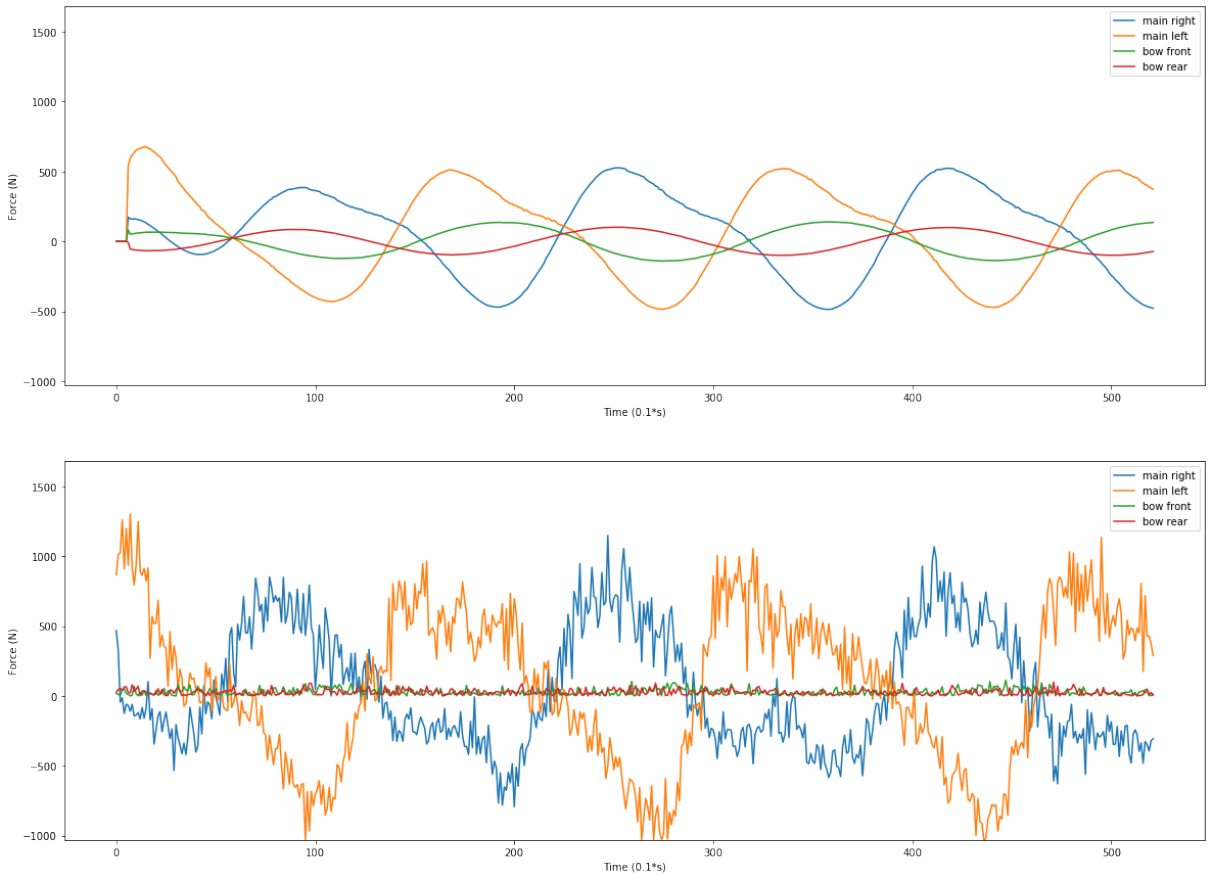


Figure 4.4: Comparison between the NMPC and RL algorithm trajectory tracking performance with payload equal to 50% of vessel's mass. Force allocation of NMPC (top) and RL (bottom) for tracking the sinewave trajectory.

The forces allocated by the controllers to achieve such behavior are presented in Figure 4.4 in the same manner as with the previous test.
The average power usage of the NMPC calculated with 4.1 with the added weight has risen to 559.3913 W and the average power usage of the RL algorithm to 742.1449 W. This means that the RL algorithm used 32.67% less power than the NMPC algorithm in this case.
As it was in the previous case, the RL algorithm retains its volatile actuation nature and the NMPC continuous. However, the RL algorithm actuation curve does not oscillate as severely as before. This can be explained with the slower response of the system caused by the excess weight leading to slower changes in the state values based on which the RL algorithm chooses its control network output.

Now we can turn to the effect of the second most significant disturbance for the Roboat vessel: current. Once again, the comparison is carried out in the same manner as the

original one with the addition of a current disturbance moving the vessel with a speed of 0.5 m/s in the direction of the negative y axis, perpendicular to the general movement direction. This direction was found to be the most difficult to deal with for the controllers. As previously stated, the maximum current speed is taken from online data, although the actual values in Amsterdam city canals are much lower.

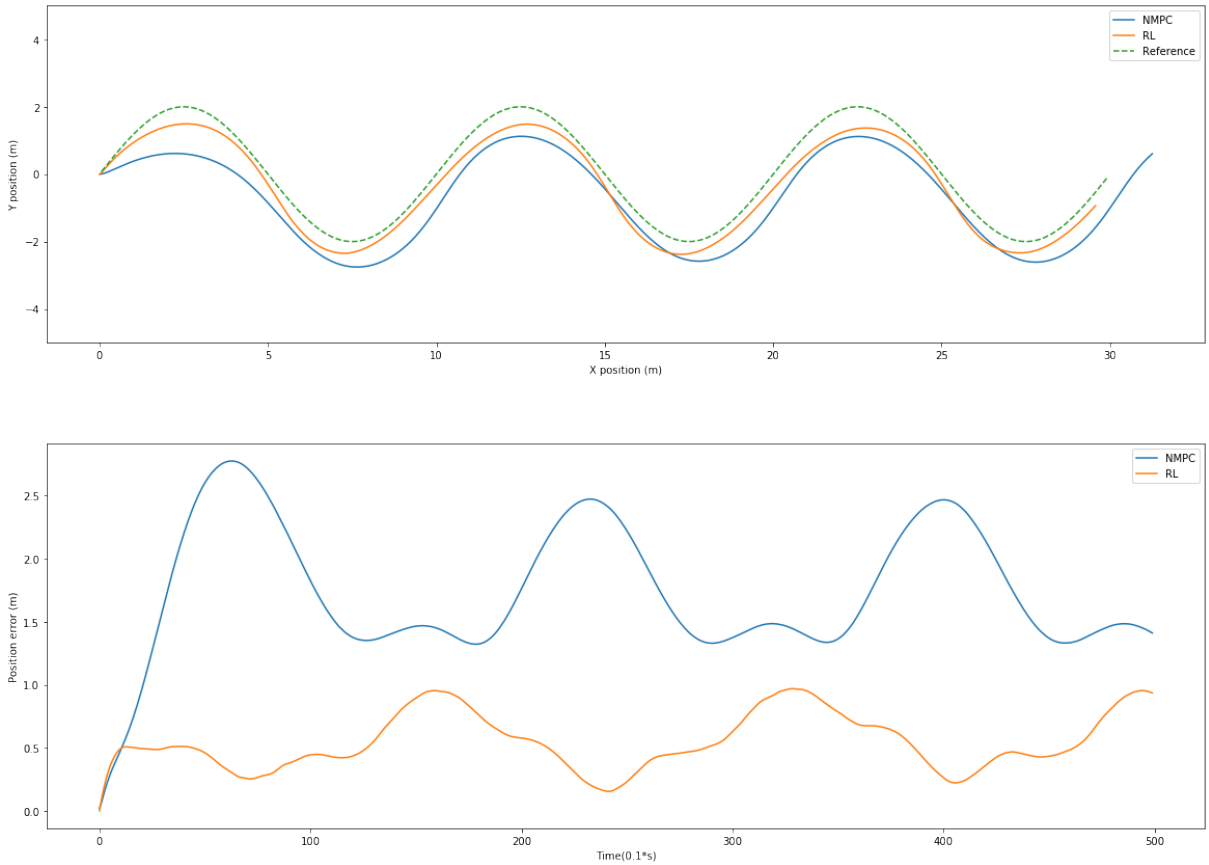


Figure 4.5: Comparison between the NMPC and RL algorithm trajectory tracking performance with a current of 0.5 m/s acting on the system. Trajectories of NMPC, RL, and reference sine on the top graph. Tracking error of NMPC and RL on the bottom graph.

The data collected for the comparison with the current disturbance is presented in Figure 4.5 in the same manner as the previous graphs. Once more, the initial state between the two simulations was almost identical, however, the RL algorithm catches up with the sine wave more aggressively.
The effect of the current disturbance is clearly visible in both cases, pushing the boat away from the reference sine. The NMPC tracking suffers much more from the disturbance since it does not output adequate forces to compensate for the current until the reference gets farther away, resulting in an initial accumulation of the position error. Once an equilib-

rium is found the error stabilizes. The RL algorithm, on the other hand, reacts quickly to the increase in position error which stabilizes around a lower value. The average position error of the NMPC during the episode was 1.6810 m, whereas for the RL algorithm it was 0.5803. The error for the RL algorithm was, therefore, 65.48% lower than that of the NMPC. The bad performance of the NMPC is expected as at every timestep it wrongly predicts the future trajectory as if there was no current causing constant sideways drift.

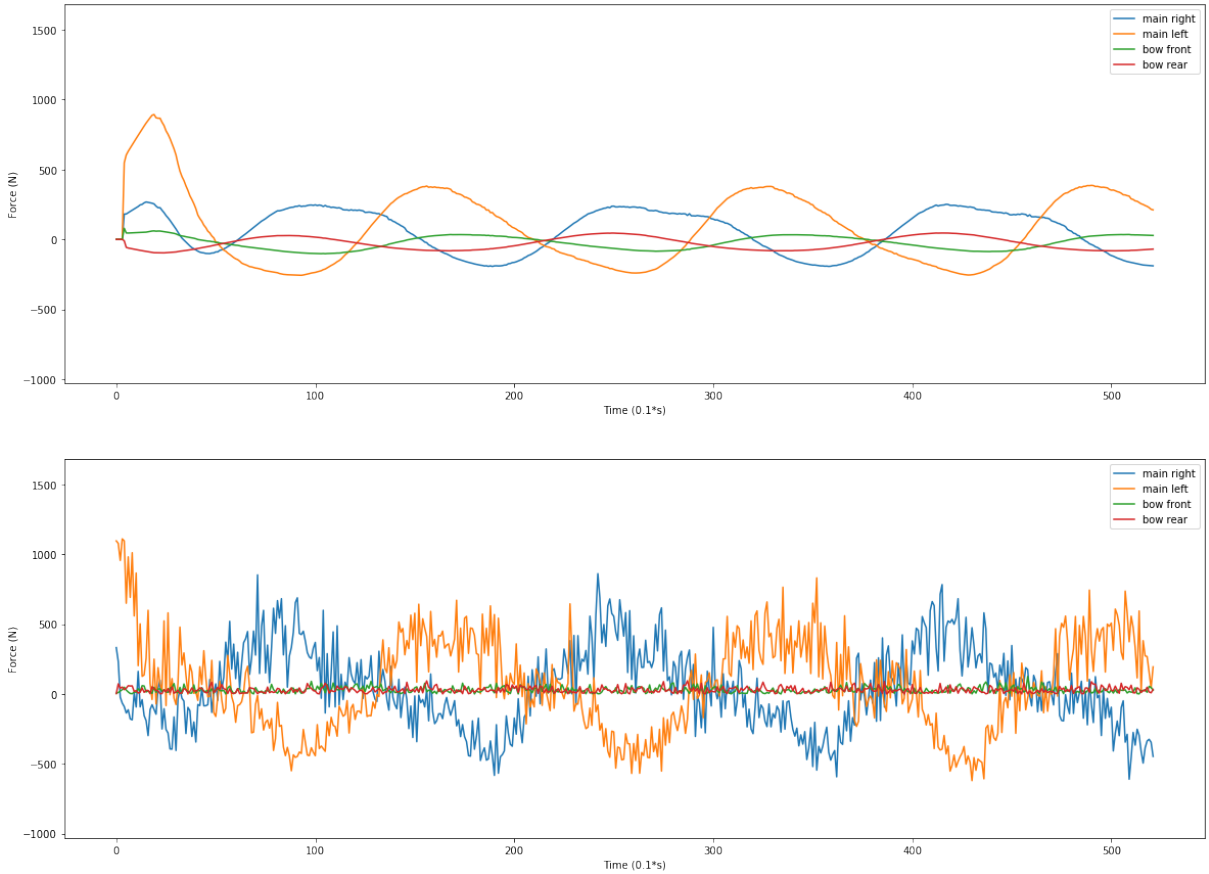


Figure 4.6: Comparison between the NMPC and RL algorithm trajectory tracking performance with a current of 0.5 m/s acting on the system. Force allocation of NMPC (top) and RL (bottom) for tracking the sinewave trajectory.

Just like before, the actuation graphs resulting in this behavior are presented in Figure 4.6. The average power usage of the NMPC in this comparison was 412.3100 W and for the RL algorithm 460.2448 W. Put in other words, the RL algorithm used up 11.63% more power than the NMPC. There are two interesting outtakes from the actuation graphs. Firstly, the initial force output of the RL algorithm is much more aggressive enabling it to catch up with the reference sine wave quicker. The NMPC on the other hand gradually increases the forces as it drifts away from the reference resulting in a more sluggish

response. Secondly, due to this slow initial reaction the NMPC later compensates the current effect throughout the episode with the main thrusters outputting positive force at the same time when moving against the current.

In the end, the effect of wind disturbance on the controller's performance will be studied. As stated in Chapter 1, the wind is approximated as a combination of forces and torques acting on a system. It is defined by its speed and the angle of attack with regards to the vessel. Moreover, the maximum speed of the ever wind measured during real world tests in Amsterdam canals was 5 on the Beaufort scale, or around 9 m/s. Once again, the most difficult task for the controllers was found to be with wind coming from the negative y axis. This was simulated for both controllers and the results are shown in the following Figures.

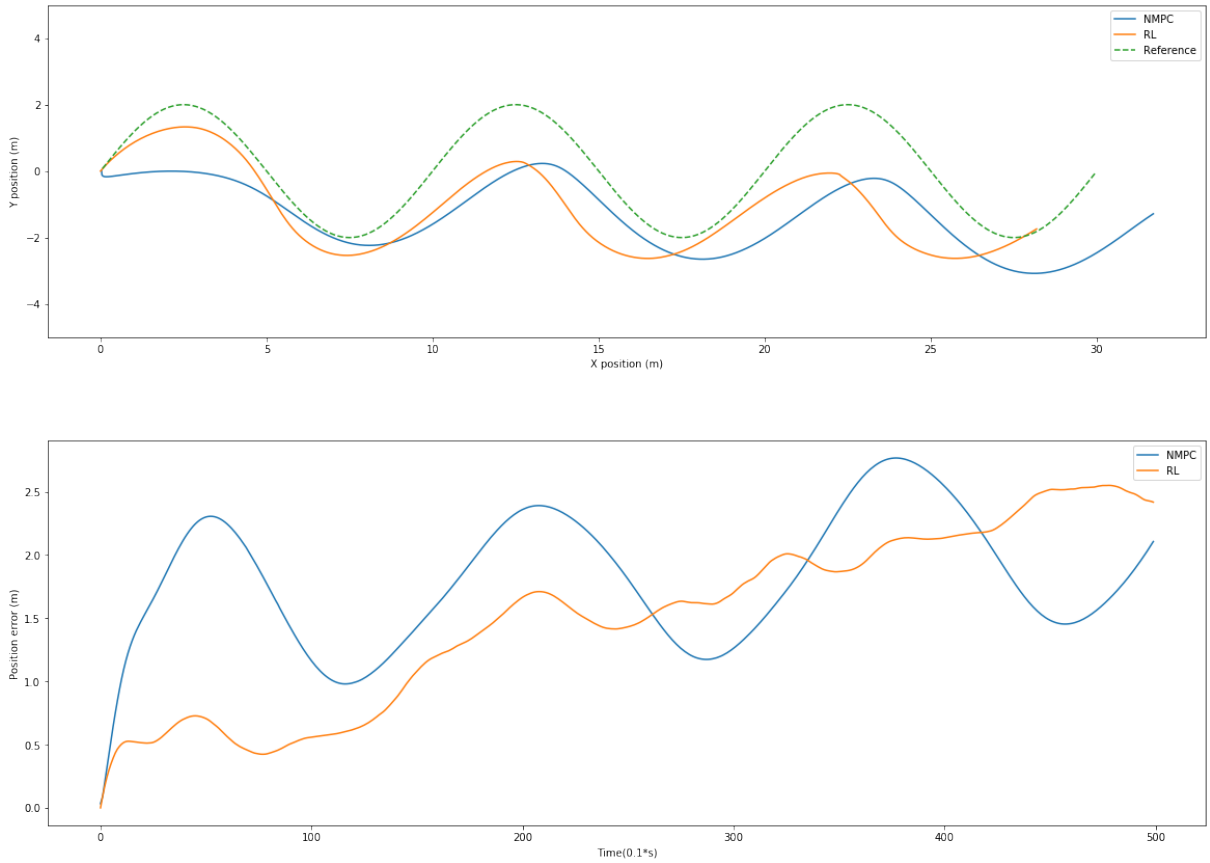


Figure 4.7: Comparison between the NMPC and RL algorithm trajectory tracking performance with a wind of speed 9 m/s acting on the system. Trajectories of NMPC, RL, and reference sine on the top graph. Tracking error of NMPC and RL on the bottom graph.

The results of the runs with wind disturbance are presented in Figure 4.7 in the same

manner as before. In the beginning, the RL algorithm once again reacts faster to the error buildup than the NMPC. From the position and error graphs, it is obvious that such a wind disturbance has a significant effect on the system.
The average position error throughout the episode for the NMPC was 1.8032 m, and 1.701 m for the RL algorithm. The RL algorithm's error is on average, therefore, 5.69% lower than that of the NMPC. When looking at the position error through time, it can be seen that both controllers fail to cancel the effect of wind with the error increasing as the episode goes on.

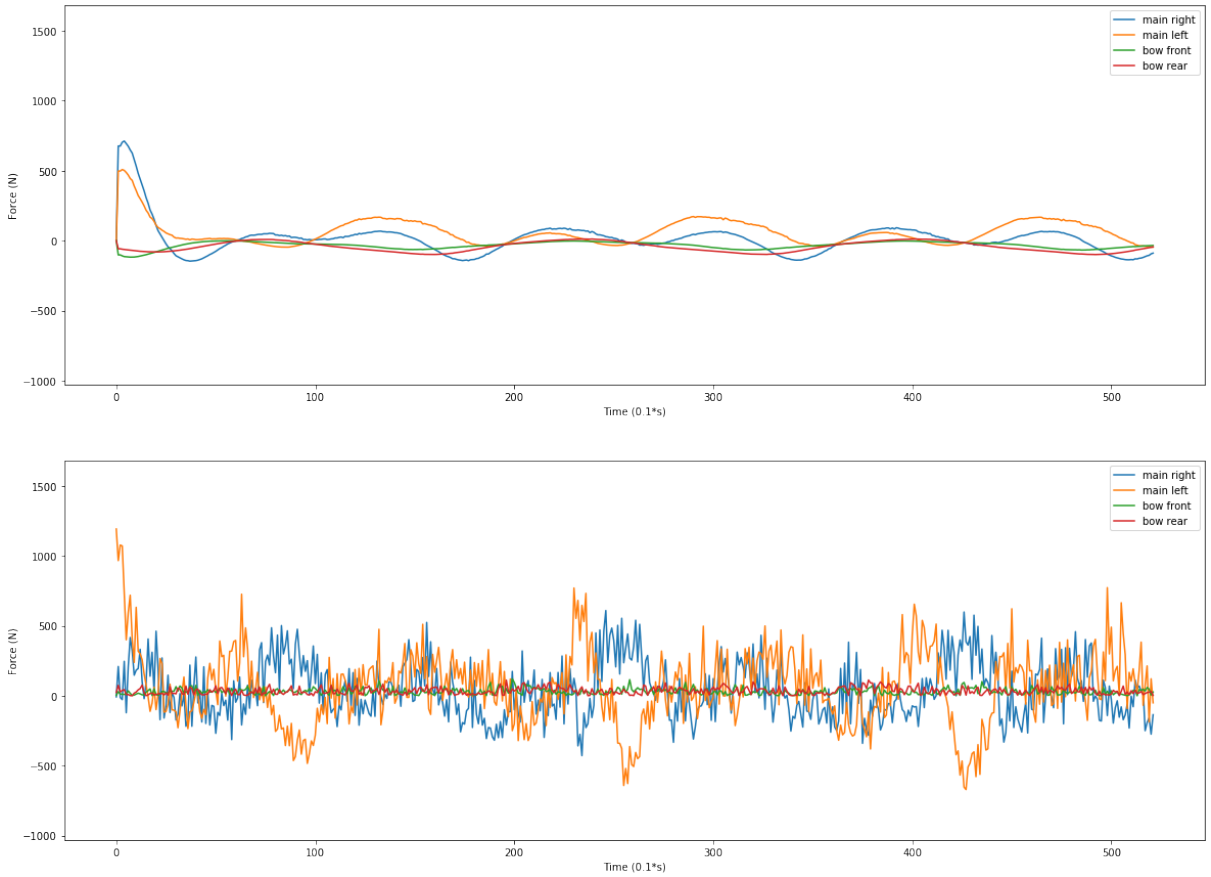


Figure 4.8: Comparison between the NMPC and RL algorithm trajectory tracking performance with a wind of speed 9 m/s acting on the system. Force allocation of NMPC (top) and RL (bottom) for tracking the sinewave trajectory.

The actuation graphs for the wind disturbance comparison can be seen in Figure 4.8 presented like in the previous cases.
In this comparison, the average power usage of the NMPC was 184.7790 W, and for the RL algorithm 300.3304 W. In other words, the RL algorithm used up 62.53% more power than the NMPC. Looking at the comparison, the RL algorithm again gives a strong

initial force to reach the reference, this time closer to the maximum thrust value of 1680 N for the main thrusters. Furthermore, the timeline can be divided into sections in which the controllers try to move forward, with both main thrusters giving positive force, and sections in which they turn, with the main thrusters working in opposite directions. It is interesting to note that the NMPC uses much less force when turning, letting the wind torque turn it clockwise. The RL algorithm on the other hand struggles to track the reference state and at times tries to turn even with position error prioritizing yaw rather than position.

Table 4.1 provides a comparison between the previously presented experiments in considered metrics in one place. In general, the RL algorithm successfully learned to track trajectories, outperforming the NMPC even in the undisturbed setting in terms of positional error. However, this comes at a significant power usage cost, as the NMPC uses significantly less power to achieve similar tracking performance. Moreover, the transient behaviour of RL is highly undesirable for real world actuators, leading to high wear and shortened life. Furthermore, it assumes that the real system responds to such behaviour accordingly which, as can be seen in the following section, does not hold.

In terms of different disturbances, they all induce more error in their own way. By looking at the values in the table it would seem that current and wind present more of a problem than varying payload. This is true for the extremes, however, varying payload is present in all the use cases of the Roboat platform, whereas the values of current and wind are in general much lower in the Amsterdam canals.

Scenario	**Undisturbed**	**Added payload**	**Current**	**Wind**
Avg. pos. error NMPC (m)	0.3018	0.6979	1.6810	1.8032
Avg. pos. error RL (m)	0.2836	0.5836	0.5803	1.701
Difference between RL and NMPC in %	-6.03%	-22.83%	-65.48%	-5.69%
Avg. power consumption NMPC (W)	323.7134	559.3913	412.3100	184.7790
Avg. power consumption RL (W)	491.3406	742.1449	460.2448	300.3304
Difference between RL and NMPC in %	51.78%	32.67%	11.63%	62.53%

Table 4.1: Comparison between the NMPC and the RL algorithm in the evaluated metrics: average position error and power consumption.

4.2. Trajectory tracking comparison on the real system

After concluding that the RL algorithm successfully performs trajectory tracking in simulation, it was time to test it on the vessel in the real world. For deployment, a new node was created in the ROS framework to enable communication with different parts of the system, such as the local planner for observations and the thrusters for actuation. The node was written in Python and enables the user to load and use the neural network as a controller for the system. As previously mentioned, the framework runs on a Intel NUC system with the control nodes working at 10 Hz just like the NMPC.
Similar to the studies explained in 2.3, when attempting to deploy a low level neural network controller directly on a real system after training only in simulation, a number of problems arise that cause the algorithm to fail. These problems range from unexpected software behavior, sensor noise and latency, to model discrepancies. Furthermore, debugging on the real system is much more time consuming and complex than in simulation. This section introduces the encountered problems and gives solutions to some of them, and hints for future work for the rest. Moreover, the best results, after applying the changes necessary for the algorithm to work in the real world, are presented.

Different planner behavior The first problem was encountered already in deploying the learned algorithm in the Roboat C++ simulator. To ensure a certain structure of the trajectory and avoid exceptions caused by badly formed trajectories, a module had been implemented to work in between the commanded trajectory and the actual future trajectory published. This module enforces the future trajectory does not exceed maximum speed of the vessel by interpolating between neighbor points in the trajectory. This, however, led to a differently structured future trajectory being fed to the controller confusing the learned algorithm. Moreover, the future trajectory would change at every step as result of the interpolation process. To fix this, a different module had to be implemented to override this behavior and get the input closer to what the algorithm learned on in the reinforcement learning simulation.

Measurement noise The second significant cause of bad performance was measurement noise from the sensors. This was only discovered once on the vessel in real life and it stems from the imperfect nature and irregular latency of the measurements. The effect of this is cleaerly visible in the vessel's odometry, which is filtered with an extended Kalman filter from the internal measurement unit (IMU) and the two GPS

units. Given the very high accuracy of the GPSs used, the odometry is calculated from their relative position at 10Hz. In between measurements, the value is filled with IMU data, working at 500Hz. As it was found, however, the GPS's measurement updates often come with a delay between the two, causing the signal to be choppy. This, in turn, due to very high certainty in the GPS's measurement in the covariance matrix, causes the filtered value to be choppy as well. This effect can clearly be seen in Figure 4.9, presenting 1s of vessel's surge motion at around 0.6 m/s in an x and y plot. This excerpt contains around 10 updates from each GPS, each one causing a small jump in the output signal.

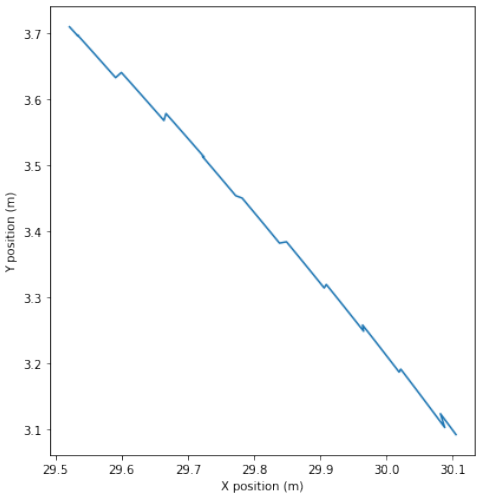


Figure 4.9: x and y plot of 1s of the vessel's odometry signal in forward motion with surge speed of around 0.6 m/s.

This was mitigated by adjusting the values in the covariance matrix to improve smoothness and adding state noise in the reinforcement learning simulator to account for the effect during the training process as well.

Model differences The third and most potent difference discovered between the simulation and the real world system is a combination of physical and thruster model errors. Firstly, the parameters of physical model were found to be slightly different than those used originally in simulation. The reason for this lies in the imperfect feedback from the thrusters, which in turn caused the parameter estimation to be slightly off. As for the thruster behavior in general, it turned out to be significantly different than expected. The thruster manufacturer provides control through controlling the force setpoint, however, the tracking performance is underwhelming. Upon deeper inspection, the thruster response was found to be a compound problem

with a command-to-thruster delay of between 0.5s - 1s, a proprietary PID controller implemented by the manufacturer to track the propeller RPM, and thruster dynamics as explained in 1.2. To reach the desired force, the proprietary software chooses an RPM setpoint for the PID from data collected in a Bullard pull test. This yields realtively good performance when moving at a constant speed, but leads to significant tracking error whenever changing speeds.
Multiple approaches were tried to get the thruster response in the reinforcement learning simulator closer to real life, such as adding random noise to the values, autoregressive and moving average processes, and thruster dynamic modelling. In the end, a combination of a random noise and a moving average with a window of size 5 applied to the actuation vector yielded best results. This made the actuation more robust and the algorithm seems to learn the slower and unreliable nature of the actuation. Furthermore, the model was finally able to function upon deployment of the model in real life. More detailed and precise modelling of the thruster behavior, from a control signal to the actual thrust output could be a very interesting study case in itself.

After implementing the mitigation techniques described above, the algorithm was able to somewhat successfully perform trajectory tracking. The following contains the results obtained in one of the test runs on the real system in Amsterdam, with the reinforcement learning getting a headstart from the NMPC to catch up with the sine, as with the delayed response of the real system it was unable to do so itself. During the experiment, two people sat onboard supervising the process, increasing the mass of the system to around 1330 kg, with very weak wind (around 2 on the Beaufort scale) and negligible current.

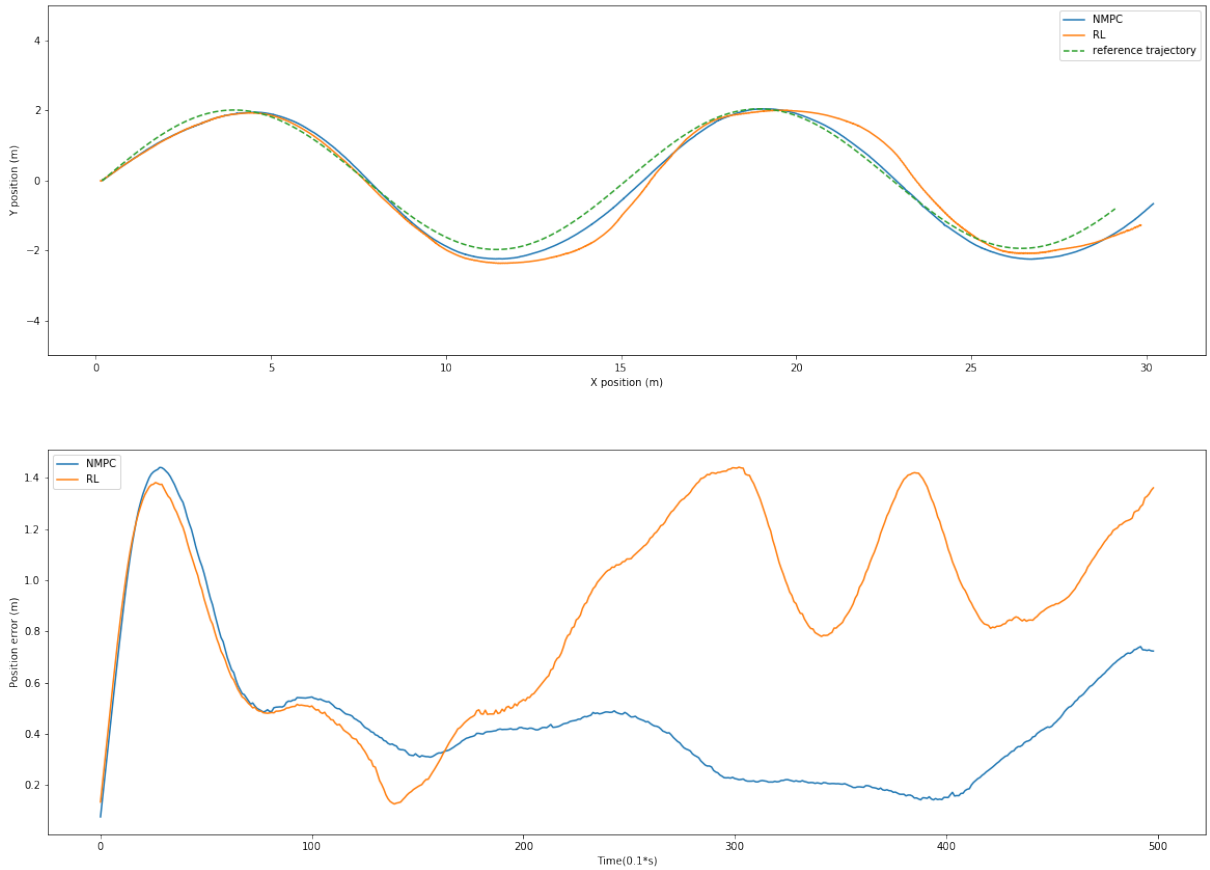


Figure 4.10: Comparison between the NMPC and RL algorithm trajectory tracking performance on the real Roboat vessel. Trajectories of NMPC, RL, and reference sine on the top graph. Tracking error of NMPC and RL on the bottom graph.

In the same manner as before, Figure 4.10 presents the comparison in trajectory tracking performance of the considered algorithms, this time performed on the real system. Both experiments are started with the NMPC in charge and once the reference is caught up with (at around 11s) the RL algorithm takes over.
During the episode the average tracking error for the NMPC was 0.4587 m whereas that of the RL algorithm was 0.8721 m, making the RL algorithm perform 90.10% worse than the NMPC. From the graph on the bottom it is clear that the RL algorithm performes significantly worse than the NMPC. It can also be observed that once RL takes over, the movement of the vessel lags behind the reference turning later than desired. This is a direct result of the delay between the control signal and thrusters reaction, which can also be felt in real life from observing the control vector and actual movement of the vessel.

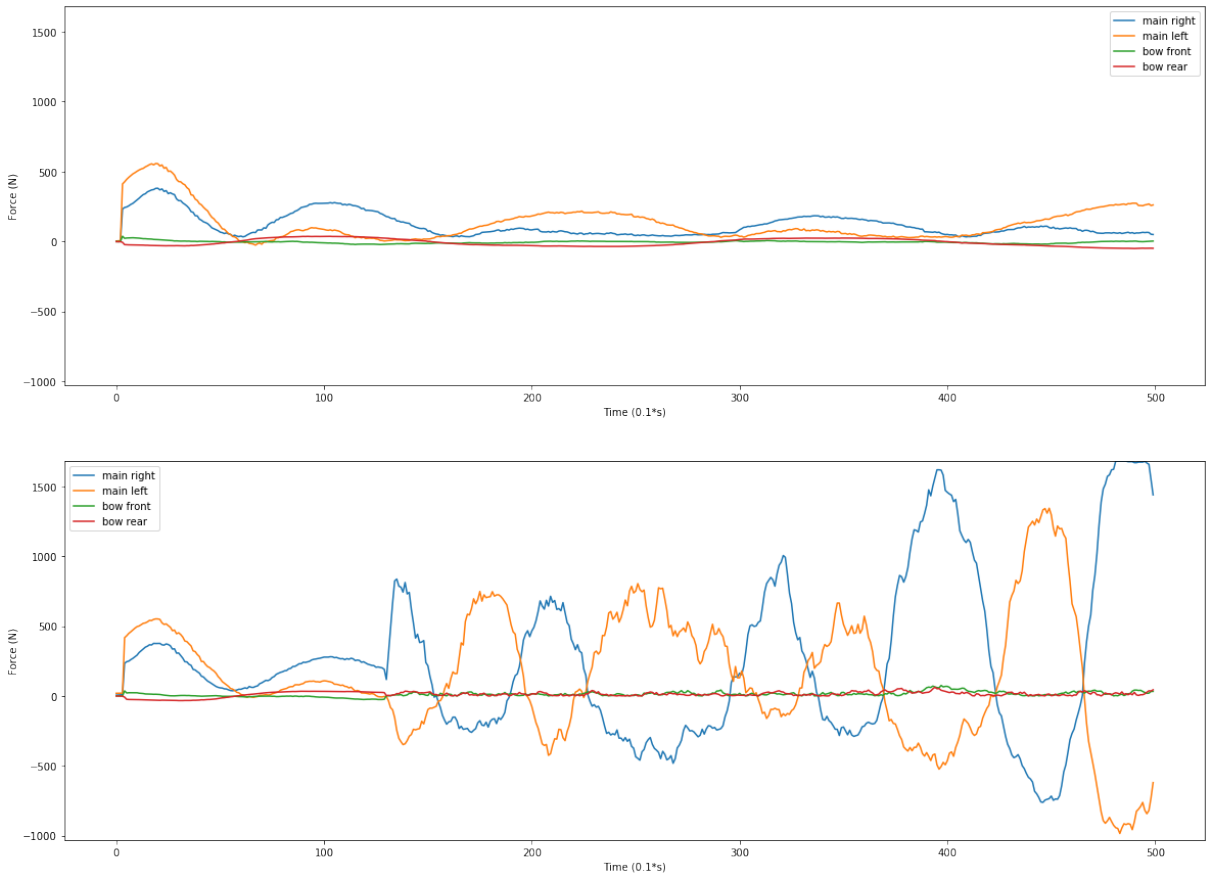


Figure 4.11: Comparison between the NMPC and RL algorithm trajectory tracking performance on the real Roboat vessel. Force allocation of NMPC (top) and RL (bottom) for tracking the sinewave trajectory.

Lastly, Figure 4.11 contains the comparison between the NMPC and RL algorithms in terms of force allocation for the previous episode. Once again, the RL takes over only after the NMPC catches up with the reference trajectory.
During the episode, the average power usage of the NMPC algorithm was 161.4688 W, whereas that of the RL algorithm was 543.6210 W. To put it into perspective, the RL algorithm really struggles, using up 336.67% more power than the NMPC when deployed on the real system. From the Figure, it is jarring to see just how much more the RL algorithm struggles, constantly overreacting to not getting the desired timely actuation and oscillating between strong left and strong right turning. This behavior is highly undesirable as when switching the direction of rotation, the propeller loses function for a short amount of time as it spins up the water flow through it, as explained in 1.2. This further exaggerates the actuation delay to beyond 2s.

In conclusion, all of this makes the real life performance very different from what the

algorithm experiences in the reinforcement learning simulator. It is also amazing to see how little the NMPC performance is impacted by these differences as it works almost the same in the real life as it does in the simulation. This is in large part thanks to the temporal and spatial smoothness of its actuation signal. Because of this, the small errors in odometry and thrust delays do not change the actuation signal significantly. The same cannot be said for the reinforcement learning algorithm, as these small differences break the Markov assumption and lead to significant changes in actuation signals. This, without adequate response, leads to poor performance in real life versus the simulation.
However, novel research, for example the Conditioning for Action Policy Smoothness algorithm (CAPS) from [27], indicates it is possible to improve the temporal and spatial smoothness of RL algorithms which gives hope for future improvements to translate the performance improvement with uncertainties and disturbances from simulation to the real system.

5 | Conclusions and future developments

The hypothesis of this thesis was that a learning-based controller can be trained to perform precise trajectory tracking for an ASV, more specifically the Roboat platform. To that end, a simulator of the system dynamics was developed based on research done on the platform and previous literature, with the modelling of the necessary uncertainties and disturbances. This simulator was then wrapper in a gym environment to facilitate the testing of reinforcement learning algorithms for the vessel and an additional module was created for the generation of various trajectories. This setup is modular in terms of vessel type, its thruster configuration, the choice of reinforcement learning algorithms, and disturbances and uncertainties.
Having done so, an agent was trained in the environment to perform trajectory tracking using the Proximal Policy Optimization algorithm. The obtained controller was then first compared in simulation with the current NMPC approach used on the vessel. Such a controller was able to outperform the NMPC in simulation in terms of trajectory tracking precision, although at a significant cost of high power consumption and erratic control signals. The controller was then deployed on the real system, where its performance deteriorated significantly, as it is not robust to differences between the real world and the simulated system. In the end, a study of problems causing the tracking to fail was performed with some of them being successfully solved and others needing further work.

To conclude, an agent trained through reinforcement learning can be used to perform trajectory tracking successfully. Obtaining such a controller takes as little as half an hour of training in simulation and it learns to deal with uncertainties and disturbances without information about them. However, it is of upmost importance to get the simulated environment as close to the real world as possible, as the described approach is not robust enough. Not doing so can lead to failure when attempting to deploy the model from simulation to real life.
Furthermore, for the learned controller to be useful in everyday function of the Roboat platform, the set of trajectories used for training has to be enriched with trajectories

containing sideways motion, acceleration, and dynamic positioning.

As for future developments of the approach, in the short term much more work should be done on the system model and robustness. Firstly, better parameter estimation is needed, with the inclusion of the Coriolis terms. This also includes the coupled effect of varying payload and payload distribution on the mass, damping, and Coriolis terms. Furthermore, a better model of the thruster dynamics is needed. When it comes to improving actuation signals, in the short term, novel approaches such as the CAPS algorithm from [27] are likely to make the system less sensitive to model discrepancies.
On the other hand in the long term, possible combinations of reinforcement learning with traditional approaches, such as those in [24], [14], and [13] should be considered. Moreover, curriculum learning for improved generalization and live parameter estimation using motion encoders could be exciting and useful new directions for the project.

Finally, this work shows that reinforcement learning might not be robust enough for critical real world applications yet, but recent developments hint it might get there in the near future.

Bibliography

[1] A. Andre do Nascimento. Robust model predictive control for marine vessels. Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2018.

[2] S. Bai, J. Z. Kolter, and V. Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling, 2018. URL `https://arxiv.org/abs/1803.01271`.

[3] J. Balchen, N. Jenssen, E. Mathisen, and S. Saelid. Dynamic positioning of floating vessels based on kalman filtering and optimal control. pages 852 – 864, 01 1981. doi: 10.1109/CDC.1980.271924.

[4] B. Bingham, C. Agüero, M. McCarrin, J. Klamo, J. Malia, K. Allen, T. Lum, M. Rawson, and R. Waqar. Toward maritime robotic simulation in gazebo. In *OCEANS 2019 MTS/IEEE SEATTLE*, pages 1–10, 2019. doi: 10.23919/OCEANS40490.2019.8962724.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016. URL `https://arxiv.org/abs/1606.01540`.

[6] C. Chen, H. Tang, J. Hao, W. Liu, and Z. Meng. Addressing action oscillations through learning policy inertia, 2021. URL `https://arxiv.org/abs/2103.02287`.

[7] M. De Paula and G. G. Acosta. Trajectory tracking algorithm for autonomous vehicles using adaptive reinforcement learning. In *OCEANS 2015 - MTS/IEEE Washington*, pages 1–8, 2015. doi: 10.23919/OCEANS.2015.7401861.

[8] G. Dulac-Arnold, D. Mankowitz, and T. Hester. Challenges of real-world reinforcement learning, 2019. URL `https://arxiv.org/abs/1904.12901`.

[9] J. Figueiredo and R. Rejaili. Deep reinforcement learning algorithms for ship navigation in restricted waters. *Mecatrone*, 3, 12 2018. doi: 10.11606/issn.2526-8260.mecatrone.2018.151953.

[10] T. Fossen. *Guidance and Control of Ocean Vehicles*. Wiley, 1994. ISBN 9780471941132. URL https://books.google.hr/books?id=cwJUAAAAMAAJ.

[11] T. Fossen and M. Paulsen. Adaptive feedback linearization applied to steering of ships. *Modeling, Identification and Control: A Norwegian Research Bulletin*, 14, 03 1995. doi: 10.4173/mic.1993.4.4.

[12] T. I. Fossen. A survey on nonlinear ship control: from theory to practice. *IFAC Proceedings Volumes*, 33(21):1–16, 2000. ISSN 1474-6670. doi: https://doi.org/10.1016/S1474-6670(17)37044-1. URL https://www.sciencedirect.com/science/article/pii/S1474667017370441. 5th IFAC Conference on Manoeuvring and Control of Marine Craft (MCMC 2000), Aalborg, Denmark, 23-25 August 2000.

[13] A. Gonzalez-Garcia, H. Castañeda, and L. Garrido. Usv path-following control based on deep reinforcement learning and adaptive control. In *Global Oceans 2020: Singapore – U.S. Gulf Coast*, pages 1–7, 2020. doi: 10.1109/IEEECONF38699.2020.9389360.

[14] A. Gonzalez-Garcia, D. Barragan-Alcantar, I. Collado-Gonzalez, and L. Garrido. Adaptive dynamic programming and deep reinforcement learning for the control of an unmanned surface vehicle: Experimental results. *Control Engineering Practice*, 111:104807, 2021. ISSN 0967-0661. doi: https://doi.org/10.1016/j.conengprac.2021.104807. URL https://www.sciencedirect.com/science/article/pii/S0967066121000848.

[15] H. Halvorsen, H. Øveraas, O. Landstad, V. Smines, T. Fossen, and T. Johansen. Wave motion compensation in dynamic positioning of small autonomous vessels. *Journal of Marine Science and Technology*, 26:1–20, 09 2020. doi: 10.1007/s00773-020-00765-y.

[16] T. Hester, M. Quinlan, and P. Stone. Generalized model learning for reinforcement learning on a humanoid robot. In *2010 IEEE International Conference on Robotics and Automation*, pages 2369–2374, 2010. doi: 10.1109/ROBOT.2010.5509181.

[17] T. Holzhüter. Lqg approach for the high-precision track control of ships. *IEE Proceedings - Control Theory and Applications*, 144:121–127(6), March 1997. ISSN 1350-2379. URL https://digital-library.theiet.org/content/journals/10.1049/ip-cta_19971032.

[18] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4 (26):eaau5872, 2019.

[19] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*, 40(4-5):698–721, jan 2021. doi: 10.1177/0278364920987859. URL https://doi.org/10.1177%2F0278364920987859.

[20] J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter. Learning quadrupedal locomotion over challenging terrain. *Science Robotics*, 5(47), oct 2020. doi: 10.1126/scirobotics.abc5986. URL https://doi.org/10.1126%2Fscirobotics.abc5986.

[21] Y. Li, H. Li, Z. Li, H. Fang, A. K. Sanyal, Y. Wang, and Q. Qiu. Fast and accurate trajectory tracking for unmanned aerial vehicles based on deep reinforcement learning. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–9, 2019. doi: 10.1109/RTCSA.2019.8864571.

[22] A. B. Martinsen and A. M. Lekkas. Curved path following with deep reinforcement learning: Results from three vessel models. In *OCEANS 2018 MTS/IEEE Charleston*, pages 1–8, 2018. doi: 10.1109/OCEANS.2018.8604829.

[23] A. B. Martinsen and A. M. Lekkas. Straight-path following for underactuated marine vessels using deep reinforcement learning. *IFAC-PapersOnLine*, 51(29):329–334, 2018. ISSN 2405-8963. doi: https://doi.org/10.1016/j.ifacol.2018.09.502. URL https://www.sciencedirect.com/science/article/pii/S2405896318321918. 11th IFAC Conference on Control Applications in Marine Systems, Robotics, and Vehicles CAMS 2018.

[24] A. B. Martinsen, A. Lekkas, S. Gros, J. Glomsrud, and T. Pedersen. Reinforcement learning-based tracking control of usvs in varying operational conditions. *Frontiers in Robotics and AI*, 7, 03 2020. doi: 10.3389/frobt.2020.00032.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013. URL https://arxiv.org/abs/1312.5602.

[26] A. F. Molland, S. R. Turnock, and D. A. Hudson. *Ship Resistance and Propulsion-Second Edition*, pages i–ii. Cambridge University Press, 2 edition, 2017.

[27] S. Mysore, B. Mabsout, R. Mancuso, and K. Saenko. Regularizing action policies for smooth control with reinforcement learning, 2020. URL https://arxiv.org/abs/2012.06644.

[28] S. Nagendra, N. Podila, R. Ugarakhod, and K. George. Comparison of reinforcement

learning algorithms applied to the cart-pole problem. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 26–32, 2017. doi: 10.1109/ICACCI.2017.8125811.

[29] M. Paravisi, D. H. Santos, V. Jorge, G. Heck, L. M. Gonçalves, and A. Amory. Unmanned surface vehicle simulator with realistic environmental disturbances. *Sensors*, 19(5), 2019. ISSN 1424-8220. doi: 10.3390/s19051068. URL `https://www.mdpi.com/1424-8220/19/5/1068`.

[30] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL `http://jmlr.org/papers/v22/20-1364.html`.

[31] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009. ISBN 0136042597.

[32] E. I. Sarda, H. Qu, I. R. Bertaska, and K. D. von Ellenrieder. Station-keeping control of an unmanned surface vehicle exposed to current and wind disturbances. *Ocean Engineering*, 127:305–324, 2016. ISSN 0029-8018. doi: https://doi.org/10.1016/j.oceaneng.2016.09.037. URL `https://www.sciencedirect.com/science/article/pii/S0029801816304206`.

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017. URL `https://arxiv.org/abs/1707.06347`.

[34] R. Tedrake. *Underactuated Robotics*. 2022. URL `http://underactuated.mit.edu`.

[35] R. Wang, J. Lehman, J. Clune, and K. O. Stanley. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions, 2019. URL `https://arxiv.org/abs/1901.01753`.

[36] W. Wang, L. Mateos, S. Park, P. Leoni, B. Gheneti, F. Duarte, C. Ratti, and D. Rus. Design, modeling, and nonlinear model predictive tracking control of a novel autonomous surface vehicle. 05 2018. doi: 10.1109/ICRA.2018.8460632.

[37] W. Wang, B. Gheneti, L. A. Mateos, F. Duarte, C. Ratti, and D. Rus. Roboat: An autonomous surface vehicle for urban waterways. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6340–6347, 2019. doi: 10.1109/IROS40897.2019.8968131.

[38] Y. Wang, J. Tong, T.-Y. Song, and Z.-H. Wan. Unmanned surface vehicle course tracking control based on neural network and deep deterministic policy gradient

algorithm. In *2018 OCEANS - MTS/IEEE Kobe Techno-Oceans (OTO)*, pages 1–5, 2018. doi: 10.1109/OCEANSKOBE.2018.8559329.

[39] J. Woo, C. Yu, and N. Kim. Deep reinforcement learning-based controller for path following of an unmanned surface vehicle. *Ocean Engineering*, 183:155–166, 2019. ISSN 0029-8018. doi: https://doi.org/10.1016/j.oceaneng.2019.04.099. URL `https://www.sciencedirect.com/science/article/pii/S0029801819302203`.

[40] R. Yu, Z. Shi, C. Huang, T. Li, and Q. Ma. Deep reinforcement learning based optimal trajectory tracking control of autonomous underwater vehicle. In *2017 36th Chinese Control Conference (CCC)*, pages 4958–4965, 2017. doi: 10.23919/ChiCC.2017.8028138.

[41] L. Zhang, L. Qiao, J. Chen, and W. Zhang. Neural-network-based reinforcement learning control for path following of underactuated ships. In *2016 35th Chinese Control Conference (CCC)*, pages 5786–5791, 2016. doi: 10.1109/ChiCC.2016.7554262.

[42] S. S. Øvereng, D. T. Nguyen, and G. Hamre. Dynamic positioning using deep reinforcement learning. *Ocean Engineering*, 235:109433, 2021. ISSN 0029-8018. doi: https://doi.org/10.1016/j.oceaneng.2021.109433. URL `https://www.sciencedirect.com/science/article/pii/S0029801821008398`.

List of Figures

List of Tables

Acknowledgements

As a final note, I would like to write a couple of words for the people who meant the most to me throughout the process.

Firstly, I want to thank my supervisor prof. Riccardo Scattolini, for having led me through the entire process and given me strong support in control theory, a field I love, but have not officially studied. He made himself ever available and helpful, timely responding any questions I had.

I would also like to thank the Roboat team and the entire AMS institute, a group of very talented and important, but wonderful people for giving me the opportunity to be a part of an exciting robotics project and to live in Amsterdam for one month, making me feel as one of them. Especially, I would like to thank Jonathan, a robotics engineer from the Roboat team and the supervisor of my work on their side, thank you for being a phenomenal mentor, as well as a great friend.

And finally, most of all I would like to thank my family and Ivana for provided me with endless love and support.