Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Henrik Bjering Strand

Autonomous Docking Control System for the Otter USV: A Machine Learning Approach

Master's thesis in Cybernetics and Robotics

Supervisor: Thor I. Fossen

June 2020

NTNU
Norwegian University of Science and Technology

Henrik Bjering Strand

Autonomous Docking Control System for the Otter USV: A Machine Learning Approach

Master's thesis in Cybernetics and Robotics
Supervisor: Thor I. Fossen
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Problem description

The goal of the project is to develop machine-learning based control algorithm for docking. The algorithms should be simulated and experimentally tested using the Otter USV. The following items should be considered in more detail:

1. Literature study on methods for docking. Appropriate research questions and requirement specifications should be formulated in order to solve the problem.
2. Develop a simulator for testing of control algorithms. The simulator should include the USV dynamics, sensory systems with realistic measurement noise and external disturbances.
3. Develop a control system for autonomous docking (including control allocation) using machine learning. Emphasis should be placed on training.
4. Simulation study and verification/validation of the results.
5. Conclude findings in a report.

Abstract

This thesis aims to use deep reinforcement learning (DRL) to develop an autonomous docking system for an underactuated Unmanned Surface Vessel (USV). The field of autonomous marine vessels is currently a hot topic and has shown the potential to increase safety and reduce the costs of operation. Various solutions for autonomous docking have been presented for fully actuated ships. However, the USV of interest in this thesis is only controlled by two fixed propellers in the rear, making it underactuated. This introduces an interesting control problem since it dramatically reduces the maneuverability and makes dynamic positioning nearly impossible.

To address this, a machine-learning environment with the USVs dynamics was developed. Two DRL algorithms were implemented and compared, namely deep deterministic policy gradient (DDPG) and proximal policy optimization (PPO). Both algorithms were trained with the same reward function, which rewards the reinforcement-learning agent based on how well it reaches the desired position and orientation. It was found that PPO performed better and was therefore used for further development. Realistic measurement noise and ocean current disturbance were added to the environment to prepare the DRL agent for a real-world application. Two models were trained, one with unknown ocean current and one with DVL measured ocean current.

The results of the simulations show that a docking controller is feasible with the use of machine learning. Both the model with and without DVL measured ocean current could handle ocean currents up to 0.2 m/s. However, as the ocean currents reached 0.5 m/s, the control system was dependant on measured ocean currents to achieve a stable docking maneuver. The work of this thesis has presented a docking system that is successful in a simulated environment. However, due to simplifications in this thesis, a solution for real-world applications still needs further work.

Sammendrag

Denne masteroppgaven utforsker muligheten for å bruke dyp forsterkende læring (eng. deep reinforcement learning, DRL) til å utvikle et autonomt dokkingsystem for en underaktuert overflatefarkost. Autonome marine fartøy er for tiden populære innen forskningsmiljøer og har vist potensiale til å både øke sikkerheten og redusere driftskostnader. Det har blitt presentert flere løsning for dokking av fullaktuerte overflatefarkoster. Problemet for dette prosjektet er at farkosten kun er drevet av to fastmonterte propeller i akter, noe som gjør den underakturert. Dette introduserer et interessant kontrolproblem siden dette dramatisk reduserer manøvrerbarheten og gjør dynamisk posisjonering nærmest umulig.

For å løse dette ble et maskinlæringsmiljø som inneholder farkostens dynamikk utviklet. To DRL-algoritmer ble implementert og sammenlignet; deep deterministic policy gradient (DDPG) og proximal policy optimization (PPO). Begge algoritmene ble trent med den samme belønningsfunksjonen, en funksjon som belønner DRL-agenten etter hvor godt den tilfredstiller kravene for posisjon og orientering. Resultatene viste at PPO utførte dokkingen bedre og ble dermed brukt til videre forsøk. Realistisk målestøy og havstrømmer ble lagt til i maskinlæringsmiljøet for å forberede DRL-agenten for testing på en virkelig farkost. To modeller ble trent, en med ukjente havstrømmer og en med DVL-målte havstrømmer.

Simuleringsresultatene viste at det er mulig å utvikle en dokkingkontroller ved å bruke maskinlæring. Begge modellene klarte å håndtere havstrømmer opp til 0.2 m/s, men etterhvert som havstrømmene nådde 0.5 m/s var dokkingkontrolleren avhengig av målinger av havstrømmene for å klare å dokke. Arbeidet i denne masteroppgaven har presentert en dokkingkontroller som er vellykket i et simulert mijø, men på grunn av forenklinger i oppgaven er det nødvendig med mer utvikling før det kan brukes på en virkelig farkost.

Preface

This thesis is submitted as a requirement for the master's thesis TTK4900 at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology in Trondheim, the spring of 2020.

I wish to thank my supervisor, Professor Thor I. Fossen, for guidance and feedback throughout the work of this thesis. I would also like to thank my co-supervisor, Pål H. Mathisen, for all his help and advice.

This thesis is a continuation of the specialization project submitted in the fall of 2019. Parts of chapters 1-4 from that are therefore included here, with some modifications. These chapters include a presentation of the Otter USV, the equations of motions for the vessel, and background material on machine learning. The USV dynamics in chapter 2 was developed in joint work with Per Gunnar Berg Torvund, and I would like to thank him for this cooperation. By this, the master thesis could focus on improving the machine learning environment and train the machine-learning model for a real-world application.

At last, I would like to thank my girlfriend and family for their support.

Henrik Bjering Strand

Trondheim, June 2020

Contents

List of Figures

List of Tables

Acronyms

AHRS Attitude and Heading Reference System. 20

AI Artificial Intelligence. 5, 43

ANN Artificial Neural Network. 2, 3, 28, 29, 31

CAD Computer-Aided Design. 4

CAS Continuous Action Space. 27

CF Center of Force. 15

CG Center of Gravity. 12, 13

CO Center of Origin. 13, 15

DAS Discrete Action Space. 27

DDPG Deep Deterministic Policy Gradient. 2, 3, 33–35, 38, 39, 61, 80, 99

DOF Degree of Freedom. 11, 12

DQN Deep Q-Network. 3, 31, 32

DRL Deep Reinforcement Learning. 2, 3, 31, 38

DVL Doppler Velocity Log. 21, 64, 86

FCNN Fully Connected Neural Network. 28

GAE Generalized Advantage Estimation. 36

GNSS Global Navigation Satellite Systems. 19, 20

GPU Graphical Processing Unit. 42

IMU Inertial Measurement Unit. 20

MDP Marcov Desicion Process. 24, 25

PPO Proximal Policy Optimization. 2, 3, 35–39, 62, 80, 85, 99

ReLU Rectified Linear Unit. 31

RL Reinforcement Learning. 2, 23–26, 71, 81, 86

RTK Real Time Kinematic. 19

SGD Stochastic Gradient Descent. 30

SSA Smallest Signed Angle. 54

TD Temporal Difference. 32, 36

TRPO Trust Region Policy Optimization. 36

USV Unmanned Surface Vehicle. 1, 5, 6, 10, 12, 49, 51, 53

VO Visual Odometry. 21

Symbols

A	Action-space	
$J(\theta)$	Loss function of parameter vector θ	
$Q(s, a)$	Action-value function	
$R(s, a)$	Reward function given by state s and action a	
S	State-space	
$V(s)$	State-value function	
V_c	Current velocity	$[m/s]$
α	Learning rate	
β_c	Crab angle of current	$[rad]$
β	Crab angle of craft	$[rad]$
χ	Course	$[rad]$
ϕ	Roll angle	$[rad]$
ψ	Yaw angle	$[rad]$
τ	Control force	$[N]$
θ	Pitch angle	$[rad]$
m	Mass of Otter	$[kg]$
n_i	Propeller shaft speed (input)	$[rad/s]$
n	Episode step number	
p	Roll velocity	$[rad/s]$
q	Pitch velocity	$[rad/s]$
r	Yaw velocity	$[rad/s]$
u	Surge velocity	$[m/s]$
v	Sway velocity	$[m/s]$
w	Heave velocity	$[m/s]$
x	Position in x direction	$[m]$
y	Position in y direction	$[m]$
z	Position in z direction	$[m]$

Chapter 1

Introduction

This introductory chapter will present the background and motivation for this thesis. The USV used for the project will be introduced in addition to an overview of the control system implemented. The assumptions and requirement specifications determined in order to set the scope of the project are presented in addition to the contributions produced by the results.

1.1 Background

The use of multi-purpose Unmanned Surface Vehicles (USVs) is increasing and showing promising results within autonomous systems (Kongsberg (2020), ECA-Group (2018)). The capability to have flexible payloads and a diverse sensory package makes a USV suitable for multiple applications. Some being seabed mapping, maintenance and inspection of offshore applications, transportation, and military use. The ability to execute missions without the need of humans present makes the USV able to operate in harsh environments, which otherwise would not be possible. In addition to removing the human error and reducing the costs of a crew, the use of autonomous vessels will also improve fuel efficiency. A study done by Kretschmann et al. (2017) showed that replacing conventional bulk carriers with autonomous vessels could contribute to reducing the emissions from shipping. The EU has decided that 30% of today's shipping by road is to be moved to the sea and railroads by 2030 (Trondheim Havn (2019)). Considering a report from TransNav (2015), showing that 60% of accidents at sea is due to human errors, autonomous marine systems are an important field of research. Even though the development of such a system is considered expensive, the long-term reward of an autonomous system can be evaluated to be high.

Docking is the procedure of pulling the vessel up to a dock in a safe and controlled way such that the cargo can be loaded or unloaded. (Van Isle Marina (2019)). The most critical aspects of docking are summarized in Murdoch et al. (2012) as low speed, a controlled approach, and thoroughly planning in regards to environmental disturbances and obstacles. A docking maneuver is considered a complex operation that requires an experienced captain. The captain has to be familiar with the ship's dynamics and be able to predict the ship's behavior based on the surrounding wind and ocean currents.

Docking is often executed in a marina where space is limited, and the speed has to be low. This increases the complexity of the operation since the thrusters' utilization becomes limited, meaning that the input from the captain has to be well planned. A study from Murdoch et al. (2012) shows that 70% of insurance claims involving dock damage is due to bad ship handling and simple mistakes made by individuals. Therefore, the development of autonomous docking algorithms could be considered a potential solution for reducing these mistakes.

Within machine learning, the field of Deep Reinforcement Learning (DRL) has proven to be successful in applications where human control tries to be imitated or even surpassed (Mnih et al. (2015)). DRL combines the use of Artificial Neural Networks (ANNs) and Reinforcement Learning (RL) to create artificial agents that can handle continuous and high-dimensional control problems. The main idea behind reinforcement learning is to let an agent explore an environment and choose an action based on the current states. At each step, the agent either receives a reward or a penalty, indicating the quality of the action chosen. This reward/penalty is given by a function called a reward function, which is defined by the user based on the wanted behavior of the system. After multiple attempts, the agent will construct an optimal policy based on the feedback received by exploring the environment. This thesis will apply and compare two DRL algorithms for continuous control tasks: Deep Deterministic Policy Gradient (DDPG)(Lillicrap et al. (2015)) and Proximal Policy Optimization (PPO)(Schulman et al. (2017)).

The use of artificial intelligence for autonomous vehicle control has provided results for both aerial, surface, and marine vessels. Self-driving cars has been developed using machine learning and deployed on full-size research vehicles (Bojarski et al. (2016), Zhang et al. (2019), Folkers et al. (2019)). The applications vary from keeping the vehicle in its lane while driving to parking in crowded parking lots. Each report shows promising results, but they still have limitations for what type of scenarios and disturbances the control systems can handle. The work of Gaudet et al. (2020) presented a theoretical solution for a Mars lunar lander. The controller was able to handle the high-dimensional, non-linear dynamics of the vessel, in addition to achieving fuel efficiency. Machine learning has also been applied to autonomous underwater vehicles, resulting in both motion controllers (Cui et al. (2017)) and docking controllers (Sans-Muntadas et al. (2017), Anderlini et al. (2019)).

For autonomous marine surface vessels, solutions using machine learning have been presented for both path following and docking. The work of Martinsen and Lekkas (2018) presents a solution for a path-following algorithm for an underactuated marine vessel. The objective is to minimize the cross-track error while following the desired path. A control policy was found using deep deterministic policy gradient (DDPG), and

the policy was shown to be stable even when under the influence of unknown ocean currents. A docking controller was constructed by Im and Nguyen (2017) using artificial neural networks (ANN) with a head-up coordinate system as input to the controller. The proposed ANN controller was able to adapt to various ports without the need for specific training on the given port by using relative bearing and distance to the desired position. However, the training data was sampled from real ship docking data, which makes the collection of a large data set difficult. Shuai et al. (2019) further builds on this by implementing a docking simulation platform, which allowed to generate a broad set of reliable docking data.

The work of Eilertsen (2019) presents a way of using a Deep Q-Network (DQN) to select the optimal actions in various situations for a marine vessel. The machine-learning agent was trained to guide a vessel from outside a port to a designated docking space while simultaneously avoiding moving obstacles. This was achieved by implementing a predefined action set from which the agent could select the next action. The trained agent could be used in two different ways, either as a high-level decision support system for a captain or as a direct controller. The thesis presented positive results for the control system in various situations and was able to guide the vessel to the docking area successfully. However, the agent was not taught how to handle environmental disturbances in addition to not knowing when the docking problem was solved. A similar approach was presented by Mothes (2019), where reinforcement learning was applied as an action-planning guidance layer for the marine vessel to guide the vessel to the designated docking position. The work of Rørvik (2020) presented a solution for docking a fully-actuated surface vessel using both DDPG and proximal policy optimization (PPO). The performance of the two DRL algorithms was compared in addition to presenting important findings regarding the development of the machine learning environment.

One particular issue regarding machine learning is the need for extensive training on a large number of training data. This presents a problem when using machine learning for real-world applications. While training, the DRL algorithm figures out the optimal solution using trial-and-error, starting with random inputs to the controllers. This is not ideal since the equipment used is often fragile and expensive. The inputs produced while training will eventually lead to a collision of some sort. Instead, one can use a simulator for the training, assuming that a model representation of the system is available.

In addition to a precise model representation, a simulated input also needs to be constructed. Depending on the type of sensory input that the vehicle possesses, there are multiple approaches. The work of Sadeghi and Levine (2017) lead to a solution where a quadrotor helicopter successfully flew and avoided obstacles in the real world,

being only trained on simulated images. The environment was construed in a 3D CAD model and used as input to the monocular camera mounted on the quadrotor. The work of Shuai et al. (2019) utilized a joystick implementation that provided manual maneuvering. This allowed for a collection of reliable data sampled from successful manual maneuvers.

When working with more complex problems, a physics engine might be necessary. The contact forces can then be modeled, and the interaction between objects can be correctly simulated. OpenAI (Andrychowicz et al. (2019)) used this approach to train their system, called Dactyl, to manipulate a block to a defined configuration. The trained model was then deployed to a Shadow Dexterous Hand (Shadow Robot Company (2019)), which successfully manipulated objects in the real world. The process from initial to the desired manipulation of the block can be seen in Figure 1.1.

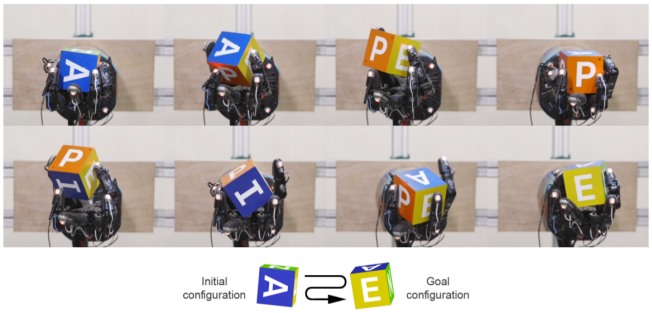


Figure 1.1: The real-world block manipulation by OpenAI's Dactyl system, only trained in a simulated environment. Image courtesy of Andrychowicz et al. (2019).

.

Classical optimization theory can often be used to solve problems that can be solved with reinforcement learning. In some way, both optimization theory and reinforcement learning are designed to solve problems the same way. Both are designed to minimize or maximize the outcome of a problem that is subject to some constraints. The difference lies in how the problems are formulated. Optimization problems are model-dependant and rely on a very accurate model to achieve successful results (Foss and Heirung (2013)). Model-free RL algorithms, such as DDPG and PPO, does not consider the model at all and instead learn the system through interactions. This makes RL more adaptable than optimization since it will be more resilient to changes or inaccuracies in the model. However, the choices made by a RL model will be harder to interpret due to the nonlinear mapping between states and action. When dealing with an optimization problem, the controller's choices can be determined by looking up the model. This makes troubleshooting easier for an optimization problem than for a RL problem.

The motivation behind this project is to investigate the possibilities within artificial intelligence (AI) to develop a docking controller for an underactuated unmanned surface vehicle (USV). Systems for guidance and control can be replaced by machine-learning models that are trained to achieve the same control objectives. The research done in this section has shown that autonomous docking could help reduce the number of accidents experienced during a docking maneuver. Multiple solutions for autonomous docking of fully actuated surface vessels have been presented, but there is still lacking research on docking systems for underactuated vessels. As long as a model of the system is available, one can develop a simulator for training in various environments. The idea is that a machine learning model trained in a physically realistic simulator can be applied directly to the real-world vehicle, and therefore reduce the quantity of training needed with the real-world USV.

1.2 System Overview

Maritime Robotics was founded in 2005 and focuses on delivering vehicles, tools, and systems that operates unmanned both in the air and on the surface. One of their products is the Otter USV, which is the smallest USV that Maritime Robotics produces. It can be used for several applications, including seabed mapping and monitoring of sheltered waters. It consists of a frame mounted on two pontoons, with a control box, batteries, and other necessary sensors mounted on top of the frame (see Figure 1.2). It has a fixed electrical motor (thruster) integrated to each of the pontoons, meaning that the difference in thrust between the two motors is necessary to turn the vessel. The length of the Otter USV is 2 meters, and the width is 1.08 meters (Maritime Robotics (2019)).

Figure 1.2: Picture of the Otter USV. Illustration from Geo-matching (2019)

.

Multiple control systems have been developed for the Otter USV. For this thesis, only the path-following control system is of interest. It's assumed that this will be used to guide the vessel close enough to the dock to activate the docking system, as illustrated in Figure 1.3. The input to the path-following controller is given by the on-board RTK GNSS system. The sensor outputs the NED position, in addition to the course of the vessel. The RTK GNSS is part of a high performance inertial sensor called *Ellipse2-D* (SBG-Systems (2018)). The path-following controller uses a waypoint generator as input reference. These waypoints are predefined by the user based on the current location of the dock and the desired path towards the dock.

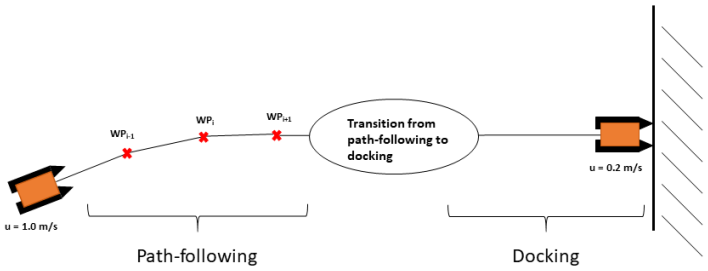


Figure 1.3: Overview of the relation between the path-following and docking controller.

.

The two inputs to the docking controller are a stereo vision camera and a doppler velocity log. The camera system calculates the position of the vessel by detecting multiple markers placed on the dock. Each marker has a unique id that can be looked up in a table to get the marker's position in NED. The camera system then calculates the relative distance between the markers and the vessel, which results in the vessel's position in NED. The doppler velocity log estimates the ocean current velocity and angle. A system diagram of the two control systems is illustrated in Figure 1.4. This overview has been given in order to put the docking control system in context with the other systems on the Otter USV. In this thesis, the details of how the pose and ocean current estimates are calculated will not be investigated, only used as input to the docking controller.

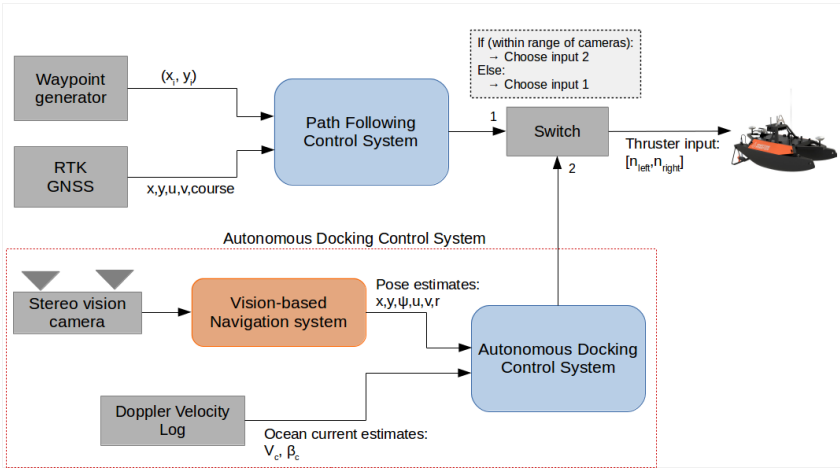


Figure 1.4: System diagram showing the two control systems on board the Otter.

1.3 Research Questions

The following research questions are of interest regarding the work of this thesis.

Q1 How can machine learning be applied to design a docking control system for an underactuated USV?

Q2 Is it possible to train the machine-learning model without measuring the environmental disturbances, such as ocean current, wind and waves?

Q3 Is it possible to achieve a control system for real-world application using a machine-learning model only trained in a simulated environment?

1.4 Objectives

The objectives of this thesis are summarized as follows:

- Describe the Otter USV equations of motion and sensory model.
- Develop an environment for the Otter USV suitable for machine learning in Python which can be used for training and simulation.
- Compare two deep reinforcement learning algorithms: Deep Deterministic Policy Gradient and Proximal Policy Optimization.
- Compare the performance of a docking controller with and without DVL measurements of ocean current velocities.

1.5 Assumptions

The following assumptions were made during development of the machine learning docking system:

- Surge speed u $\in$ [0.2, 1] m/s, with cruising speed of 1 m/s while approaching the dock. This is the operating condition chosen in order to ensure a safe and controlled docking maneuver.
- Ocean current speed $V_c \in [0, 0.5]$ m/s. This is the USV's ideal operating condition without saturating the actuators.
- Thruster dynamics are not considered. Due to the fast dynamics of the thrusters, changes in thruster output are assumed instant. However, the input is low-pass filtered in order to reduce actuator wear and tear.

- The USV states are measured using stereo vision positioning system operating at 20 Hz, with 2 cm position accuracy and 0.1° heading accuracy.
- No wind or wave disturbances present. Only ocean currents are considered for the implementation in this thesis.
- Loss of signal, signal drift, signal freeze, and wild points for the sensor measurements are not present, however, this should be considered in future work.

1.6 Requirement Specifications

When developing the machine learning docking system, the following requirement specifications were defined for the controller performance, software, and hardware:

R1 The USV is able to complete the docking maneuver with an accuracy of 1 meter from the desired docking position, with a surge speed $u \leq 0.2$m/s.

R2 The USV is able to operate under the influence of measurement noise and in ocean currents up to 0.5 m/s.

R3 The software is developed using an object-oriented programming language, making the software modular and convenient to implement with the existing systems on the Otter USV. The software is also developed using open-source libraries for the machine-learning implementation.

R4 The hardware choices must be able to provide a CPU with enough computing power to run the machine-learning algorithm, a GPU for online training, and enough battery capacity to power it. In addition, the form factor of the hardware has to be small enough to be mounted on the Otter USV.

1.7 Contributions

The work of this thesis has resulted in the following contributions:

- The development of a reinforcement-learning environment for the Otter USV in Python. The environment is suitable for training with various reinforcement-learning algorithms and is easily adjusted to fit the action and state space of the problem. The simulator includes realistic dynamics for the USV in addition to realistic measurement noise and ocean currents.
- A proposal for how reinforcement learning can be applied to achieve autonomous docking. Remarks and findings when developing a reward function have been

presented, in addition to how the state vector should be developed.

- A novel docking system for an underactuated USV using machine learning. Autonomous docking has previously been achieved before for fully actuated marine vessels. However, this thesis has adapted this to handle the underactuated dynamics of the Otter USV. The docking system has been developed with a real-world application in mind, using values for measurement accuracy found in consumer-available products.
- The results from comparing a reinforcement-learning model with and without measured ocean current have been verified. This showed that it is recommended to measure the ocean current in order to achieve a model that performs well under the influence of strong ocean currents.

1.8 Outline

The thesis consists of eight chapters:

- **Chapter 1** presents the background and motivation behind the thesis. An overview of the system is presented along with research questions, assumptions, requirement specifications and contributions.
- **Chapter 2** presents the equations of motion and sensory model for the USV.
- **Chapter 3** presents the theory and development within deep reinforcement learning. The main concepts are explained and put in context with the two reinforcement-learning algorithms used.
- **Chapter 4** present the software and hardware choices for the thesis.
- **Chapter 5** presents the design and implementation details. This includes the machine learning environment, the choices made for the reward function and the parameters used for DDPG and PPO.
- **Chapter 6** explains the simulation setup used for training. Initial and termination values are presented in addition to the three cases used for testing the docking controller.
- **Chapter 7** presents the results from testing the trained machine-learning models and discusses the results from each case.
- **Chapter 8** concludes on the results and presents suggestions for future work.
- **Appendix A** contains the physical parameters of the Otter USV.

Chapter 2

Otter USV Model

2.1 Kinematics of the Otter USV

In order to describe the position and orientation of a marine craft moving freely in 3 dimensions it's necessary to use 6 degrees of freedom (DOFs), 3 translational and 3 rotational components (Fossen (2011)). The 3 translational components consists of **surge**, **sway** and **heave**, while the 3 rotational components consists of **roll**, **pitch** and **yaw**, see Figure 2.1.

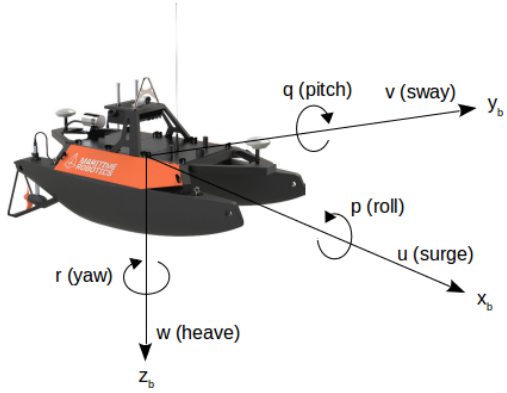


Figure 2.1: The 6 degrees of freedom for the Otter USV.

The notation in Figure 2.1 is adopted from the Society of Naval Architects and Marine Engineers SNAME (1950). Table 2.1 gives a description for each of the components shown in Figure 2.1

Table 2.1: Notation from SNAME (1950)

DOF		Forces and moments	Linear and angular velocities	Positions and Euler angles
1	motions in the x-direction (surge)	X	u	x
2	motions in the y-direction (sway)	Y	v	y
3	motions in the z-direction (heave)	Z	w	z
4	rotation about the x-axis (roll)	K	p	ϕ
5	rotation about the y-axis (pitch)	M	q	θ
6	rotation about the z-axis (yaw)	N	r	ψ

2.2 The Otter USV model

The equations of motion for the Otter USV are represented in a compact marine craft model in 6 DOF as presented in (Fossen; 2011, p. 13).

$$\mathbf{M}\dot{\boldsymbol{\nu}} + \mathbf{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \mathbf{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \mathbf{g}(\boldsymbol{\eta}) + \mathbf{g}_0 = \boldsymbol{\tau} + \boldsymbol{\tau}_{\text{wind}} + \boldsymbol{\tau}_{\text{wave}} \tag{2.1}$$

with $\boldsymbol{\nu}$ and $\boldsymbol{\eta}$ defined as

$$\begin{aligned} \boldsymbol{\nu} &= [u, v, w, p, q, r]^\top \\ \boldsymbol{\eta} &= [x, y, z, \phi, \theta, \psi]^\top \end{aligned} \tag{2.2}$$

where $\boldsymbol{\nu}$ and $\boldsymbol{\eta}$ are generalized velocities and positions used to describe motions in 6 DOF and $\boldsymbol{\tau}$ are the generalized forces acting on the craft. In this model $\mathbf{M}$, $\mathbf{C}(\boldsymbol{\nu})$ and $\mathbf{D}(\boldsymbol{\nu})$ denotes the inertia, coriolis and damping matrices, $\mathbf{g}(\boldsymbol{\eta})$ is the generalized gravitational and buoyancy force-matrix and $\mathbf{g}_0$ consists of static restoring forces and moments due to ballast systems and water tanks.

2.2.1 Inertia Matrices

In order to find $\mathbf{M}$ and $\mathbf{C}(\boldsymbol{\nu})$ the rigid-body inertia matrix $\mathbf{M}_{RB}$ and the rigid-body coriolis and centripetal forces-matrix $\mathbf{C}_{RB}(\boldsymbol{\nu})$ in CG are calculated (Fossen; 2011, p. 49):

$$\mathbf{M}_{\text{RB}}^{\text{CG}} = \begin{bmatrix} (m + m_p)\mathbf{I}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{I}_g \end{bmatrix} \tag{2.3a}$$

$$\mathbf{C}_{\text{RB}}^{\text{CG}}(\boldsymbol{\nu}) = \begin{bmatrix} (m + m_p)\mathbf{S}(\omega_{b/n}^b) & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & -\mathbf{S}(\mathbf{I}_g\omega_{b/n}^b) \end{bmatrix} \tag{2.3b}$$

where

$$\boldsymbol{\omega}_{b/n}^b = [p, q, r]^\top \tag{2.4}$$

and m_p is the payload mass for the Otter, $\mathbf{S}(\mathbf{x})$ is the skew-symmetric matrix of $\mathbf{x}$ and $\mathbf{I}_g$ is the inertia matrix. $\mathbf{I}_g$ was denoted as

$$\mathbf{I}_g := \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{yx} & I_y & -I_{yz} \\ -I_{yz} & -I_{zy} & I_z \end{bmatrix} = m \begin{bmatrix} R_{44}^2 & 0 & 0 \\ 0 & R_{55}^2 & 0 \\ 0 & 0 & R_{66}^2 \end{bmatrix} \tag{2.5}$$

where $\mathbf{R}_{44}$, $\mathbf{R}_{55}$ and $\mathbf{R}_{66}$ are the radii of gyration. Since $\mathbf{M}_{RB}$ and $\mathbf{C}_{RB}(\boldsymbol{\nu})$ are defined in CG, it is necessary to transform the matrices to CO by using the transformation matrix $\mathbf{H}(\mathbf{r}_g^b)$, which is denoted as:

$$\mathbf{H}(\mathbf{r}_g^b) := \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{S}^\top(\mathbf{r}_g^b) \\ \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3} \end{bmatrix}, \quad \mathbf{H}^\top(\mathbf{r}_g^b) = \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{S}(\mathbf{r}_g^b) & \mathbf{I}_{3\times3} \end{bmatrix} \tag{2.6}$$

The transformation of $\mathbf{M}_{RB}$ and $\mathbf{C}_{RB}(\boldsymbol{\nu})$ from CG to CO can then be done using (2.6):

$$\mathbf{M}_{\mathrm{RB}}^{\mathrm{CO}} = \mathbf{H}^\top(\mathbf{r}_g^b)\mathbf{M}_{RB}^{CG}\mathbf{H}(\mathbf{r}_g^b) \tag{2.7a}$$

$$\mathbf{C}_{\mathrm{RB}}^{\mathrm{CO}}(\boldsymbol{\nu}) = \mathbf{H}^\top(\mathbf{r}_g^b)\mathbf{C}_{RB}^{CG}\mathbf{H}(\mathbf{r}_g^b) \tag{2.7b}$$

A marine USV has to take the resistance of the fluid into account when finding the $\mathbf{M}$ and $\mathbf{C}(\boldsymbol{\nu})$ matrices. This is done by including hydrodynamic added mass, $\mathbf{M}_A$ and $\mathbf{C}_A(\boldsymbol{\nu})$. These matrices was found using the following equations (Fossen; 2011, p. 118-121)

$$\mathbf{M}_A = -\begin{bmatrix} X_{\dot{u}} & 0 & 0 & 0 & 0 & 0 \\ 0 & Y_{\dot{v}} & 0 & 0 & 0 & 0 \\ 0 & 0 & Z_{\dot{w}} & 0 & 0 & 0 \\ 0 & 0 & 0 & K_{\dot{p}} & 0 & 0 \\ 0 & 0 & 0 & 0 & M_{\dot{q}} & 0 \\ 0 & 0 & 0 & 0 & 0 & N_{\dot{r}} \end{bmatrix} \tag{2.8}$$

$$\mathbf{C}_A(\boldsymbol{\nu}) = -\begin{bmatrix} 0 & 0 & 0 & 0 & -Z_{\dot{w}}w & Y_{\dot{v}}v \\ 0 & 0 & 0 & Z_{\dot{w}}w & 0 & -X_{\dot{u}}u \\ 0 & 0 & 0 & -Y_{\dot{v}}v & X_{\dot{u}}u & 0 \\ 0 & -Z_{\dot{w}}w & Y_{\dot{v}}v & 0 & -N_{\dot{r}}r & M_{\dot{q}}q \\ Z_{\dot{w}}w & 0 & -X_{\dot{u}}u & N_{\dot{r}}r & 0 & -K_{\dot{p}}p \\ -Y_{\dot{v}}v & X_{\dot{u}}u & 0 & -M_{\dot{q}}q & K_{\dot{p}}p & 0 \end{bmatrix} \tag{2.9}$$

The following assumptions were made:

$$\begin{aligned} X_{\dot{u}} &= -0.1 \cdot m \\ Y_{\dot{v}} &= -1.5 \cdot m \\ Z_{\dot{w}} &= -1.0 \cdot m \\ K_{\dot{p}} &= -0.2 \cdot R_{44} \\ M_{\dot{q}} &= -0.8 \cdot R_{55} \\ N_{\dot{r}} &= -1.7 \cdot R_{66} \end{aligned} \tag{2.10}$$

The $\mathbf{M}$ and $\mathbf{C}(\boldsymbol{\nu})$ matrices was then found by summing the rigid-body and added mass matrices

$$\mathbf{M} = \mathbf{M}_{\text{RB}}^{\text{CO}} + \mathbf{M}_A \tag{2.11a}$$

$$\mathbf{C}(\boldsymbol{\nu}) = \mathbf{C}_{\text{RB}}^{\text{CO}}(\boldsymbol{\nu}) + \mathbf{C}_A(\boldsymbol{\nu}) \tag{2.11b}$$

2.2.2 Restoring Forces

Since the Otter is modeled in 6 degrees of freedom, the motions in heave, roll and pitch can't be represented by a zero-frequency model. The natural frequencies in these second-order mass-damper-spring systems are dominating and needs to be modeled by the following equations:

$$\omega_{\text{heave}} = \sqrt{\frac{G_{33}}{M_{33}}} \tag{2.12a}$$

$$\omega_{\text{roll}} = \sqrt{\frac{G_{44}}{M_{44}}} \tag{2.12b}$$

$$\omega_{\text{pitch}} = \sqrt{\frac{G_{55}}{M_{55}}} \tag{2.12c}$$

where G_{33}, G_{44} and G_{55} is defined by the transverse ($\overline{\text{GM}}_T$) and longitudinal metacentric height ($\overline{\text{GM}}_L$) (Fossen; 2011, p. 67).

$$G_{33} = 2\rho g A_{\text{w,pont}} \tag{2.13a}$$

$$G_{44} = \rho g \nabla \overline{\text{GM}}_T \tag{2.13b}$$

$$G_{55} = \rho g \nabla \overline{\text{GM}}_L \tag{2.13c}$$

With ∇ and $A_{\text{w,pont}}$ given by

$$\nabla = \frac{m + m_p}{\rho} \tag{2.14a}$$

$$A_{\text{w,pont}} = C_{\text{w,pont}} \cdot L \cdot B_{\text{pont}} \tag{2.14b}$$

This can then be used to find the restoring matrix $\mathbf{G}^{CF}$ in Center of Force (CF) (Fossen; 2011, p. 181).

$$\mathbf{G}^{\text{CF}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & G_{33} & 0 & 0 & 0 \\ 0 & 0 & 0 & G_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & G_{55} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.15}$$

Which has to be transformed from CF to CO by using the transformation matrix $\mathbf{H}(\mathbf{r}_f^b)$ from (2.6):

$$\mathbf{G} = \mathbf{H}^\top(\mathbf{r}_f^b)\mathbf{G}^{\text{CF}}\mathbf{H}(\mathbf{r}_f^b) \tag{2.16}$$

where $\mathbf{r}_f^b$ = [-0.2, 0, 0]$^\top$ is the distance from CF to CO. This can then be used to find $\mathbf{g}(\eta)$ in (2.1):

$$\mathbf{g}(\eta) \approx \mathbf{G}\boldsymbol{\eta} \tag{2.17}$$

Lastly, the forces and moments $\mathbf{g}_0$ due to the ballast tanks is given by the following

equation (Fossen; 2011, p. 75):

$$\mathbf{g}_0 = \begin{bmatrix} 0 \\ 0 \\ -Z_{\text{ballast}} \\ -K_{\text{ballast}} \\ -M_{\text{ballast}} \\ 0 \end{bmatrix} \tag{2.18}$$

Where Z_{ballast}, K_{ballast} and M_{ballast} are heave, roll and pitch moments due to ballast. The value of these was found by manual pre-trimming as shown in the following equation (Fossen; 2011, p. 76):

$$\mathbf{G}\boldsymbol{\eta} + \mathbf{g}_0 = 0 \tag{2.19}$$

from where $\mathbf{g}_0$ was found to be:

$$\mathbf{g}_0^{3,4,5} = \begin{bmatrix} 0 \\ 320 \\ 0 \end{bmatrix} \tag{2.20}$$

2.2.3 Damping Forces

The linear viscous damping matrix $\mathbf{D}(\nu)$ is given by:

$$\mathbf{D}(\nu) = -\begin{bmatrix} X_u & 0 & 0 & 0 & 0 & 0 \\ 0 & Y_v & 0 & 0 & 0 & 0 \\ 0 & 0 & Z_w & 0 & 0 & 0 \\ 0 & 0 & 0 & K_p & 0 & 0 \\ 0 & 0 & 0 & 0 & M_q & 0 \\ 0 & 0 & 0 & 0 & 0 & N_r \end{bmatrix} \tag{2.21}$$

Where the linear damping terms on the diagonal of the damping matrix $\mathbf{D}(\nu)$ is defined

by the following equations (Fossen; 2011, p. 125):

$$-X_u = B_{11v} = \frac{M_{11}}{T_{\text{surge}}} \tag{2.22a}$$

$$-Y_v = B_{22v} = 0 \tag{2.22b}$$

$$-Z_w = B_{33v} = 2\zeta_{\text{heave}}\omega_{\text{heave}}M_{33} \tag{2.22c}$$

$$-K_p = B_{44v} = 2\zeta_{\text{roll}}\omega_{\text{roll}}M_{44} \tag{2.22d}$$

$$-M_q = B_{55v} = 2\zeta_{\text{pitch}}\omega_{\text{pitch}}M_{55} \tag{2.22e}$$

$$-N_r = B_{11v} = \frac{M_{66}}{T_{\text{yaw}}} \tag{2.22f}$$

2.2.4 Cross-Flow Drag for Sway and Yaw

The nonlinear damping forces in sway and the yaw moment are applied as presented in (Fossen; 2011, p. 127).

$$Y = -\frac{1}{2}\rho \int_{-\frac{L}{2}}^{\frac{L}{2}} T(x)C_d^{2\text{D}}(x)|v_r + xr|(v_r + xr)dx \tag{2.23a}$$

$$N = -\frac{1}{2}\rho \int_{-\frac{L}{2}}^{\frac{L}{2}} T(x)C_d^{2\text{D}}(x)x|v_r + xr|(v_r + xr)dx \tag{2.23b}$$

where $v_r = v - v_c$ is the relative sway velocity and $C_d^{2\text{D}}(x)$ = Hoerner(B,T) is calculated with the Matlab MSS toolbox (Fossen and Perez (2004)).

2.2.5 Control Allocation

As stated under assumptions in Section 1.5, wind and waves were neglected, meaning $\boldsymbol{\tau}_{\text{wind}} = \boldsymbol{\tau}_{\text{wave}} = 0$. The control forces and moments was calculated using the following equation (Fossen; 2011, p. 413)

$$\boldsymbol{\tau} = \mathbf{TKu} \tag{2.24}$$

Where $\mathbf{T}$ is the actuator configuration matrix, $\mathbf{K}$ is a diagonal matrix of thrust coefficients and $\mathbf{u}$ is the control variable given by

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} n_1|n_1| \\ n_2|n_2| \end{bmatrix} \tag{2.25}$$

where $n_{1,2}$ is the propeller revolutions per minute (rpm) for the left and right thruster

respectively. Since the thrusters only act on the heading and the surge of the vessel, the $\boldsymbol{\tau}$ matrix will be

$$\boldsymbol{\tau} = [\tau_1 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; \tau_6]^\top \tag{2.26}$$

where τ_1 and τ_6 are the control inputs for surge and yaw respectively. Furthermore the thrust coefficients are equal for both of the thrusters, only depending on positive or negative rotation of the propellers. Using this in (2.24) gives

$$\begin{bmatrix} \tau_1 \\ \tau_6 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -l_1 & -l_2 \end{bmatrix} \begin{bmatrix} k_1 & 0 \\ 0 & k_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{2.27}$$

where

$$\begin{aligned} l_1 &= -l_2 = -Y_{\text{pont}} \\ k_i &= \begin{cases} k_{\text{pos}}, & \text{if } n_i > 0 \\ k_{\text{neg}}, & \text{otherwise} \end{cases} \end{aligned} \tag{2.28}$$

Solving (2.27) for $\mathbf{u}$ yields the following

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} k_1 & 0 \\ 0 & k_2 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 1 \\ -l_1 & -l_2 \end{bmatrix}^{-1} \begin{bmatrix} \tau_1 \\ \tau_6 \end{bmatrix} \tag{2.29}$$

The propellers of the USV is quadratic and modelled with $u_{1,6} = n_{1,6}|n_{1,6}|$ in (2.27). The general solution to the inverse of this is $u_{1,6} = \text{sgn}(\text{u}_{1,6})\sqrt{|\text{u}_{1,6}|}$. The controller input for both the controllers can therefore be modeled as

$$\begin{bmatrix} n_1 \\ n_2 \end{bmatrix} = \begin{bmatrix} \text{sgn}(\text{u}_1)\sqrt{|\text{u}_1|} \\ \text{sgn}(\text{u}_2)\sqrt{|\text{u}_2|} \end{bmatrix} \tag{2.30}$$

Which is bounded as follows

$$\begin{aligned} n_{\text{max}} &= \sqrt{\frac{0.5 \cdot 24.4 \cdot g}{k_{\text{pos}}}} \\ n_{\text{min}} &= \sqrt{\frac{0.5 \cdot 13.6 \cdot g}{k_{\text{neg}}}} \end{aligned} \tag{2.31}$$

2.3 Sensory Systems

In order to use the docking controller for a real-world application, the USV needs to be able to estimate its position and velocity. The ocean current velocity and angle also needs to be estimated. This section will introduce and discuss some possible sensors or methods that can be utilized to estimate the needed states for the controller.

2.3.1 Position and Velocity

For a docking situation, it is not sufficient to only use a GNSS system. Even though a RTK-GNSS system can provide accuracy at a centimeter-level, it can not comply with the strict requirements for redundancy. GNSS is dependant on a clear view of the sky in order to receive the satellite signal. For a small vessel as the Otter USV, this line of sight might be blocked by larger objects in the surroundings, like other ships or constructions. Also, a GNSS system is sensitive for occasional high noise content, multipath effects, low bandwidth, and interference or jamming (Aqel et al. (2016)). Due to these weaknesses, a docking controller cannot be regarded as safe with GNSS as the only positioning system. Instead, an onboard sensory system that can precisely output the relative position of the USV has to be utilized. The following will introduce and discuss different sensory systems for position estimation.

Lidar

Lidar, which stands for Light Detection and Ranging, is a method that uses light to measure the distance between the sensor and the surroundings. A map of the environment is created by using a pulsing laser and measure the time of flight of the reflections (National Ocean Service (2020)). The calculation for how far the returning light has traveled is done with the equation:

$$\text{Distance} = \frac{\text{Speed of light} \times \text{Time of flight}}{2} \tag{2.32}$$

Lidar can also be designed such that it spins in a circle. This will create a sensor that returns a 360-degree point cloud of the surroundings. The use of a fine laser-beam allows lidar to estimate distance with a high resolution. By using filter techniques, the sensor is also able to remove certain materials in order to remove noise (Horaud et al. (2016)).

There exists a large variety of lidar systems on the market, from simple low-cost 2D lidars (ROS components (2020)) to high-end 3D sensor systems (Velodyne Lidar (2020)). Range, resolution, and field of view become better as the quality increases. Which sensor to choose depends on the required accuracy and precision.

Ultrasonic Sensor

An ultrasonic sensor measures distance by emitting sound waves and measure the time of the reflected sound waves. This is the same principle as used for lidar, but with sound instead of light (Arrow (2020)). The use of sound waves instead of light has its benefits and can detect objects more efficiently in some situations. Light-based sensors are more affected by sunlight and may struggle to detect transparent objects. However, sound-based sensors will become unreliable if the object is made out of an absorbing material or shaped such that the sound is reflected away from the receiver. It is also hard to make a sound wave as narrow as a laser, which reduces how accurately the direction can be determined.

IMU

An Inertial Measurement Unit (IMU) consists of three motion sensors; three-axis rate gyros, accelerometers, and magnetometers. The IMU measures angular rate, force, and magnetic field. It uses software to combine these measurements and output orientation and heading. This is often referred to as an attitude and heading reference system (AHRS). A stand-alone IMU solution will drift due to sensor biases, misalignment, and temperature variations (Fossen; 2011, p.328). This drift can be removed by combining the IMU with a GNSS in a state observer. As with lidars, there exists a large quantity of different IMUs, ranging from low-cost sensors with limited accuracy (Sparkfun (2020)) to expensive high-end IMU sensors (SBG-Systems (2018)).

Optical Cameras

Vision-based systems can be employed for localization tasks and have been shown to be more accurate and reliable than other sensor-based localization systems, such as GNSS (Howard (2008)). In addition, optical cameras are a significantly cheaper solution than other proximity sensors as lidar and ultrasonic sensors. A localization task can be achieved using only consumer-grade cameras (Aqel et al. (2016)). Camera images can be used for both indoor and outdoor navigation, and the images captured by a camera can provide a large amount of information that can be used for several purposes. Another positive attribute of optical cameras is that they are passive, meaning that they do not suffer from interference often encountered when other proximity sensors are used.

However, optical cameras are sensitive to environmental conditions, such as lightning, image blurs, shadows, and harsh weather, like fog, rain, or snow. In addition, the image processing would require a large amount of memory and computational power, making it an expensive task.

The use of an optical camera for pose estimation is called Visual Odometry (VO) and consists of performing incremental online estimation using an image sequence captured by the camera. The most common VO methods utilize either monocular or binocular cameras. Monocular VO systems are preferred when size is essential. By only using a single camera, the deployment is made easy, and many calibration errors, which binocular systems are prone to, are mitigated. However, monocular systems suffer from scale uncertainty and cannot determine depth from only a single image. Binocular cameras, or stereo cameras, utilizes two cameras with a fixed and known baseline. By using triangulation, the position in three dimensions can be calculated from a single image. However, the stereo cameras require more calibration than monocular camera systems, in addition to being more expensive.

By combining visual odometry with fiducial markers (Olson (2011)), the position of the vessel can be determined at a centimeter level. Fiducial markers are 2D planar targets with artificial features, making it possible to recognize and distinguish multiple markers in a single image. The markers' position is precisely measured and known to the vessel; hence, the relative position of the vessel to the markers can be calculated using classical computer vision algorithms.

2.3.2 Ocean Current

During a docking procedure, the USV will be affected by the surrounding ocean currents. These currents are considered a disturbance and have to be considered in the design of the control law. Both the current velocity and angle of attack are unknown to the USV at the start of the docking. Some form of online estimation of the disturbance is therefore needed in order to ensure a safe and controlled docking.

One solution is to apply the work of Moe et al. (2014). The paper proposes an ocean-current observer for estimating unknown ocean currents affecting an underactuated surface vessel. The method extends the results of Borhaug and Pettersen (2006) by developing a control method for path following using virtual Serret-Frenet reference frames. This guidance law, combined with the ocean-current observer, is used to achieve a path following algorithm with UGAS stability properties under explicit conditions.

Another approach to estimating the ocean-current disturbances is to utilize a Doppler Velocity Log (DVL). A DVL estimates the vessel's velocity relative to the sea bottom

using the Doppler shifting of acoustic signals (Rudolph and Wilson (2012)). The sensor consists of four angled transducers, each sending a sound wave. The transducers receive the echo from each sound wave, and the frequency shift between the transmitted and received signal can be determined. The change in frequency is used to calculate the velocity of the DVL along each of the transducer axes, which again is used to determine the velocity of the vessel along its coordinate axes. By measuring the time of flight of the sound waves, the DVL can also determine the distance between the transducers and the sea bottom, i.e., the altitude of the vessel. Figure 2.2 illustrates the principle of the DVL sensor.

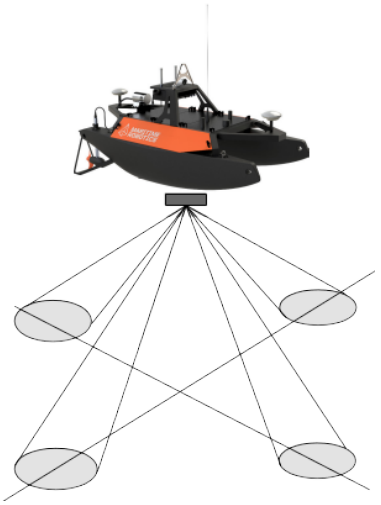


Figure 2.2: Working principle of the DVL with four transducers.

2.4 Summary

This chapter has given a brief overview of the equations contained in the Otter USV model used for simulations. The theory behind the sensory model on the USV has also been presented.

Chapter 3

Machine Learning Theory

This chapter presents relevant theory needed for the implementation of the machine-learning controller. Sutton and Barto (2015) is used as the main source of information for the fundamental reinforcement learning theory.

3.1 Reinforcement Learning

Reinforcement learning (RL) is a kind of machine learning where an agent is set to solve a problem without being told how to proceed throughout its environment. Instead, the RL-agent has to interact with and explore the environment. The behavior is then adapted based on the consequence of its actions in order to reach its goals. The agent's possible states and actions are available in the state space and the action space, respectively.

The RL-agent will at each time step t receive information about its state $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the state space, and based on this state information select an action $A_t \in \mathrm{A}(S_t)$, where $\mathrm{A}(S_t)$ is the action space available in state S_t. After the action is chosen and executed the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and information about its new state, S_{t+1}. An illustration of the RL learning-cycle can been seen in Figure 3.1. During training the agent creates a mapping from states to actions, called the agents policy π_t. The policy is probabilistic and an action is chosen from a distribution denoted $A_t \sim \pi(s, a) = P(a|s)$, where $P(a|s)$ is the probability that $A_t = a$ if $S_t = s$.

This type of goal-directed learning is inspired by how intelligent beings learn. A toddler is not told how to operate its limbs or look around. Instead, it receives tons of information on how the environment responds to its actions. This way, the toddler develops and becomes an active decision-making agent that seeks to achieve its goal, even though its environment is full of uncertainties.

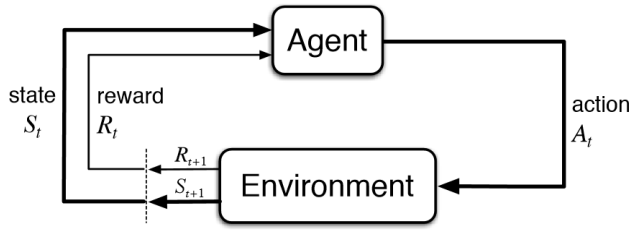


Figure 3.1: Reinforcement Learning illustration. Image courtesy of Sutton and Barto (2015).

In a RL-model, one usually define four main subelements: a *policy*, a *reward signal*, a *value function*, and a *model* of the environment (Sutton and Barto (2015)).

Definition 3.1.1. *Policy: defines what action the RL-agent will choose based on the current states received from the environment.*

Definition 3.1.2. *Reward signal: a single number received by the RL-agent from the environment at each time step. The agent will try to maximize this value through its actions during training.*

Definition 3.1.3. *Value function: specifies the expected long-term reward when in the current state and considering the most likely next steps.*

Definition 3.1.4. *Environment model: a representation of the environment such that the next state and reward can be predicted during training.*

Markov Decision Process

A Markov Decision Process (MDP) is a task where all states satisfy the Markov property. The Markov property states that any given state contains all the necessary information, and all information encountered so far can be disregarded. This means that all future states only depends on the current state, not the path taken to reach this state. A MDP contains a set of states, actions, rewards, and decision probabilities, which completely specifies the dynamics of the system. This is an important property in reinforcement learning and is used to describe the environment in which the agent is investigating.

The Bellman Equation

The Bellman equation is the basis for solving a reinforcement learning problem. The equation is used to solve a MDP by finding the optimal policy which maximizes the reward received over time. The most significant advantage of using the Bellman equations is that the value of any state can be expressed as a value of other states. This means that if S_{t+1} is known, then S_t can be calculated, making it easy to create iterative solutions for MDPs. The state value is equal to the maximum reward, given by the optimal action, and the discounted value from the next state, s'.

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s') \right) \tag{3.1}$$

Where *R(s,a)* is the reward received for taking action a in state s, γ is the discount factor and $P(s,a,s')$ is the transition probability for ending up in state s' if starting in state s and take action a. The discounted term, $V(s')$, in (3.1) can be expanded to an infinite sequence containing all future states. This infinite sequence can instead be grouped together, forming an expression for the quality of a given action in a given state.

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s,a,s') \max_{a'}(Q(s',a')) \tag{3.2}$$

The quality function $Q(s,a)$ in (3.2) has shown to give better results in reinforcement learning due to not needing to know the transition and reward function in advance in order to find the expected value.

Exploration vs Exploitation

One challenge specific for reinforcement learning is to find the balance between exploration and exploitation. At the beginning of training, the environment is completely unknown to the RL-agent. While training, the agent has to explore the environment in order to find actions that will accumulate a high reward, but it also needs to exploit what is already learned in order to maximize the reward. This trade-off is often hard to determine since both exploration and exploitation has to be performed. If only the greedy actions are exploited, the agent might get stuck performing suboptimal actions.

In order to increase both training speed and the overall result, one can apply exploration noise. When adding noise to the action space, the likelihood of choosing an action is randomized, which again will improve the efficiency of the exploration.

Better exploration with parameter noise

Exploration is an essential part of training a RL-agent, and the trade-off between exploration and exploitation is always challenging to balance. To increase the quality of the exploration executed by the agent, it is common practice to apply noise to the action space. This will help randomize the agent's actions, resulting in better exploration of the environment. An alternative to this approach was presented by Plappert et al. (2018), where they apply noise to the parameter space instead of the action space. This is illustrated in Figure 3.2. Three conclusions were drawn from their experiments: firstly, parameter-space noise outperforms action-space noise in most environments and performs equally in the rest. Secondly, parameter space noise gives better exploration in sparse reward environments. Their last experiment showed that parameter space noise gives equal sample efficiency as the state-of-the-art evolution strategies for deep policies. The article's work shows that parameter-space noise works for both discrete and continuous environments resulting in more stable exploration and faster convergence to an optimal policy.

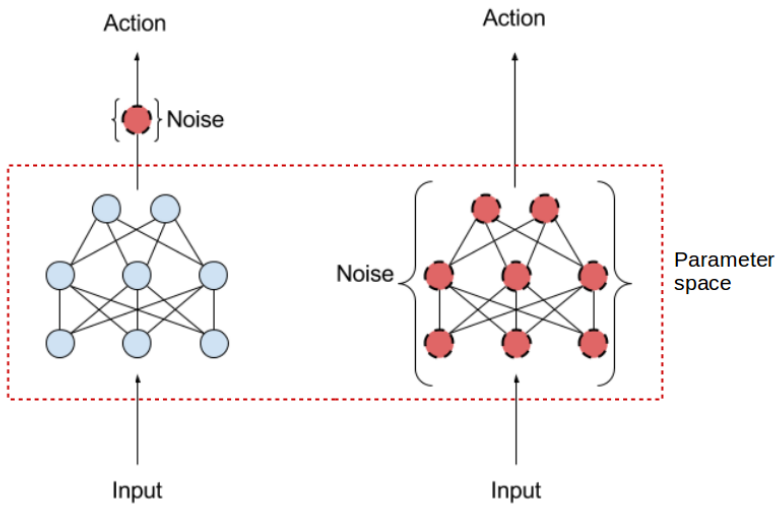


Figure 3.2: Action space noise compared to parameter space noise

Reward Function

The reward function used for a reinforcement learning problem is highly dependant on the purpose or goal of the agent. In order to make the agent learn, each step has to be properly rewarded. Since the goal is to maximize the long-term reward, the reward function has to indicate which action the agent should do next to increase its reward further. There are different ways of penalizing the RL agent. One could give a reward of -1 for all moves outside the goal and then give a reward of +5 when reaching the goal. This kind of reward function could be appropriate for motivating the agent to use as few steps as possible. However, it doesn't say anything about how close the agent is to a solution. Instead, one can use a continuous, smooth function where the gradient grows larger closer to the goal. The agent will then receive small rewards when far from the goal, and then the reward grows bigger as the agent approaches the target.

Terminal states, such as going out of the environment or reaching the goal, should also be highly penalizing or rewarding. This way, the agent wants to reach the goal fast instead of collecting reward around the goal.

The long-term accumulated reward can the expressed with the return function G_t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.3}$$

where the discount rate γ, $0 \leq \gamma \leq 1$, defines the current value of future rewards R.

Discrete vs Continuous Action Spaces

Within reinforcement learning, the action space is normally categorized as either discrete or continuous. The Discrete Action Space (DAS) consists of a finite number of actions possible for each state the agent can experience. This allows for faster learning since the amount of exploring needed is limited. However, the DAS does not perform well when the set of possible actions grows too big. An example of a DAS is the game of Pac-Man, where the actions are limited to up, down, left, and right in each state.

The Continuous Action Space (CAS) is used for problems where the agent needs to predict a continuous output. Path-following for a car is an example of such a problem, where the agent predicts a steering angle based on the current state. It is possible to discretize the action space of a CAS such that the algorithms in DAS can be used, but this often results in a sparse coverage of the action space.

Supervised vs Unsupervised Learning

When working with supervised learning, the model is trained on a labeled set of data. The training data contains both the input and the corresponding output. The goal for supervised learning is then to create a function that maps the input to the correct output. If done correctly, then the model can be used to predict the correct output when given new, unlabeled input. This type of learning is useful for classification problems, but not suited for learning from interaction.

Unsupervised learning is the case where the data set only contains the input data without the labeled output data. The goal of unsupervised learning is to find a model that represents the hidden structure or distribution in unlabeled data. This type of learning is used for clustering data and association data.

Reinforcement learning is considered to be different from unsupervised learning. While unsupervised learning aims to find similarities and differences in data, reinforcement learning tries to maximize the long-term reward which the agent receives by choosing the optimal path and actions.

3.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are the backbone of modern machine learning algorithms. The structure takes inspiration from the human brain, where neurons receive input signals and produce an output based on the information received. The ANN neurons are connected in an acyclic graph, where the outputs of some neurons become the input of other neurons. Figure 3.3a shows the structure of a Fully Connected Neural Network (FCNN), the layer type most commonly used. In a FCNN, all neurons in the previous layer are connected to each neuron in the next layer, while no neuron within the same layer is connected (Karpathy (2019)).

The computation done by a single neuron is illustrated by Figure 3.3b and can be mathematically expressed as

$$a_j^l = \sigma\left(\sum_{k=1}^{n}(w_{j,k}^l x_i^{l-1}) + b_j^l\right) \tag{3.4}$$

where x_i^{l-1} is the input, from layer *l-1*, to the node and is multiplied with the weight parameter $w_{j,i}^l$, in layer *l*. All products of inputs and weights are summed over all n nodes in layer *l-1*, then added a bias b_j^l before being sent through an activation function σ. The output a_j^l is the value which is passed on to the next layer in the network.

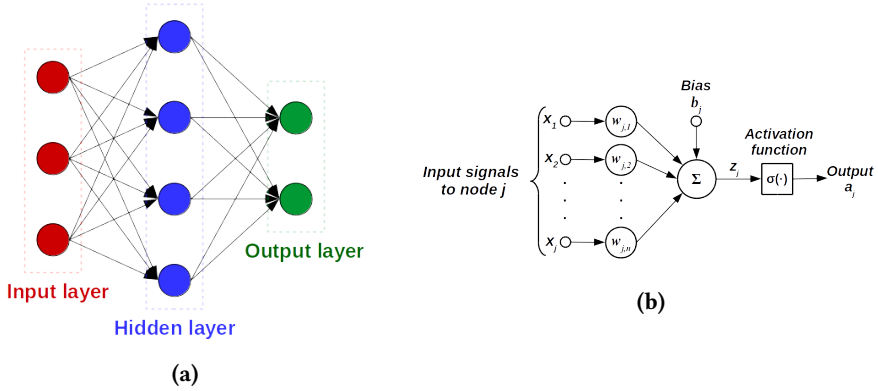


Figure 3.3: (a) Fully Connected Neural Network. (b) Single neuron computation.

The single node operation from (3.4) can be expanded to calculate the activation for the whole layer with the use of matrices.

$$\mathbf{z}^{(L)} = \mathbf{w}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)} \tag{3.5a}$$

$$\mathbf{a}^{(L)} = \sigma(\mathbf{z}^{(L)}) \tag{3.5b}$$

where the matrices has the following dimensions: weight matrix $\mathbf{w}^{(L)} \in \mathbb{R}^{j\times k}$, bias vector $\mathbf{b}^{(L)} \in \mathbb{R}^{j\times 1}$, and output vector $\mathbf{a}^{(L)} \in \mathbb{R}^{j\times 1}$.

The goal when training an ANN is to minimize the given loss function, C. The most common loss function is a quadratic function of the difference between the output $\mathbf{a}^{(L)}$ and target vector, $\mathbf{y} \in \mathbb{R}^{j\times 1}$:

$$C = \frac{1}{2}||\mathbf{a}^{(L)} - \mathbf{y}||^2 \tag{3.6}$$

The minimization of the cost function (3.6) is done with backpropagation, an optimization algorithm which updates the trainable weights $\mathbf{w}^{(L)}$ and biases $\mathbf{b}^{(L)}$ using the corresponding gradients of the loss function, $\nabla_w C$ and $\nabla_b C$. This algorithm is also referred to as gradient descent.

$$\mathbf{w}^{(L)} = \mathbf{w}^{(L)} - \alpha\nabla_w C \tag{3.7a}$$

$$\mathbf{b}^{(L)} = \mathbf{b}^{(L)} - \alpha\nabla_b C \tag{3.7b}$$

where $\mathbf{w}^{(L)}$ and $\mathbf{b}^{(L)}$ are the trainable weights and biases in layer L, and the parameter α is the learning rate controlling the step size for each iteration. The learning rate is a tuning parameter and has to be carefully selected. If the step size is too large, the changes in the weights and biases might be too large and will never reach the local minima. On the other hand, if the learning rate is too small, the learning process will

become very slow. $\nabla_w C$ and $\nabla_b C$ are the gradients of the loss function with respect to the weights and biases respectively. They are calculated using chain rules:

$$\frac{\partial C}{\partial \mathbf{w}^{(L)}} = \frac{\partial C}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L)}} \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{w}^{(L)}} \tag{3.8a}$$

$$\frac{\partial C}{\partial \mathbf{b}^{(L)}} = \frac{\partial C}{\partial \mathbf{a}^{(L)}} \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L)}} \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{b}^{(L)}} \tag{3.8b}$$

where the partial derivatives are calculated as follows:

$$\frac{\partial C}{\partial \mathbf{a}^{(L)}} = (\mathbf{a}^{(L)} - \mathbf{y}) \tag{3.9a}$$

$$\frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L)}} = \sigma'(\mathbf{z}^{(L)}) \tag{3.9b}$$

$$\frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{w}^{(L)}} = \mathbf{a}^{(L-1)} \tag{3.9c}$$

$$\frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{b}^{(L)}} = 1 \tag{3.9d}$$

Stochastic gradient descent (SGD)(Bottou (2012)) is a variant of gradient descent. As the network grows larger, the calculation of gradient descent becomes inefficient and computationally expensive. SGD solves this by dividing the training data into smaller subsets and then randomly selecting samples. Another optimization algorithm is Adaptive Moment Estimation (Adam)(Kingma and Ba (2014)), which further builds on SGD by introducing adaptive learning rates for the different parameters.

Activation Functions

The activation function helps to bound the output from the neuron, which can be anything in the range -inf to +inf (Karpathy (2019)). The activation function helps the neuron decide whether it should send its value to the next neuron or not. An example of an activation function commonly used is the Sigmoid function. This function returns a value between 0 and 1, has a smooth curve, and is very useful if the output is represented as a probability. Another popular activation function is the hyperbolic tangent function, which outputs a value between -1 and 1. The tanh function has the nice property that zero on the input gives zero at the output in addition to create good gradients for values around zero.

Sigmoid and tanh both have a problem with saturation for large negative or positive values. This introduces the problem of vanishing gradients when the networks consist of multiple hidden layers. This means that the gradient of the first layers becomes

very small, making the changes made to the weights minimal. A solution to this is the Rectified Linear Unit (ReLU), where the output is equal to the input if the input is positive and 0 otherwise. The common approach for training multi-layered networks is to use the ReLU-activation function on the hidden layers and then Sigmoid or tanh on the output layer.

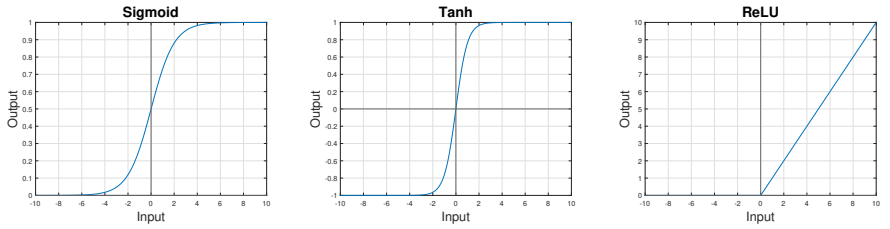


Figure 3.4: Illustration of the Sigmoid, tanh and ReLU activation functions.

3.3 Deep Learning

Deep learning introduces the use of multi-layered networks. By constructing multiple hidden layers in the networks, one is able to increase the performance and accuracy of the machine-learning algorithms. In addition to learning what output to predict given an input, a deep network also understands basic features of the input. The breakthrough of deep learning came alongside with the development of the ReLU activation function, which gave a solution to the problem of vanishing gradients. This allowed for the use of gradient descent methods to efficiently train multi-layered networks.

3.4 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) introduces the use of Artificial Neural Networks (ANNs) for predictions instead of using constant matrices determined during training. The early forms of reinforcement learning were limited to domains where the state spaces were low-dimensional and fully observable. This yielded results in some application, but they were dependant on that useful features were handcrafted. This changed when the field of deep neural networks managed to develop a working artificial agent, today known as the deep Q-network (DQN). This introduced a new type of network structure where end-to-end reinforcement learning could be used to learn policies from high-dimensional sensory inputs.

An implementation of a deep Q-network was introduced by Mnih et al. (2015), based on the Q-learning algorithm from Watkins and Dayan (1992). This agent was tested on the

domain of classic Atari 2600 games, using only the pixels and game scores as input. The agent was applied to 49 different games with the same algorithm, network architecture, and hyperparameters. This resulted in scores that exceeded all preceding algorithms and could be compared with a professional game tester.

The development of DQN was a big step in the right direction regarding autonomous control in high-dimensional observation spaces. However, the algorithm isn't able to handle anything other than discrete, low-dimensional action spaces. When working with real-world applications, such as physical control tasks, one has to handle continuous and high-dimensional action spaces. In order to apply DQN to such applications, the action spaces have to be discretized. This introduces a problem due to the high dimension of the discretized action space. The number of actions would increase exponentially with the degrees of freedom in the system. Another problem occurs if the discretization is executed poorly, which can lead to a sparse representation of the action domain.

Actor-Critic

Reinforcement learning is commonly divided into two different methods: value-based and policy-based. Value-based methods rely on finding a suitable value function that assigns each state-action pair a value. After training, the agent will have a look-up table containing the best action for each state based on the largest value found during training. Policy-based methods do not use a value function to find the optimal policy, but instead designs the policy based on the reward received at the end of an episode.

The Actor-Critic architecture utilizes the advantages of both methods by using a value-based critic to criticize the action chosen by the policy-based actor. This allows for updating the policy at each time step instead of only at the end of each episode. The evaluation given by the critic to the actor is described by the Temporal Difference (TD) error:

$$\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t) \tag{3.10}$$

where V_t is the value function from the critic at time t in the current state S_t and next state S_{t+1}. This relationship is illustrated in Figure 3.5. Actor-Critics methods remains popular because they require minimal computation before selecting an action and they are able to learn an explicitly stochastic policy (Sutton and Barto (2015)).

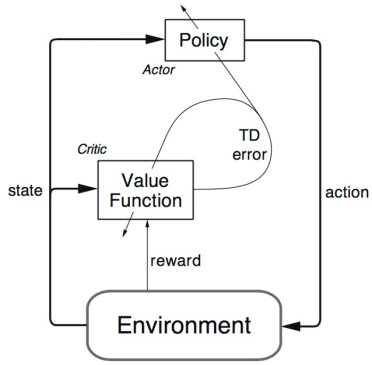


Figure 3.5: Actor-Critic illustration. Image courtesy of Sutton and Barto (2015).

Target Networks

Target networks are copies of the actor and critic networks. The weights of the target networks are constrained to track the weights of the main networks slowly. During training, the actions are predicted by the main networks, while the learning and calculation of losses are based on the target networks. This type of architecture has shown to increase stability during training.

Replay Buffer

A replay buffer is a fixed size buffer where previous experiences are stored. During training, the newest experiences replace the oldest ones in the buffer. The network can then sample a random batch from this buffer during training, often referred to as experience replay. This way of learning from previous experience enables the possibility to learn from individual experiences multiple times and recall rare occurrences. This way, damaging correlations are avoided, and the agent is allowed to train on multiple samples simultaneously.

Deep Deterministic Policy Gradient

The work of Lillicrap et al. (2015) presents an algorithm that handles the problem of continuous action spaces, called Deep Deterministic Policy Gradient(DDPG). The DDPG algorithm is a model-free, off-policy, actor-critic algorithm where the agent learns policies using deep function approximators. The DDPG algorithm builds on the results of Silver et al. (2014), where it was proven that the deterministic policy gradient is the expected gradient of the action-value function. These findings make the estimation of the policy gradient more efficient than it would be with its stochastic counterpart. The

DDPG algorithm was tested on over 20 physical control tasks and showed results that can be compared to a state-of-the-art planning algorithm.

The DDPG algorithm consists of four networks; critic Q(s, a|θ^Q), actor μ(s|θ^μ) and their respective target networks Q' and μ'. The DDPG algorithm also implements a replay buffer $\mathcal{R}$ in addition to the target networks. Action noise is applied in order to ensure sufficient exploration by the agent. The noise process $\mathcal{N}$ used in the original algorithm was an Ornstein-Uhlenbeck process (Uhlenbeck and Ornstein (1930)), and is added to the action from the actor network at each time step before being executed. After the action a_t is executed, the observed reward r_t and new state s_{t+1} are stored in the replay buffer $\mathcal{R}$ in addition to the previous state s_t and action a_t.

The next step of the algorithm is to take a random mini-batch sample from the replay buffer and use this to update the critic network. The critic network is updated by minimizing the loss function L, which is the average squared difference between y_i and the estimated reward value for the critic in state s_i and action a_i, for each sample in the mini-batch. The estimated long-term reward, y_i, is the result of adding the observed reward r_i, given state s_i and action a_i, and the discounted estimate given by the target critic network based on the action proposed by the target actor network in the next state s_{i+1}.

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{3.11a}$$

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \tag{3.11b}$$

The actor policy is then updated using the sampled policy gradient. This gradient is the average of the product of the gradients of the critic's value estimate and the actor's weight gradients.

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i} \tag{3.12}$$

The last step of the algorithm is to slowly update the weights of the target networks to track the learned networks, based on the constant $\tau \ll 1$. This way of constraining the target values to change slowly improves the stability of learning.

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \tag{3.13a}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \tag{3.13b}$$

This process is repeated for M number of episodes with T number of states. Pseudo

code for the DDPG algorithm is shown in Algorithm 1.

Algorithm 1 DDPG algorithm, as proposed by Lillicrap et al. (2015).

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

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Proximal Policy Optimization

The Deep Deterministic Policy Gradient updates the policy by taking a step in the direction of the steepest ascent in expected reward. However, a problem occurs when it comes to deciding the size of the step. Training time decreases as the step becomes larger, but if the step becomes too large, it can prevent the policy from converging. As a worst case, the policy might have a bad update which it can't recover from in later updates. This problem can be avoided by using a small learning rate for the agent, but this is not ideal as it will slow down the convergence.

The work of Schulman et al. (2017) proposes a new family of policy gradient methods aiming to create the largest possible step, with guaranteed improvement. The algorithm is called Proximal Policy Optimization (PPO) and is an improvement of the Trust Region

Policy Optimization (TRPO) by Schulman et al. (2015b). The PPO algorithm keeps some of the benefits from TRPO, as efficiency and reliable performance, but improves by being simpler to implement and more general by using only first-order optimization.

TRPO updates its policy by maximizing the objective function:

$$L^{\text{CPI}}(s, a, \theta_k, \theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a) \right] = \hat{\mathbb{E}}_t \left[r_t(s, a, \theta_k, \theta) A^{\pi_{\theta_k}}(s, a) \right] \tag{3.15}$$

where s and a is the state-action pair, θ_k is the previously saved policy parameter vector and θ is the current policy parameter vector. The policy $\pi_\theta(a, s)$ is parameterized with the parameter vector θ and $r_t(s, a, \theta_k, \theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$ is the probability ratio, comparing the new policy with the old. This gives $r_t(s, a, \theta_k, \theta) = 1$ if $\theta = \theta_k$. The expectation $\hat{\mathbb{E}}_t[...]$ indicates the empirical average over a finite batch of samples.

The advantage function $A^\pi(s, a)$ in (3.15) is a variant of a generalised advantage estimation (GAE) function (Schulman et al. (2015a)), which estimates the advantage as the discounted sum of TD errors:

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{3.16}$$

where δ_t^V is the TD error (3.10) of the value function V and is an estimate of the advantage of the action a_t. γ is the discount factor and $\lambda \in (0, 1)$ is used to control the trade-off between bias and variance in the generalized advantage estimator.

The advantage function used by Schulman et al. (2017) in the PPO-algorithm is a variation of (3.16) which is well suited for policy gradient implementations and is denoted as:

$$A^\pi(s, a) = -V(s_t) + r_t + \gamma r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T) \tag{3.17}$$

where T is the number of time steps which the policy is ran sampling a trajectory segment. The variable t specifies the time index in $[0, T]$ within a given length-T trajectory segment, and γ is the discount factor.

The maximization of L^{CPI} needs to be constrained otherwise the policy updates will be too large. PPO solves this by modifying the objective such that changes that move the probability ratio, $r_t(s, a, \theta_k, \theta)$, away from 1 is penalized.

$$L^{\text{CLIP}}(s, a, \theta_k, \theta) = \hat{\mathbb{E}}_t \left[min \left(L^{\text{CPI}}, clip \left[r_t(s, a, \theta_k, \theta), 1 - \epsilon, 1 + \epsilon \right] A^{\pi_{\theta_k}}(s, a) \right) \right] \tag{3.18}$$

where the first term in the *min* is L^{CPI} from (3.15) and the second term, $clip(...)A^{\pi_{\theta_k}}$, modifies the surrogate objective by clipping the probability ratio, which keeps r_t inside the interval of $[1-\epsilon, 1+\epsilon]$. Achiam and Abbeel (2018) simplified (3.18) in order to better explain the *clip()*-functionality.

$$L^{\text{CLIP}}(s, a, \theta_k, \theta) = \hat{\mathbb{E}}_t \left[min \left(L^{\text{CPI}}(s, a, \theta_k, \theta), g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right) \right] \tag{3.19a}$$

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A, & A \geq 0 \\ (1-\epsilon)A, & A < 0 \end{cases} \tag{3.19b}$$

For each iteration in the PPO algorithm, N actors collects experience for T time steps by applying the policy π_{θ_k} and observing the resulting rewards and next states. After T time steps, the advantage estimates $A^{\pi_{\theta_k}}$ are calculated and used in combination with the sampled data from the N agents to acquire the objective function L^{CLIP}. The policy is then updated with the expression:

$$\theta_{k+1} = \underset{\theta}{argmax} \underset{s,a \sim \pi_{\theta_k}}{\mathbb{E}} \left[L^{CLIP}(s, a, \theta_k, \theta) \right] \tag{3.20}$$

The pseudo code for the PPO algorithm is found in Algorithm 2.

Algorithm 2 PPO algorithm, as proposed by Schulman et al. (2017)

for iteration=1, 2,... **do**
  for actor=1, 2,..., N **do**
    Run policy π_{θ_k} in environment for T timesteps
    Compute advantage estimates $A_1^{\pi_{\theta_k}}$,..., $A_T^{\pi_{\theta_k}}$
  end for
  Optimize surrogate L^{CLIP} w.r.t. θ, with K epochs and minibatch size $M \leq NT$
  Update the policy parameter vector by maximizing the PPO-Clip objective
  $\theta_k \leftarrow \theta_{k+1}$
end for

Off-policy vs On-policy

The work of this thesis applies two different DRL algorithms; Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO). Both algorithms are applicable in continuous state and action space, model-free, and online algorithms. However, they differ in DDPG being off-policy and PPO being on-policy. An off-policy algorithm is able to learn from experience from previously learned policies, while on-policy algorithms only learn based on experience from the current policy. Off-policy algorithms tend to have a slower convergence rate, but is less likely to converge to a bad optimum as on-policy algorithms can be prone to. The off-policy property might be useful in a real-world application since experience from a simulator can be utilized.

Safety in Artificial Intelligence

One of the biggest concerns regarding the use of artificial intelligence is how to make sure the system will not cause an accident or cause harm to humans. An accident can be described as an unintended and harmful behavior due to poor design choices. Dikmen and Burns (2017) performed a study on humans' trust in autonomous vehicles. The study showed that trust is highly dependant on the system to make safe and predictable decisions. The paper from Amodei et al. (2016) presents five problems related to accident risk and proposes ways to avoid them. The problems are categorized into three sub-groups: the problem happens due to having the wrong objective function, the objective function cannot be evaluated at a high enough rate due to high complexity, and unwanted behavior while training.

- **Avoiding Negative Side Effects:** The developer has to make sure that the agent will not cause any harm or disturb the environment while reaching its objective. The objective function will need to penalize the agent if any unwanted alterations to the environment are made.
- **Avoiding Reward Hacking:** The developer has to ensure that the agent cannot cheat the reward function by performing "illegal" actions that utilize flaws in the reward function to maximize the reward.
- **Scalable Oversight:** The agent would need to be able to evaluate the expensive aspects of the objective even though they are rarely visited.
- **Safe Exploration:** While exploring, the agent will have to be limited not to execute actions that could damage the agent. These dangerous situations could be hard-coded in a collision avoidance algorithm, but in complex domains, this

might not be that simple.

- **Robustness to Distributional Shift:** The developer has to ensure that the agent will have a robust behavior even though the environment is different from the one in which the agent was trained.

For real-world applications, the problem of safe exploration is critical. While exploring in a simulated environment, a bad action will only affect the score negatively, but in the real world it can damage the environment or the agent. The work of Hans et al. (2008) presents a method for safe exploration containing a safety function and a backup policy. The safety function returns a value for the state's degree of safety while the backup policy returns the system back to a safe state.

3.5 Summary

This chapter has presented the theory needed in order to understand the basics about reinforcement learning in addition to how a machine-learning algorithm can be implemented for a system. An overview of the Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO) has been given.

Chapter 4

Software and Hardware Choices

This chapter will summarize the software and hardware choices for this thesis. Four requirement specifications for the docking system were defined in Section 1.6. These requirements considered the limited space on the USV in addition to creating demands on the software implementation.

4.1 Software

The software choices were made based on the third requirement specification, **R3**, where an object-oriented programming language is desired. The Otter USV is a versatile platform where multiple sensor packages can be deployed at once. By utilizing an object-oriented programming language, the docking system becomes more modular and easier to implement with the existing systems. The requirement also states that the machine-learning implementation should be based on open-source libraries. It's not desirable to spend time on implementing machine-learning algorithms when it already exists multiple solutions with large communities and continuous development. Time can instead be spent on adapting the dynamical model to fit an existing machine learning implementation. The software was developed using Ubuntu 16.04.

Python

Python is a high-level, object-oriented programming language with many extensions and toolboxes. It offers concise and readable code and is platform-independent, meaning that a program developed on one machine can run on other machines with minimal changes. Python has grown to be the preferred language for machine learning, resulting in a wast selection of packages and resource repositories. It is also surrounded by a large community that constantly updates and debugs the existing frameworks.

OpenAI's Stable Baseline

Stable Baselines (Hill et al. (2018)) is a GitHub repository from OpenAI developed for Python. The repository contains a large set of reinforcement algorithms in addition to well documented functions and classes. This allows for easy setup and experimentation with an advanced toolset without spending time on implementation details.

The framework from OpenAI requires the following packages:

- Python3 ($\geq$ 3.5)
- Numpy 1.16.4
- Tensorflow-gpu 1.14.0
- Cmake
- OpenMPI

OpenAI's Gym Environments

Gym (Brockman et al. (2016)) is a toolbox for reinforcement-learning environments. Gym contains a large set of already made environments, but it also allows for making custom environments. Once the environment is made one can start to develop and compare different reinforcement-learning algorithms on the environment.

CUDA

CUDA is a parallel processing platform developed by Nvidia which utilizes the computers Graphical Processing Unit (GPU). When training neural networks, this parallel processing is highly desirable in order to speed up the training.

4.2 Hardware

This thesis will only develop a software solution for the autonomous docking system and not implement any hardware. However, it is desirable to make the simulations as realistic as possible. To achieve this, some suitable hardware was investigated to find correct performance values for the simulator. Therefore, this section can be reviewed as a proposal for possible hardware choices that are suitable for a real-world application. The following hardware was selected in order to comply with the fourth requirement specification, **R4**, while still providing enough processing power to handle the docking system.

Embedded computer platform

A docking system like this requires a large amount of computing power. The computer must be able to handle image processing from the positioning sensor, input from the DVL sensor, and run the trained network for the docking controller, in real-time. Sufficient GPU power is also required to be able to further train the docking controller online in a real-world application. The computer's size is also limited by the platform of the USV, making this a challenging task. The Jetson family of embedded platforms from Nvidia (2020) provides a solution for this. These developer kits are compact modules made especially for AI development. They provide CUDA supported GPUs and CPUs with sufficient processing power. The board only requires 19V supply voltage and has a power consumption of only 10W. Figure 4.1 illustrates the Jetson Xavier, the latest version from Nvidia, with a footprint of only 103×90.5 mm.

Figure 4.1: Jetson Xavier NX Developer Kit. Image courtesy of Nvidia (2020).

Stereo vision camera

Optical cameras were presented in Section 2.3 as a preferred method for position estimation in confined areas. There exist multiple providers of camera systems, varying from only providing image output (e.g., FLIR (2020)) to delivering a complete package with positional tracking and object detection. One commercially available solution which provides the latter is the Zed 2 stereo camera from Stereo Labs (2020), see Figure 4.2. The camera uses neural networks to provide high precision 6-DOF localization and mapping. With a size of only 175×30 mm and a data rate of 25 Hz, the Zed 2 presents itself as a valid choice for this application.

Figure 4.2: Zed 2, stereo camera. Image courtesy of Stereo Labs (2020).

DVL ocean current estimator

Commercial DVLs has previously been rather expensive and not accessible to small research teams (Rudolph and Wilson (2012)). However, a new product from Water Linked (2020) has made DVLs more accessible and affordable for developers. The size of the unit is considerably smaller than its competitors on the market, making it ideal for research and development on small ROVs/USVs. Figure 4.3 show the size comparison of the DVLs from Water Linked (2020) and two of its competitors, Teledyne Marine (2020) and Nortek (2020), here mounted on a BlueROV2 from Blue Robotics (2020).

(a) **(b)** **(c)**

Figure 4.3: Size comparison of commercial DVLs. (a) Teledyne WHN1200
(b) Nortek DVL1000 (c) Waterlinked DVL A50. Image courtesy of Water Linked (2020).

Chapter 5

Design and Implementation

This chapter presents an overview of the implemented system. The chapter is divided into sections as follows:

- Section 5.1 presents the environment developed for the Otter USV.
- Section 5.2 presents the state and action vector.
- section 5.3 presents the reward function and termination parameters.
- Section 5.4 presents the parameters used for DDPG and PPO.
- Section 5.5 presents the sensor implementation used.

5.1 Machine Learning Environment

A custom environment for the Otter USV was needed in order to use reinforcement learning to develop a controller for autonomous docking. The dynamical model presented in Chapter 2 was implemented in Python and used to simulate the dynamics of the USV in the environment. The environment was developed based on OpenAI's toolkit called Gym (Brockman et al. (2016)). The compatibility with Gym was necessary in order to be able to use the DDPG and PPO algorithms implemented by OpenAI. The Gym environment consists of a class with four functions: the initialization function, the step function, the reset function, and the render function.

- **The initialization function** is called once at the beginning of the training session. The function takes no additional parameters and sets the initial states of the USV. Training parameters such as maximum allowed distance from the port, desired goal distance from the port, minimum and maximum thruster output, and the maximum number of steps are determined in this function. The action space and observation space are also initialized here.
- **The step function** is called at each time step. This function takes only the action predicted by the neural network as input. The action is sent to the Otter dynamical model to get the updated states from the USV. These states are then used to calculate the accumulated penalty/reward. If the states are within the

desired values, then a boolean value is returned, representing a finished episode. The step function returns a list of four parameters: the next states, the reward accumulated in the step, the boolean value indicating if the episode is done, and a string containing additional information about the status of the environment.

- **The reset function** is called at the end of an episode as long as the training isn't finished. This function takes no inputs and resets the USV's states back to a randomized starting position. This is important in order to make sure that the USV experiences the whole environment. The reset function returns the new initialized states.
- **The render function** is a visualization of the USV's behavior. A graphical interface is used to verify that the trained model behaves as expected.

The predicted action from the neural network is interpreted as input values to the two fixed thrusters. The sequence diagram in Figure 5.2 illustrates how the training is executed and how each of the class functions communicates.

The simulator constructed for the render function can be seen in Figure 5.1. A simple 2D model was created for this thesis since only horizontal movement is of interest. This allows the user to get a visualization of how the USV behaves with the trained model. The initial position and heading of the USV can be selected by the user. The ocean current velocity and angle are illustrated in the lower-left corner.

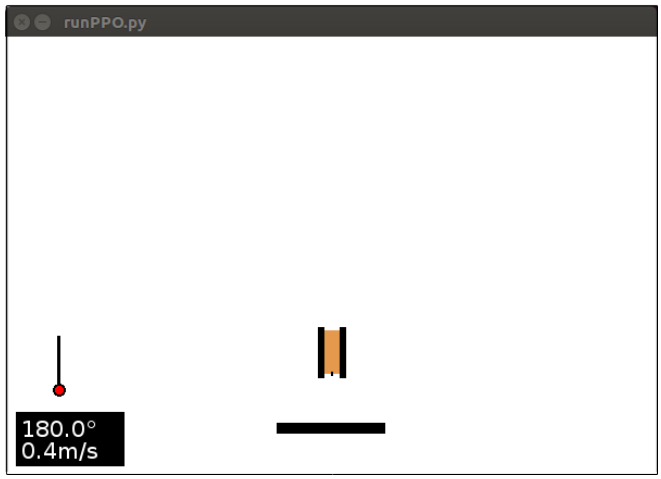


Figure 5.1: The simulator for the machine-learning environment

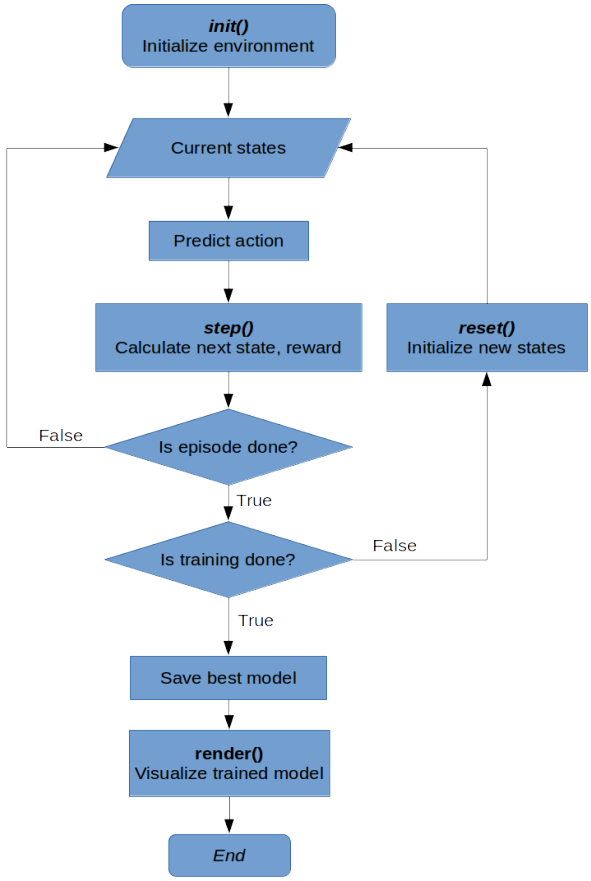


Figure 5.2: A sequence diagram for the training of the reinforcement learning model.

5.2 Observation and Action Space

An important part of developing a machine learning environment is to define which states and actions are needed to achieve a successful model. The observation space contains a vector of states which is chosen in order to explain the behavior and dynamics of the USV to the machine learning agent. This state vector is used as input to the machine learning model, which calculates the correct action vector. This action vector is then applied to the actuators on the USV.

5.2.1 Observation Space

The design of the observation space is important in order to achieve an agent that learns the desired control objective. The agent uses a combination of observed states and accumulated rewards to optimize the objective. The correct selection of states has to be found by trial and error since both too little, and too much information will lead to the agent not understanding the essential aspects of its objective and produce an unsuccessful policy. The selection of states also has to consider the design choices in the reward function. By having the agent observe all the states which are rewarded, it is able to faster see the correlation between the given reward and state-action pair.

- $(\tilde{x}, \tilde{y}, \tilde{\psi})$: Cartesian and angular distance to desired position in NED frame.
- (u, v, r): surge, sway and angular rate in body frame.
- e_d: Euclidean distance using the relative position in NED frame.
- n: Episode step number
- (V_c, β_c): Ocean current velocity and angle in NED frame.

The states $\tilde{x}, \tilde{y}, \tilde{\psi}$ and, e_d are states which describes the USV's pose related to the desired docking position. The step number, n, is observed in order to penalize the time spent by the USV to dock successfully. The linear and rotational velocities u, v, and r doesn't contribute to describing the position of the vessel, but is needed to help the agent understand the dynamics of the USV. The ocean current velocity and angle, V_c and β_c, are added to the state vector when the disturbance is measured and assumed known to the agent. The states are normalized to obtain a mean of zero and fit in the range [-1,1]. Since the neural networks use tanh as activation function, this normalization will help speed up training and lead to faster convergence (Stöttner (2019)).

State vector 1

The first state vector constructed was used for the initial training and testing of the network model. This vector contains all the information needed for the agent to know the relative position of the USV $(\tilde{x}, \tilde{y}, \tilde{\psi}, e_d)$, the dynamics of the USV (u, v, r) and the time spent on the episode, n. However, the state vector does not supply the agent with any information about the ocean-current disturbances. This results in the following state vector:

$$\mathbf{x}_1 = \left[\tilde{x}, \tilde{y}, \tilde{\psi}, u, v, r, e_d, n\right] \tag{5.1}$$

State vector 2, with ocean current

The second state vector was used for the training sessions where the ocean current is assumed known to the agent. The measured ocean current velocity, V_c, and angle, β_c, are added to the first state vector in (5.1). By making these measurements available to the agent, this information can be used to create a model that correctly controls the USV while taking the disturbance into account.

$$\mathbf{x}_2 = \left[\tilde{x}, \tilde{y}, \tilde{\psi}, u, v, r, e_d, n, V_c, \beta_c\right] \tag{5.2}$$

5.2.2 Action Space

The USV-model has two fixed actuators in the rear. The action space constructed such that the agent has full accessibility to the actuators. The action vector is therefore chosen as follows:

$$\mathbf{a} = \left[n_{\text{left}}, n_{\text{right}}\right] \in [-101.737, 103.931] \text{ rad/s} \tag{5.3}$$

5.3 Reward Function

A considerable amount of time has been spent on developing the reward function for the docking controller. The goal is to develop a reward function that finds the optimal path from the starting position outside the dock and guides the USV to the desired docking position with a safe surge speed. Since the USV is underactuated, it cannot move directly sideways, which makes dynamic positioning almost impossible. Therefore, in order to safely dock the USV, it has to have the correct heading before reaching the desired docking position. In addition, the USV is allowed to hit the dock with a low constant surge speed in order to keep the USV still after impact.

The following parameters were chosen for the reward function:

- Euclidean distance, e_{d}
- Differentiated euclidean distance, $\dot{e}_{\mathrm{d}}$
- Heading error, e_{ψ}
- Surge speed, u
- Step number, n
- Action rate, $\dot{\mathbf{a}}$

One approach could be to reward the agent only when the correct position and orientation are reached. However, this sudden step in reward will result in a sparse reward function, which makes the convergence time longer. A better approach is to give the agent more reward as it approaches the desired states. This approach is called reward shaping (Grzes and Kudenko (2008)) and will significantly shorten the convergence time. An efficient way to create reward shaping is to use Gaussian functions. These functions provide good convergence properties since the maximum reward is given when the error is zero (Martinsen and Lekkas (2018)). The difference between a step function and a Gaussian function is illustrated in Figure 5.3.

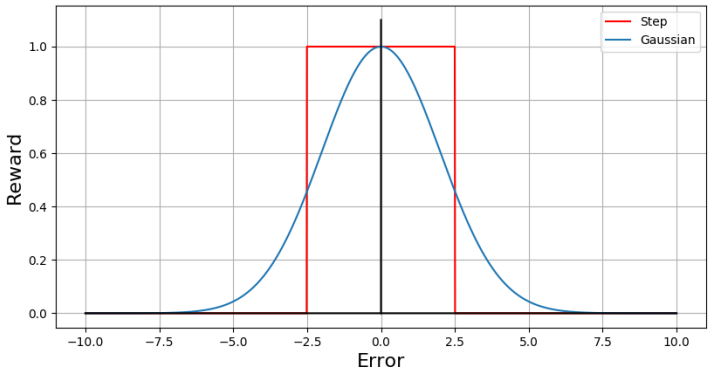


Figure 5.3: Gaussian reward compared to step reward.

5.3.1 Position reward

The first part of the reward function is based on the position of the USV using the euclidean distance. This will reward the USV as it approaches the desired position. The euclidean distance from the desired position to the vessel is calculated as follows:

$$e_d = \sqrt{(x_d - x)^2 + (y_d - y)^2} \tag{5.4}$$

The position reward is defined such that the agent only receives reward if the heading error, e_ψ, is smaller than $\frac{\pi}{2}$, in order to only reward the USV if it is facing towards the dock. In addition, the change of position, $\dot{e}_d$ must be smaller than zero, meaning that the vessel is in motion towards the dock. This results in the following reward using the calculated euclidean distance in (5.4):

$$r_{e_d} = \begin{cases} C_{e_d} e^{-\frac{e_d^2}{2\sigma_{e_d}^2}}, & \text{if } e_\psi \leq \frac{\pi}{2} \text{ and } \dot{e}_d < 0 \\ 0, & \text{otherwise} \end{cases} \tag{5.5}$$

where $C_{e_d} > 0$ is the amplitude of the reward and $\sigma_{e_d} > 0$ is the standard deviation, both are constants. The position reward, r_{e_d}, in (5.5) is illustrated in Figure 5.4.

Figure 5.4: Position reward, r_{e_d}, with $C_{e_d} = 2.0$ and $\sigma_{e_d} = 2.5$.

In Figure 5.5 the position reward, r_{e_d}, is illustrated in the NED-frame, within the boundaries defined at $x \in [-2, 10]$ and $y \in [-5, 5]$. This show how the reward is zero when the USV is at the boundaries and grow larger as the USV approaches the dock at position (x_d, y_d) = (0,0).

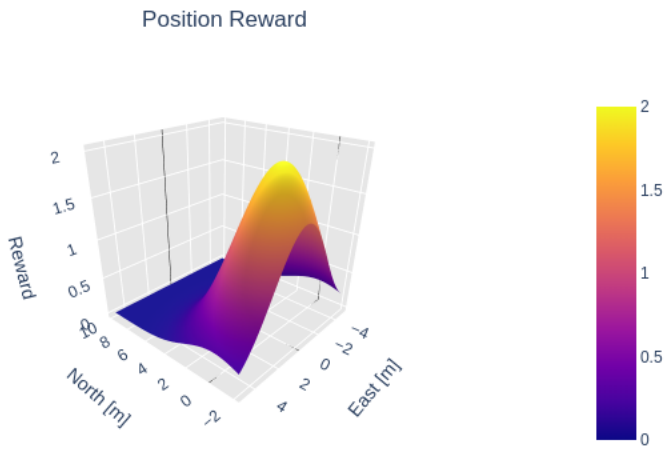


Figure 5.5: Position reward, r_{e_d}, related to position in NED.

5.3.2 Differentiated euclidean distance

Only rewarding the position is not enough for the agent to learn how to perform a successful docking maneuver. It is also important to reward the USV for constantly decreasing its distance to the desired position. This is done by calculating the change of position using numerical differentiation:

$$\dot{e}_d = \frac{e_{d,k} - e_{d,k-1}}{h} \tag{5.6}$$

where $e_{d,k}$ is the current distance, $e_{d,k-1}$ is the distance at the previous time step and h is the simulator step size.

The distance rate is used to penalize the USV if it stops moving towards the desired position. By not rewarding the USV for moving towards the dock, the agent's movement isn't constrained by this function. Instead, it will inform the agent when it is moving in the wrong direction. Therefore, the agent can explore and find the optimal path towards the dock and not be penalized as long the distance to the dock decreases. In order to assure a smooth function at the transition from 0 to -1, the tanh function is used. The reward function for distance rate is denoted as follows:

$$r_{\dot{e}_d} = -\frac{C_{\dot{e}_d}}{2}(\tanh(K\dot{e}_d) + 1) \tag{5.7}$$

where $C_{\dot{e}_d} > 0$ is the amplitude of the penalty and $K > 0$ is used to decide the gradient of the transition from 0 to -1. The reward function in (5.7) is illustrated in Figure 5.6.

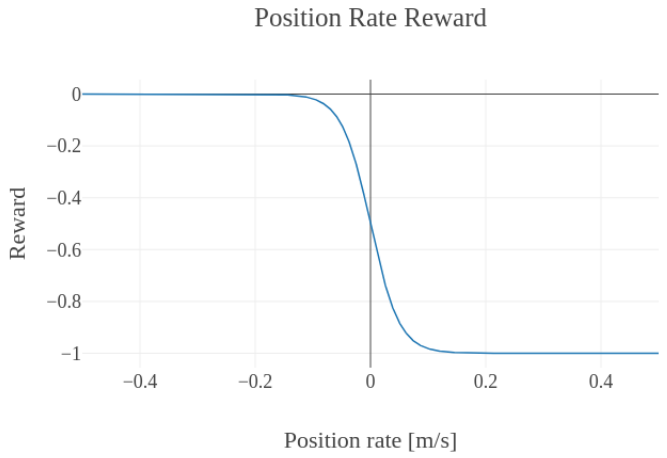


Figure 5.6: Position rate reward, $r_{\dot{e}_d}$, with $C_{\dot{e}_d} = 1.0$ and $K = 20$.

5.3.3 Heading reward

Rewarding the correct heading is important when approaching the dock since the USV is underactuated. Since the vessel cannot move directly sideways, the heading has to be correct before approaching the desired position. However, it is not desirable to restrict the heading of the vessel when far from the dock. This will not allow the USV to correct its position and orientation in order to dock successfully. Therefore, the heading is only rewarded when the position error, e_d, is smaller than 3.0 meters. Another requirement in order to receive heading reward is that the distance rate is negative, $\dot{e}_d < 0$, which will stop the RL-agent from cheating the reward function by stopping outside the dock and accumulate reward for correct heading.

In order to create a versatile model that could be applied to different docking positions, the heading error is used for the reward. This allows for comparing the heading of the USV with the desired heading, which is defined based on the desired docking position. The heading error is calculated using the Smallest Signed Angle (SSA) function from Fossen and Perez (2004), which maps an angle to the interval $[-\pi, \pi)$:

$$e_\psi = \mathrm{SSA}(\psi_\mathrm{d} - \psi) \tag{5.8a}$$

$$\mathrm{SSA}(\phi) = \mathrm{modulo}(\phi + \pi, 2\pi) - \pi \tag{5.8b}$$

where ψ is the heading of the USV in NED frame and ψ_d is the desired heading defined by angle perpendicular to the dock. The modulus operator returns the remainder of the division between the two inputs. The heading reward is calculated using (5.8) in the following Gaussian function.

$$r_{e_\psi} = \begin{cases} \mathrm{C}_{e_\psi} \mathrm{e}^{-\frac{{e_\psi}^2}{2{\sigma_{e_\psi}}^2}}, & e_d \leq 3.0\mathrm{m} \text{ and } \dot{e}_d < 0 \\ 0, & \text{otherwise} \end{cases} \tag{5.9}$$

where $C_{e_\psi} > 0$ is the amplitude of the reward and $\sigma_{e_\psi} > 0$ is the standard deviation, both are constants. The heading reward in (5.9) is illustrated in Figure 5.7.

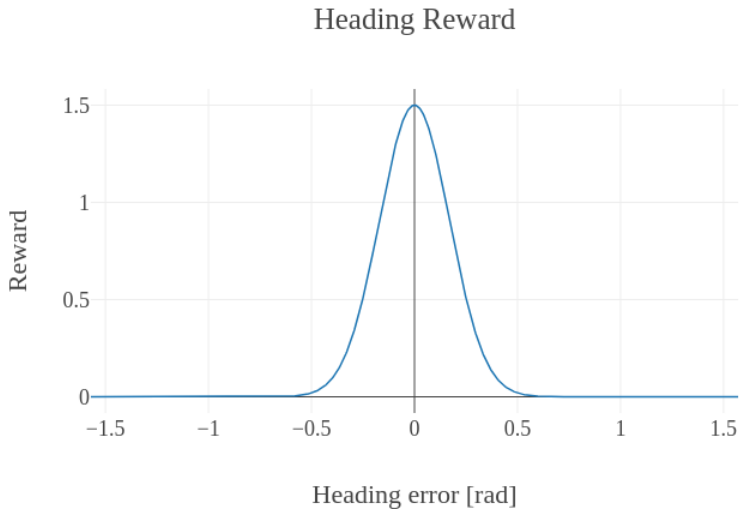


Figure 5.7: Heading reward, r_{e_ψ}, with $C_{e_\psi} = 1.5$ and $\sigma_{e_\psi} = 0.17$.

5.3.4 Surge reward

It is vital to restrict the surge speed of the USV when in the docking area. This is to assure a safe and controlled docking. At the beginning of the docking scenario, when the USV is far from the dock, it is allowed to keep a surge speed of 1.0 m/s. However, when the USV is within 3.0 meters from the dock, it is required to reduce the surge speed to 0.2 m/s, which is the desired docking surge speed.

Another requirement regarding the surge speed is to keep the USV constantly moving forwards. Therefore, it's beneficial to penalize the RL-agent when the surge speed deviates too far from the desired velocity. This results in the following reward function for surge speed.

$$r_u = (C_u + \alpha)e^{-\frac{(u-u_d)^2}{2\sigma u^2}} - \alpha \tag{5.10}$$

where $C_u > 0$ are the amplitude of the reward and σ_u is the standard deviation. The constant $\alpha > 0$ is the offset which moves the Gaussian function down such that it converges to $-\alpha$ instead of zero when the vessel deviates too far from the desired surge velocity. The desired surge velocity, u_d, changes as the vessel approaches the dock, and is defined as follows:

$$u_d = \begin{cases} 1.0\text{m/s}, & \text{if } e_d \geq 3.0m \\ 0.2\text{m/s}, & \text{otherwise} \end{cases} \tag{5.11}$$

The reward function for surge (5.10) when the USV is closer than 3.0 meters to the dock is illustrated in Figure 5.8.

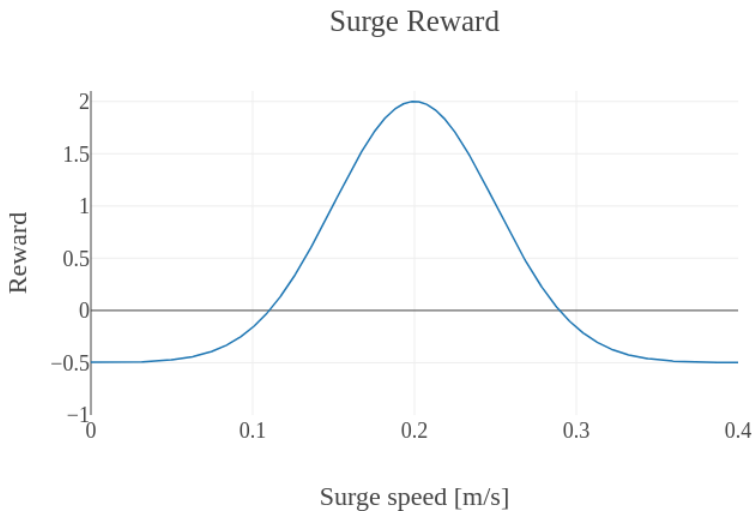


Figure 5.8: Surge reward, r_u, with $C_u = 2.0$, $\sigma_u = 0.05$, $u_d = 0.2$ m/s, and $\alpha = 0.5$.

5.3.5 Step number penalty

The step number penalty is implemented in order to control the time spent by the USV on completing the docking maneuver. In addition, the step number penalty is used to stop the USV from "cheating" the reward function. This cheating behavior occurs when the USV does not complete the episode, but instead learns how to exploit the reward given for position and surge speed. The step number penalty function first normalizes the step number, then raises it by a constant $\beta \in [1, 2)$. This results in a curve where the gradient grows larger as the step number reaches the maximum value. This will penalize the agent less at the beginning of the episode, but as time is spent, the penalty will grow larger.

$$r_n = C_n \left(\frac{n}{n_{\max}} \right)^{\beta} \tag{5.12}$$

where $C_n > 0$ is the amplitude of the penalty and $n_{max} > 0$ is the episode length. The step number penalty in (5.12) is illustrated in Figure 5.9.

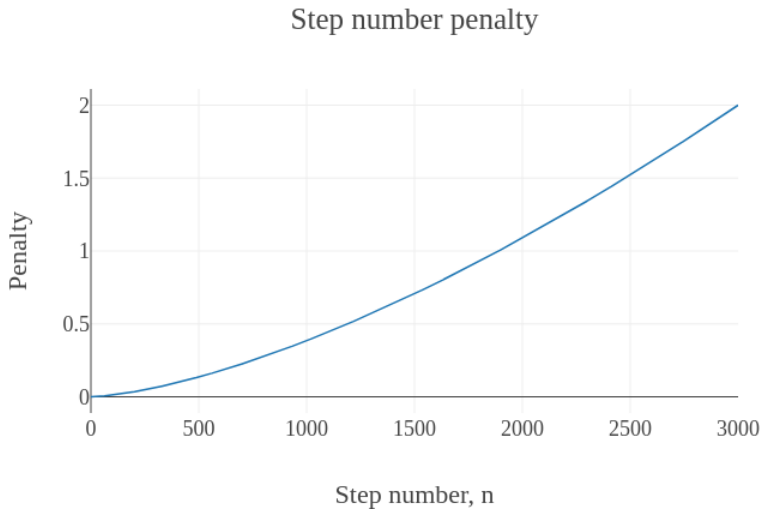


Figure 5.9: Step penalty, r_n, with $C_n = 2.0$, $\beta = 1.5$ and $n_{max} = 3000$.

5.3.6 Action penalty

When using reinforcement learning to develop a controller, one might experience an undesirable use of actuators in the optimal solution. The RL-agent does not distinguish between a solution with aggressive and less aggressive control action. From the RL-agent's perspective, the behavior of the USV might appear identical for both cases. For a real-world application, aggressive use of thruster input will introduce a lot of wear and tear on the actuators, which is not desirable. In order to reduce the use of control action, a small penalty is added to the derivative of the thruster input, $\mathbf{a}$ (5.3). This penalty should be large enough to remove the aggressive use of the actuators, but small enough to not stop the RL-agent from exploring the environment. This will result in a reward function that favors smoother and slower control action.

$$r_{\dot{a}} = C_{\dot{a}}(|\dot{n}_{left}| + |\dot{n}_{right}|) \tag{5.13}$$

where $C_{\dot{a}} > 0$ is the amplitude of the penalty.

5.3.7 Episode termination

While training the agent, each episode is either successful or unsuccessful. The termination of an episode is triggered by the USV reaching a certain state. An episode is denoted unsuccessful if one of the following two things happens:

1. **Out of bound:** The RL-agent steers the USV outside of the defined boundaries of the environment.
2. **Step limit reached:** The step number reaches the maximum allowed value, resulting in the episode timing out; $n \geq n_{\max}$

An episode is denoted successful if the RL-agent is able to dock the USV. The quality of a docking operation is quantified based on a set of requirements. The USV needs to have a position, heading, and surge velocity within a certain range in order to a docking to be accepted. The four following requirements have to be fulfilled in order to terminate an episode:

1. **Correct distance from dock:** The euclidean distance, e_d, from (5.4) is smaller than the required value; $e_d \leq e_{d,\text{goal}}$.
2. **Correct x-position:** The north position of the USV is on the correct side of the dock; x $\geq$ 0.0 m.
3. **Correct surge velocity:** The surge velocity is lower than the allowed docking velocity, but still larger than zero; 0.0 m/s $\leq$ u $\leq u_d$ m/s
4. **Correct heading:** The heading error, e_ψ, from (5.8) is smaller than a threshold, ensuring that the USV is facing towards the dock; $e_\psi \leq e_{\psi,\text{goal}}$

The outcome of an episode is summarized in two terminal states: F and G. F is the fail condition, which is activated if one of the two fail conditions are true. G is the goal condition, which indicates that all four requirements for a successful docking maneuver are fulfilled.

$$F = \begin{cases} f, & \text{if episode is unsuccessful} \\ 0, & \text{otherwise} \end{cases} \tag{5.14a}$$

$$G = \begin{cases} g, & \text{if episode is successful} \\ 0, & \text{otherwise} \end{cases} \tag{5.14b}$$

where the two constants, $f < 0$ and $g > 0$, are the reward or penalty received at the end of an episode. These values are often larger than the rewards received at each step, since it is beneficial to give the RL-agent a clear indication that the episode is finished.

The RL-agent will therefore seek to complete the episode before the maximum step number is reached and keep within its boundaries in order to receive the large reward at the end of the episode.

5.3.8 Total episode reward

The total accumulated reward for an episode is the sum of the rewards received at each step in addition to the terminal reward, depending on the outcome of the docking operation. Two reward functions were used during training, one with action rate penalty and one without.

5.3.8.1 Without action rate penalty

$$R(n) = \sum_{0}^{n} (r_{e_d} + r_{\dot{e}_d} + r_{e_\psi} + r_u - r_n) + F + G \tag{5.15}$$

5.3.8.2 With action rate penalty

$$R(n) = \sum_{0}^{n} (r_{e_d} + r_{\dot{e}_d} + r_{e_\psi} + r_u - r_n - r_{\dot{a}}) + F + G \tag{5.16}$$

5.4 Network Models

The work of this theses will compare the two algorithms Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO). The reward function will be the same for both algorithms, and the comparison will be based on the performance of the policies produced by the two RL-agents. However, both algorithms have to be tuned individually in order to even be able to produce a working policy. This process is tedious and relies a lot on trial and error testing. Each algorithm consists of multiple parameters, and each parameter has to be chosen with respect to the others in order for the model to be successful. The work of Islam et al. (2017) resulted in an article that presents a series of tests and comparisons of different parameter combinations. The findings in this report were used as a baseline for the parameters chosen, and then some adjustments were made in order to fit the models to this application.

An overview of the architecture is illustrated in Figure 5.10. The USV model is unknown to the reinforcement learning agent, and the only communication is through the state and action vector, $\mathbf{x}_n$ and $\mathbf{a}$, which was presented in Section 5.2.

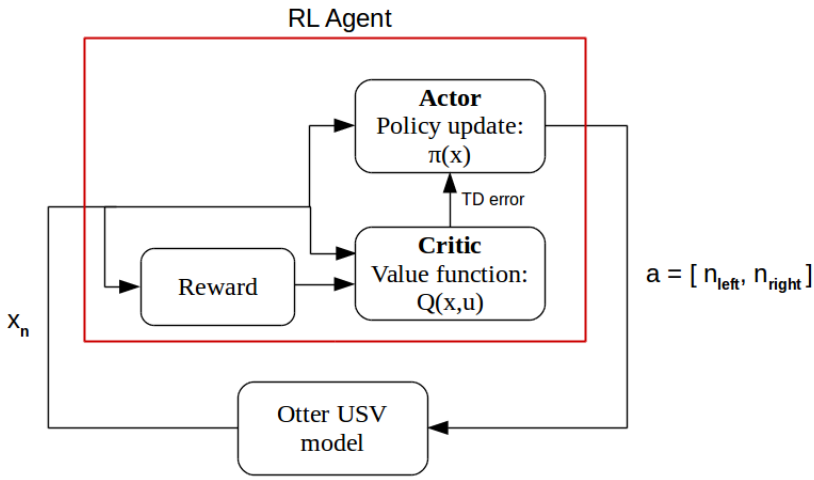


Figure 5.10: Overview of how the RL agent communicates with the USV model.

5.4.1 Deep Deterministic Policy Gradient

As discussed in Section 3.4, the deep deterministic policy gradient (DDPG) algorithm (Lillicrap et al. (2015)) is well suited for problems with high-dimensional state space and continuous action spaces. The DDPG model used is based on the framework implemented by OpenAI (Hill et al. (2018)) with some modifications. The pseudo-code is presented in Algorithm 1. The original DDPG model used Ornstein-Uhlenbeck (Uhlenbeck and Ornstein (1930)) as action noise to motivate exploration. However, as described in Section 3.1, parameter noise has shown to give more stable learning than action noise (Plappert et al. (2018)) and is therefore chosen for this project. The adaptive parameter noise function will gradually reduce the standard deviation of the noise from the initial value to the desired value as the training converges. The paper from Lillicrap et al. (2015) used SGD for the training of the network, but this implementation uses the ADAM optimizer (Kingma and Ba (2014)). Table 5.1 summarizes the parameter values set for the DDPG model used while training the agent.

Table 5.1: The parameters chosen for DDPG.

Model part	Parameter	Value
Actor-Critic	Actor, number of hidden layers	2
	Actor, number of nodes in each hidden layer	[400,300]
	Actor, learning rate	1e-4
	Critic, number of hidden layers	2
	Critic, number of nodes in each hidden layer	[400,300]
	Critic, learning rate	1e-3
	Hidden layers activation function	ReLU
	Output layer activiation function	Tanh
DDPG	Discount rate, γ	0.99
	Replay buffer size	1 000 000
	Mini-batch sample size	128
	τ, target network update rate	0.001
	Parameter noise, initial standard deviation	1.0
	Parameter noise, desired standard deviation	0.1

5.4.2 Proximal Policy Optimization

The Proximal Policy Optimization (PPO) algorithm is based on the framework implemented by OpenAI (Hill et al. (2018)), where the pseudo code is illustrated in Algorithm 2 in Section 3.4. As recommended by Schulman et al. (2017), ADAM optimizer (Kingma and Ba (2014)) was used during training. The parameters used during training is summarized in Table 5.2.

Table 5.2: The parameters chosen for PPO.

Model part	**Parameter**	**Value**
Actor-Critic	Actor, number of hidden layers	2
	Actor, number of nodes in each hidden layer	[400,300]
	Actor, learning rate	2.5e-4
	Critic, number of hidden layers	2
	Critic, number of nodes in each hidden layer	[400,300]
	Critic, learning rate	2.5e-4
	Hidden layers activation function	ReLU
	Output layer activiation function	Tanh
PPO	Discount rate, γ	0.99
	Mini-batch sample size	125
	Number of epochs	10
	Clipping range, ϵ	0.2
	GAE paramter, λ	0.95

5.5 Sensor Implementation

This section will describe how the sensory system for the USV position and ocean current estimation was implemented in the simulator in order to simulate realistic measurement noise and accuracy. The performance values are picked based on the hardware choices made in Section 4.2. By applying these realistic values, the simulator will provide measurements to the reinforcement-learning agent, which will be closer to what the real-world application will experience.

5.5.1 Position Estimation

As discussed in Section 2.3.1, a stereo camera is a good alternative for a position estimator for the USV during the docking procedure. The stereo camera from Stereo Labs (2020), Zed 2, was presented as a valid sensor choice in Section 4.2. This solution provides a high enough data rate and measurement precision for the docking system, and will be the basis for the measurement performance used in the simulator. The Zed 2 camera provides a data rate of 25 Hz, with a position drift of 0.35% and heading drift of 0.005°/m.

The data rate of 25 Hz is considered the best case under ideal conditions. However, this positioning system will be implemented in a cascade of multiple sensors. In order to make room for further data processing the sampling frequency in the simulator is set to 20Hz. The precision values are set to 2cm for the NED position and 0.1° for heading. The performance values determined for the position estimation are summarized in Table 5.3.

Table 5.3: Performance values set for position estimates in the simulator.

Parameter	**Value**
Data rate	20 Hz
Depth range	0.2-20 m
Position accuracy	2 cm
Heading accuracy	0.1°

5.5.2 External Disturbance

Ocean currents were implemented as an external disturbance in the simulator. The ocean current velocity, V_c, and angle, β_c, are given in NED frame and can be decomposed into the following surge and sway velocity in the USV body frame:

$$u_c = V_c \cos(\beta_c - \psi) \tag{5.17a}$$

$$v_c = V_c \sin(\beta_c - \psi) \tag{5.17b}$$

where ψ is the heading angle of the USV measured in NED. The velocity vector, $\boldsymbol{\nu}$, of the USV in (2.1) then has to be replaced by the relative velocities given by the following equation:

$$\boldsymbol{\nu}_r = \boldsymbol{\nu} - \boldsymbol{\nu}_c \tag{5.18}$$

where $\boldsymbol{\nu}_c = [u_c, v_c, 0, 0, 0]$ is the velocity of the ocean currents expressed in body frame of the USV. Using (5.18) will change the equation of motions presented in (2.1) to:

$$\mathbf{M}\dot{\boldsymbol{\nu}}_r + \mathbf{C}(\boldsymbol{\nu}_r)\boldsymbol{\nu}_r + \mathbf{D}(\boldsymbol{\nu}_r)\boldsymbol{\nu}_r + \mathbf{G}\boldsymbol{\eta} + \mathbf{g}_0 = \boldsymbol{\tau} \tag{5.19}$$

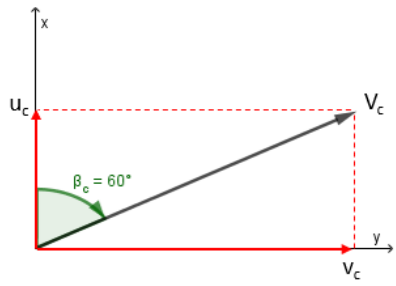


Figure 5.11: Decomposition of ocean current V_c at $\beta_c = 60°$.

Ocean Current Estimator

In order to make the simulator as realistic as possible, a suitable ocean current estimator sold on the consumer marked was applied. The ocean current affecting the vessel can be measured using a Doppler Velocity Log (DVL), which was presented in Section 4.2. The DVL sensor from Water Linked (2020) was found to be suitable for the USV due to its small size and performance. The performance data of the sensor is shown in Table 5.4.

Table 5.4: Performance and accuracy for DVL A50 from Water Linked.

Parameter	**Value**
Sample frequency	4-26 Hz
Min altitude	5 cm
Max altitude	50 m
Max velocity	2.6 m/s
Long term accuracy	0.1 cm/s

In the simulator, the ocean current can be added to the environment as a disturbance. Therefore, it is beneficial to create a disturbance with the same sample frequency and long term accuracy as the DVL sensor from Water Linked can estimate. This may help provide a smoother transition from a simulator to a real-world application. The DVL sensor is able to provide a sampling frequency in the range 4-26 Hz, depending on the altitude. For this thesis, the sampling frequency will be set to 20 Hz, making it the same as the position estimator.

The noise amplitude is randomly selected from a normal (Gaussian) distribution. The DVL sensor has a long term accuracy of 0.1 cm/s, which will be the standard deviation of the distribution. The mean of the distribution is set to 0 m/s. The ocean current velocity and angle is set to be constant throughout an episode. The noise added to the ocean current is illustrated in Figure 5.12.

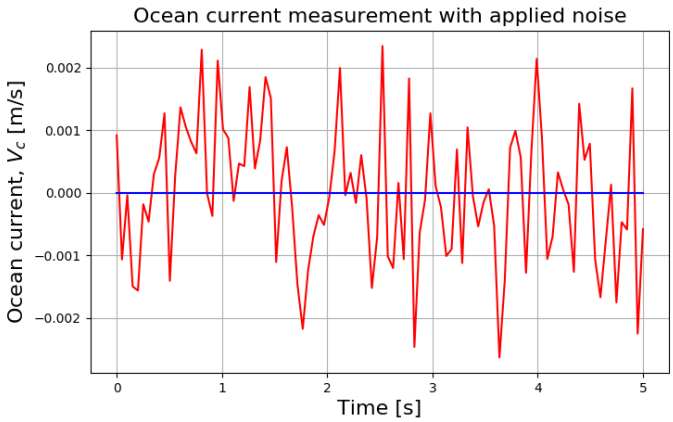


Figure 5.12: Noise added to ocean current estimate at 20 Hz with 0.1 cm/s accuracy.

Chapter 6

Simulation Setup

This chapter will present and explain initial values and boundaries for the environment during training. The reinforcement learning agent will be trained and tested for three different cases where the next case builds further on the previous; (1) comparison of DDPG and PPO, (2) the best model from case 1 with realistic sample rate and measurement noise, but without disturbance, (3) the same as case 2, but with disturbance.

The state vector, $\mathbf{x}_i$, will be modified between the cases, but the action vector, $\mathbf{a}$, presented in Section 5.2 will remain the same for all cases. The two thrusters are fixed and will not be modified between any of the cases.

$$\mathbf{a} = \left[n_{\text{left}}, n_{\text{right}}\right] \tag{6.1}$$

6.1 Environment Parameters

The reinforcement learning agent will be deployed in three different environments: without ocean current disturbance, with unknown disturbance, and with measured disturbance. These three cases are constructed in order to compare the different results and discuss whether the performance improves if the disturbance is measured or not. However, the following parameters of the environment will remain the same for all cases:

- **Environment boundaries:** The outer boundaries of the environment are constant for all cases. These boundaries are set with regard to what is realistic for a real-world application. These boundaries are also important in order to limit the space available for the RL-agent to explore. This way, the reward function is able to penalize the agent if these boundaries are exceeded. The boundaries are defined in NED with the dock in origin, (x,y)=(0,0). The north position, x, is allowed to be negative, which means that the USV can travel through the dock. This is done such that the RL-agent is able to explore the whole environment. However, the reward function is designed such that an episode is not successful as long as the north position is negative, thus teaching the agent that this area is invalid.

  - x ∈ [-2, 10] m
  - y ∈ [-5, 5] m

- **Episode length:** The maximum number of time steps, n, is also constant for all cases. This value is set in order to limit the amount of time the RL-agent is allowed to explore the environment. This limit is set by considering the minimum time required by the USV to complete an episode successfully, but also by giving the RL-agent enough time to explore the environment at the beginning of training. The episode length is also used in the reward function in order to penalize the agent if the time limit is exceeded.
  - $n_{\max} = 3000$
- **Step size:** The step size corresponds to the sampling frequency of the sensor systems deployed on the USV. This value will set to correspond to the largest sample frequency of the sensors onboard. Two different step sizes will be used, one for comparison of the two network models, h_1, and one for testing with realistic measurements, h_2.
  - $h_1 = 0.02s$ (50 Hz)
  - $h_2 = 0.05s$ (20 Hz)

6.2 Initial Values

In order to help the agent explore the whole environment, it is important to choose initial values which introduce the agent to all possible scenarios. The initial values are set for the four states; x-position, y-position, heading, and surge velocity. The values are chosen in the range that are plausible states at the moment where the docking controller is enabled. A random uniform distribution is used to select a value in the defined range.

The initial position in NED is set within the defined boundaries of the environment.

$$x_0 \in [5.0, 9.5] \ \text{m}, \qquad y_0 \in [-4.0, 4.0] \ \text{m} \tag{6.2}$$

The initial heading is set based on the east-position. It's assumed that the path-following controller will leave the USV in an orientation such that the vessel is facing towards the dock. The initial heading value is therefore divided into two quadrants: negative

y-position and positive y-position.

$$\psi_0 \in \begin{cases} [150, 180]^\circ, & \text{if } y \in [-4.0, 0.0]m \\ [180, 210]^\circ, & \text{if } y \in [0.0, 4.0]m \end{cases} \tag{6.3}$$

The initial surge velocity is set in a range around 1.0 m/s which is the desired velocity at the beginning of the docking operation.

$$u_0 \in [0.8, 1.2] \text{ m/s} \tag{6.4}$$

6.3 Termination Values

The USV has to meet some requirements for position, velocity, and heading in order to qualify an episode as successful, as described in Section 5.3.7. These values can be changed and adapted based on the wanted behavior. The requirements set for this implementation are based on the requirement specification **R1** defined in Section 1.6. The values are summarized in Table 6.1.

Table 6.1: Desired values for successful episode termination.

	Parameter	**Value**
1.	Desired euclidean distance	$e_d \leq 1.0m$
2.	Desired north-position	$x \geq 0$ m
3.	Desired surge	$0.0 \leq u \leq 0.2$ m/s
4.	Desired heading error	$\lvert e_\psi \rvert \leq 45^\circ$

All four parameters have to be fulfilled simultaneously. Each parameter is therefore defined within a range, making it possible for the RL-agent to explore and find the optimal combination. This creates an area of acceptance in front of the dock where the USV can complete the docking maneuver, as illustrated in Figure 6.1.

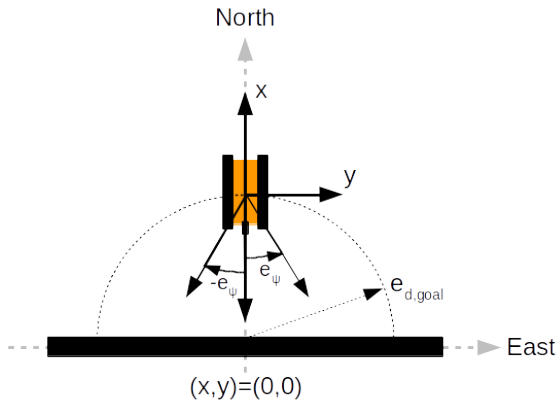


Figure 6.1: Illustration of the requirements for successful episode termination.

6.4 Case 1: Comparison of network models

This first case is used to decide which of the two network models that will be used for the two other cases. Since both Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO) are suitable for this controller implementation, it's interesting to compare the two and decide which performs best. The same reward function will be used for both network models and will be the basis for how good the performances are. The success rate and deviation from the desired position and velocity will also be a measure for the performance.

The network models themselves are tuned individually in order to achieve successful models. The parameters used are summarized in Table 5.1 and Table 5.2. The step size for these simulations are set to $h_1 = 0.02s$ (50 Hz). The RL-agents are not exposed to any measurement noise or disturbances in theses tests. This is done in order to be able to compare the performance under ideal conditions.

Case 1 will utilize the state vector without the measured ocean current velocity and angle, $\mathbf{x}_1$, which was presented in Section 5.2.

$$\mathbf{x}_1 = \left[\tilde{x}, \tilde{y}, \tilde{\psi}, u, v, r, e_d, n\right] \tag{6.5}$$

6.5 Case 2: Best model - without disturbance

The second case is performed in order to confirm that the RL agent is able to successfully dock the USV without any ocean current disturbances present. However, the measurements will be exposed to measurement noise in order to simulate with realistic values, as described in Section 5.5.1. The step size is increased to $h_2 = 0.05s$ (20 Hz) such that it corresponds to the sampling rate of the position estimator. A significant amount of time has been spent on developing the machine learning algorithm and the environment itself, so this test is important in order to verify that the simulator is suitable for further testing. Case 2 will utilize the same state vector as case 1, $\mathbf{x}_1$:

$$\mathbf{x}_1 = \left[\tilde{x}, \tilde{y}, \tilde{\psi}, u, v, r, e_d, n\right] \tag{6.6}$$

6.6 Case 3: Best model - with disturbance

In this case, the RL-agent is exposed to ocean current disturbance, as described in Section 5.5.2. Both the ocean current velocity, V_c, and angle, β_c, are set within a range which the USV is likely to encounter in a marina. The controller is created for a dock which is stationary at the shore facing north, making it only possible for the ocean current to have a direction from the open water towards the dock. In order to avoid actuator saturation, the ocean current velocity has an upper limit of 0.5 m/s. The ranges of the velocities and angles are defined as follows:

$$V_c \in [0.0, 0.5] \quad \text{m/s} \tag{6.7a}$$

$$\beta_c \in [120, 240]^\circ \tag{6.7b}$$

Both the ocean current velocity and angle are randomly selected from a normal distribution and stay constant throughout the episode. However, measurement noise is added to the constant values, and a sampling frequency of 20 Hz is used. An illustration of how the ocean current affects the USV at the different angles are shown in Figure 6.2. The measurement noise remains the same as for case 2, as described in Section 5.5.1. The same is for the step size, $h_2 = 0.05s$, resulting in a sampling frequency at 20 Hz.

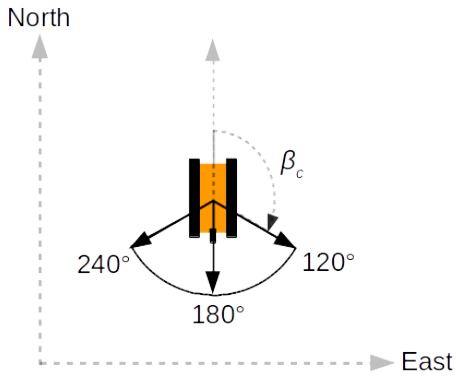


Figure 6.2: Illustration of the ocean angles, β_c, affecting the USV in case 3.

6.6.1 Unknown ocean current

This training will be executed with unknown ocean current, meaning that the disturbance isn't measured or estimated by the vessel. This case will utilize the same state vector as case 1 and 2 since the disturbance still remains unknown to the RL-agent:

$$\mathbf{x}_1 = \left[\tilde{x}, \tilde{y}, \tilde{\psi}, u, v, r, e_d, n\right] \tag{6.8}$$

6.6.2 DVL measured ocean current

This training is executed in order to compare the performance of the RL-agent with and without measured ocean current. Both the ocean current velocity, V_c, and angle, β_c, are implemented as before, using the ranges defined in (6.7). The same measurement noise and sample frequency are used as in the training scenario where the ocean current is unknown.

This training will utilize the state vector that observes the two states, V_c and β_c, presented in Section 5.2. By having the states available for the RL-agent, it creates a form of feed-forward controller, which might make it perform better and achieve a more stable controller.

$$\mathbf{x}_2 = \left[\tilde{x}, \tilde{y}, \tilde{\psi}, u, v, r, e_d, n, V_c, \beta_c\right] \tag{6.9}$$

6.7 Reward Function Parameters

The weights of the rewards and penalties used in the reward function are summarized in Table 6.2. Most of the parameters remain the same for all three cases, but some modifications are done between the cases. Case 1 was trained with a penalty on the surge, $\alpha = 0.5$, but this was removed when training case 2 and 3. The surge penalty resulted in some unwanted behavior when the sample rate was reduced from 50 Hz to 20 Hz. This is explained in more detail in Section 7.2. The action rate penalty, $C_{\dot{a}}$ was not added before in case 3.

Table 6.2: Reward function parameters for each case.

	C_{e_d}	$C_{\dot{e}_d}$	C_{e_ψ}	C_u	C_n	$C_{\dot{a}}$	σ_{e_d}	σ_{e_ψ}	σ_u	α	β
Case 1	2.0	1.0	1.5	2.0	2.0		2.5	0.17	0.05	0.5	1.5
Case 2	2.0	1.0	1.5	2.0	3.0		2.5	0.17	0.05	0.0	1.5
Case 3	2.0	1.0	1.5	2.0	3.0	0.02	2.5	0.17	0.05	0.0	1.5

The the reward and penalty for reaching terminal states, F and G (5.14), are the same for all three cases.

$$g = 300, \qquad f = -200 \tag{6.10}$$

Chapter 7

Results and Discussion

This chapter will present the results from testing the three cases presented in Chapter 6. A discussion of the results are done after each case. All simulations are done with the assumptions made in Section 1.5. The following tests were performed:

- **Case 1:** DDPG vs PPO, 50 Hz, no ocean current or measurement noise
- **Case 2:** PPO, 20 Hz, with measurement noise, but no ocean current
- **Case 3:** PPO, 20 Hz, with measurement noise and ocean current. Comparison of model with and without measured ocean current.

The two first tests were done in order to confirm the development of the machine-learning model. The third test was done in order to verify the two first requirement specifications, **R1** and **R2**.

7.1 Case 1

The first case will compare the performance of Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO). Both network models are trained in the same environment, with the same initial values, as presented in Section 6.2. The step size of the simulator is $h_1 = 0.02s$ (50 Hz) and maximal episode length is $n_{max} = 3000$ steps. The parameters used for DDPG and PPO are summarized in Table 5.1 and Table 5.2 respectively. Both models were trained for 10,000,000 time steps.

Figure 7.1 show the accumulated reward for each episode during training. Both models can be seen to have negative episode reward for the first 2000 episodes, and then converging after around 4000 episodes. The PPO algorithm converges to an average reward at approximately 1900, while the DDPG algorithm peaks at approximately 800. Both models produced RL-agents that successfully docks the USV and meets the requirements for position, surge velocity, and heading when approaching the dock. However, there are also requirements for surge and heading throughout the docking procedure. The plots of the average reward can indicate that the PPO algorithm will better comply to these requirements than the DDPG.

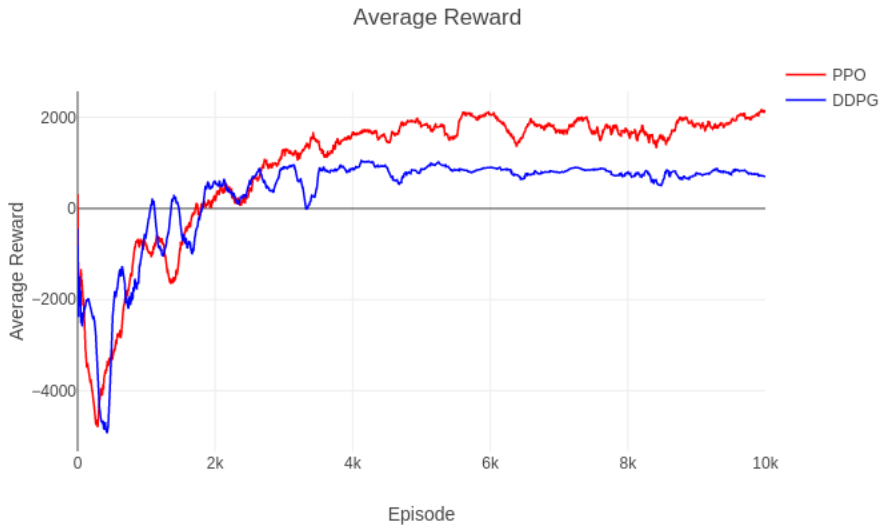


Figure 7.1: Training reward comparison between DDPG and PPO.

Episode Comparison

In order to better compare the performance of the two models, the RL-agents are initialized with the same position, orientation, and velocity, and tested for five episodes. The initial values of each episode are shown in Table 7.1.

Table 7.1: Case 1: Initial values for the test episodes.

Parameter	**Ep.1**	**Ep.2**	**Ep.3**	**Ep.4**	**Ep.5**
Initial position, (x_0, y_0)	(9,-3)	(9,-2)	(9,0)	(9,2)	(9,3)
Initial heading, ψ_0	150°	160°	180°	200°	210°
Initial surge, u_0	1.0 m/s	1.0 m/s	1.0 m/s	1.0 m/s	1.0 m/s

The two models were compared in how well they comply with the reward function. The important states are NED position, (x,y), surge velocity, u, and heading error, e_ψ. First, the models are compared in how they approach the dock, as illustrated in Figure 7.2. The reward function for the position (5.5) does not inform the RL-agent of how the path towards the dock should be. It only rewards the agent as long as the distance to the dock is decreasing. This is done in order to let the agent explore the environment and decide what the optimal path is. Therefore, due to differences in the algorithms of

DDPG and PPO, the paths towards the dock are different. It is difficult to decide if one path is better than the other, and this parameter alone cannot be used to decide whether one algorithm is better than the other.

All five episodes are successful for both models, meaning that the requirements for position, velocity, and heading defined in Section 6.3 are met. What can be observed from the NED plot in Figure 7.2, is that PPO's path is more consistent than DDPG for all five episodes. Especially for episode 3, where $(x_0, y_0) = (9, 0)$m, it can be seen that PPO chooses a path directly towards the dock, while DDPG chooses a path in negative east direction. This can be considered less optimal from the developer's perspective. Another thing to notice is PPO's path when $e_d < 3.0$ m, which is when the desired surge velocity is decreased from 1.0 m/s to 0.2 m/s (5.11) and the reward for heading error, e_ψ, is enabled (5.9). This creates a sudden change in the USV's path and may be a result of the PPO agent slightly cheating the reward function.

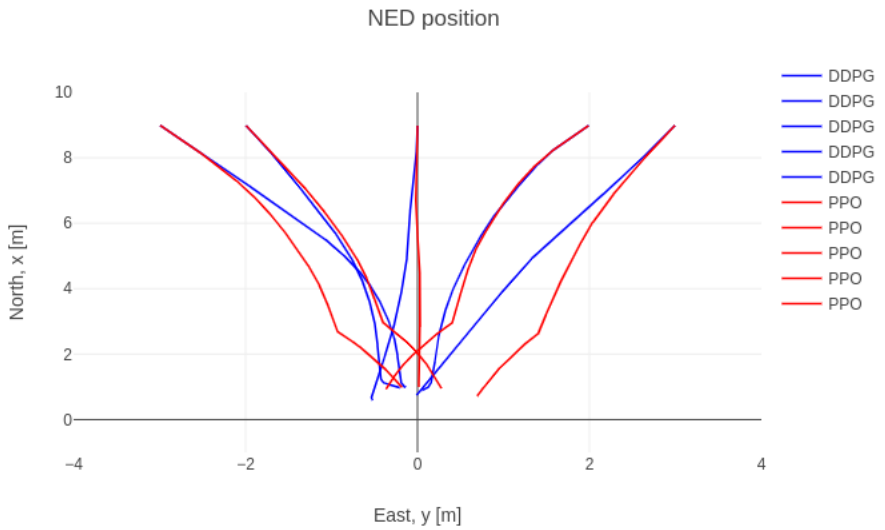


Figure 7.2: Comparison in NED position between DDPG and PPO.

All fives episodes starts with the same initial surge velocity, $u_0 = 1.0$ m/s. As defined in the reward function (5.11), it's desirable for the USV to keep a surge velocity of 1.0 m/s as it approaches the dock. When the euclidean distance becomes smaller than 3.0 meters, the desired surge is reduced to 0.2 m/s in order to ensure a safe docking speed. The reward functions also require a surge speed less than 0.2 m/s in order to complete the episode, as described in Section 6.3. Both models meet the requirements for surge

velocity when terminating the episode. However, the velocity on the way towards the dock is better satisfied by PPO than DDPG.

The surge velocity in each episode is illustrated in Figure 7.3. The PPO model can be seen to strictly follow the required surge velocity of 1.0 m/s while more than 3 meters from the dock and then quickly reducing the velocity to 0.2 m/s as it approaches the dock. The DDPG model doesn't comply with the requirements as good as PPO. In four of the five episodes, it can be observed that the DDPG model is able to hold a surge velocity of 1.0 m/s for the first meters of the docking procedure. However, as it reaches the dock, it increases the velocity to approximately 2.5 m/s before again reducing the velocity to 0.2 m/s in order to complete the episode. The fifth episode can be observed to increase the surge velocity to 2.5 m/s as the episode starts. Then reducing it around 5 meters from the dock and then increase it again. This might be explained by the DDPG model not being able to explore the state space enough to experience the reward received when meeting the desired velocity requirement. Instead, it speeds up to complete the episode and reduce the penalty received for not meeting the requirements.

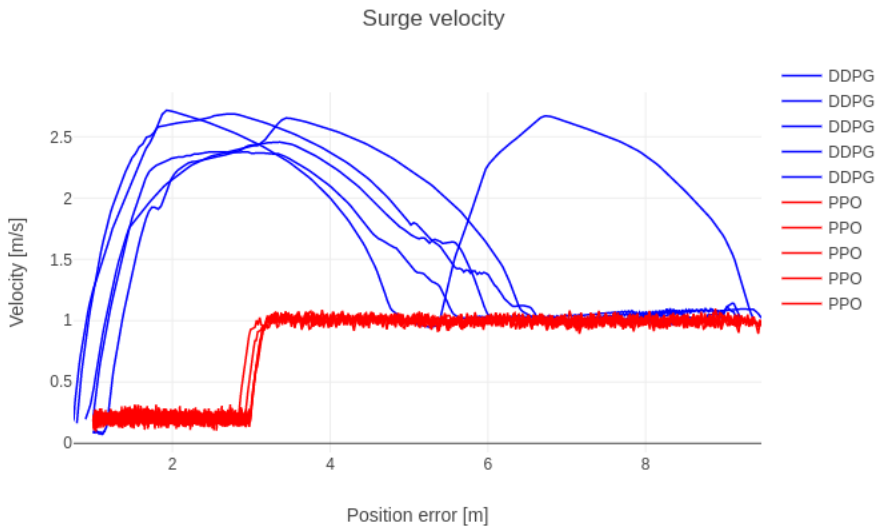


Figure 7.3: Comparison in surge velocity between DDPG and PPO.

The initial heading was the same for both models in each episode, as described in Table 7.1. The reward for heading error (5.9) doesn't activate before the position error, e_d, is smaller than 3.0 meters. This was done in order to not limit the RL-agent from exploring the environment and find the optimal path towards the dock itself. The desired

heading only becomes important as the docking procedure comes to an end. Ideally, the USV hits the dock head-on, meaning that the heading error is as close to zero as possible when the episode terminates.

As illustrated in Figure 7.4, the PPO follows the heading requirements strictly as the position error becomes less than 3.0 meters. In can also be observed that the heading error in each episode is slowly decreasing towards zero, resulting in a smooth path towards the dock. The DDPG model is not able to reduce the heading error to zero, but is still within the requirements for episode termination.

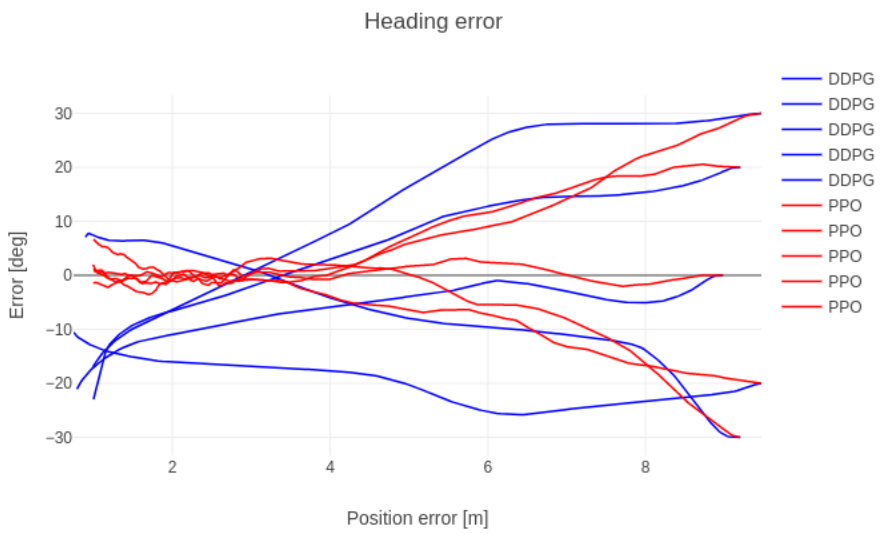
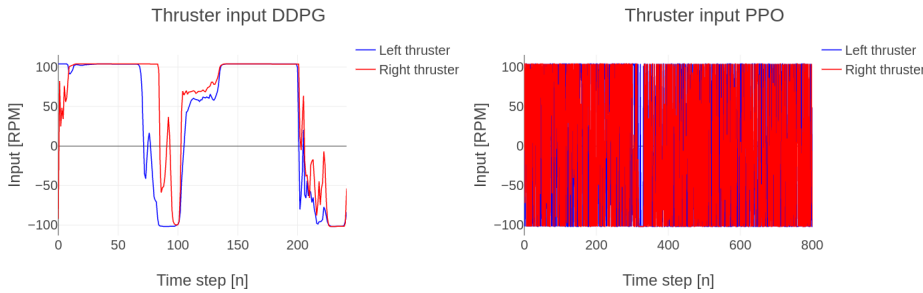


Figure 7.4: Comparison in heading error between DDPG and PPO.

The thruster input produced by the DDPG and PPO can be observed to be quite different, as illustrated in Figure 7.5. The DDPG input is less prone to oscillations compared to the PPO input. However, it changes between keeping the input at max and min throttle, which also isn't ideal. The thruster input produced by the PPO model is much more aggressive than DDPG. However, since the use of input is not penalized, the agent does not experience this behavior as unwanted.



(a) DDPG **(b)** PPO

Figure 7.5: Thruster input for the two DRL-models.

Discussion

These tests were performed in order to compare the two models produced by Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO). Both models were trained with the same reward function in order to compare how well they complied with the requirements. The observations by looking at the NED position, surge velocity, heading error, and thruster input have provided some insightful results. The PPO model complies better with the reward function and overall performs more in a way that is expected by the developer. In the NED plot, in Figure 7.2, the PPO model showed some signs of "cheating" the reward function by controlling the vessel in an unrealistic way. This might be a product of the RL-agent learning to strictly follow the requirements for euclidean distance, surge velocity, and heading reward. By quickly oscillating the thruster input, the agent is able to cheat the behavior of the USV and might reveal that the mathematical model lacks some dynamical constraints.

This case was implemented as a first attempt to produce a successful machine-learning model for autonomous docking. Therefore, the thruster input was not restricted in order to reduce the number of parameters that needed tuning. Even though the thruster input is more aggressive for PPO than for the DDPG model, the PPO algorithm showed to be easier to learn to follow the designed reward function. PPO was therefore chosen for further implementation.

Another feature that makes the PPO algorithm better suited for further testing is that it has fewer parameters that need tuning in order to make the training successful. The algorithm is also measured to be almost 1.5x faster while training, compared to the DDPG algorithm. Considering that the model has be trained for at least 20 hours in order to conclude if the changes to the model are working, this is an important feature in regards to further development.

7.2 Case 2

This next case will use Proximal Policy Optimization (PPO) to train a more realistic model in order to prepare it for a real-world application. The sampling rate was changed, and measurement noise was added to the machine-learning environment. The sensor system on the USV was presented in Section 5.5, and it was decided to use a sampling rate of 20 Hz on both the position estimator and the ocean current estimator. However, for this case, the ocean current disturbance is not added to the environment. This case is implemented in order to develop a model that can control the vessel under ideal conditions. The accuracy and amplitude of the measurement noise for the position and velocity estimates are summarized in Table 5.3.

When the sampling rate was reduced from 50 Hz to 20 Hz, the model showed signs of rushing to complete the episode. This resulted in a surge velocity that was much higher than desirable, as observed with the DDPG model in case 1. To solve this, the penalty in the reward function for surge (5.10) was removed. This gives the RL-agent more incentive to explore the environment. The changes made to the surge reward are illustrated in Figure 7.6.

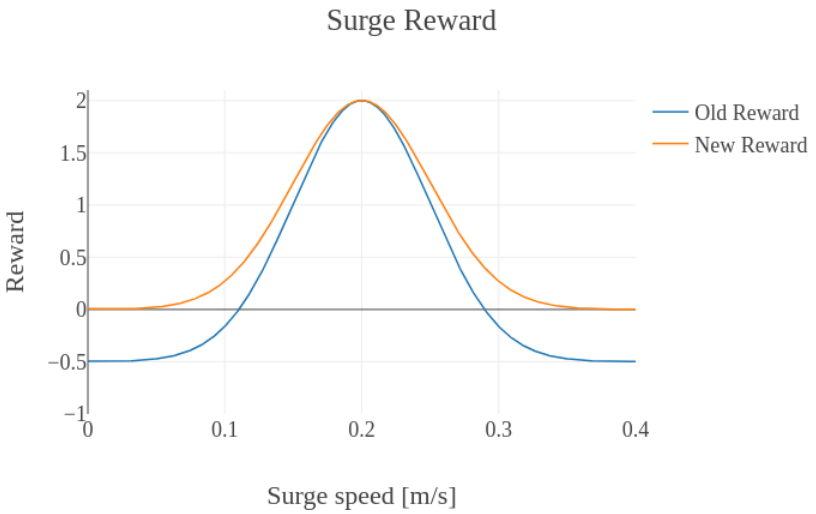


Figure 7.6: New reward function for surge velocity, at desired surge $u_d = 0.2$ m/s.

The PPO model was tested for 50 episodes, with initial NED position $x_0 = 9.0$ m and $y_0 \in [-4, 4]$ m, initial heading $\psi_0 \in [150, 210]°$, and initial surge velocity $u_0 \in [0.8, 1.2]$m/s. The initial north position, x_0, was kept at a constant distance, while the other initial states, y_0, ψ_0, u_0, were selected randomly from their respective range. This way, the PPO model was tested across the whole state space. The NED positions for all 50 episodes are illustrated in Figure 7.7.

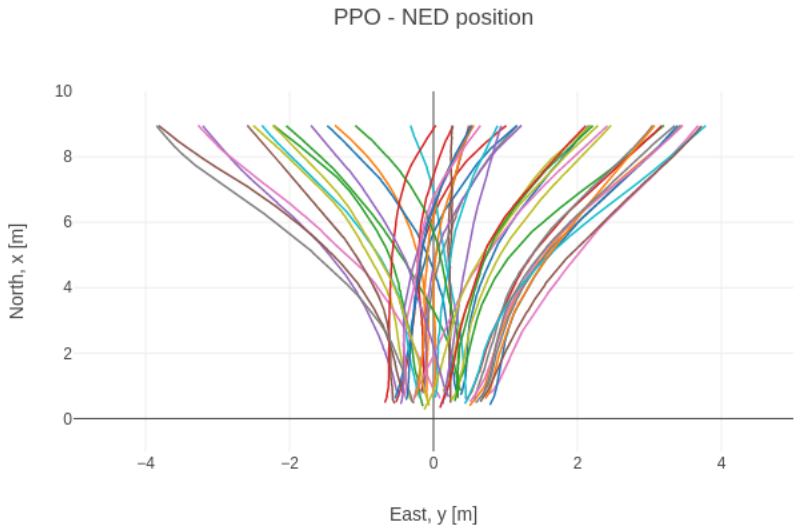


Figure 7.7: PPO NED position for 50 episodes.

The surge velocities for all 50 episodes are illustrated in Figure 7.8. Each episode was initialized with a random surge velocity in the range $u_0 \in [0.8, 1.2]$m/s. The desired surge velocities remains the same as for case 1, with $u_d = 1.0$ m/s if $e_d \geq 3.0$m, and $u_d = 0.2$ m/s otherwise. All episodes can be observed to keep a surge velocity between 0.8 m/s and 1.3 m/s when approaching the dock. When the position error becomes smaller than 3.0 meters the vessel starts to gradually slow down to 0.2 m/s.

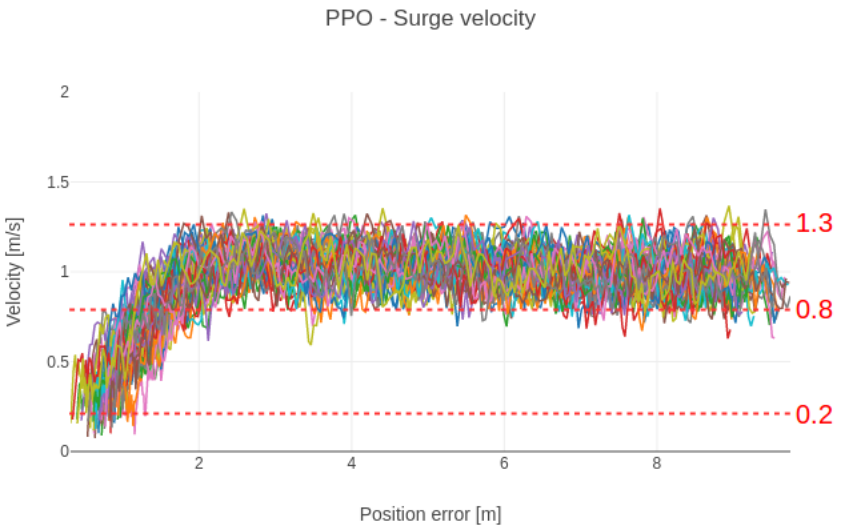


Figure 7.8: PPO surge velocity for 50 episodes.

Each episode are initialized with a random heading in the range $\psi_0 \in [150, 210]^\circ$, as described in Section 6.2. The corresponding heading error is illustrated in Figure 7.9, where $e_\psi = 0$ if $\psi = 180^\circ$. All episodes are able to finish with a heading error smaller than $\pm 18^\circ$, which is well within the requirements for completing an episode. The episodes initialized in the outer edge of the environment, (x_0, y_0)=(9, $\pm$4) m, are the ones where both position error, e_d, and heading error, e_ψ, are largest in Figure 7.9. For these episodes, the heading error first increases before converging towards zero. This means that the vessel first turns in towards zero east position, $y = 0$ m, before turning directly in towards the dock.

The thruster input from a single episode is illustrated in Figure 7.10. The thruster input is still not penalized and results in actuator saturation and aggressive oscillations.

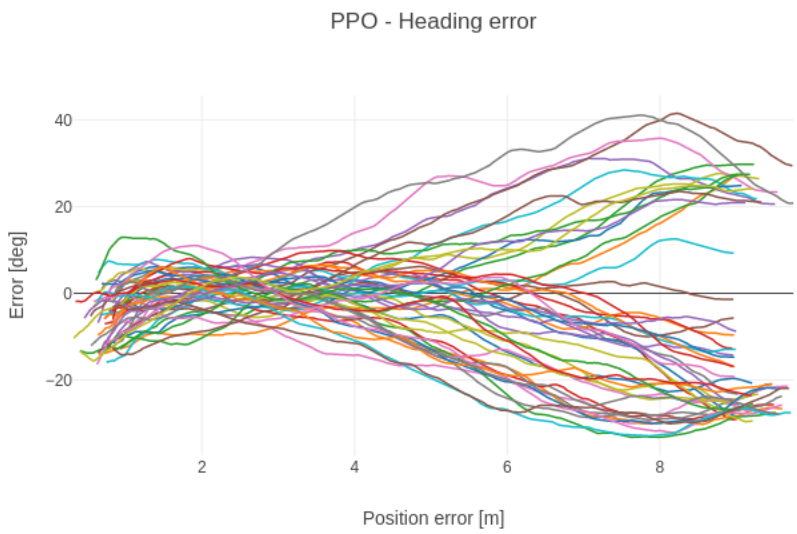


Figure 7.9: PPO heading error for 50 episodes.

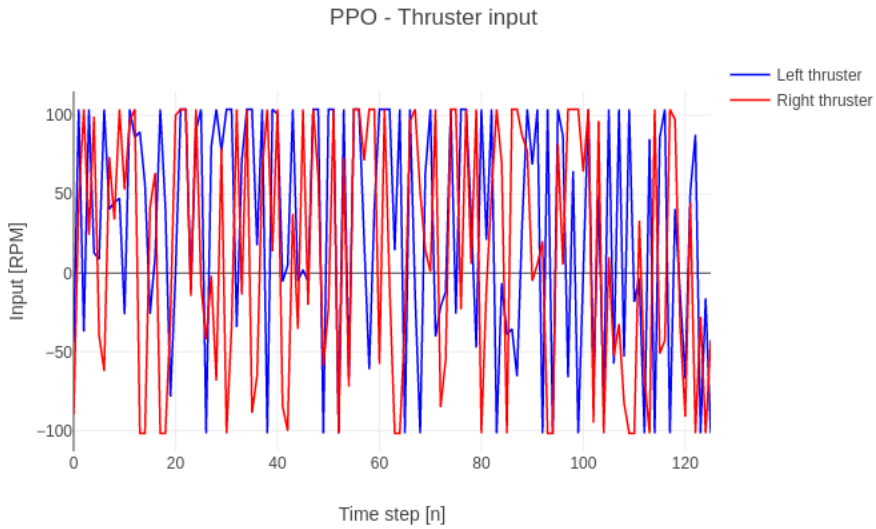


Figure 7.10: PPO thruster input for a single episode.

Discussion

Case 2 trained a model using Proximal Policy Optimization (PPO) with a sampling rate of 20 Hz and measurement noise. The model showed better performance with regards to NED position, surge velocity, and heading error compared to the PPO model in case 1. In case 1, the model showed some signs of unrealistic behavior by applying too aggressive thruster input, and therefore cheating the dynamic model. A reason for this improved behavior might be a result of removing the penalty for not keeping the desired surge velocity, as described in Figure 7.6. By removing this penalty, the RL-agent is more encouraged to explore the environment and find a combination of states which better satisfy the reward function.

The paths towards the dock in Figure 7.7 can be observed to be smoother than in case 1. From a developer's perspective, most of the paths look optimal in how they choose the shortest path to satisfy the requirements for both euclidean distance and heading error to complete the episode, as defined in Table 6.1. The vessel first turns in towards the center of the dock at $y = 0$ m, and then steers directly towards the dock.

The surge velocity wasn't followed as strictly as for case 1, but the surge speed was still reduced from 1.0 m/s to 0.2 m/s in a controlled manner. Even though the surge speed wasn't kept at a constant 0.2 m/s at 3 meters from the dock, this controlled descent was considered acceptable in order to ensure a safe docking maneuver. Similar to the surge velocity, the heading error isn't corrected as strictly towards zero as for case 1. This corresponds well with the behavior which was wanted in order to handle the underactuated vessel.

The behavior of the vessel is considered smoother in case 2 than in case 1, as seen in the plots for NED position, surge velocity and heading error. This is reflected in the thruster input, which is illustrated in Figure 7.10. Compared to the thruster input in case 1, the input in case 2 does not oscillate as much, but will still inflict too much actuator wear and tear. This is considered too aggressive for a real-world application. In order to improve the thruster input, a penalty should be applied to the thruster input rate. This might teach the RL-agent to apply a smoother control input.

7.3 Case 3

The RL-agent was trained in an environment exposed to ocean current disturbance, as described in Section 6.6. At the beginning of each episode, the ocean current velocity was randomly chosen from the uniform distribution defined in (6.7) and stayed constant for the whole episode. Two models were trained, one where the ocean current disturbance was unknown and one where the ocean current was measured using a doppler velocity log (DVL) sensor. The position and ocean current measurements are exposed to noise, as described in Section 5.5, and both sensors run at 20 Hz. Action rate penalty, $r_{\dot{\mathbf{a}}}$ (5.13), was added to the reward function in order to reduce the aggressive thruster input experienced in case 1 and 2. This test was done in order to verify that the two first requirement specifications, **R1** and **R2**, were met.

The average rewards from the two training sessions are illustrated in Figure 7.11. It can be observed that the two models perform almost equally with regards to accumulated reward. It is therefore hard to conclude if one is better than the other just by looking at the episode reward.

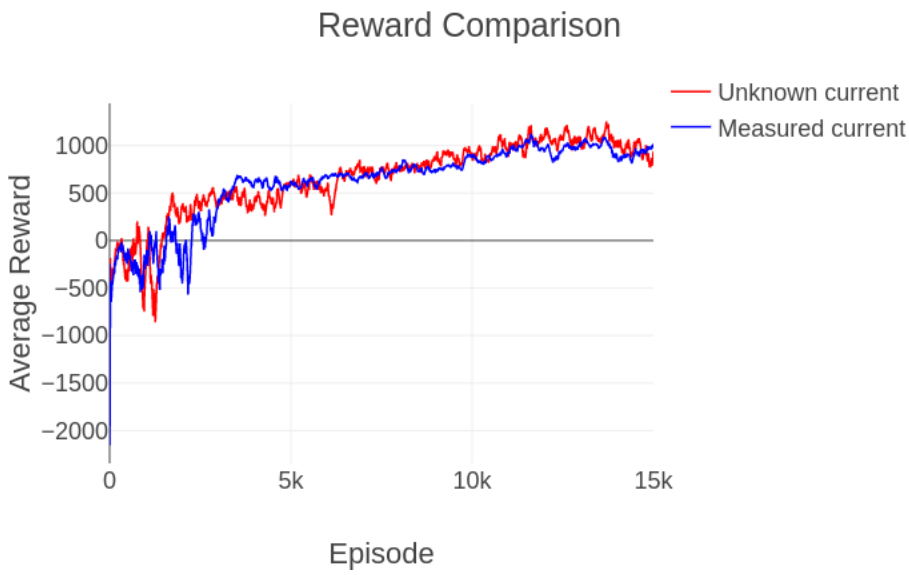


Figure 7.11: Comparison of average reward for model trained with no current, unknown current and known current.

A set of tests was implemented in order to compare the performance of the two models. It is desirable to test the models across the whole environment to confirm that the docking system can handle all situations when taking over from a path-following controller.

The NED position was initialized with north-position at $x_0 = 9.0$ m and east-position spanning the range $y_0 \in [-4, 4]$ m with 1 meter between each episode. Initial heading is initialized in the range $\psi_0 \in [150, 210]^\circ$, and surge speed is randomly initialized in the range $u_0 \in [0.8, 1.2]$ m/s. The nine test episodes are summarized in Table 7.2

The thruster input was observed to be too aggressive in case 1 and case 2. To handle this, the reward function was modified to include a penalty on the input rate, $\dot{\mathbf{a}} = [\dot{n}_{\text{left}}, \dot{n}_{\text{right}}]$, as described in (5.13) in Section 5.3.

Table 7.2: Case 3: Initial values for episodes when exposed to ocean current disturbance.

	Initial position, (x_0, y_0)	**Initial heading**, ψ_0	**Initial surge**, u_0
Ep.1	(9,-4) m	150°	∈[0.8, 1.2]m/s
Ep.2	(9,-3) m	160°	∈[0.8, 1.2]m/s
Ep.3	(9,-2) m	170°	∈[0.8, 1.2]m/s
Ep.4	(9,-1) m	175°	∈[0.8, 1.2]m/s
Ep.5	(9, 0) m	180°	∈[0.8, 1.2]m/s
Ep.6	(9, 1) m	185°	∈[0.8, 1.2]m/s
Ep.7	(9, 2) m	190°	∈[0.8, 1.2]m/s
Ep.8	(9, 3) m	200°	∈[0.8, 1.2]m/s
Ep.9	(9, 4) m	210°	∈[0.8, 1.2]m/s

7.3.1 Unknown ocean current

The model which experiences an unknown ocean current was tested with an ocean current $V_c = 0.4$m/s and $\beta_c = 180°$, with the initial values described in Table 7.2. The NED positions for each episode are illustrated in Figure 7.12. Episode 1-7 can be observed to completing the episode successfully, while episode 8 and 9 struggles to find the correct path in towards the dock and ends up completing behind the dock.

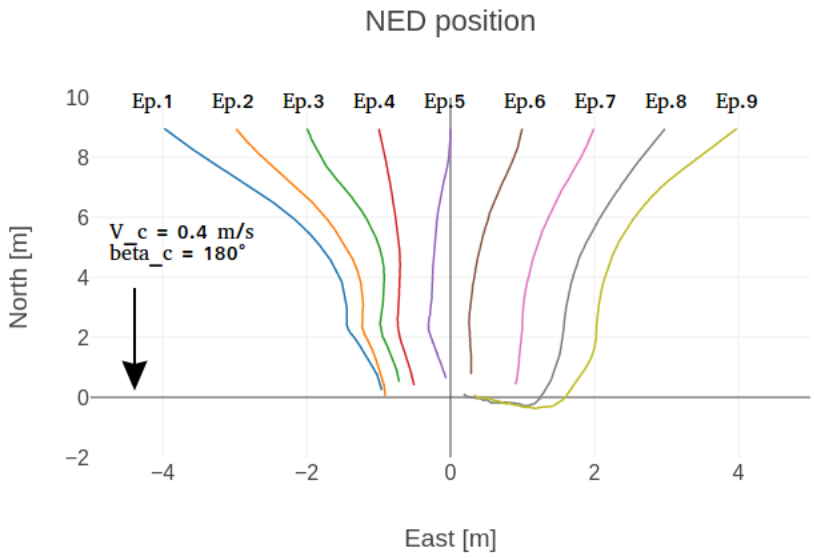


Figure 7.12: NED position when exposed to ocean current $V_c = 0.4$m/s with angle $\beta_c = 180°$

The heading error for each episode is illustrated in Figure 7.13. The heading error converges towards zero, but experiences some oscillations when the surge velocity is reduced at 3.0 meters from the dock. The ocean current at 0.4 m/s hits the vessel from behind, so it struggles to keep a straight path while simultaneously maintaining the correct surge speed.

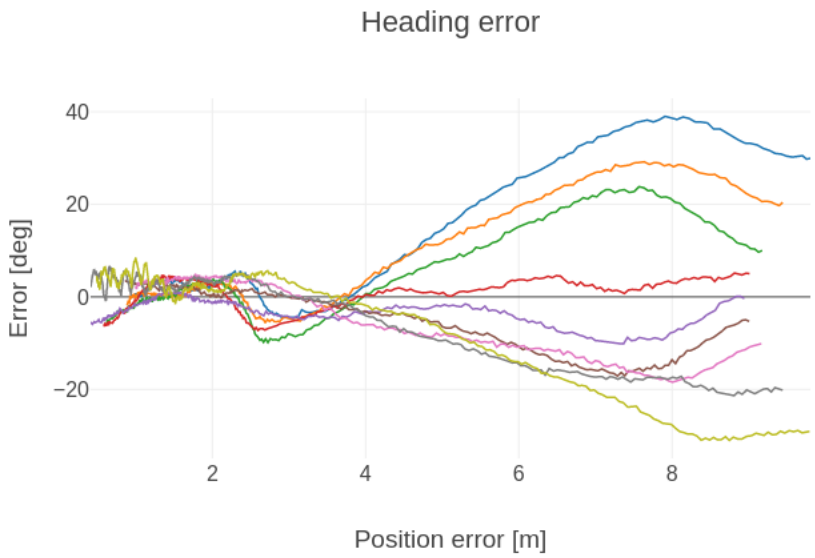


Figure 7.13: Heading error when exposed to ocean current $V_c = 0.4$m/s with angle $\beta_c = 180^\circ$

The surge velocities are illustrated in Figure 7.14. Episode 1 (blue line) and episode 2 (orange line) struggles to keep a surge speed of 1 m/s at the beginning of the episode. They are initialized at $y_0 = -4.0$m and $y_0 = -3.0$m, meaning that the vessel needs to turn in towards the center of the dock at $y = 0$ m before docking. This results in the RL-agent struggling to comply with the surge requirement while the ocean currents hit form the side. As it reaches the correct position to hit the dock head-on, it manages to comply with the surge requirement. Episodes 3-7 handles the surge requirements fine throughout the whole episode, while episodes 8 and 9 come to a standstill while struggling to complete the episode.

The thruster input is illustrated in Figure 7.15. The added action rate penalty can be observed to produce a more appropriate thruster input than for case 1 and case 2. The actuator saturation is removed, but the input is still prone to aggressive oscillations.

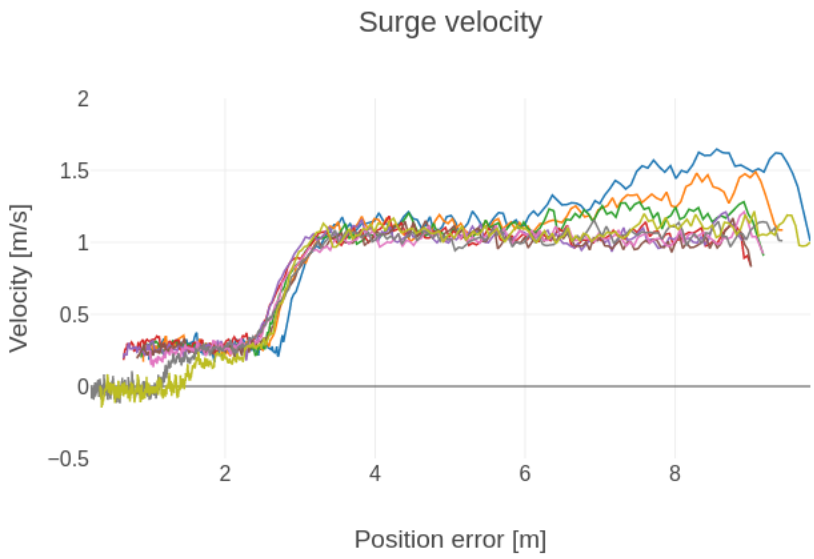


Figure 7.14: Surge velocity when exposed to ocean current $V_c = 0.4$m/s with angle $\beta_c = 180°$

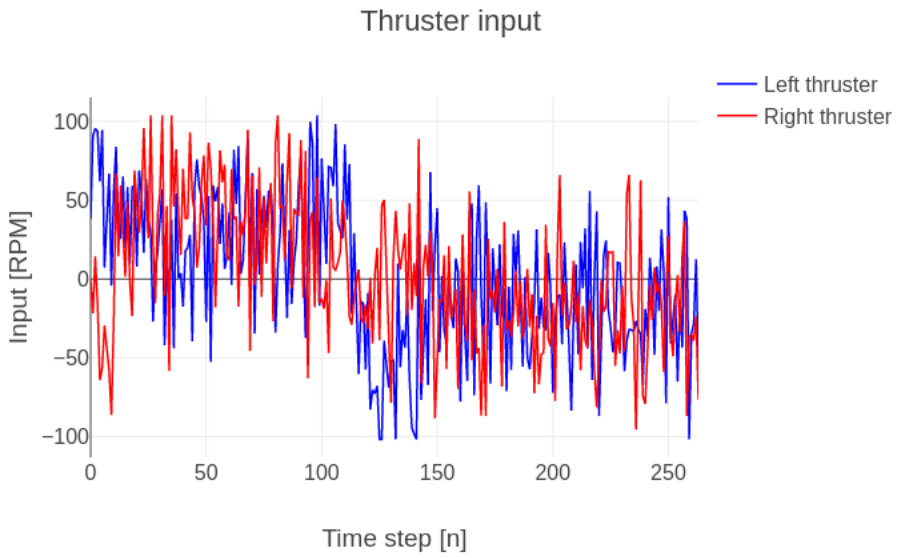
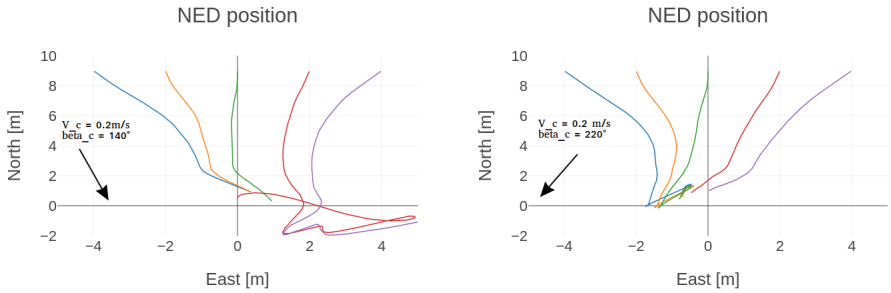


Figure 7.15: Thruster input when exposed to ocean current $V_c = 0.4$m/s with angle $\beta_c = 180°$

The model trained with unknown ocean current has shown to be able to handle ocean currents that hit perpendicular to the dock at $\beta_c = 180°$. However, when then the ocean current hits from the sides, the model performs strictly worse. To check the performance of the model, two tests were done: one with angle $\beta_c = 140°$ and one with $\beta_c = 220°$. The ocean current velocity had to be reduced to $V_c = 0.2$ m/s due to the model not being able to handle any higher velocity at these angles. Five episodes were tested, with the initial values of episodes 1, 3, 5, 7, and 9 in Table 7.2.

The results from the two tests are illustrated in Figure 7.16. The models are able to handle the ocean current when it is initialized on the same side of the dock as the ocean current comes from. The vessels are then able to float with the ocean current towards the dock and complete the episode successfully. However, when the vessel has to drive towards the ocean current, the model fails. The vessel drives directly towards the dock and is not able to handle the current. Instead, the vessel is pushed away from the dock without any means of correcting for it.

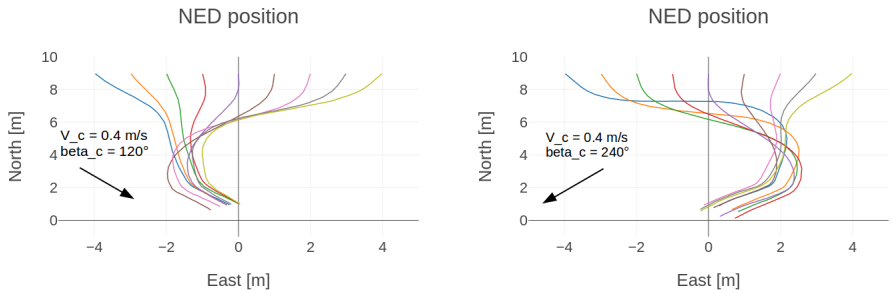


(a) $V_c = 0.2$m/s and $\beta_c = 140°$ **(b)** $V_c = 0.2$m/s and $\beta_c = 220°$

Figure 7.16: NED position when exposed to ocean current $V_c = 0.2$m/s with $\beta_c = 140°$ and $\beta_c = 220°$.

7.3.2 DVL measured ocean current

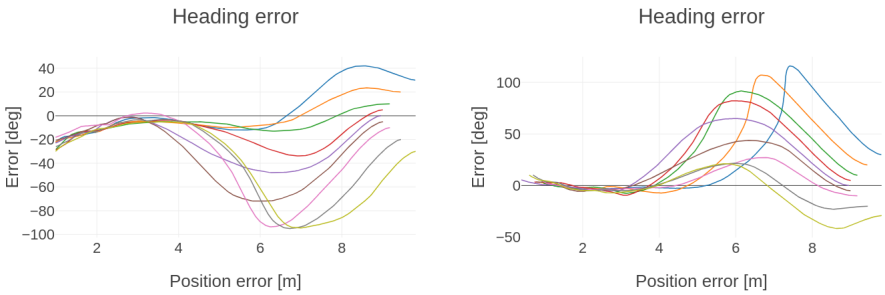
The model trained with measured ocean current was tested in two scenarios: one with ocean current angle $\beta_c = 120°$ and one with $\beta_c = 240°$, both with ocean current velocity $V_c = 0.4$ m/s. The initial values for both tests are summarized in Table 7.2. The NED positions for the two tests are illustrated in Figure 7.17. The model is able to dock for all nine episodes in both tests successfully. The vessel chooses a path where it drives towards the current until it reaches a position where it can use the current to its advantage and float towards the dock.



(a) $V_c = 0.4$m/s and $\beta_c = 120°$ **(b)** $V_c = 0.4$m/s and $\beta_c = 240°$

Figure 7.17: NED position when exposed to ocean current $V_c = 0.4$m/s with minimum and maximum current angle.

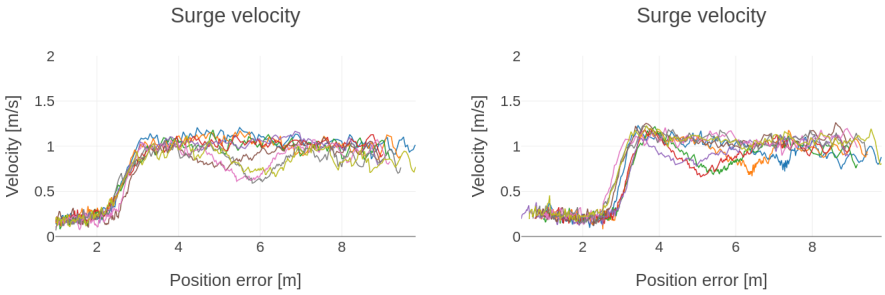
The heading error for the two tests is shown in Figure 7.18. As designed, the reward function for heading error (5.8) doesn't penalize the RL-agent for having a large heading error when more than 3.0 meters from the dock. This results in a model where the vessel corrects its position such that it can keep a heading error close to zero as the docking completes. As it reaches the dock, the vessel is allowed to have a small sideslip in order to compensate for the ocean currents.



(a) $V_c = 0.4$m/s and $\beta_c = 120°$ **(b)** $V_c = 0.4$m/s and $\beta_c = 240°$

Figure 7.18: Heading error when exposed to ocean current $V_c = 0.4$m/s with minimum and maximum current angle.

Figure 7.19 illustrates the surge velocities for the two tests. The model is able to follow the desired surge speed of 1 m/s until the position error is smaller than 3.0 meters. Then the desired surge is reduced to 0.2 m/s. Some of the episodes experiences a surge velocity a bit lower than 1 m/s, but the descent to 0.2 m/s is still controlled.



(a) $V_c = 0.4$m/s and $\beta_c = 120°$ **(b)** $V_c = 0.4$m/s and $\beta_c = 240°$

Figure 7.19: Surge velocity when exposed to ocean current $V_c = 0.4$m/s with minimum and maximum current angle.

The thruster input is illustrated in Figure 7.20. The actuators experience some saturation at the beginning of the episodes as the vessel corrects its position before completing the docking maneuver. As for the model trained with unknown ocean current, the oscillations in the thruster input is greatly reduced compared to case 1 and 2.

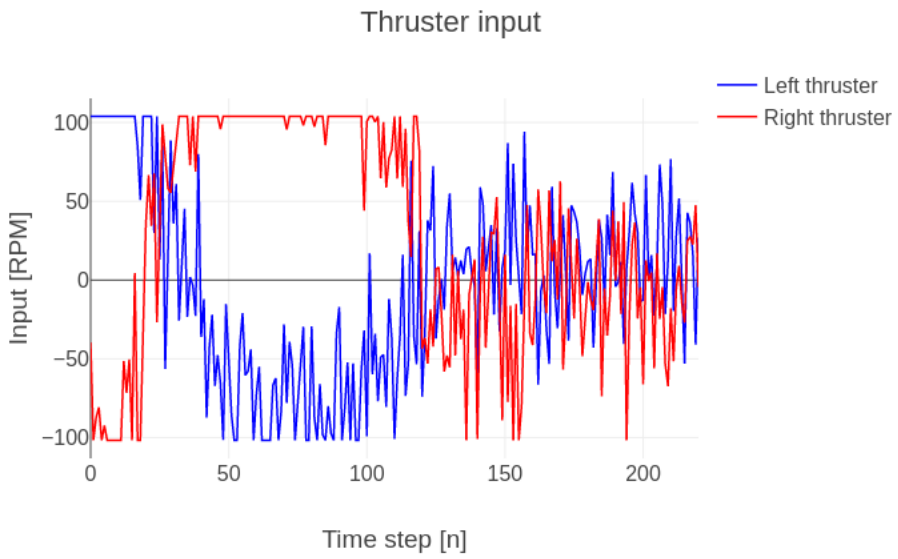


Figure 7.20: Thruster input when exposed to ocean current $V_c = 0.4$m/s and angle $\beta_c = 120°$.

This section has so far presented how the model is able to handle ocean currents up to $V_c = 0.4$ m/s, coming from the minimum and maximum angle. A last test was done to demonstrate how it struggles when the ocean current is turned up to $V_c = 0.5$ m/s. As Figure 7.21 illustrates, the vessel is still able to steer itself towards the dock. Episodes 1-3 failed to dock successfully, but episode 4-9 managed. As shown in Figure 7.22, both surge velocity and heading error are controlled as designed when the vessel approaches the dock. The problem for episode 1-3 occurs in the last stage of the docking maneuver. The vessel isn't able to position itself to make the position error smaller than the desired value of $e_d \leq 1.0$ m. The vessel then spins out and drives outside of the boundaries.

The thruster input for episode 5 is illustrated in Figure 7.23. This shows how the USV has learned to use the whole action space in order to handle the strong ocean currents.

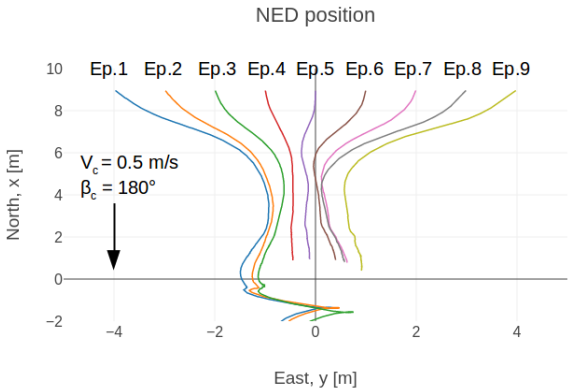
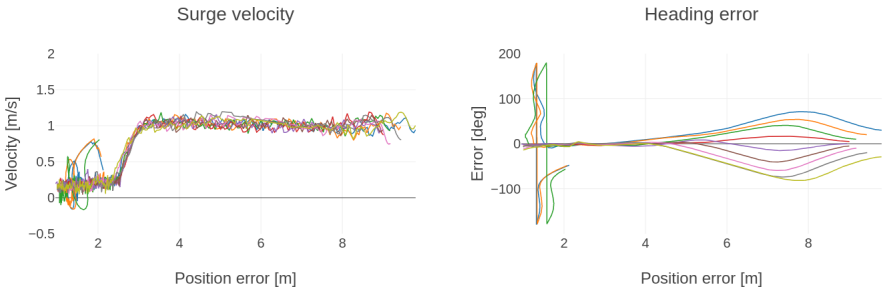


Figure 7.21: NED position with ocean current $V_c = 0.5$ m/s and angle $\beta_c = 180°$.



(a) Surge velocity **(b)** Heading error

Figure 7.22: Surge and heading error with ocean current $V_c = 0.5$ m/s with $\beta_c = 180°$.

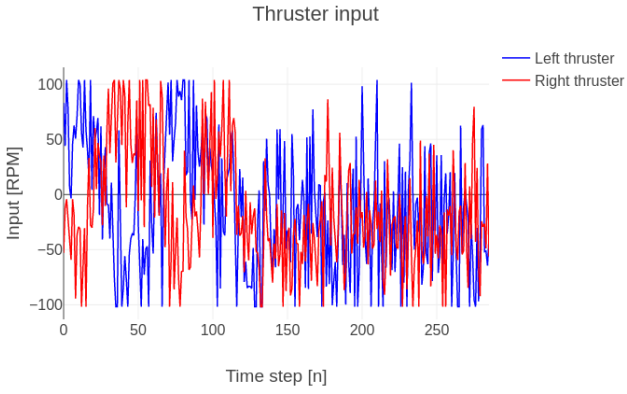


Figure 7.23: Thruster input with ocean current $V_c = 0.5$ m/s and angle $\beta_c = 180°$.

Discussion

Case 3 has trained two models: one exposed to unknown ocean current and one with DVL measured ocean current. This was done in order to compare the performance of the two models and research whether the performance is improved if the ocean current is measured. The first model with unknown ocean current was able to handle an ocean current that hits perpendicular to the dock, at $\beta_c = 180°$. The NED position, heading error, and surge velocity converge towards the desired values, and the episodes completes successfully. However, the USV shows signs of struggling to handle the unknown current. This becomes more clear when the model is tested with an ocean current coming from $\beta_c = 140°$ and $\beta_c = 220°$ and $V_c = 0.2$ m/s. The vessel does not choose a path that puts it in a position where it's able to handle the current. Instead, it follows a path similar to the model trained in case 2. This results in the vessel being pushed away from the dock. Even though the model is trained with relative positions and heading, it's not able to estimate the ocean current only based on this. To improve this, the model needs more training with these edge cases. Since all states are randomly chosen from a normal distribution, it requires an enormous amount of time steps in order to expose the RL-agent to these maximum values enough times to learn how to do this successfully.

The second model was trained with DVL measured ocean current. The average reward from the two training sessions was presented in Figure 7.11. The second model could be observed to accumulate slightly less reward, and this can be explained by looking at the performance in the nine test episodes. The first model learned to drive straight towards the dock, and this will be good enough for most episodes. Since the ocean current values are randomly chosen, it's more likely to be exposed to slow currents hitting the dock around 180°. Since it spends less time on completing the docking maneuver, it's less penalized by the step number penalty (5.12).

Even though the second model accumulates slightly less reward, it performs remarkably better when exposed to strong ocean current coming from steep angles. As seen in Figure 7.17, the vessel chooses a path that utilizes the ocean current to its advantage. Since the USV is underactuated and cannot move directly sideways, it learns to float with the current and instead spends its thruster input on keeping the correct heading and surge speed. The model is able to handle ocean currents up to $V_c = 0.4$ m/s coming from the minimum and maximum angle, $\beta_c = 120°$ and $\beta_c = 240°$. When the ocean current is turned up to 0.5 m/s, the vessel struggles to dock when the ocean current comes from the side. As for the first model, this could be a result of a lack of training on these edge cases. The USV could also be unable to dock in the strong currents due

to too little thrust. The USV might need an additional thruster in order to perform a controlled docking maneuver.

The use of action rate penalty, $r_{\dot{\mathbf{a}}}$, resulted in a smoother action input compared to case 1 and 2. The input from the first model with unknown current does not saturate the input as much as the second model. However, for this case, the maximum thruster input is needed in order to handle the strong ocean currents. By looking at the thruster input for the second model in Figure 7.20, the vessels stop to saturate the input after approximately 120 time steps. After this, the input is smoother, but still changing with a high frequency. This is still an unrealistic thruster input when preparing for a real-world application. Even though the Otter USV has fast thruster dynamics, this input is still too fast. This could perhaps be handled by low-pass filtering the thruster input before applying it to the USV.

7.4 Other Remarks

This section will present how an attempt was made to filter the action input.

Low-pass filtered action input

The results from case 3 in Section 7.3 showed how aggressive actuator use was reduced using an action rate penalty. However, the signal still has a too high frequency. In a real-world application, the dynamics of the actuators will behave as a low-pass filter. There was made an attempt to implement this for the model used in this thesis. The low pass filter was implemented using the work of Haugen (2008):

$$T_f \dot{y}(t_k) + y(t_k) = u(t_k) \tag{7.1a}$$

$$\dot{y}(t_k) \approx \frac{y(t_k) - y(t_{k-1})}{h} \tag{7.1b}$$

where $y(t_k)$ is the filtered signal, $u(t_k)$ is the unfiltered action input and h is the sample rate. Solving for $y(t_k)$ gives the following:

$$y(t_k) = (1 - a)y(t_{k-1}) + au(t_k) \tag{7.2}$$

where $a = \frac{h}{T_f + h}$ and $T_f \geq 5h$.

The trained model showed worse input utilization than the model trained without LP-filter. The filter is unknown to RL-agent, and it seems the agent is not able to learn how to handle the vessel when the input is filtered. Instead, it tries to control the vessel

by using even more aggressive thruster input, as shown in Figure 7.24. Due to time constraints, this was not further investigated, but should be in future work.

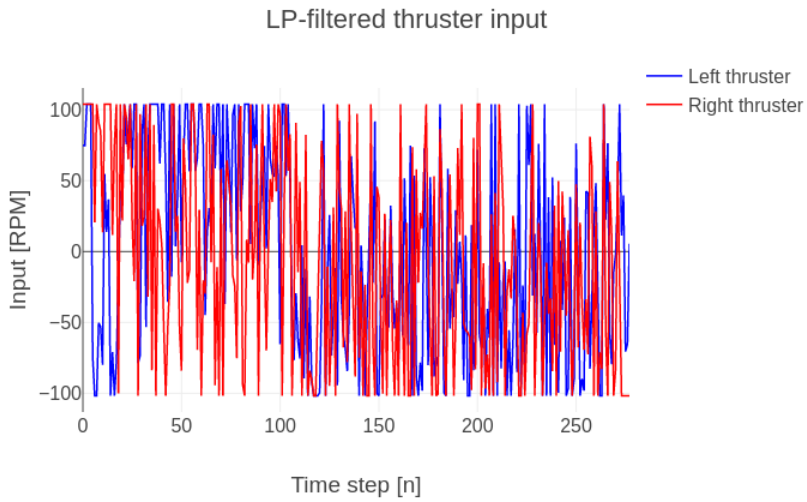


Figure 7.24: Low-pass filtered thruster input, with $V_c = 0.4$ m/s and $\beta_c = 180°$.

Chapter 8

Conclusion

This thesis has investigated how an autonomous docking system can be implemented for an underactuated USV using machine learning. The main focus has been on how reinforcement learning can be applied and what should be considered in the development of the reinforcement-learning environment and the reward function. The development has been divided into three cases, where each case builds further on the previous one. This was done in order to explain the development process and argue for the choices made along the way. A set of requirement specifications were defined in Section 1.6 for the controller performance, software, and hardware. These were used as a basis for the choices made while designing the reward function and choosing the hardware which was used for decided the measurement noise and sampling rate. This chapter will conclude the findings and present proposals for future work.

8.1 Overview

A set of objectives were presented in Section 1.4. The work of this thesis has managed to complete all the desired objectives:

- The equations of motion for the Otter USV was thoroughly presented in Chapter 2. The sensory model has also been presented. The sensory models for position and ocean current velocity were presented in theory in order to argue for the selection of sensor type. This was further used to propose a consumer-available product that could be used as a basis for the measurement noise and sampling rate.
- A machine-learning environment for the Otter USV was developed. The environment is flexible, and the parameters can easily be changed in order to fit the desired dynamic model and sensor setup. It allows for training with different reward functions and can be tested for various docking scenarios. The sequence diagram in Figure 5.2 describes the different steps of the training procedure. The environment is designed such that it can be trained using OpenAI's framework.
- A comparison of two deep reinforcement learning algorithms, Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO), was performed. The results in case 1 in Section 7.1 showed that PPO was easier to train and

followed the reward function better than DDPG. The PPO algorithm was also more efficient, resulting in faster training. This made it the best choice for further development in case 2 and 3.

- The performance of a docking controller with and without DVL measured ocean current was compared in case 3 in Section 7.3. This showed that the model trained with measured ocean current performed strictly better than the other. The path chosen utilizes the ocean current to hit the dock at the correct position, orientation, and velocity. It was able to handle ocean currents up to $V_c = 0.4$ m/s when the current comes from the sides, $\beta_c = 120°$ and $\beta_c = 240°$. The model trained with unknown current had to be initialized right in front of the dock in order to handle currents up to $V_c = 0.2$ m/s coming from the sides.

8.2 Research Questions

Throughout the work of this project, the research questions presented in Section 1.3 has been answered.

Q1: Machine learning has proven successful in the design of an autonomous docking system. By having a model of the dynamics of the Otter USV, a custom environment can be implemented. By designing the correct reward function, a model can be trained to meet the requirement specifications set by the developer.

Q2: This thesis only implemented ocean current as environmental disturbance. The model trained with unknown ocean current learned a general docking maneuver that does not take the ocean current into consideration. The model was able to dock while under the influence of slow ocean currents. However, as the currents get stronger, the model struggles to dock successfully. The model trained with measured ocean current learned a docking maneuver that takes the ocean current into consideration and uses this to choose a path that utilizes it. It's therefore recommended to measure the ocean current in order to develop a more reliable docking controller.

Q3: The development of the docking controller has had a real-world application in mind. However, some simplifications have been made in order to restrict the scope of the thesis. White noise without bias was added to the measurements, and the measurements were always available. The model has never experienced signal loss, signal drift, or wild points. This is an ideal situation that is not realistic when planning a real-world application. Further training is therefore needed with measurement sampled from real-world sensors. However, the trained model presented in this thesis can be used as a foundation for real-world training.

8.3 Future Work

The work done in this thesis has been based on a set of assumptions and simplifications in order to narrow the scope of the task. This section will reflect on these simplifications and propose how this can be improved in future work. The goal of these proposals is to make the simulations even more realistic and prepare the docking controller for a real-world application.

8.3.1 Thruster dynamics

The change in thruster output was assumed instant. A penalty was added to the input rate in order to reduce the aggressive action input observed in case 1 and 2. This removed the saturation, but the signal was still high frequent. In an attempt to reduce the frequency, a low-pass filter was added between the action input from the RL-agent and the dynamic model. Since the model would not be able to react to the high-frequency input instantly, it was expected that the RL-agent would learn to reduced its action input. However, this only resulted in the model performing worse than the ones without filtered input. Due to time constraints, this could not be further investigated.

The thruster dynamics need to be better implemented before a real-world test can be executed. This could maybe be achieved by implementing more input constraints in the model. Another approach could be to research further into the action rate penalty. The weight of this penalty was difficult to determine. If it was too high, the RL-agent would not explore the environment. If it was too low, then the input frequency was not reduced at all. This weight has to be tuned in regards to the rest of the rewards and penalties in the reward function. Therefore, it could help to find a new combination of rewards in order to achieve a better model.

8.3.2 Further environment development

The environment used for training is a simplified representation of a real-world scenario. The model is trained to handle a docking maneuver in a 8×10 meter area outside the dock. Ocean current is the only environmental disturbance, and no other vessels or obstacles are present.

Improvements to the environment could be to increase the area of operation. This could result in a docking system which could be enabled even earlier and farther from the dock. Both stationary and moving obstacles should also be added to the environment. The position of these obstacles should then be added to the state space of the RL-agent. By adding a penalty to the distance between the vessel and the obstacle, the agent

could learn to avoid them. Wind and waves could also be added to the environment, depending on the weather conditions at the dock of interest.

8.3.3 Measurement noise

The measurement noise used in this thesis is white and without bias. This is simplified compared to what would be encountered in a real-world application. Future work should implement more realistic measurement noise, which contains loss of signal, signal freeze, signal drift, and wild points.

8.3.4 Reward function improvement

The reward function used for this thesis proved successful and provided a foundation for further work. However, as the environment becomes larger and more disturbances are added, it may require some modifications to the reward function. It could be enough to tune the weights of the rewards and penalties, or it may require that more states are observed.

References

Achiam, J. and Abbeel, P. (2018). Proximal policy optimization - spinning up documentation, https://spinningup.openai.com/en/latest/algorithms/ppo.html.

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J. and Mané, D. (2016). Concrete problems in ai safety, *Cornell University* .

Anderlini, E., Parker, G. and Thomas, G. (2019). Docking control of an autonomous underwater vehicle using reinforcement learning, *Applied Sciences* **9**: 3456.

Andrychowicz, O., Baker, B., Chociej, M., Józefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., Schneider, J., Sidor, S., Tobin, J., Welinder, P., Weng, L. and Zaremba, W. (2019). Learning dexterous in-hand manipulation, *The International Journal of Robotics Research* p. 027836491988744.

Aqel, M., Marhaban, M. H., Saripan, M. I. and Ismail, N. (2016). Review of visual odometry: types, approaches, challenges, and applications, *SpringerPlus* **5**.

Arrow (2020). Ultrasonic sensors: how they work, https://www.arrow.com/en/research-and-events/articles/ultrasonic-sensors-how-they-work-and-how-to-use-them-with-arduino. Accessed: 27-04-2020.

Blue Robotics (2020). Bluerov2, https://bluerobotics.com/store/rov/bluerov2/. Accessed: 13-05-2020.

Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J. and Zieba, K. (2016). End to end learning for self-driving cars, *NVIDIA Corporation* .

Borhaug, E. and Pettersen, K. (2006). Los path following for underactuated underwater vehicle, *Proc. 7th IFAC Conference on Manoeuvring and Control of Marine Craft*, IFAC.

Bottou, L. (2012). *Stochastic Gradient Descent Tricks*, Vol. 7700, pp. 421–436.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W. (2016). Openai gym.

Cui, R., Yang, C., Li, Y. and Sharma, S. (2017). Adaptive neural network control of auvs with control input nonlinearities using reinforcement learning, *IEEE* .

Dikmen, M. and Burns, C. (2017). Trust in autonomous vehicles: The case of tesla autopilot and summon, *IEEE* .

ECA-Group (2018). Eca group usv solutions, https://www.ecagroup.com/en/find-your-eca-solutions/usv. Accessed:2019-05-10.

Eilertsen, E. (2019). High-level action planning for marine vessels using reinforcement learning, *NTNU* .

FLIR (2020). Blackfly s usb3, https://www.flir.com/products/blackfly-s-usb3/. Accessed: 05-06-2020.

Folkers, A., Rick, M. and Büskens, C. (2019). Controlling an autonomous vehicle with deep reinforcement learning.

Foss, B. and Heirung, T. (2013). Merging optimization and control, *NTNU* .

Fossen, T. I. (2011). *Handbook of Marine Craft Hydrodynamics and Motion Control*, John Wiley & Sons, Ltd.

Fossen, T. I. and Perez, T. (2004). Marine systems simulator (MSS), https://github.com/cybergalactic/MSS.

Gaudet, B., Linares, R. and Furfaro, R. (2020). Deep reinforcement learning for six degree-of-freedom planetary landing, *Advances in Space Research* .

Geo-matching (2019). Maritime robotics otter usv, https://geo-matching.com/usvs-unmanned-surface-vehicles/otter-usv. Accessed: 27-08-2019.

Grzes, M. and Kudenko, D. (2008). Plan-based reward shaping for reinforcement learning, *2008 4th International IEEE Conference Intelligent Systems*, Vol. 2, pp. 10–22–10–29.

Hans, A., Schneegaß, D., Schäfer, A. and Udluft, S. (2008). Safe exploration for reinforcement learning, *ESANN* .

Haugen, F. (2008). Derivation of a discrete-time lowpass filter, http://techteach.no/simview/lowpass_filter/doc/filter_algorithm.pdf. Accessed: 06-06-2020.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S. and Wu, Y. (2018). Stable baselines, https://github.com/hill-a/stable-baselines. Accessed: 10-02-2020.

Horaud, R., Hansard, M., Evangelidis, G. and Clément, M. (2016). An overview of depth cameras and range scanners based on time-of-flight technologies, *Machine Vision and Applications* **27**.

Howard, A. (2008). Real-time stereo visual odometry for autonomous ground vehicles, pp. 3946 – 3952.

Im, N.-K. and Nguyen, V. (2017). Artificial neural network controller for automatic ship berthing using head-up coordinate system, *International Journal of Naval Architecture and Ocean Engineering* **10**.

Islam, R., Henderson, P., Gomrokchi, M. and Precup, D. (2017). Reproducibility of benchmarked deep reinforcement learning tasks for continuous control.

Karpathy, A. (2019). Modeling one neuron, `http://cs231n.github.io/neural-networks-1/`. Accessed: 22-10-2019.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization, *International Conference on Learning Representations* .

Kongsberg (2020). Sounder usv system, `https://www.kongsberg.com/no/maritime/products/marine-robotics/autonomous-surface-vehicles/sounder-unmanned-surface-vehicle/`. Accessed: 10-05-2020.

Kretschmann, L., Burmeister, H.-C. and Jahn, C. (2017). Analyzing the economic benefit of unmanned autonomous ships: An exploratory cost-comparison between an autonomous and a conventional bulk carrier, *Research in Transportation Business & Management* **25**.

Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D. (2015). Continuous control with deep reinforcement learning, *ICLR 2016* .

Maritime Robotics (2019). The otter, `https://www.maritimerobotics.com/otter`. Accessed: 29-08-2019.

Martinsen, A. B. and Lekkas, A. M. (2018). Straight-path following for underactuated marine vessels using deep reinforcement learning, *IFAC* .

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D. (2015). Human-level control through deep reinforcement learning, *ICLR 2016* .

Moe, S., Caharija, W., Pettersen, K. Y. and Schjølberg, I. (2014). Path following of underactuated marine surface vessels in the presence of unknown ocean currents, *2014 American Control Conference*, pp. 3856–3861.

Mothes, B. (2019). Reinforcement learning for autodocking of surface vessels, *NTNU* .

Murdoch, E., Clarke, C. and Dand, I. W. a. (2012). *A masters guide to: Berthing*, The Standard Club.

National Ocean Service (2020). What is lidar?, https://oceanservice.noaa.gov/facts/lidar.html. Accessed: 27-04-2020.

Nortek (2020). Doppler velocity logs - dvl1000, https://www.nortekgroup.com/products/dvl-1000-300m. Accessed: 13-05-2020.

Nvidia (2020). Jetson xavier nx, https://developer.nvidia.com/embedded/jetson-xavier-nx-devkit. Accessed: 05-06-2020.

Olson, E. (2011). Apriltag: A robust and flexible visual fiducial system, *2011 IEEE International Conference on Robotics and Automation*, pp. 3400–3407.

Plappert, M., Houthooft, R., Dhariwal, P., Sidor, S., Chen, R., Chen, X., Asfour, T., Abbeel, P. and Andrychowicz, M. (2018). Parameter space noise for exploration, *ICLR* .

ROS components (2020). Tfmini low cost lidar, https://www.roscomponents.com/en/lidar-laser-scanner/246-ce30-a.html. Accessed: 27-04-2020.

Rudolph, D. and Wilson, T. (2012). Doppler velocity log theory and preliminary considerations for design and construction.

Rørvik, E. (2020). Automatic docking of an autonomous surface vessel, *NTNU* .

Sadeghi, F. and Levine, S. (2017). Cad2rl: Real single-image flight without a single real image, https://arxiv.org/pdf/1611.04201.pdf. Accessed: 24-10-2019.

Sans-Muntadas, A., Pettersen, K., Brekke, E. and Kelasidi, E. (2017). Learning an auv docking maneuver with a convolutional neural network, https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2493045/root84351.pdf?sequence=2&isAllowed=y.

SBG-Systems (2018). Rms inertial sensors, http://www.cnsens.com/PDF/Ellipse.pdf. Accessed: 2019-11-19.

Schulman, J., Moritz, P., Levine, S., Jordan, M. and Abbeel, P. (2015a). High-dimensional continuous control using generalized advantage estimation.

Schulman, J., Moritz, P., Levine, S., Jordan, M. and Abbeel, P. (2015b). Trust region policy optimization.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O. (2017). Proximal policy optimization algorithms.

Shadow Robot Company (2019). Shadow dexterous hand^{TM}, https://www.shadowrobot.com/products/dexterous-hand/. Accessed: 24-10-2019.

Shuai, Y., Li, G., Cheng, X., Skulstad, R., Xu, j., Liu, H. and Zhang, H. (2019). An efficient neural-network based approach to automatic ship docking, *Ocean Engineering* .

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D. and Riedmiller, M. (2014). Deterministic policy gradient algorithms, *31st International Conference on Machine Learning, ICML 2014* **1**.

SNAME (1950). *Nomenclature for treating the motion of a submerged body through a fluid*, Vol. 1-5 of *Technical and research bulletin*, The Society of Naval Architects and Marine Engineers.

Sparkfun (2020). Inertial measurement unit, https://www.sparkfun.com/products/13762. Accessed: 27-04-2020.

Stereo Labs (2020). Zed 2 stereo camera, https://www.stereolabs.com/zed-2/. Accessed: 05-06-2020.

Stöttner, T. (2019). Why data should be normalized before training a neural network, https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d. Accessed: 06-06-2020.

Sutton, R. and Barto, A. (2015). *Reinforcement Learning: An Introduction 2nd edition*, The MIT Press, Cambridge, London.

Teledyne Marine (2020). Doppler velocity logs (dvls), http://www.teledynemarine.com/dvls. Accessed: 13-05-2020.

TransNav (2015). Further studies on the colregs (collision regulations), http://www.transnav.eu/Article_The_Further_Studies_On_The_COLREGs_Demirel,33,551.html. Accessed: 14-05-2020.

Trondheim Havn (2019). Et havnekraftig trøndelag strategiplan 2019-2030, https://trondheimhavn.no/wp-content/uploads/2019/06/trondheim-havn-strategi-2030.pdf. Accessed: 14-05-2020.

Uhlenbeck, G. E. and Ornstein, L. S. (1930). On the theory of the brownian motion, *Physical Review 36 (5)* .

Van Isle Marina (2019). Anchoring, mooring & docking learning the (getting) ins and outs of boating, https://vanislemarina.com/anchoring-mooring-docking/. Accessed: 14-05-2020.

Velodyne Lidar (2020). Puck ^{TM}, https://velodynelidar.com/products/puck/. Accessed: 27-04-2020.

Water Linked (2020). Dvl a50, https://waterlinked.com/dvl/. Accessed: 24-04-2020.

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning, *Machine Learning* **8**(3): 279–292.

Zhang, P., Xiong, L., Yu, Z., Fang, P., Yan, S., Yao, J. and Zhou, Y. (2019). Reinforcement learning-based end-to-end parking for automatic parking system, *Sensors* .

Appendices

A Physical Parameters

Parameter	Description	Value	Unit
L	Length	2.0	[m]
B	Beam	1.08	[m]
m	Mass	55.0	[kg]
r_g	Center of gravity for the hull	$[0.2 \;\; 0 \;\; -0.2]^\top$	[m]
R_{44}	Radii of gyration	$0.4 \cdot \mathrm{B}$	[m]
R_{55}	Radii of gyration	$0.25 \cdot \mathrm{L}$	[m]
R_{66}	Radii of gyration	$0.25 \cdot \mathrm{L}$	[m]
B_{pont}	Beam of one pontoon	0.25	[m]
Y_{pont}	Distance from centerline to waterline area center	0.395	[m]
$C_{w,pont}$	Waterline area coefficient	0.75	[–]
$C_{b,pont}$	Block coefficient	0.4	[–]

Table A1: Physical parameters of the Otter USV

Henrik Bjering Strand

Autonomous Docking Control System for the Otter USV: A Machine Learning Approach

NTNU
Norwegian University of
Science and Technology