

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Marine Technology

Kristoffer Borgen Knudsen

Deep Learning for Station Keeping of AUVs

Master's thesis in Marine Technology
Supervisor: Ingrid Schjøtlberg
June 2019

Kristoffer Borgen Knudsen

Deep Learning for Station Keeping of AUVs

Master's thesis in Marine Technology
Supervisor: Ingrid Schjøllberg
June 2019

Norwegian University of Science and Technology
Faculty of Engineering
Department of Marine Technology

 **NTNU**
Norwegian University of
Science and Technology



MASTER THESIS IN MARINE TECHNOLOGY

Spring 2019

FOR

Kristoffer Borgen Knudsen

Deep Learning for Station Keeping of AUVs

Control of underwater vehicles remains an active research topic within the literature. Multiple challenges exist for controlling an underwater vehicle, including highly nonlinear effects due to hydrodynamics and limited sensing capabilities, which renders the state space unobservable. Control based models seek to model the underlying dynamics but suffer from balance between tractable computation and performance. Machine Learning (ML) control techniques show promise as an alternative to classical model-based approaches.

Station keeping is defined as the ability of a vehicle to maintain a constant position and orientation (pose) with regard to a reference object and is crucial for executing underwater operations efficiently. Due to this, this thesis aims to apply ML techniques in station keeping of AUVs.

Objective

The master thesis aims to investigate the possibilities of applying deep learning to accomplish station keeping control of AUVs.

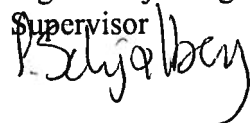
Tasks:

1. Develop a project plan for the master thesis work.
2. Perform a literature review on state-of-the-art ML methods for control.
3. Develop a ML-based control design for station keeping.
4. Train the controller in a simulated environment.
5. Validate the performance in a simulated and real-life environment.
6. Discuss the results.
7. Draw the conclusions from the studies and discuss possible further research steps.
8. Write report.

Supervisor : Ingrid Schjølberg
Co-supervisor : Mikkel Cornelius Nielsen

Submitted : January 15th 2019
Deadline : June 11th 2019

Ingrid Schjølberg

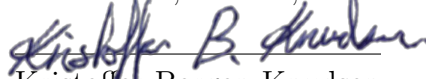
Supervisor


Preface

This thesis is the final product of a *Master of Science* degree in marine technology, with specialisation in cybernetics, at the Norwegian University of Science and Technology. The research conducted in this thesis spans over the period from January 2019 to June 2019.

The thesis aims to discover the possibilities for *station keeping* of autonomous underwater vehicles by the use of *machine learning*, especially focusing on *deep learning* principles. The reader is assumed to have some prior knowledge within machine learning, hydrodynamics and control theory.

Trondheim, June 5., 2019



Kristoffer Borgen Knudsen

Acknowledgement

I would like to express gratitude towards my supervisor, Professor Ingrid Schjøberg, for allowing me to investigate the possibilities on this subject, as well as guidance throughout the project period. Furthermore, I want to give a special thanks to Post-doctoral Fellow Mikkel Cornelius Nielsen for his continuous support in the development of the simulation environment, as well as the valuable discussions throughout the project period.

K.B.K

Abstract

This thesis investigates the possibilities of applying machine learning techniques, more specifically *deep learning*, in station keeping of autonomous underwater vehicles, and it is a continuation of the preliminary work done by Knudsen [10].

The usage of deep learning in this thesis surrounds the development of a sufficient controller design to accomplish station keeping, meaning that the vehicle is kept stable in all six degrees of freedom (DOF), which is fundamental for performing underwater operations efficiently. The process of doing underwater control design is complicated, which is mainly due to the complex underwater environment. The environment makes the control nonlinear since the vehicle is sensitive to flow and hydraulic resistance. In turn, the classical model-based approaches become challenging to apply. These disadvantages, together with the rapid developments in artificial intelligence (AI), have triggered the interest of using machine learning (ML) techniques in underwater control designs.

To develop and train an ML-based controller for accomplishing station keeping, there is used a dynamic model of the BlueROV2, which is a small, remotely operated vehicle. The vehicle is controlled in all 6-DOF, though suggesting the use of a dual control design which encompasses a *Deep Deterministic Policy Gradient* (DDPG) algorithm in conjunction with a *Proportional-Derivative* (PD) controller. To sufficiently train the algorithm, two simulation environments, Gazebo and Robot Operating System (ROS), are used in combination with the machine learning framework TensorFlow. In the simulated environment, the BlueROV2 dynamic model is not connected by a tether, and is therefore assumed to behave as an autonomous underwater vehicle.

The simulation results shows that it is possible to sufficiently train a machine learning algorithm to accomplish station keeping at an arbitrary pose, in all 6-DOF. To accomplish this the dual control design splits the fully actuated BlueROV2 by using the DDPG algorithm to control the thrust input in surge x and sway y , and the PD controller to control the thrust input in heave z , pitch ϕ , roll θ and yaw ψ . Validation of the results shows that the vehicle accomplish station keeping with error values in the order of $10^{-2}m$ in simulation, and $10^{-1}m$ in real-life experiments. The main reasons for the differences are flaws in the dynamic model used in training, as well as weaknesses related to the real-life pose measurement equipment.

The work in this thesis has also resulted in an abstract submitted to the IEEE Oceans 2019 conference in Seattle, which is given in Appendix B.

Sammendrag

Denne oppgaven undersøker mulighetene for å anvende maskinlæring, mer spesifikt *dyp læring*, for å holde et autonomt undervannsfartøy i en fast ønsket posisjon (eng: station keeping). Dette er en videreføring av forarbeidet som ble gjort av Knudsen på temaet [10].

Bruken av dyp læring i denne oppgaven omhandler utviklingen av et tilstrekkelig kontrolldesign for å oppnå station keeping, som betyr at fartøyet holdes konstant i alle seks frihetsgrader (6-DOF, eng: six degrees of freedom). Station keeping egenskaper er ekstremt viktig for å kunne gjennomføre undervannsoperasjoner effektivt. Undervanns kontrolldesign er komplisert, noe som er mye grunnet det komplekse undervannsmiljøet. Miljøet resulterer i ikke-lineær kontroll siden fartøyet er sensitiv til strømning og hydraulisk motstand. Dette gjør at klassiske model-baserte kontrolldesign er vanskelig å anvende. Disse vanskelighetene, sammen med den raske utviklingen som har vært i kunstig intelligens, har skapt en stor interesse rundt mulighetene for å bruke maskinlæringsteknikker i undervanns kontrolldesign.

For å utvikle samt trene en kontroller basert på maskinlæring for station keeping brukes en dynamisk modell av en BlueROV2, som er et lite fjernstyrt undervannsfartøy. Fartøyet blir kontrollert i 6-DOF ved å foreslå bruken av et todelt kontrolldesign som består av en *Deep Deterministic Policy Gradient* (DDPG) algoritme og en *Proportional-Derivative* (PD) kontroller. For å trene kontrolleren brukes to simuleringsmiljøer, Gazebo og Robot Operating System (ROS), sammen med maskinlæringsrammeverket TensorFlow. I simulering har BlueROV2en ikke en fysisk tilkobling til en operatør, noe som gjør at det er antatt at den opptrer som et autonomt undervannsfartøy.

Resultatene fra simulering viser at det er mulig å tilstrekkelig trene en maskinlæringsalgoritme for å oppnå station keeping i 6-DOF. Dette blir gjort ved å bruke DDPG algoritmen til å kontrollere thrust bidraget i x og y , og PD kontrolleren til å kontrollere thrust bidraget i z , ϕ , θ og ψ . Ved å validere resultatene viser det seg at station keeping blir oppnådd med avvik i størrelsesordenen $10^{-2}m$ i simulering og $10^{-1}m$ for det virkelige systemet. Hovedgrunnen til forskjellene er at den dynamiske modellen ikke er helt eksakt samt mangler i det virkelige posisjonssystemet.

Arbeidet med denne oppgaven har også resultert i et abstract til IEEE Oceans 2019 konferansen i Seattle. Bidraget finnes i Appendix B.

Contents

Preface	ii
Acknowledgement	iii
Abstract	iv
1 Introduction	1
1.1 Background	1
1.2 Traditional Methods for Station Keeping of AUVs	2
1.3 Motivation for ML-based Control Designs	4
1.4 Contributions	5
1.5 Structure of the Report	6
2 Machine Learning - A Literature Review	7
2.1 Machine Learning	7
2.2 An Introduction to Reinforcement Learning	9
2.2.1 Markov Decision Processes	10
2.2.2 Dynamic Programming	13
2.2.3 Policy Gradients	14
3 Deep Reinforcement Learning	20
3.1 Deep Deterministic Policy Gradients	20
3.1.1 Q-Learning	22
3.1.2 Policy Learning	24
3.1.3 Exploration vs. Exploitation	24
3.2 Challenges in Policy Gradient Methods	25
3.3 Trust Region Policy Optimisation	27
3.3.1 Minorize-Maximization Algorithm	27
3.3.2 Trust Region	28
3.3.3 Importance Sampling	28
3.3.4 TRPO	30

3.4	Proximal Policy Optimisation	31
3.4.1	Optimisation	31
3.4.2	Adaptive Kullback-Leibler Penalty	33
3.4.3	Clipped Surrogate Objective	33
4	Modeling	36
4.1	Reference Frames	37
4.2	Dynamic Model of Underwater Vehicles	38
4.2.1	Mass Matrix	39
4.2.2	Coriolis and Centripetal Force Matrices	39
4.2.3	Damping Matrices	40
4.2.4	Hydrostatic Terms	40
5	Method	41
5.1	Controller Implementation for the BlueROV2	41
5.1.1	Implementation of the PD Controller	42
5.1.2	Implementation of the DDPG Algorithm	43
5.1.3	Controller Architecture	46
5.1.4	Reward Function	49
5.2	Simulation and Experiment Setup	49
5.2.1	Simulation Setup	49
5.2.2	Experiment Setup	50
6	Reward Shaping	53
6.1	Body-frame Error	53
6.2	Reward Shaping with Time Constraint	54
6.3	DDPG without Heading	56
7	Results	60
7.1	DDPG without Heading	61
7.1.1	Training Results	63
7.1.2	Validation Results: Simulation	65
7.1.3	Validation Results: MC-lab Experiments	67
7.2	DDPG with Heading	78
7.2.1	Training Results	78
8	Discussion	82
8.1	Quality of Pose Measurement Sensors	82
8.2	Dynamic Model	83
8.3	Limitations in Software	84
8.4	Difficulties in Reward Function Design	84

8.5	Possibilities of TRPOs and PPOs	85
9	Conclusions	87
10	Further Work	89
A	BlueROV2 Parameters	95
A.1	Rigid Body Mass Matrix	95
A.2	Added Mass Matrix	95
A.3	Damping Matrices	96
B	IEEE Oceans 2019 Seattle - Abstract	97

List of Figures

1.1	GNC Signal Flow [4]	2
2.1	Machine Learning	8
2.2	Reinforcement Learning Architecture [26]	10
2.3	Look-ahead Tree [26]	12
2.4	Policy Iteration by Acting Greedily [26]	14
2.5	The Bell Curve [23]	16
2.6	Baysian Distribution [23]	17
2.7	Markov Chains	18
3.1	Policy Model Represented as a Net	26
3.2	The Minorize-Maximization Algorithm [5]	27
3.3	Performance of Clipped Surrogate Objective [22]	35
4.1	BlueROV2 by BlueRobotics [1]	36
4.2	Body-fixed and Earth-fixed Reference Frames	37
5.1	Controller Architecture for the BlueROV2	46
5.2	Gazebo and Robot Operating System (ROS)	50
5.3	MC-lab Setup	51
5.4	Qualisys Mapping	52
6.1	Circular Movement in x_e	55
6.2	Lack of Heading Information	56
6.3	Conflict Between ψ and $[x, y]$	58
7.1	DDPG without Heading: Norm of the Error State	62
7.2	DDPG without Heading: Total Reward per Episode	64
7.3	DDPG without Heading: Number of Steps per Episode	64
7.4	DDPG without Heading: Number of PD Penalties per Episode	65
7.5	DDPG without Heading: Performance Validation, $[x_d, y_d, \psi_d]=[2,0,0]$	66
7.6	DDPG without Heading: Performance Validation, $[x_d, y_d, \psi_d]=[2,2,0]$	66

7.7	Experiment 1: DDPG State Error	68
7.8	Experiment 1: PD State Error	69
7.9	Experiment 1: Force	69
7.10	Experiment 1: Torque	70
7.11	Experiment 2: DDPG State Error	71
7.12	Experiment 2: PD State Error	72
7.13	Experiment 2: Force	72
7.14	Experiment 2: Torque	73
7.15	Experiment 3: DDPG State Error	74
7.16	Experiment 3: PD State Error	75
7.17	Experiment 3: Force	75
7.18	Experiment 3: Torque	76
7.19	Experiment 3: DP 4-corner Test	77
7.20	DDPG with Heading: Total Reward per Episode	79
7.21	DDPG with Heading: Number of Steps per Episode	80
7.22	DDPG with Heading: Number of PD Penalties per Episode	80
8.1	DDPG: Reward Function	85

List of Algorithms

1	DDPG Algorithm	48
2	Reward Function with Time Constraint	54
3	DDPG without Heading: Reward Function	62

List of Tables

4.1	The Notation of SNAME (1950) for Marine Vessels	38
6.1	Q-Matrix: without Heading	57
6.2	Q-Matrix: with Heading	57
7.1	DDPG: Learning Parameters	60

Acronyms

Adam adaptive moment estimation.

AI artificial intelligence.

ANN artificial neural network.

AUV autonomous underwater vehicle.

BODY body-fixed reference frame.

CNN convolutional neural network.

COB centre of buoyancy.

COG centre of gravity.

DDPG deep deterministic policy gradient.

DP dynamic positioning.

DPG deterministic policy gradient.

DRL deep reinforcement learning.

GNC guidance, navigation and control.

GNSS global navigation system.

IMU inertial measurement unit.

INS inertial navigation system.

IS importance sampling.

KL kullback-leibler.

MBSE mean-squared bellman error.

MC marine cybernetics.

MCMC markov chain monte-carlo.

MDP markov decision process.

ML machine learning.

MM minorize-maximization.

MP markov property.

NED north-east-down reference frame.

PG policy gradient.

PID proportional, integral, derivative control.

PPO proximal policy optimisation.

RL reinforcement learning.

ROS robot operating system.

ROV remotely operated vehicle.

SPG stochastic policy gradient.

TD temporal difference.

TRPO trust region policy optimisation.

VS visual servoing.

Chapter 1

Introduction

This thesis is a continuation of the work done by Knudsen on *station keeping of AUVs*, which presented a preliminary investigation of the possibilities on the subject [10]. Knudsen showed that through the use of machine learning (ML) techniques, more specifically deep reinforcement learning (DRL), it was possible to control an AUV to reach the desired pose in a simulated environment. However, the developed algorithm, or agent, was not successful in doing station keeping at the pose. The studies done in this thesis aims to succeed in achieving station keeping of AUVs through DRL techniques, both in a simulated environment and in real-life experiments.

1.1 Background

Remotely Operated Vehicles (ROVs) are unmanned and highly manoeuvrable underwater vehicles either connected to a surface-vessel or land, through tethers. The tethers transmit commands and control signals between an operator and the ROV, resulting in remote navigation of the vehicle. Primarily, ROVs have been used to perform inspection tasks, for example, pipeline inspections and exploration of the oceans. ROVs are commonly divided into classes based on parameters such as weight, power, capabilities and size. Larger ROVs, which can hold more sensors and mechanical tools, are usually used for intervention tasks on subsea installation sites. Smaller ROVs usually perform tasks related to exploration, such as exploration of minuscule areas, e.g. cavities or pipeline cracks [3].

The main difference between ROVs and *Autonomous Underwater Vehicles* (AUVs) is that the latter are *un-tethered*. As for ROVs, a surface-vessel or land is used to

communicate with AUVs, but the communication is now usually through a satellite. The removal of tethers makes the AUVs more manoeuvrable and extends their application area. However, when removing the tethers, the AUVs also become highly dependent on battery life, which is now limited. Because of this, AUVs typically perform less power consuming task compared to ROVs, such as surveillance operations.

One of the long-term benefits in the development of AUV techniques is reduced cost. Today, both ROVs and AUVs require either connection with a surface-vessel or by land. Using a surface-vessel is a costly operation, mainly because of vessel day rates. Furthermore, *recovery* and *launch*, meaning taking the vehicle up and down from the ocean are complex and costly operations. Due to this, the long-term goal of designing *life-of-field* AUVs, meaning that the AUVs are *resident* at the installation site, could be a great economic benefit. However, to accomplish this, there has to be further development in robust sensors and autonomous control systems.

1.2 Traditional Methods for Station Keeping of AUVs

Motion Control is achieved through three different branches; *guidance*, *navigation* and *control* (GNC), and it contains design of systems that automatically control or remotely control devices or vehicles that are moving underwater, on the surface or in space [4]. The flow between these three systems, and how they interact through data and signal transmission, is illustrated in Figure 1.1.

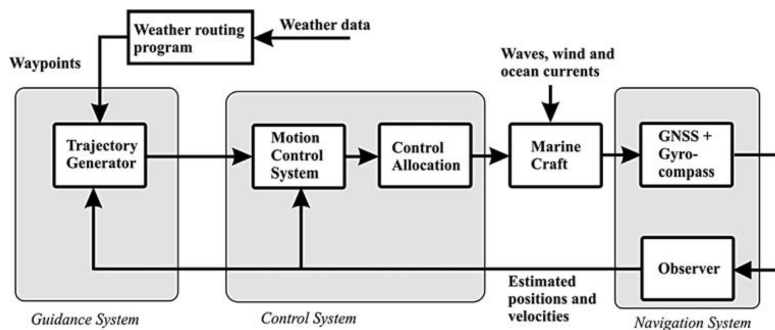


Figure 1.1: GNC Signal Flow [4]

The branches in GNC systems are defined as follows.

- **Guidance** deals with computation of the desired pose, velocity and acceleration, which is transmitted into the control system. Guidance systems usually consists of motion sensors, external data (such as weather data in Figure 1.1) and a computer.
- **Navigation** is the process of directing a craft by determining the pose, course and distance travelled. To do this, global navigation systems (GNSS), inertial navigation systems (INS) and motion sensors, such as inertial measurement units (IMU), are usually used.
- **Control** (motion control) deals with computing the necessary control forces and moments based on some *control objective*, for example station keeping.

Station keeping control falls into the last branch in GNC, and is defined as the ability of a vehicle to maintain a constant position and orientation (pose), relative to a reference object [20]. Station keeping capabilities are essential for the performance of AUVs, to reduce risk, as well as executing intervention tasks efficiently. In relation to a surface-vessel, station keeping can be viewed as the equivalent to *dynamic positioning* (DP) [4] above the surface.

Sufficient control design is crucial to accomplish station keeping capabilities, and today, most solutions to this surrounds *classical* control designs. Traditional controller architecture of underwater vehicles are usually based around the principle of *Proportional-Integral-Derivative* (PID) algorithms. In all simplicity, the PID controller is a control-loop feedback algorithm, which continuously calculates an error value based on some control objective. For example, if the goal is to do station keeping of an underwater vehicle, the error value would be based on some desired pose and the current pose of the vehicle. The PID algorithm is given in Equation 1.1, where K_p , K_i and K_d denote the coefficients for the proportional, integral and derivative gains, respectively.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d(t)}{dt} \quad (1.1)$$

The algorithm uses the *proportional* term to reduce the error, the *integral* term to remove possible stationary deviation¹ and the *derivative* term takes the error values rate of change into account, in order to compensate for future trends of error.

¹a constant error

1.3 Motivation for ML-based Control Designs

Accomplishing a sufficient PID controller for a given control objective is done by tuning the controller gains in Equation 1.1. The methods of tuning a PID controller are many [4], but they usually surrounds using the mass- and damping matrices of the system, as well as defining the natural frequency ω_n and the natural damping ζ_n , resulting in *constant* gains. However, the underwater environment is highly complex, making the autonomous control nonlinear since flow and hydraulic resistance easily influences the AUVs motions [29]. The result of this is that traditional control techniques, such as nonlinear control with PID [8], is a difficult design process. These challenges, in combination with the rapid developments in *artificial intelligence* (AI), has made many scholars interested in the possibility of applying machine learning (ML), which is a method of accomplishing AI, in these types of control designs.

As mentioned above, the main reason for investigating the possibilities of ML in underwater control is that identifying and modelling a real-life system is challenging and in some cases infeasible due to unobservability and highly nonlinear effects, which is the case in the underwater environment [15]. The investigation done in this thesis surrounds the branch within ML that is called reinforcement learning (RL). In RL, there is no need to model any system or possess any knowledge about the environment, meaning that the challenges related to this are non-existent. In traditional control design, the controller gains, e.g. PID gains, are tuned based on parameters such as the control objective and the sea state in the operating environment. This tuning makes them less agile when it comes to changes in these parameters. On the other hand, an ML-based controller does not care about either of these and in theory, ML-based controllers should perform just as good independent of the environment and control objective. Furthermore, the RL framework, discussed in detail in Chapter 2, basically describes a closed loop system, which is the basis for every traditional control design.

The result of this is that ML, and especially RL techniques, have experienced a steep increase in both popularity and application during the recent years. From this, several promising RL based techniques have emerged, and in the following section, some of these are discussed.

In 2015 Mariano De Paula and Gerardo D. Acosta proposed a trajectory tracking algorithm for autonomous vehicles using *Adaptive Reinforcement Learning* [14], but this was too general for usage in AUV control. One of the more significant difficulties, which is also one of the primary goals of AI in control, is to solve complex tasks from unprocessed, high-dimensional and sensory input [12]. Based on this,

David Silver et al. proposed a *Deterministic Policy Gradient* (DPG) algorithm, for performing complex tasks with high dimensionality and perceptible input [25]. This algorithm showed significantly better performance than using *Stochastic Policy Gradients* (SPG), as well as having usage within nonlinear optimisation problems as well. Yu Runsheng et al. based their research on this when they in 2017 proposed a *Deep Deterministic Policy Gradient* (DDPG) for trajectory tracking control of AUVs [29], which showed significant improvements compared to traditional methods.

The preliminary studies proposed a *dual* control design, by combining a PD controller with a DDPG algorithm based on the work of Silver et al. and Runsheng et al. [10]. The investigation done in this thesis builds further on this research, where the primary goal is to achieve sufficient station keeping capabilities of the BlueROV2.

1.4 Contributions

The main contributions from this thesis are listed below.

1. An introduction into how DRL techniques can be used in control of underwater vehicles.
2. An implementation of a dual control design, which encompasses a DDPG algorithm in conjunction with a PD controller.
3. An implementation of a DRL controller in Gazebo and Robot Operating System (ROS).
4. A verification of that it is possible to train ML-based algorithms in a simulated environment and apply them to a real-life system.
5. A development of a sufficient reward function design to achieve station keeping capabilities.
6. A trained DRL based controller which accomplishes station keeping with error values in the order of $10^{-2}m$ in simulation, and $10^{-1}m$ on a real-life system.
7. An abstract has been submitted to the IEEE Oceans 2019 conference in Seattle, presented in Appendix B.

1.5 Structure of the Report

This thesis aims to showcase how machine learning techniques, and especially deep reinforcement learning, can be of advantage in doing station keeping control of underwater vehicles. To do this, the following chapters are presented.

Chapter 2 presents the overlying concepts of machine learning and a profound investigation into the principles of reinforcement learning and how this can be used in underwater control designs.

Chapter 3 introduces deep reinforcement learning, which is the basis of how to use ML in underwater control designs. This chapter considers three state-of-the-art algorithms within the DRL family: Deep Deterministic Policy Gradients (DDPGs), Trust Region Policy Optimisation (TRPO) and Proximal Policy Optimisation (PPO).

Chapter 4 presents the dynamic model of the BlueROV2.

Chapter 5 presents the method of developing a dual controller design for the BlueROV2. This chapter also presents the simulation and experiment setup, which is used for training and validation.

Chapter 6 discuss sufficient reward function design to accomplish station keeping.

Chapter 7 presents the training results and validation results from the investigation.

Chapter 8 contains a discussion about the results and the prospects of using machine learning in control.

Chapter 9 concludes the thesis.

Chapter 10 discuss further work on the subject.

Chapter 2

Machine Learning - A Literature Review

This chapter presents an introduction into the concepts of machine learning (ML), especially focusing on the principles of reinforcement learning (RL) and policy gradients (PG), which is the essence of Deep Deterministic Policy Gradient (DDPG) algorithms.

2.1 Machine Learning

In 1959 Arthur Samuel coined the concept of machine learning as *a field of study that gives computers the ability to learn without being explicitly programmed*. The concept was further defined in 1998 when Tom Mitchell proposed a more precise definition as

A computer program is said to learn from experience E with respect to some task T and some performance measure P , if its performance on T , as measured by P , improved with experience E [19].

To clarify this definition, one can look at an AUV performing an intervention task at an installation site. By doing this specific task, T , over and over again, the performance measure, P , would be an indicator of how well the task is performed. If P increases with the experience E from doing T over and over again, the AUV is learning. ML techniques are usually divided into three distinctive methods. These are displayed in Figure 2.1.

1. *Supervised Learning*

2. *Unsupervised Learning*
3. *Reinforcement Learning*

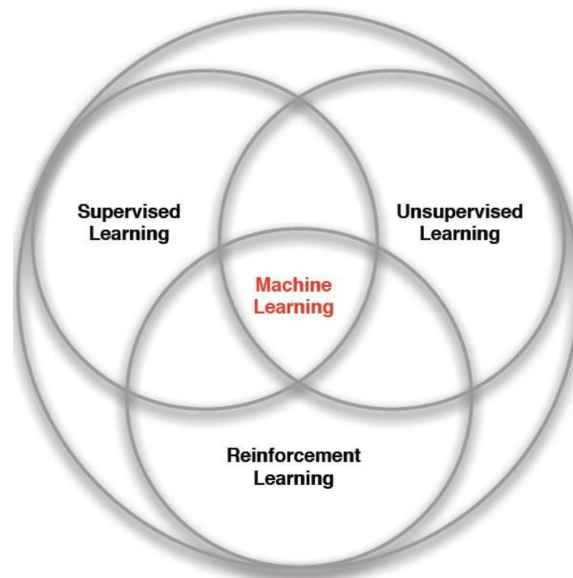


Figure 2.1: Machine Learning

Supervised learning is the concept of using known data distributions to train an algorithm [24]. The algorithm is trained using large sets of data, preferably in the order of 10^4 and greater, and the goal is to teach the algorithm how to recognise any new data input. From a scientific perspective, the algorithm has access to both the input data x and the output data y , and tries to do model fitting based on these. Supervised learning is very common in image classification tasks, where *Convolutional Neural Networks*¹ (CNN) are the state-of-the-art method. In image classification tasks the network is trained by using large sets of data, including images (input) and their respective labels (output). If the network is sufficiently trained, the overall goal is to use a new image as input and have the algorithm output the correct label.

¹see chapter 2.3.1 in Knudsen [10]

Unsupervised learning, on the other hand, is not given data to learn from, but the algorithm is left alone to discover interesting features in the data [11]. Mathematically the algorithm has the input data, x , but no information about the output data. This branch of learning is further divided into two sub-branches; *Association* and *Clustering*. In association, the algorithm tries to discover *rules*, which describes large portions of the data. For example, *when the incident P occurs, L also tends to happen*. In clustering, the algorithm looks for clusters in the data, such as specific behaviour or trends.

2.2 An Introduction to Reinforcement Learning

In *Reinforcement Learning* (RL) the overall goal is to train an agent² to perform correct actions in an environment. This is done by giving the agent a *reward* based on how good it was to take a particular *action* in the current state. The reward, R_t , is defined as a scalar feedback signal, indicating how good it was to take that particular action at that specific step in time, t . The overall goal is to maximise the total cumulative reward over all time-steps such that the optimal *policy* is found, starting in any state. A policy in reinforcement learning can be interpreted as a strategy. By defining the reward as a scalar signal, this means that we assume that every action in the real world can be weighted against each other, meaning that even ethical dilemmas are solvable.

As stated, the goal of reinforcement learning is to maximise the total cumulative reward over all actions in order to find the optimal policy. This is done by taking an action, A_t , in the environment. An important note here is that taking an action at the time-step t might have a long-term consequence, meaning that the agent does not necessarily experience the consequence of taking that action in the next time-step. This is the same principle as, e.g. a financial investment.

In Figure 2.2, the relationship within reinforcement learning is visualised. At each time-step t the agent will execute an action A_t , receive an observation O_t , and a scalar reward R_t . The environment will receive an action A_t , and return an observation O_t and a scalar reward R_t to the agent.

²algorithm, computer program

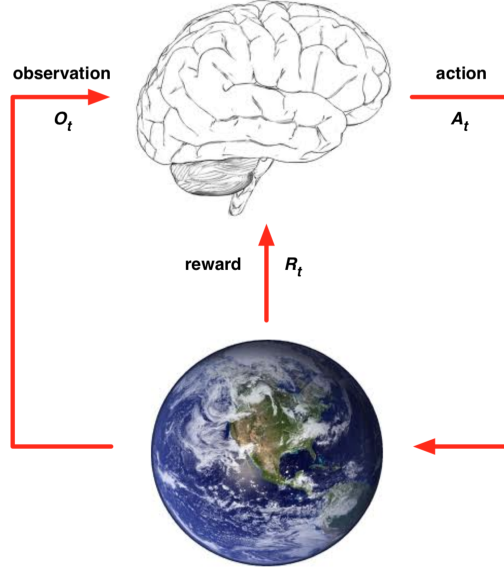


Figure 2.2: Reinforcement Learning Architecture [26]

2.2.1 Markov Decision Processes

When introducing the concepts of reinforcement learning it is necessary to determine what types of environments that are possible to *solve* using this type of algorithm. A *Markov Decision Process* (MDP) describes a fully *observable* environment for reinforcement learning [26]. A given state, S_t , is said to be *Markov* if and only if it satisfies the *Markov Property* (MP), given by

$$P[S_{t+1}] = P[S_{t+1}|S_1, \dots, S_t] \quad (2.1)$$

In all generality the Markov Property states that *the future is independent of the past given the present*, meaning that the state will capture all the relevant information from the history, but the history may be erased once the state is known. Observe that the Markov Property in Equation 2.1 defines a conditional independence property and can therefore be represented by a *chain Bayesian network*. The benefit of having this representation is revealed in the introduction to policy gradients.

In an environment where all states are Markov, a MDP is a Markov reward process with decision. This is given by a tuple

$$\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$$

where,

- \mathcal{S} - a finite set of states.
- \mathcal{A} - a finite set of actions.
- \mathcal{P} - a state transition probability matrix.
- \mathcal{R} - a reward function, where $\mathcal{R}_s^a = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$.
- γ - a discount factor, $\gamma \in [0, 1]$.

The reason for using a discount factor, γ , on the reward has several advantages. From a mathematical perspective it is convenient to discount, as well as making sure that infinite returns are avoided. Furthermore, there is also an uncertainty about the future that is unknown, and the agent do not know if an immediate reward is more valuable than a future reward. In MDPs, three definitions are essential; the *policy* π , the *state-value* function $V_\pi(s)$, and the *action-value* function $q_\pi(s, a)$.

- The *policy* determines how the agent behaves in the environment, meaning which policy it is following. The policy is dependent on the current state S_t and the action A_t taken in that state. This is defined as

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \quad (2.2)$$

- The *state-value function* is the expected return G by starting in a state s and following a specific policy π . The state-value function is defined as

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \quad (2.3)$$

- The *action-value function* takes the chosen action into account. Computing the expected return G from starting in a state s , taking an action a_t and following a policy π . The action-value function is defined as

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \quad (2.4)$$

These three definitions are essential for optimising the MDP. In order to use them the *Bellman Expectation Equation* is introduced, which transforms the definitions above into a new set of *solvable* definitions [26]. This means that both the state-value function and the action-value function can be decomposed into an immediate reward, plus the discounted value of the successor state. The Bellman Expectation equations are given in Equation 2.5 and 2.6.

- *Bellman Expectation Equation* for the state-value function $V_\pi(s)$ is given by

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) (\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_\pi(s')) \end{aligned} \quad (2.5)$$

- *Bellman Expectation Equation* for the action-value function $q_\pi(s, a)$ is given by

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \\ &= \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a') \end{aligned} \quad (2.6)$$

By solving the Bellman Expectation equations, the algorithm is able to look one step ahead based on the current state and the policy it is following. This concept can be explained through a *look-ahead tree*, which is illustrated in Figure 2.3. Overall, the look-ahead tree lets the agent evaluate how *good* it is to take a particular action, given the current state and policy.



Figure 2.3: Look-ahead Tree [26]

The overall goal of the MDP is to find the optimal behaviour for the agent, which means that the MDP is *solved* when the agent has found the optimal value function, resulting in the best possible policy. This is done by taking the maximum value function, \max_π , over both the state-value function and the action-value function. The optimal policy for the agent is then found by maximising over the optimal action-value function.

In conclusion, MDPs, which are environments consisting of a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, is very suitable to solve with reinforcement learning. The tuple above also shows that an underwater environment can be defined as a MDP. The state (S) would be the location of the AUV in the environment, the action (A) would be the direction the AUV is heading in the given state, and the reward (R) would say something

about how good it is for the AUV to go in this specific direction, based on the control objective. By defining the underwater environment as an MDP, it shows that reinforcement learning can be applied to *solve* control problems in the underwater environment. Furthermore, the feedback from the environment shows that the reinforcement learning framework essentially is a feedback system, which is the essence of every traditional control design.

2.2.2 Dynamic Programming

Solving MDPs often results in a complex and computational expensive process, which is where *dynamic programming* (DP) comes in handy. DP is a method for solving complex problems, where the problem is broken down into sub-problems, and solved separately, before combining the solutions [26]. DP works very well for problems with the following properties; *optimal substructure*. These properties state that the problem can be decomposed into overlapping sub-problems, resulting in solutions that can be cached and reused. In MDPs, using the Bellman equations results in a recursive decomposition, with the value function both storing and reusing solutions, which means that MDPs satisfies both properties.

In reinforcement learning, the DP algorithm takes the MDP as input, and return the optimal value function (V^*) and an optimal policy (π^*). The way it does this is by iteratively applying the Bellman Expectation Equations, given in Equation 2.5 and 2.6. This is done by starting with an arbitrary initial value function and sweep over all states in each iteration. The policy is improved by acting *greedily* with respect to V_π , given by $\pi' = greedy(V_\pi)$. Acting greedily means that the agent chooses the action that it believes has the best long-term reward. This concept is illustrated in Figure 2.4. The algorithm initialises an arbitrary state-value and policy function, evaluates them, and acts greedily to receive an improved state-value and policy function. By iteratively doing this, the policy will *always* converge to the optimal value function V^* and the optimal policy π^* [26].

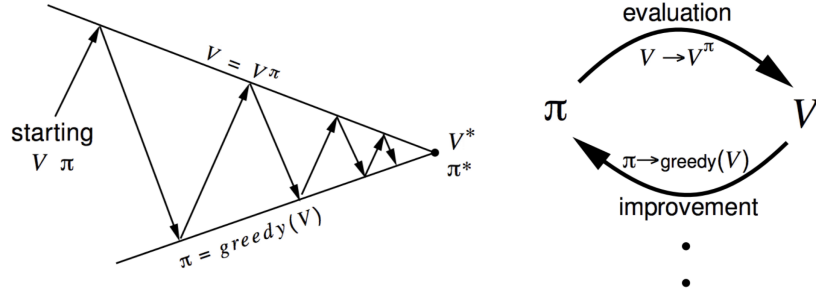


Figure 2.4: Policy Iteration by Acting Greedily [26]

2.2.3 Policy Gradients

As stated throughout this chapter, the objective of reinforcement learning is to maximise the total future cumulative reward r , in order to find the optimal policy π . The total reward for a given policy can be found by solving

$$J(\theta) = \mathbb{E}_{\pi}[r(\tau)] \quad (2.7)$$

In Equation 2.7, $r(\tau)$ is the total reward for following a given policy, and θ is a defined set of parameters to describe the given policy π_{θ} . From the definition of MDPs, it is known that every MDP has *at least* one optimal policy, as well as at least one stationary³ and deterministic⁴ policy. To find this optimal policy, the agent needs to find the parameters θ that maximise J , which is commonly done in machine learning literature through *gradient ascent* (or descent) methods [7]. Gradient ascent methods are based on starting with an arbitrary set of parameters and then stepping through by the use of an update rule. This is given by

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \quad (2.8)$$

In order to find the optimal policy, the agent needs to estimate the gradient of the update rule that contains the optimal set of parameters θ_{t+1} . One way to do this is through integration, but integrals present a challenge because of their computational complexity and costs. To avoid integral computation, the first step is to reformulate

³the policy only returns action distributions depending on the last visited state

⁴the policy deterministically selects actions based on the current state

the gradient by expanding the total reward gradient.

$$\nabla \mathbb{E}_\pi[r(\tau)] = \nabla \int \pi(\tau)r(\tau)d\tau \quad (2.9)$$

$$= \int \nabla \pi(\tau)r(\tau)d\tau \quad (2.10)$$

$$= \int \pi(\tau)\nabla \log \pi(\tau)r(\tau)d\tau \quad (2.11)$$

$$= \mathbb{E}_\pi[r(\tau)\nabla \log \pi(\tau)] \quad (2.12)$$

This is known as the *Policy Gradient Theorem*, which states that *the derivative of the expected reward is the expectation of the product of the reward and gradient of the log of the policy π_θ* [7].

$$\nabla \mathbb{E}_{\pi_\theta}[r(\theta)] = \mathbb{E}_{\pi_\theta}[r(\tau)\nabla \log \pi_\theta(\tau)] \quad (2.13)$$

The next step is to expand the definition of the policy $\pi_\theta(\tau)$. This results in

$$\pi_\theta(\tau) = \mathcal{P}(s_0) \sum_{t=1}^T \pi_\theta(a_t|s_t)p(s_{t+1}, r_{t+1}|s_t, a_t) \quad (2.14)$$

where \mathcal{P} is the ergodic distribution⁵ when starting in some arbitrary state s_0 , π_θ is the policy and p is the environmental dynamics, which determines which new state the agent will transition into. The terms are updated through the product rule of probability, which is allowed because of the Markov Property. Remember that the Markov Property states that every action is independent of the previous action, meaning that the product rule can be applied. T represents the total number of time steps. By taking the logarithm of Equation 2.14 the result becomes

$$\log \pi_\theta(\tau) = \log \mathcal{P}(s_0) + \sum_{t=1}^T \log \pi_\theta(a_t|s_t) + \sum_{t+1}^T \log p(s_{t+1}, r_{t+1}|s_t, a_t) \quad (2.15)$$

$$\nabla \log \pi_\theta(\tau) = \sum_{t=1}^T \nabla \log \pi_\theta(a_t|s_t) \quad (2.16)$$

$$\longrightarrow \nabla \mathbb{E}_{\pi_\theta}[r(\tau)] = \mathbb{E}_{\pi_\theta}[r(\tau) \left(\sum_{t+1}^T \nabla \log \pi_\theta(a_t|s_t) \right)] \quad (2.17)$$

This result shows that neither the ergodic distribution of states \mathcal{P} nor the environmental dynamics p provides relevant information to the agent, meaning that the

⁵a distribution that has the same behaviour averaged over time as averaged over space

agent don't really need to possess any information about these. Both \mathcal{P} and p are variables that are extremely difficult to measure exact, so removing these greatly reduces the complexity of the problem. This results in what is formally known as *model-free* algorithms, simply because the environment is not modelled. As stated in Chapter 1, one of the problems with traditional model-based controllers is that identifying and modelling a real-life system is challenging and in some cases infeasible. However, as shown in Equation 2.17, this problem is removed when using policy gradients.

Although the ergodic distribution of states and the environmental dynamics have been proved to be negligible in Equation 2.17, the integral term (expectation \mathbb{E}_{π_θ}) is still present. In order to deal with this term a common solution is to sample a large data set of trajectories, and average them out [7]. Although this is an approximation of the problem, it is an unbiased one and similar to the approximation of continuous space integrals into discrete domains. This approximation is known as *Markov Chain Monte-Carlo* (MCMC), and it is commonly used to approximate parametric probability distributions in Bayesian networks⁶.

Markov Chain Monte-Carlo

In its simplest form *Markov Chain Monte-Carlo* (MCMC) methods are used to approximate the posterior distribution of a parameter of interest by randomly sampling in a probabilistic space [23]. In these methods, a parameter of interest defines the specific parameter we are interested in, e.g. height in a region. The distribution defines the mathematical representation of every possible value for this parameter, and the probability of observing each value. This definition is often described through the Bell curve, displayed in Figure 2.5.

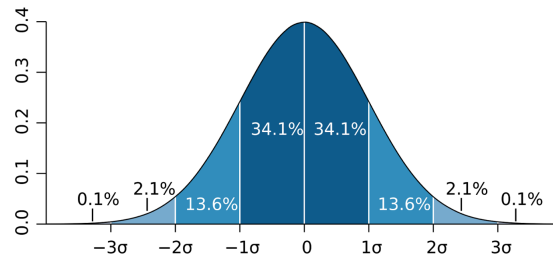


Figure 2.5: The Bell Curve [23]

⁶a type of probabilistic graphical models that model conditional dependencies between states

However, in Bayesian networks, the distribution has an additional layer of interpretation. This additional layers contains *beliefs*, meaning what our beliefs about the parameter of interest are. For example what is believed to be the human height in a region, which in a Bayesian distribution would be represented as in Figure 2.6.

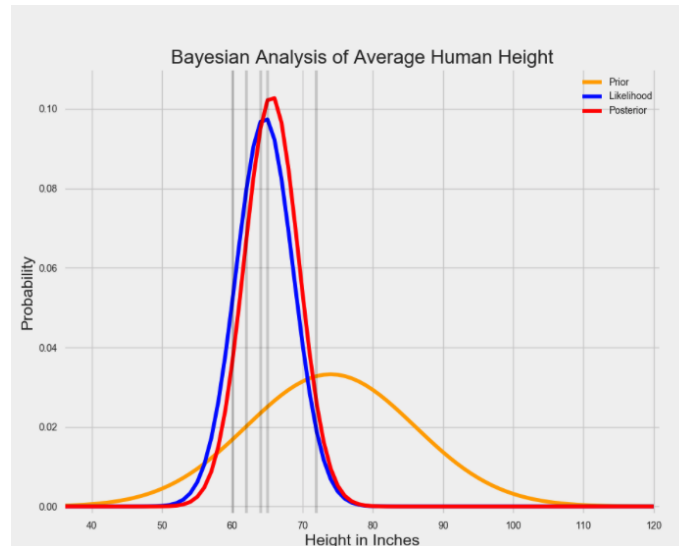


Figure 2.6: Bayesian Distribution [23]

The belief is represented by the *prior distribution* since this is our belief prior to seeing the data. On the other hand, the *likelihood distribution* represents the likelihood of the observed data. By combining these two distributions, the *posterior distribution* is achieved, which tells something about which parameter values that maximise the probability of observing a specific data, given our prior beliefs. As shown in Figure 2.6, the posterior distribution shows that we are relatively confident in the likelihood distribution from the data, but still believe that the average height is a little higher than what the data suggests. Determining the posterior distribution in the average height case is fairly simple since both the prior and likelihood distributions have a convenient curve. However, this is often not the case, and it can be impossible to determine the posterior distribution analytically. This is where the MCMC methods are applied.

First of all, MCMC methods are a type of *Monte Carlo* methods. Monte Carlo is a type of dynamic programming (DP) methods, and consist of simulations that repeatedly samples random numbers to estimate some fixed parameter. So, instead of estimating the parameter analytically, which often is impossible, the Monte Carlo simulation provides an approximation of the parameter. The second layer of MCMC

methods is *Markov Chains*, which describes a probabilistic relationship within a sequence of events. Each event in the cycle is the result of a set of outcomes, and each outcome determines which outcome that follows, based on a fixed set of probabilities.

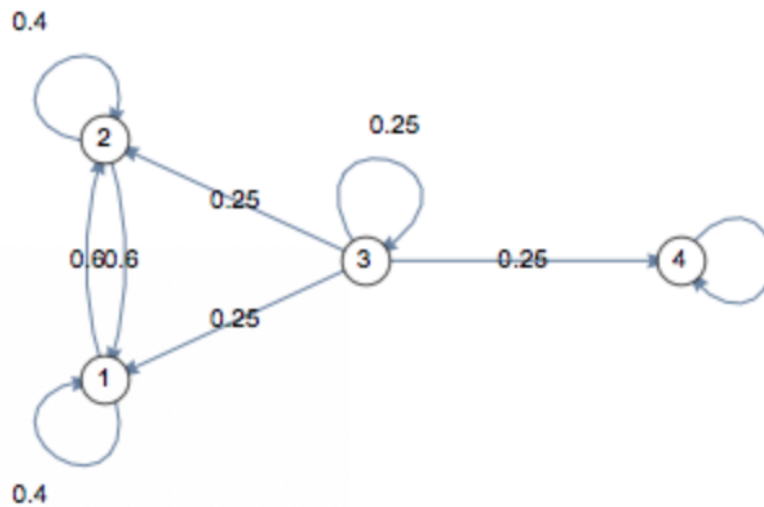


Figure 2.7: Markov Chains

The concept of Markov Chains is illustrated in Figure 2.7, where 4 states and their respective transition probabilities are shown. For example, if starting in state 3, there is a 25% probability of transitioning into state 1, 2 or 4, as well as a 25% probability of staying in state 3. This shows that Markov Chains are also *memoryless*, meaning that all information needed to predict the next event is available in the current state. Meaning that no information about the history of events is necessary [23]. This property is formally known as the *Markov Property*, which has been discussed in previous sections.

With some knowledge about how Monte Carlo simulations and Markov Chains work, we can begin to understand how MCMC methods are able to estimate the posterior distribution of a parameter of interest. Initially, MCMC methods choose an arbitrary parameter value, and through Monte Carlo simulation generate random values based on some rule of what determines a good parameter value [23]. To do this, the simulation looks at a pair of parameter values and computes the probability of each value explaining the data, based on the prior beliefs. The better value is added to a Markov Chain, which is updated as the simulation continuous and eventually outputs the posterior distribution of the property.

By doing this an approximation of Equation 2.17 is received, which can then be

used to find the optimal policy by finding the parameters θ that maximise J in

$$J(\theta) = \mathbb{E}_{\pi}[r(\tau)] \quad (2.18)$$

As stated previously, the use of policy gradients is common in reinforcement learning techniques, and ML in general, to find the optimal policy. The next chapter discuss *Deep Reinforcement Learning* (DRL) techniques, which further substantiates the usage and importance of policy gradients in RL.

Chapter 3

Deep Reinforcement Learning

This chapter includes a further investigation into reinforcement learning, more specifically, the branch of learning that is denoted *Deep Reinforcement Learning* (DRL). These are methods using the principles of *Artificial Neural Networks* (ANNs), which are based on the idea of how biological nervous systems, such as the human brain, operates [18]. In all simplicity, a neural network¹ is a connected graph with input neurons, output neurons and weighted edges [2]. In recent years there has been a rapid increase in new methods for deep reinforcement learning, primarily focusing on benchmark games such as Atari², AlphaGO³ and AlphaZero⁴.

From this rapid increase in both popularity and research, three methods have shown great promise, and are considered *state-of-the-art* methods within deep reinforcement learning. The first is known as *Deep Deterministic Policy Gradients* (DDPG), developed by Googles DeepMind, the second is *Trust Region Policy Optimisation* (TRPO), developed at UC Berkeley, and the third is named *Proximal Policy Optimisation* (PPO), developed by OpenAI Gym.

3.1 Deep Deterministic Policy Gradients

Deep Deterministic Policy Gradient (DDPG) algorithms are characterised as a type of policy gradients that are off-policy, model-free and actor-critic. Recall from Section 2.2.3 that policy gradients are algorithms which continuously computes noisy

¹see chapter 2.3.1 in Knudsen [10]

²collection of computer games such as pong and space invaders.

³computer program that plays the board game Go

⁴computer program that plays the board game Chess

estimates from the gradient of the expected reward from following a specific policy, to optimise the policy end-to-end. Furthermore, by being actor-critic, the policy function (actor) is independent of the value function (critic). The fact that these two are independent means that the policy function performs some action in the current state, while the value function computes a *temporal difference*⁵ (TD) error signal based on the current state and the reward from performing that particular action.

Being an off-policy algorithm means that the policy function does not make any assumption if whether or not the agent is following the actual policy. The result from this is that the policy function, which is used to generate behaviour, is unrelated to the policy that is evaluated and improved. The benefit of having this separation is that the policy might be deterministic, e.g. greedy⁶, while the policy function can continue to sample all possible actions in every state. Meaning that although the policy itself is deterministic, and thereby only choose the action with the highest expected reward, the policy function separately evaluates all possible actions in every state. The reason for doing this is to make sure that the agent does not converge towards a sub-optimal policy. The last characteristic is that the algorithms are model-free, which means that the environment is not modelled. A result of this is that the agent does not know anything about the underlying environmental dynamics, and the benefit of this is that the computational complexity of the problem is significantly reduced compared to model-based methods.

DDPG algorithms are closely connected to *Q-learning*, in the way that they use off-policy data and the Bellman equation to learn a Q-function, and then use the Q-function to learn the optimal policy [17]. The optimal policy is found by solving

$$a^*(s) = \underset{a}{\operatorname{arg\,max}} Q^*(s, a) \quad (3.1)$$

which is based on the fact that if the action-value function $Q^*(s, a)$ is known, the optimal action $a^*(s)$ can be determined in every state, and from this the optimal policy $\pi^*(s)$ can be found. Recall from Chapter 2 that Equation 3.1 is simply another way to define the policy gradient problem in Equation 2.7. DDPG algorithms are specially designed for environments with *continuous action spaces*, which is revealed through the computation of the *max* over all actions $\underset{a}{\operatorname{max}} Q^*(s, a)$. In finite action spaces, meaning that there exist a finite number of discrete actions, the max computation is no problem since the Q-values for each action can be computed separately and compared. However, continuous action spaces cannot exhaustively be

⁵an agent learning from an environment through episodes with no prior knowledge of the environment [28]

⁶the policy chooses the action with highest expected reward in each state

evaluated, and solving the optimization problem is highly non-trivial, which results in calculating $\max_a Q^*(s, a)$ being a very computational expensive routine.

Because of this, the function $Q^*(s, a)$ is assumed to be differentiable with respect to the action argument in continuous action spaces [17]. By assuming this an efficient, gradient-based, learning rule for the policy $\pi(s)$ can be created. Consequently, the expensive \max computation is approximated as $\max_a Q^*(s, a) \approx Q^*(s, \pi(s))$.

3.1.1 Q-Learning

As previously mentioned, DDPG algorithms are closely connected to Q-learning, in the way that they use the Bellman equation to learn a Q-function, which is then used to learn the optimal policy. In Equation 2.5 the Bellman equation for the action-value function $Q^*(s, a)$ was given by

$$\begin{aligned} q_\pi(s, a) &= E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \\ &= E_\pi[r(s, a) + \gamma \max_{a'} Q^*(s', a')] \end{aligned} \quad (3.2)$$

where $q_\pi = Q^*$. In order to understand how this Bellman equation contributes to learning an approximator to $Q^*(s, a)$ imagine that the approximator is a neural network $Q_\phi(s, a)$, with parameters ϕ and a set of transitions $\mathcal{D} = (s, a, r, s', d)$. Here, s and s' defines the current and previous state, a is the action taken, r is the reward received and d indicates if the state is *terminal* or not. The terminal state is defined as the state where a trajectory, or episode, is finished, e.g checkmate in a chess game. The variable d is either 1 or 0, which represents *true* or *false*, respectively. The goal is to satisfy the Bellman equation, and in order to estimate how close Q_ϕ is to achieve this a *mean-squared Bellman error* (MBSE) function is used, which is given in Equation 3.3.

$$L(\phi, \mathcal{D}) = E_{(s,a,r,s',d) \in \mathcal{D}} [(Q_\phi(s, a) - (r + \gamma(1 - d) \max_{a'} Q_\phi(s', a')))^2] \quad (3.3)$$

DDPG, and Q-learning algorithms in general, are based on minimising this MSBE loss function $L(\phi, \mathcal{D})$ in order to satisfy the Bellman equation. To accomplish this, DDPG algorithms utilises two main tricks, which are described in the following sections.

Replay Buffer

The first trick is the use of a *replay buffer*, which contains a set \mathcal{D} of previous experiences, and is a common implementation in standard deep neural network

training algorithms. In Equation 3.3, the replay buffer is represented by the term

$$Q_\phi(s, a) \tag{3.4}$$

The size of the replay buffer has a great influence on the quality of training, in the sense that it should be sufficiently large such that it contains enough diverse experience, but if its too large training can become immensely slow. To determine how to balance this trade-off between enough experience and training time, one should tune the hyper-parameters of the algorithm. Furthermore, recall that DDPG algorithms are off-policy, which is revealed through the replay buffer as well. The replay buffer contains information about old experiences, which possibly were experienced using *outdated* policies. Using experience from outdated policies is allowed because of the Bellman equation, which does not care about which transition tuples that are used, how the actions are selected, or what happens after a given transition, because the optimal Q-function should satisfy the Bellman equation for *all* possible transitions [17].

Target Network

The second trick involves a target network, represented in Equation 3.3 as the term

$$r + \gamma(1 - d)\max_a Q_\phi(s', a') \tag{3.5}$$

This term is called the target because minimising the MBSE loss function is effectively the same as trying to make the Q-function be more like this target. This target is problematic because it depends on the same set of parameters ϕ that the algorithm is trying to train, which results in the MSBE minimisation being unstable. Resolving this issue is done through the use of a copy network, which uses a set of parameters that comes *close* to ϕ , with a time delay. This network, ϕ_{targ} , is created by copying from the main network with some fixed step interval. In DDPG algorithms, the creation of the target network is done through *Polyak* averaging.

$$\phi_{targ} \leftarrow \rho\phi_{targ} + (1 - \rho)\phi \tag{3.6}$$

Where the Polyak hyper-parameter ρ has a value between 0 and 1. Recall that computing $\max_a Q^*(s, a)$ in continuous action spaces is a challenge, and using a *target policy network* to compute an action that approximately maximises $Q_{\phi_{targ}}$ is how DDPG algorithms deals with this challenge. This procedure is similar to the update function in Equation 3.6, mainly to use Polyak averaging on the policy parameters over the whole training period. The result from doing this is that the

Q-learning part of a DDPG algorithm becomes the procedure of minimising the following MSBE loss function through gradient descent

$$L(\phi, \mathcal{D}) = \underset{((s,a,r,s',d) \in \mathcal{D})}{E} [Q_\phi(s, a) - (r + \gamma(1 - d) Q_{\phi_{target}}(s', \pi_{\theta_{target}}(s')))]^2 \quad (3.7)$$

where $\pi_{\theta_{target}}$ is the target policy.

3.1.2 Policy Learning

As mentioned in the previous section, the Bellman equation is used to learn the Q-function, and the Q-function is then used to learn the optimal *policy*. The overall goal of policy learning in DDPGs is to determine a deterministic policy $\pi_\theta(s)$, which gives the action that maximises $Q_\phi(s, a)$ in every state. Since the action space is continuous, which resulted in the Q-function being assumed differentiable with respect to the action, gradient ascent (see Section 2.2.3) can be used to solve the maximisation problem in equation 3.8, resulting in the optimal policy.

$$\max_{\theta} E_{s \sim \mathcal{D}} [Q_\phi(s, \pi_\theta(s))] \quad (3.8)$$

3.1.3 Exploration vs. Exploitation

An important note when dealing with DDPG algorithms, and reinforcement learning in general, is the relationship between *exploration* and *exploitation* of actions. This is important because the policy in DDPG algorithms is deterministic, meaning that if the agent is exploring on-policy, in the beginning, it is not certain that it would experience enough variety of actions to find useful learning signals. This relationship specifically involves the trade-off between exploitation, meaning that the agent exploits the *learned* best action to take in a given state, and exploration, meaning that the agent *tries* a new action in a given state. This trade-off deals with the fact that although the agent has found what it believes to be the optimal action in a given state, there could always exist an unexplored action that is better. Due to this, the possibility for exploration is implemented, such that a sub-optimal policy is avoided.

The trade-off between exploration and exploitation is usually determined by a parameter ϵ , which is effectively the same as introducing noise into the system. If $\epsilon \rightarrow 0$ the agent only exploits previous learned optimal actions, and if $\epsilon \rightarrow 1$ the agent always explores new actions in a given state. When the agent is starting to explore the environment, it has no prior knowledge about it. Due to this, the agent should explore and exploit actions. However, as the agent discovers more about the

environment, meaning that it increases its *situational awareness*, the probability of the *learned* policy being optimal increases. This means that ϵ should decrease as the knowledge about the environment increases.

3.2 Challenges in Policy Gradient Methods

As described in the last section, DDPG algorithms are a type of policy gradient (PG) methods. Recall from Chapter 2 that the basic concept of PG methods is to use gradient ascent (or descent) to follow policies in the direction of the steepest increase in total reward. However, PG methods are a type of first-order optimisation, which results in difficulties in curved areas, and creates issues in convergence towards an optimal policy [6]. *Trust Region Policy Optimisation* (TRPO) and *Proximal Policy Optimisation* (PPO) methods tries to resolve these issues, but to understand how it is necessary to dig deeper into the challenges related to PG methods.

In PG methods, a policy θ is optimised to achieve the maximum expected discounted reward. This is given by

$$\max_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad (3.9)$$

In order to do this optimisation, PG methods computes the steepest ascent (or descent) direction of the rewards, denoted g in Equation 3.11, and then update the policy θ in this direction [6].

$$g = \nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^{\infty} \gamma^t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) A^{\pi_{\theta}}(s_t, a_t) \right] \quad (3.10)$$

$$\theta_{k+1} = \theta_k + \alpha g \quad (3.11)$$

Here, αg is a gradient step taken in the steepest direction. The gradient step in Equation 3.11 is a first-order derivative, which essentially means that the reward function is assumed to be flat. This assumption poses a problem if the reward function is steep or curved, which could result in terrible actions if the step is too large. Imagine that the reward function looks like a hill, and if the agent chooses to take a too large step at the top of the hill, it falls down the hill. When the agent resumes exploration, it does so from a worse state using a bad local policy, meaning that it probably needs a longer time to recover. At the same time, if the step is too small the agents learning rate decreases, resulting in learning becoming immensely slow.

Furthermore, the reward function also results in difficulties when it comes to defining a sufficient learning rate. Imagine that the hill is flat on the top, which means that the learning rate should be higher than average to accomplish a sufficient learning speed. However, if the step is too large, and it falls down the hill, this learning rate is suddenly too high, which results in an exploding policy update [6]. This problem is one of the main reasons why PG methods are suffering from convergence problems.

The third challenge related to PG methods is that the whole trajectory, or training episode, is sampled for only one policy update, and the policy cannot update at every time step.

$$g = \nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^{\text{inf}} \gamma^t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) A^{\pi_{\theta}}(s_t, a_t) \right] \quad (3.12)$$

one policy update per trajectory

The reason for this can be explained by imagining the policy model as a net, as in Figure 3.1. If the probability of $\pi(s)$ is increased at one point, the surrounding points will be pulled up. Similar response will be the case for the states within a trajectory, and if the policy was to be updated at every time step this will essentially result in pulling the net up multiple times at the same spots.



Figure 3.1: Policy Model Represented as a Net

The result from this is that the changes at each time-step affect and magnify each other, which results in the training process becoming unstable. Due to this, the policy cannot update at every time step, and since there likely are thousands of steps in one trajectory, this is not *sample efficient*, resulting in PG methods needing 10 million or more training steps to reach convergence [6]. The need of 10 million or more steps is very computationally expensive, especially when combined with the convergence problems discussed in the previous section as well.

3.3 Trust Region Policy Optimisation

Based on the challenges in policy gradient methods, it is clear that policy changes should be limited, and a goal would be that every policy change should guarantee an improvement in rewards. Because of this, it is necessary to develop a more robust optimisation method in order to produce better policies. This is where *Trust Region Policy Optimisation* (TRPO) enters. TRPO uses three different concepts to address the problems related to policy gradients; *Minorize-Maximization algorithm*, *Trust Region* and *Importance Sampling*.

3.3.1 Minorize-Maximization Algorithm

The first question to address is how to accomplish a guaranteed improvement in expected reward for every policy change. This is achieved through the *Minorize-Maximization* (MM) algorithm, which is shown in Figure 3.2.

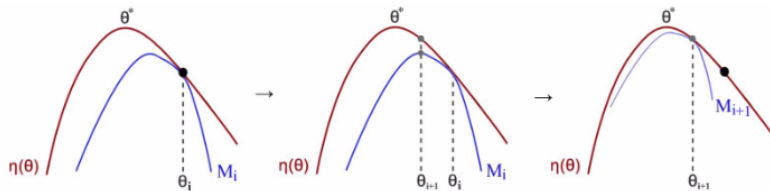


Figure 3.2: The Minorize-Maximization Algorithm [5]

The goal is to approximate the expected reward η (red line), by iteratively maximising a lower bound function M (blue line). This is done by starting with an initial arbitrary policy guess and find the lower bound M for that approximation [6]. The local optimal point for M , θ_i , is then used as the next guess. By iteratively doing this, as shown in Figure 3.2, the policy eventually converges to the optimal

policy. The result of this is that through the MM algorithm, the expected reward is guaranteed to improve for every policy change.

3.3.2 Trust Region

In TRPO methods, trust region optimisation is used instead of gradient descent (or ascent), which was the case in PG methods. Trust region optimisation works by first determining the maximum step size the agent should explore, and then locate the optimal point within the trust region. The objective of trust region optimisation is defined in equation 3.13.

$$\begin{aligned} \max_{s \in \mathbb{R}^n} m_k(s) \\ \text{s.t. } \|s\| \leq \delta \end{aligned} \quad (3.13)$$

Here, the goal is to find an optimal point m within the thrust region radius δ , and by iteratively repeating this process, eventually reaching the optimal policy. One of the problems with PG methods was that it is mainly an on-policy method, meaning that the agent stick with what could be a bad policy. Recall that in PG methods the agent could fall off the hill. On the other hand, in trust region methods, the region size δ can be readjusted in each iteration based on the curvature of the reward function. If the divergence between the new and current policy is getting too large, the trust region can be decreased and increased if it is getting too small. The benefit of doing this is that the agent has much better control of the learning procedure, as well as increased learning speed.

3.3.3 Importance Sampling

As previously discussed, PG methods have poor sample efficiency as well, which was due to the fact that the current policy is used to compute the policy gradient.

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}(\tau)} [\nabla_{\theta}(\tau) r(\tau)] \quad (3.14)$$

This is revealed in Equation 3.14, since the reward $r(\tau)$ is calculated from the trajectory using the current policy $\mathbb{E}_{\tau \sim \pi_{\theta}(\tau)}$. This means that when the policy changes, new samples are collected, and the old samples are not reusable, which results in poor sample efficiency. On the other hand, in TRPO methods *Importance Sampling* (IS) is used, which lets the agent choose to sample from the old policy instead. This is done by sampling data from an old policy q and calibrate the result through the probability ration between the new policy p and q . This is given by

$$\mathbb{E}_{x \sim q} \left[\frac{f(x)p(x)}{q(x)} \right] \quad (3.15)$$

where $f(x)$ is the function evaluated. From this, Equation 3.14 can be rewritten to use samples from an old policy instead. This is given by

$$\nabla_{\theta'} J(\theta') = \mathbb{E}_{\tau \sim \pi_{\theta}(\tau)} \left[\sum_{t=1}^T \nabla_{\theta'} \log \pi_{\theta'}(a_t | s_t) \left(\prod_{t'=1}^t \frac{\pi_{\theta'}(a_{t'} | s_{t'})}{\pi_{\theta}(a_{t'} | s_{t'})} \right) \left(\sum_{t'=t}^T r(s_{t'}, a_{t'}) \right) \right] \quad (3.16)$$

where θ and θ' is the old and new policy, respectively. One note here is that Equation 3.15 has the variance given in Equation 3.17 [21].

$$\frac{1}{N} (\mathbb{E}_{x \sim P} \left[\frac{P(x)}{Q(x)} f(x)^2 \right] - \mathbb{E}_{x \sim P} [f(x)]^2) \quad (3.17)$$

This means that if the ratio $P(x)/Q(x)$ is large, meaning that the two policies are far apart, the variance of the estimation gets very large. The result of this is that the agent cannot use old samples for too long, and the agent should frequently resample.

Using IS also means that the objective function can be changed. Recall that in policy gradient methods this was given by

$$g = \nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^{\infty} \gamma^t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) A^{\pi_{\theta}}(s_t, a_t) \right] \quad (3.18)$$

$$\theta_{k+1} = \theta_k + \alpha g \quad (3.19)$$

From the work done by Schulman et. al in [21] it is shown that this derivative can be reversed as the objective function given in Equation 3.20.

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t [\log \pi_{\theta}(a_t | s_t) \hat{A}_t] \quad (3.20)$$

where γ is set equal to 1 for simplicity, and by expressing this as importance sampling the result becomes

$$L_{\theta_{old}}^{IS}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)} \hat{A}_t \right] \quad (3.21)$$

By having two policies, π_{θ} and $\pi_{\theta_{old}}$, in the objective function the agent is now able to control *and* restrict policy change, which was one of the goals from the challenges in PG methods.

3.3.4 TRPO

By using the concepts above the objective function for TRPO methods is defined by Schulman et al. as

$$\begin{aligned}
 & \max_{\pi'} \mathcal{L}_{\pi}(\pi') - C \sqrt{\mathbb{E}_{s \sim d^{\pi'}} [D_{KL}(\pi' || \pi)[s]]} \\
 & \text{or} \\
 & \max_{\pi'} \mathcal{L}_{\pi}(\pi') \\
 & \text{s.t. } \mathbb{E}_{s \sim d^{\pi}} [D_{KL}(\pi' || \pi)[s]] \leq \delta
 \end{aligned} \tag{3.22}$$

where $\mathcal{L}_{\pi}(\pi')$ is defined as

$$\begin{aligned}
 \mathcal{L}_{\pi}(\pi') &= \frac{1}{1 - \gamma} \mathbb{E}_{s \sim d^{\pi}, a \sim \pi} \left[\frac{\pi'(a|s)}{\pi(a|s)} A^{\pi}(s, a) \right] \\
 &= \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t \frac{\pi'(a_t|s_t)}{\pi(a_t|s_t)} A^{\pi}(s_t, a_t) \right]
 \end{aligned} \tag{3.23}$$

and

$$d^{\pi}(s) = (1 - \gamma) \sum_{t=0}^{\infty} \gamma^t P(s_t = s | \pi) \tag{3.24}$$

Here, A is the *advantage function*, which says something about the expected improvement in rewards between the two policies π and π' , and d is the discounted future state distribution [21]. From this, \mathcal{L} can be interpreted as a way of using importance sampling to estimate the advantage function. Furthermore, D_{KL} is *Kullback-Leibler* (KL) divergence, which is a measure of how one data distribution p is different from another data distribution q , or in this case the difference in data distribution between two policies. This is given by

$$D_{KL}(P||Q) = \sum_{x=1}^N P(x) \log \frac{P(x)}{Q(x)} \tag{3.25}$$

The objective function in Equation 3.22 provides two possible objectives, either a KL penalised objective, or a KL constrained objective. Theoretically, these two are the same if the computational resources are unlimited. However, in practice, they are not. δ imposes a hard constraint in order to control bad policy changes and is in practice much easier to tune than the constant C . Because of this, TRPO implementations usually prefer using trust region constraint (KL constrained).

3.4 Proximal Policy Optimisation

Proximal Policy Optimisation (PPO) algorithms are a *newer* family of algorithms in the deep reinforcement learning environment, originally proposed by Schulman et al. from OpenAI Gym in 2017 [22]. Compared to standard policy gradient methods, such as DPPGs, which does one gradient update each data sample, the PPOs enables multiple epochs⁷ of minibatch⁸ updates. Meaning that PPOs utilises multiple updates each iteration, by passing over a full subset of the training data each update. Furthermore, OpenAI gym argues that PPOs are easier to implement, more general and have better sample complexity than TPROs, although it prohibits some of the same benefits.

As stated throughout, the rapid development in AI has led to several proposed methods for reinforcement learning with neural network function approximators. However, there is still room for improvement, especially when it comes to developing methods that are both *scalable*, *data efficient* and *robust*. Meaning that the methods are applicable to large state and action spaces, not computational *heavy*, and successful on a variety of problems. DDPG algorithms and deep Q-learning methods, in general, have problems related to both robustness and data efficiency. Often leading to problems in convergence towards the optimal solution, while TRPO algorithms consist of complicated algorithms and many architectural flaws related to noise and parameter sharing [22].

In PPO algorithms these problems are resolved by only using first-order optimisation, but at the same time retain the data efficiency and performance of TPROs. To do this, Schulman et al. proposes a novel objective function⁹ with *clipped* probability ratios, and policies that are optimised by alternating between sampling policy data and performing epochs of optimisation on the sampled data.

3.4.1 Optimisation

To optimise the policy, PPO algorithms also utilise the *Minorize-Maximization* (MM) algorithm, which was defined in Figure 3.2. Furthermore, PPO algorithms use the KL-divergence to limit policy change. The KL-divergence was defined for

⁷one pass over the full training set

⁸a subset of all the training data

⁹a function that is desired to maximise or minimise

two data distributions p and q as

$$D_{KL}(P||Q) = \mathbb{E}_x \log \frac{P(x)}{Q(x)} \quad (3.26)$$

In the same way as TRPO methods, the PPO uses the KL-divergence to evaluate the difference between two policies, and make sure that the new policy does not differ too much from the old one. By doing this, it turns out that to limit policy change, such that the agent does not make bad decisions, the lower bound of M in the MM algorithm can found as

$$M = \mathcal{L}(\theta) - C * \bar{KL}, \text{ where} \quad (3.27)$$

$$\mathcal{L}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \quad (3.28)$$

where π_θ and $\pi_{\theta_{old}}$ are the new and current policy, respectively. \bar{KL} is KL-divergence and C is a constant. $\mathcal{L}(\theta)$ is the expected advantage function, which was defined in the previous section. The lower bound of M can be further defined as follows [5].

$$\eta(\tilde{\pi}) \geq L_\pi(\tilde{\pi}) - CD_{KL}^{max}(\pi, \tilde{\pi}), \text{ where} \quad (3.29)$$

$$C = \frac{4\epsilon\gamma}{(1-\gamma)^2} \quad (3.30)$$

$$D_{KL}^{max}(\pi, \tilde{\pi}) = \max_s D_{KL}(\pi(-|s)||\tilde{\pi}(-|s)) \quad (3.31)$$

$$\epsilon = \max_{s,a} [A_\pi(s, a)] \quad (3.32)$$

The maximum KL-divergence $D_{KL}^{max}(\pi, \tilde{\pi})$ is computational heavy to calculate, and because of this its requirements are *relaxed* a bit by using mean KL-divergence instead. This is allowed because of the advantage function \mathcal{L} . The advantage function will get less accurate as the error between the new and current policy increases, where the distance has an upper bound defined as the second term in M . Since the upper bound of the error is *within* the trust region, the new policy is guaranteed to be better than the old policy. However, if the optimal policy *exceeds* the trust region, the accuracy might be far off and the policy cannot be trusted. Thus, a relaxed requirement by using the mean KL-divergence is sufficient, and the whole objective function can be summarised as

$$\underset{\theta}{\text{maximise}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \quad (3.33)$$

$$\text{subject to} \quad \hat{\mathbb{E}}_t [KL[\pi_{\theta_{old}}(-|s_t), \pi_\theta(-|s_t)]] \leq \delta$$

However, δ is here very small and considered too conservative. By inserting a tuneable hyperparameter β instead, the conditions are relaxed, and the objective function

can be re-written as

$$\underset{\theta}{\text{maximise}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] - \beta \hat{\mathbb{E}}_t [KL[\pi_{\theta_{old}}(-|s_t), \pi_{\theta}(-|s_t)]] \quad (3.34)$$

In order to make M easy to optimise Taylor expansion is used on the results in Equation 3.33 and 3.34, which results in the following solution for the policy update.

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T F^{-1} g}} F^{-1} g \quad (3.35)$$

The second term in this solution is a natural policy gradient, which involves both a second-order derivative F and its inverse F^{-1} . However, second-order derivatives are computational expensive operations, and PPO algorithms aim not to do these operations. This is solved by pushing the first-order derivative solution as close as possible to the second-order derivative solution through the use of *soft constraints*. The idea for doing this is that it is better to have a bad policy decision once in a while compared to calculating a second-order solution, but by adding a soft constraint as well we make sure that the policy stays within the trust region, meaning that the probability of bad decisions are reduced. PPO utilises this through two different methods, *adaptive KL penalty* and *clipped surrogate objective*, which are explained in the next sections.

3.4.2 Adaptive Kullback-Leibler Penalty

In PPO with adaptive KL penalty the objective is formulated such that the second term in Equation 3.34 serves as a *penalty*, where β controls the weight of the penalty.

$$\underset{\theta}{\text{maximise}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] - \beta \hat{\mathbb{E}}_t [KL[\pi_{\theta_{old}}(-|s_t), \pi_{\theta}(-|s_t)]] \quad (3.36)$$

The weight β will penalise the objective if the new policy differs from the current policy. The KL-divergence between the new and current policy is given by the second term, $\hat{\mathbb{E}}_t [KL[\pi_{\theta_{old}}(-|s_t), \pi_{\theta}(-|s_t)]]$. If this term is larger than the target value, β is reduced, and if it is smaller, β is increased to expand the trust region.

3.4.3 Clipped Surrogate Objective

From Schulman et al. it is proved that PPO with adaptive KL penalty achieves similar performance as of TRPO, and with speed close to gradient descent methods.

However, PPO with clipped surrogate objective has been shown to do even better [22]. In PPOs, two policy networks are used, where $\pi_\theta(a_t|s_t)$ is the current policy that is optimised and $\pi_{\theta_k}(a_t|s_t)$ is the previous policy used to collect samples. The current policy is again optimised through

$$\underset{\theta}{\text{maximise}} \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \quad (3.37)$$

Doing this means that the current policy can be improved and optimised by using samples collected from an old policy, which increases sample efficiency. However, as the current policy is optimised and improved it will get further away from the old policy. As the difference increases the variance in the estimation will increase, and the probability of bad decisions increases as well. Due to this, the two policy networks $\pi_\theta(a_t|s_t)$ and $\pi_{\theta_k}(a_t|s_t)$ needs to be regularly synchronised. This is done by letting

$$\pi_{\theta_{k+1}}(a_t|s_t) \leftarrow \pi_\theta(a_t|s_t) \quad (3.38)$$

By computing the ratio between the two terms in Equation 3.38, the difference between the two policy networks can be measured as

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \quad (3.39)$$

This ratio is then used to clip the the estimated advantage function if the difference between the two policies are too large. This results in the clipped surrogate objective function, displayed in Equation 3.40.

$$L_{\theta_k}^{CLIP}(\theta) = \hat{\mathbb{E}}_t [\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t\theta, 1 - \epsilon, 1 + \epsilon)\hat{A}_t)] \quad (3.40)$$

Here ϵ is a tuneable hyperparameter, set has $\epsilon = 0.2$ in [22]. This makes sure that if the probability ratio between the two policies exceeds the range $(1 - \epsilon, 1 + \epsilon)$, the advantage function is clipped, meaning that the probability ratio is ignored when the objective is improved and included if it makes the objective worse [22]. This can be shown by plotting a single step t of the clipped surrogate objective, illustrated in Figure 3.3.

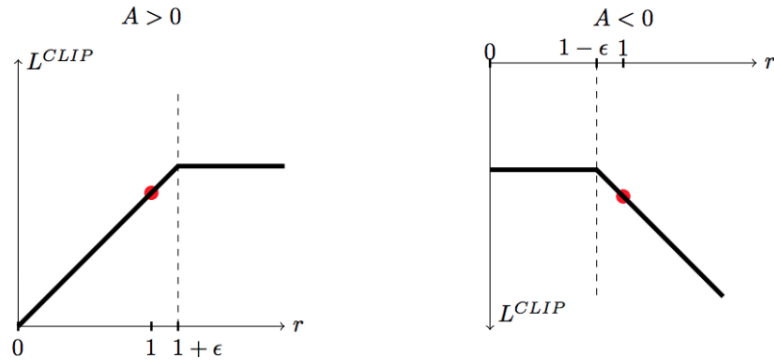


Figure 3.3: Performance of Clipped Surrogate Objective [22]

Observe in Figure 3.3 that the clipping is based on whether the advantage function is greater or less than zero. If minibatch gradient ascent is used the loss function L^{CLIP} can be maximised, meaning that the problem is indeed a first-order problem.

This chapter has introduced three state-of-the-art algorithms within the deep reinforcement learning family, and showcased their benefits and challenges. The preliminary studies on the subject evaluated the performance of a DDPG based controller architecture [10], and as stated in this chapter the DDPG algorithm has problems related to convergence. However, the preliminary studies showed that the algorithm was able to learn how to reach a desired pose despite of these difficulties. Due to this, the controller implementation in the next chapters will surround a DDPG based controller architecture.

Chapter 4

Modeling

This thesis builds on the preliminary work done by Knudsen on station keeping of AUVs [10], and the following chapter on the model dynamics is based on this work.

In order to evaluate the possibilities of applying DRL techniques in AUV control, simulations will be conducted using the BlueROV2 by *BlueRobotics* [1]. The BlueROV2 has a 6-thruster vectored configuration, open-source electronics and software, and works well for inspections, research, and adventuring. The simulation model, provided by Nielsen et al. in [16], does not include the BlueROV2's tether, which means that the vehicle is *assumed to perform as an AUV* in simulation.



Figure 4.1: BlueROV2 by BlueRobotics [1]

4.1 Reference Frames

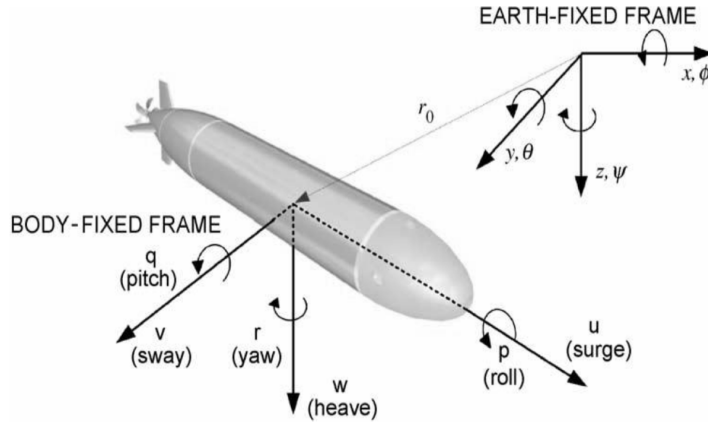


Figure 4.2: Body-fixed and Earth-fixed Reference Frames

In Figure 4.2, the BlueROV2's 6 degrees of freedom (DOF) in the *Body-fixed* and *Earth-fixed* reference frames are illustrated. Here, the *North-East-Down* (NED) reference frame is used as the earth-fixed reference frame. When analysing motion in 6-DOF it is convenient to define motion in different reference, or coordinate, frames. The reference frames are given by

- **NED** is defined as a coordinate system $\{n\} = (x_n, y_n, z_n)$ with origin o_n relative to the Earth's reference ellipsoid, commonly defined as the tangent plane to the surface of the Earth moving with the vessel [4]. Here, the x_n axis points towards true North, the y_n axis points towards true East and the z_n axis points downwards normal to the surface.
- **BODY** is defined as a coordinate system $\{b\} = (x_b, y_b, z_b)$ with origin o_b centred at the vessels centre of mass, and moving with the body. Here, the x_b axis is directed from aft to fore, the y_b transverse axis is directed towards starboard and the z_b normal axis directed from top to bottom.

The notations in Figure 4.2 are defined according to SNAME (1950) [4], which is given in Table 4.1. The SNAME notation is used throughout this thesis.

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, heel)	K	p	ϕ
5	rotation about the y axis (pitch, trim)	M	q	θ
6	rotation about the z axis (yaw)	N	r	ψ

Table 4.1: The Notation of SNAME (1950) for Marine Vessels

When dealing with multiple frames, it is necessary to *rotate* between them, such that any parameter can be expressed in all frames. This is done through a *rotation matrix*. For the BlueROV2, the linear and angular velocities, given in Table 4.1, are defined in the (Body) frame. From Fossen [4] the *Euler angles*; roll (ϕ), pitch (θ) and yaw (ψ), can be used to decompose the velocity vector $\mathbf{v}_{b/n}^b$ into the (NED) frame. This is given by

$$\mathbf{v}_{b/n}^n = \mathbf{R}_b^n(\Theta_{nb})\mathbf{v}_{b/n}^b \quad (4.1)$$

where the rotation matrix from (Body) to (NED) is given by

$$\mathbf{R}_b^n(\Theta_{nb}) = \begin{bmatrix} c\psi c\theta & -s\psi c\theta + c\phi s\theta s\phi & s\psi s\theta + c\psi c\phi s\theta \\ s\psi c\theta & c\phi c\theta + s\phi s\theta s\psi & -c\psi s\theta + s\theta s\psi c\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix} \quad (4.2)$$

where $c = \cos$ and $s = \sin$.

4.2 Dynamic Model of Underwater Vehicles

From Fossen [4], the underwater dynamic equation of motions is defined as

$$M\dot{v} + C(v)v + D(v)v + g(\mu) + \delta = \tau \quad (4.3)$$

$$\longrightarrow (M_{RB} + M_A)\dot{v} + (C_{RB}(v) + C_A(v))v + (D_L + D_Q)v + g(\mu) + \delta = \tau \quad (4.4)$$

For simplicity, the external forces, such as environmental forces τ_{env} , are here neglected. Consequently, τ is the force and torque applied by the controller, while δ is a model uncertainty vector. The terms in Equation 4.4 are defined in the following sections, and their values are given in Appendix A.

4.2.1 Mass Matrix

The Mass matrix consist of *rigid-body* terms and *hydrodynamic* terms. The hydrodynamic terms are due to *added mass*, which is mass created by the acceleration of the surrounding water when a vehicle is moving through a stationary fluid [4].

$$\mathbf{M}_{RB} = \begin{bmatrix} m * I_{3x3} & 0_{3x3} \\ 0_{3x3} & I_g \end{bmatrix} \quad (4.5)$$

$$\mathbf{I}_g = \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ I_{yx} & I_y & -I_{yz} \\ -I_{zx} & -I_{zy} & I_z \end{bmatrix} \quad (4.6)$$

$$\mathbf{M}_A = \begin{bmatrix} X_{\dot{u}} & X_{\dot{v}} & X_{\dot{\omega}} & X_{\dot{p}} & X_{\dot{q}} & X_{\dot{r}} \\ Y_{\dot{u}} & Y_{\dot{v}} & Y_{\dot{\omega}} & Y_{\dot{p}} & Y_{\dot{q}} & Y_{\dot{r}} \\ Z_{\dot{u}} & Z_{\dot{v}} & Z_{\dot{\omega}} & Z_{\dot{p}} & Z_{\dot{q}} & Z_{\dot{r}} \\ K_{\dot{u}} & K_{\dot{v}} & K_{\dot{\omega}} & K_{\dot{p}} & K_{\dot{q}} & K_{\dot{r}} \\ M_{\dot{u}} & M_{\dot{v}} & M_{\dot{\omega}} & M_{\dot{p}} & M_{\dot{q}} & M_{\dot{r}} \\ N_{\dot{u}} & N_{\dot{v}} & N_{\dot{\omega}} & N_{\dot{p}} & N_{\dot{q}} & N_{\dot{r}} \end{bmatrix} \quad (4.7)$$

$$(4.8)$$

Here, \mathbf{m} denotes the total mass of the vehicle, and $\mathbf{I}_x, \mathbf{I}_y, \mathbf{I}_z$ represents the moment of inertia about the body axis. Because of symmetry, the products of inertia, here represented on the off-diagonal, will have the form $\mathbf{I}_{xy} = \mathbf{I}_{yx}, \mathbf{I}_{yz} = \mathbf{I}_{zy}$ and $\mathbf{I}_{zx} = \mathbf{I}_{xz}$. The added mass matrix consist of hydrodynamic derivatives, which is given by

$$N_k = \frac{\partial N}{\partial \dot{k}} \quad (4.9)$$

where $N = (X, Y, Z, K, M, N)$ and $k = (u, v, \omega, p, q, r)$. For example, $X_{\dot{u}}$ represents the added mass force coefficient in surge due to an acceleration in surge.

4.2.2 Coriolis and Centripetal Force Matrices

The Coriolis-Centripetal matrix also consists of rigid-body and hydrodynamic terms. From Fossen [4], the rigid-body Coriolis-Centripetal matrix can be found by defining \mathbf{M}_{RB} as

$$\mathbf{M}_{RB} = \mathbf{M}_{RB}^T = \begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{bmatrix} > 0 \quad (4.10)$$

where $\mathbf{M}_{21} = \mathbf{M}_{12}^T$. \mathbf{C}_{RB} is then found by

$$\mathbf{C}_{RB}(\nu) = \begin{bmatrix} 0_{3x3} & -S(M_{11}\nu_1 + M_{12}\nu_2) \\ -S(M_{11}\nu_1 + M_{12}\nu_2) & -S(M_{21}\nu_1 + M_{22}\nu_2) \end{bmatrix} \quad (4.11)$$

where $\nu_1 := v_{b/n}^b = [u, v, \omega]^T$, $\nu_2 := \omega_{b/n}^b = [p, q, r]^T$ and S is the cross-product operator. The hydrodynamic Coriolis-Centripetal matrix can be found in the same way, by using \mathbf{M}_A . This yields

$$\mathbf{M}_A = \mathbf{M}_A^T = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} > 0 \quad (4.12)$$

$$\mathbf{C}_A(\nu) = \begin{bmatrix} 0_{3 \times 3} & -S(A_{11}\nu_1 + A_{12}\nu_2) \\ -S(A_{11}\nu_1 + A_{12}\nu_2) & -S(A_{21}\nu_1 + A_{22}\nu_2) \end{bmatrix} \quad (4.13)$$

4.2.3 Damping Matrices

As stated previously, the environmental forces are neglected in Equation 4.4, which means that skin friction and vortex shedding will dominate the damping term. This leads to the simplified hydrodynamic damping terms in Equation 4.14 and 4.15.

$$\mathbf{D}_L = - \begin{bmatrix} X_u & 0 & 0 & 0 & 0 & 0 \\ 0 & Y_v & 0 & 0 & 0 & 0 \\ 0 & 0 & Z_\omega & 0 & 0 & 0 \\ 0 & 0 & 0 & K_p & 0 & 0 \\ 0 & 0 & 0 & 0 & M_q & 0 \\ 0 & 0 & 0 & 0 & 0 & N_r \end{bmatrix} \quad (4.14)$$

$$\mathbf{D}_Q = - \begin{bmatrix} X_{u|u}|u| & 0 & 0 & 0 & 0 & 0 \\ 0 & Y_{v|v}|v| & 0 & 0 & 0 & 0 \\ 0 & 0 & Z_{\omega|\omega}|\omega| & 0 & 0 & 0 \\ 0 & 0 & 0 & K_{p|p}|p| & 0 & 0 \\ 0 & 0 & 0 & 0 & M_{q|q}|q| & 0 \\ 0 & 0 & 0 & 0 & 0 & N_{r|r}|r| \end{bmatrix} |\mathbf{v}| \quad (4.15)$$

4.2.4 Hydrostatic Terms

The hydrostatic matrix, expressed as $g(\mu)$ in Equation 4.4, accounts for the *restoring forces*, which is defined as the relationship between the weight W and the buoyancy B . By assuming that the vehicle is *neutrally buoyant* it will satisfy $W = B$, and by further assuming that the *centre of gravity* (COG) and the *centre of buoyancy* (COB) are located vertically on the z axis, the hydrostatic matrix can be defined as

$$g(\mu) = [0, 0, 0, \bar{B}G_z W \cos(\phi) \sin(\phi), \bar{B}G_z W \sin(\theta), 0]^T \quad (4.16)$$

where $\bar{B}G_z = z_g - z_b$, with z_g being the z component in COG, and z_b being the z component in COB, and $\mu = [N, E, D, \phi, \theta, \psi]$ defined in the (NED) and (Body) reference frames.

Chapter 5

Method

This chapter describes the procedure of developing a sufficient DRL-based station keeping controller for the BlueROV2. The controller builds on the design proposed by Knudsen in the preliminary studies [10] and consists of a dual controller which encompasses a DDPG algorithm in conjunction with a PD controller.

5.1 Controller Implementation for the BlueROV2

The BlueROV2 has 13 states, which are defined hereafter as

- 3 *position* states defined in the (NED) frame as $[x, y, z]$
- 4 *orientation* states defined in the (NED) frame as the quaternions $[\epsilon_1, \epsilon_2, \epsilon_3, \eta]$
- 3 *linear velocity* states defined in the (Body) frame as $[u, v, w]$
- 3 *angular velocity* states defined in the (Body) frame as $[p, q, r]$

The orientation states are defined as *quaternions* such that the issue related to singularity in pitch $\phi = \pm 90^\circ$ when using *Euler angles* is resolved. However, in order for the controller to use these states they need to be transformed into Euler angles. The singularity will not be an issue when doing this, since the transformation does not affect the model measurements. The rotation matrix from (Body) to (NED) for the quaternions is defined by Fossen [4] as

$$\mathbf{R}_b^n(\mathbf{q}) = \begin{bmatrix} 1 - 2(\epsilon_2^2 + \epsilon_3^2) & 2(\epsilon_1\epsilon_2 - \epsilon_3\eta) & 2(\epsilon_1\epsilon_3 + \epsilon_2\eta) \\ 2(\epsilon_1\epsilon_2 + \epsilon_3\eta) & 1 - 2(\epsilon_1^2 + \epsilon_3^2) & 2(\epsilon_2\epsilon_3 - \epsilon_1\eta) \\ 2(\epsilon_1\epsilon_3 - \epsilon_2\eta) & 2(\epsilon_2\epsilon_3 + \epsilon_1\eta) & 1 - 2(\epsilon_1^2 + \epsilon_2^2) \end{bmatrix} \quad (5.1)$$

Furthermore, the relationship between the quaternions and Euler angles is defined as

$$\mathbf{R}_b^n(\Theta_{nb}) := \mathbf{R}_b^n(\mathbf{q}) \quad (5.2)$$

which is equal to

$$\begin{bmatrix} c\psi c\theta & -s\psi c\theta + c\phi s\theta s\phi & s\psi s\theta + c\psi c\phi s\theta \\ s\psi c\theta & c\phi c\theta + s\phi s\theta s\psi & -c\psi s\theta + s\theta s\psi c\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \quad (5.3)$$

where $c = \cos$ and $s = \sin$. This results in the following Euler angles, expressed by the quaternions, which is used in the controller.

$$\phi = \text{atan2}(R_{32}, R_{33}) = \text{atan2}(2(\epsilon_2\epsilon_3 + \epsilon_1\eta) = 1 - 2(\epsilon_1^2 + \epsilon_2^2)) \quad (5.4)$$

$$\theta = -\sin^{-1}(R_{31}) = -\sin^{-1}(2(\epsilon_1\epsilon_3 - \epsilon_2\eta)) \quad (5.5)$$

$$\psi = \text{atan2}(R_{21}, R_{11}) = \text{atan2}(2(\epsilon_1\epsilon_2 + \epsilon_3\eta) = 1 - 2(\epsilon_2^2 + \epsilon_3^2)) \quad (5.6)$$

ϕ and θ defines the pitch and roll angle, respectively. In order to simplify the implementation of a controller based on reinforcement learning it is convenient to not include these two states in the algorithm. The reason for this is that the overall goal is to accomplish station keeping, and these two states are not as important as the rotation about the z-axis, ψ . To further simplify the problem, the position in z is also neglected, since constant depth is assumed.

Although these three states; ϕ , θ and z , are not accounted for by the reinforcement learning algorithm, meaning that the algorithm do not produce thrust input for these direction/orientations, their states will affect the behaviour of the vehicle. In order to resolve this a classical control algorithm is first implemented, such that these three states are stable on their own. Then, the reinforcement learning algorithm is implemented for controlling the remaining states.

5.1.1 Implementation of the PD Controller

As mentioned in Chapter 1, the PID controller is a common controller architecture in underwater designs, but in most cases, only using PD control is sufficient. In order to accomplish station keeping of the AUV, the PD controller should adjust thrust in the z direction as well as the rotation in pitch ϕ and roll θ . The PD controller is

defined as follows.

$$\tau = K_p e(t) + K_d \frac{d(t)}{dt} \quad (5.7)$$

$$\rightarrow \begin{bmatrix} \tau_z(t) \\ \tau_\phi(t) \\ \tau_\theta(t) \end{bmatrix} = K_p \begin{bmatrix} e(t)_z \\ e(t)_\phi \\ e(t)_\theta \end{bmatrix} + K_d \begin{bmatrix} \omega(t) \\ p(t) \\ q(t) \end{bmatrix} \quad (5.8)$$

5.1.2 Implementation of the DDPG Algorithm

As mentioned in Chapter 3, the DDPG is a policy-gradient algorithm, which uses a stochastic behaviour policy for good exploration, but estimates a deterministic target policy, which is easy to learn. To implement this, two *neural networks* are used, a *control* neural network (actor) and an *evaluation* neural network (critic), based on the experiments done by Runsheng et al. [29].

In order to define the neural networks, the equation of motions for underwater vehicles, defined in Equation 4.4, is used. By assuming the pitch rotation ϕ and the roll rotation θ *small*, the rotation matrix from (Body) to (NED) can be expressed as

$$\mathfrak{R}(\psi) = \begin{pmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (5.9)$$

Furthermore, by assuming the mass matrix, $M = M_{RB} + M_A$, *not singular*, Equation 4.4 can be modified as

$$\dot{v} = M^{-1}(\tau - D(v)v - g(\eta) - C(v)v - \delta) \quad (5.10)$$

$$\mathfrak{R}(\psi)v = \dot{\eta} \quad (5.11)$$

Applying the first-order Taylor expansion to this results in

$$v(t+1) = M^{-1}(\tau + G(t)) \quad (5.12)$$

$$\eta(t+1) = \mathfrak{R}(\psi(t))v(t) \quad (5.13)$$

$$G(t) = (-D(v(t))v - g(\eta(t)) - C(v(t))v(t) - \omega) \quad (5.14)$$

where ω is noise from the environment, and t is a certain moment of the system. From this, the controller τ can be defined as a function of the position and velocity of the previous moment. This is given by

$$\tau(t) = \mu(v(t), \eta(t)) \quad (5.15)$$

which can be simplified as

$$\tau(t) = \mu(s_t) \quad (5.16)$$

$$s_t = [v(t), \eta(t)]^T \quad (5.17)$$

The controller τ applies a thrust matrix input to the system in the given state, which means that it can be defined as an action (a_t) that the vehicle executes in the given state (s_t). Since the goal is to accomplish station keeping of the AUV at a desired state s_d , this means that the actor function is designed to minimise the state error through minimising the reward function, given by

$$R(s_t, a_t) = \int_t^\infty \gamma^{-(k-t)} r(s_t, a_t) dk \quad (5.18)$$

where $\gamma \in (0, 1)$ is the discount factor, which is used to reduce the influence from possible future states, defined in Chapter 2. This means that an optimisation problem can be defined as

$$\underset{s_t, a_t}{\operatorname{argmin}} \{R(s_t, a_t)\} \quad (5.19)$$

$$\text{s.t. } a_{min} \leq a_t \leq a_{max} \quad (5.20)$$

$$s_{min} \leq s_t \leq s_{max} \quad (5.21)$$

In Equation 5.21, a_{min} and a_{max} are equal to the minimum and maximum thrust input from the controller, respectively. The BlueROV2 has a maximum velocity of 2 [m/s] [1], but in simulation this is set to half of this, resulting in $a_{min} = -1[N]$ and $a_{max} = +1[N]$.

Critic Function

To solve the **argmin** optimisation problem, the critic function is defined as

$$Q(s_t, a_t) = R(s_t, a_t) = \int_t^\infty \gamma^{-(k-t)} r(s_t, a_t) dk \quad (5.22)$$

$$\text{Discretize} \longrightarrow R(s_t, a_t) = \sum_{i=t}^\infty \gamma^{(i-t)} r(s_t, a_t) \quad (5.23)$$

From the **Bellman Equation**, defined in Chapter 2, it is given that

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}) \quad (5.24)$$

This gives the **optimal** critic function as

$$\hat{Q}(s_t, a_t) = \underset{s_t, a_t}{\operatorname{argmin}}(Q(s_t, a_t)) \quad (5.25)$$

By replacing the critic function with a *neural* network this results in

$$Q(s_t, a_t) \longrightarrow Q(s_t, a_t|\omega) \quad (5.26)$$

To evaluate the policy it is first necessary to find the optimal critic function in Equation 5.25. This is done by defining a Loss function

$$Loss = \frac{1}{2}(y_t - Q(s_t, a_t|\omega))^2 \quad (5.27)$$

$$y_t = r(s_t, a_t) + \gamma Q(s_t, a_t|\omega) \quad (5.28)$$

By then using the policy gradient algorithm from Silver et al. [25] with a sampled batch from (s_t, a_t) , the average Loss function, and its gradient, are given as

$$Loss = \frac{1}{N} \sum_{i=1}^N (y_i - Q(s_i, a_i|\omega))^2 \quad (5.29)$$

$$\nabla_{\omega} Loss = -\frac{2}{N} \sum_{i=1}^N (y_i - Q(s_i, a_i|\omega)) \frac{\partial Q(s_i, a_i|\omega)}{\partial \omega} \quad (5.30)$$

The weight, ω , is updated by

$$\omega_{t+1} = \omega_t + \alpha \nabla_{\omega} Loss \quad (5.31)$$

where α is the learning rate.

Actor Function

The critic function is now used to update the actor function. This is done by replacing $\mu(s_t)$ with a neural network, $\mu(s_t|\theta)$. By substituting $a_t = \mu(s_t|\theta)$ into $Q(s_t, a_t|\omega)$ the results becomes

$$J = Q(s_t, \mu(s_t|\theta)|\omega) \quad (5.32)$$

The differentiation of J is then given by

$$\nabla_{\theta} J = \frac{\partial Q(s_t, \mu(s_t|\theta)|\omega)}{\partial a_t} \frac{\partial \mu(s_t|\theta)}{\partial \theta} \quad (5.33)$$

In order to update the network, **Adam** [9] is used, which is a stochastic optimization method used to update the network weights iteratively. This is given by

$$m_{t+1} = \wp \times m_t + (1 - \wp) \nabla_{\theta} \mu \quad (5.34)$$

$$\mathfrak{S}_{t+1} = \beta \times \mathfrak{S}_t + (1 - \beta) \nabla_{\theta} \mu \quad (5.35)$$

$$\hat{m}_t = \frac{m_t}{1 - \wp^t} \quad (5.36)$$

$$\hat{\mathfrak{S}}_t = \frac{\mathfrak{S}_t}{1 - \beta^t} \quad (5.37)$$

$$\theta_{t+1} = \theta_t - \eta \frac{1}{\sqrt{\hat{\mathfrak{S}}_t}} \hat{m}_t \quad (5.38)$$

Here, \wp and β are the Adam learning rates.

5.1.3 Controller Architecture

The proposed solution is to split the fully actuated BlueROV2, such that the PD controller controls the states z, ϕ and θ , in conjunction with a DDPG algorithm that controls the states x, y and ψ . This results in the *dual* controller architecture visualised in Figure 5.1.

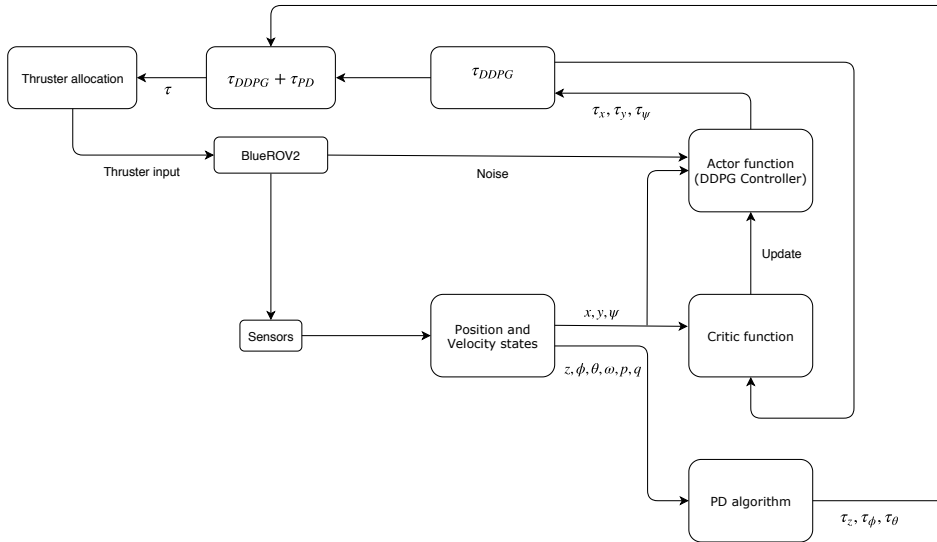


Figure 5.1: Controller Architecture for the BlueROV2

From the BlueROV2's sensors the controller receives the pose (position and orientation) and velocities in the given state. The PD controller receives the pose states $[z, \phi, \theta]$ and the velocity states $[\omega, p, q]$, to calculate the needed thrust input in $[z, \phi, \theta]$. At the same time, the DDPG algorithm will receive the pose states $[x, y, \psi]$, to calculate the needed thrust input in these states. This results in the thrust input matrix in the given state, s_t , being equal to

$$\tau = \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} \quad (5.39)$$

The thrust input matrix τ is transmitted through the thruster allocation matrix, which determines how much force and torque that needs to be generated in each thruster such that τ is satisfied.

The DDPG algorithm is displayed in Algorithm 1. As shown in the algorithm, it clips the action a_t based on the constraint $a_t \in [-1, 1]$ and a factor ϵ . ϵ is here the exploitation versus exploration factor, discussed in Chapter 3. Both networks also need to be discounted by a *learning rate* α , which is a hyperparameter controlling how much the weights in the networks are adjusted with respect to the loss gradient [30]. A small learning rate makes sure that no local minimum in the loss function is missed, which is preferable. However, a small learning rate could also result in a longer time needed to reach convergence. Due to this, the learning rate should be decreased over the number of episodes, since the agent obtains larger and larger knowledge about the environment.

Algorithm 1: DDPG Algorithm

Initialise the network classes $Q(s_t, a_t|\omega)$ and $\mu(s_t|\theta)$ with weights ω and θ .
 Initialise the replay buffer R as a memory class, M, to hold s_t, a_t, r_t and s_{t+1} .

for ep **in** $MAX_EPISODES$ **do**

 Initialise episode reward to zero

 Initialise episode step, t , to zero

while $t < MAX_EP_STEPS$ **do**

 Initialise desired state, s_d

 Initialise desired state PID, $s_{d,PID}$

 get position states, x, y, z , and orientation states, $\epsilon_1, \epsilon_2, \epsilon_3, \eta$

 get the Euler angles ϕ, θ, ψ from the orientation states

 compute $s_t = (x, y, \psi)$

 Choose action $a_t = \mu(s_t|\theta)$

 clip action based on a_{max}, a_{min} and ϵ

 compute the error state, s_e

 compute the error state PID, $s_{e,PID}$

 compute $r(s_t, a_t)$

 episode reward $+= r(s_t, a_t)$

 store in transition $R(s_t, a_t, r_t, s_{t+1})$

 Randomly select N arrays from R

 compute $y_i = r_i + \gamma Q(s_i, a_i|\omega)$

 compute $Loss = \frac{1}{N} \sum_{i=1}^N (y_i - Q(s_i, a_i|\omega))^2$

 compute $\nabla_{\omega} Loss = \frac{1}{N} \sum_{i=1}^N (y_i - Q(s_i, a_i|\omega)) \frac{\partial Q(s_i, a_i|\omega)}{\partial \omega}$

 update weight $\omega_{t+1} = \omega_t + \alpha \nabla_{\omega} Loss$

 compute $\nabla_{\theta} J = \frac{1}{N} \sum_{i=1}^N \frac{\partial Q(s_i, a_i|\mu(s_i|\theta))}{\partial a_i} \frac{\partial \mu(s_i|\theta)}{\partial \theta}$

 compute $m_t = \wp \times m_{t-1} + (1 - \wp) \nabla_{\theta} \mu$

 compute $\mathfrak{S}_t = \beta \times \mathfrak{S}_{t-1} + (1 - \beta) \nabla_{\theta} \mu$

 compute $\hat{m}_t = \frac{m_t}{1 - \wp^t}$

 compute $\hat{\mathfrak{S}}_t = \frac{\mathfrak{S}_t}{1 - \beta^t}$

 update weight $\theta_{t+1} = \theta_t - \eta \frac{1}{\sqrt{\hat{\mathfrak{S}}_t}} \hat{m}_t$

 update weight $\omega' = \rho \omega + (1 - \rho) \omega'$ (ρ is learning rate)

 update weight $\theta' = \rho \theta + (1 - \rho) \theta'$

 Get velocity states $[\omega_t, p_t, q_t]$

 Get τ_{PID} from velocity states and $s_{t,PID} = [z, \phi, \theta]$

 Set the output thrust equal to: $\tau[0, 1, 5] = a_t$ and $\tau[2, 3, 4] = \tau_{PID}$

end

end

5.1.4 Reward Function

Optimal design of the reward function, $r(s_t, a_t)$, in Algorithm 1 is essential for the behaviour of the algorithm. As mentioned throughout this thesis, the reward is a feedback from the environment, based on how good it was to take a particular action in a given state. Due to this, the reward function needs to be designed in such a way that the agent accomplishes station keeping. The agent is continuously looking for the highest total cumulative reward in order to determine the optimal policy starting in any state. Because of this, different reward function designs needs to be evaluated, more about this in the next chapter.

5.2 Simulation and Experiment Setup

In order to sufficiently train and validate the dual controller, both simulation- and real-life experiments are used. The idea here is that the agent is trained in simulation, which is computational cheap and safe, and then validated in real-life.

5.2.1 Simulation Setup

A DRL algorithm requires many training samples for the algorithm to reach convergence towards an optimal policy. Therefore, simulation-based training provides the foundation of the algorithm. To do this, the simulation environments Gazebo and Robot Operating System (ROS) are used together with the UUV_SIMULATOR [13] and a dynamic model of the BlueROV2 [16]. Gazebo, which offers the ability to accurately and efficiently simulate a robotic system in complex environments, is used to simulate the Marine Cybernetics (MC) lab at Marin Teknisk Senter in Trondheim, Norway. ROS is used to simulate the dynamics of the BlueROV2, and combining these two results in the simulator in Figure 5.2.

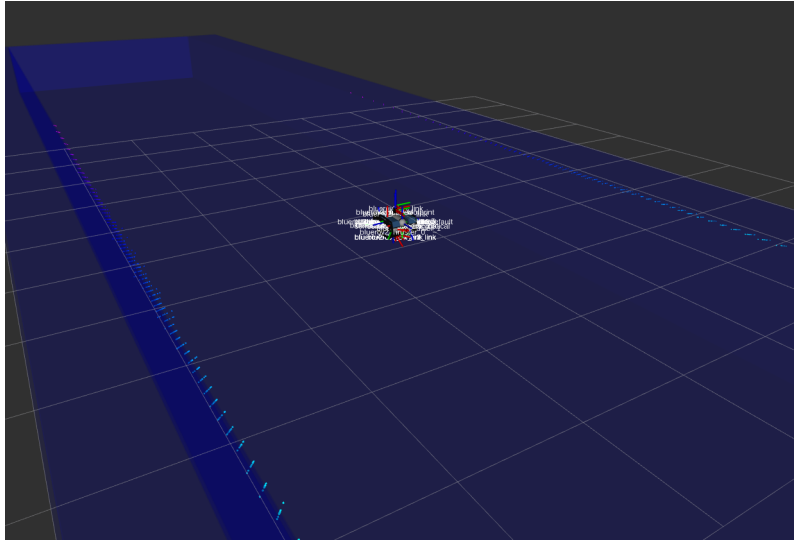


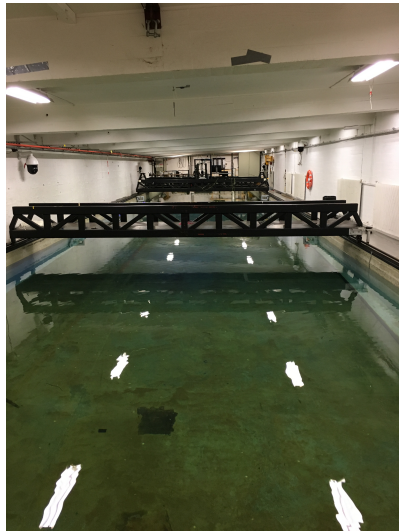
Figure 5.2: Gazebo and Robot Operating System (ROS) with the UUV_SIMULATOR [13]

For the implementation of the controller design, the programming framework *TensorFlow* is chosen. TensorFlow is an open-source machine learning framework, which works well in combination with the *Python* programming language.

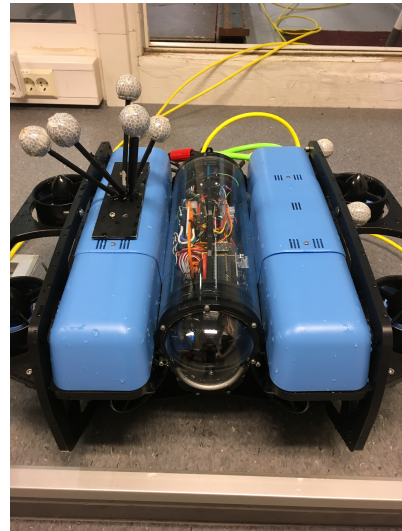
The UUV_SIMULATOR provides a set of topics for the vehicle, including sensor measurements, which are updated and published at every time-step. The DDPG and PD algorithm can subscribe to these topics, meaning that at every time-step they receive the (NED) position, orientation and velocity values. The computed thrust matrix is again published back to the ROS model, which enables the vehicle to move in the simulated MC-lab. The results from this is that the algorithm can store the pose and the corresponding thrust matrix at every time-step, and by that facilitate the possibility of learning.

5.2.2 Experiment Setup

The simulation results are validated on the actual BlueROV2 vehicle in the MC-lab at Marin Teknisk Senter. One of the larger differences between simulation and the real environment is pose estimation. In the real system, the pose is estimated from *Qualisys*, which is a motion-capture system installed in the MC-lab. The Qualisys setup is shown in Figure 5.3a, 5.3b and 5.4.



(a) MC-lab, Trondheim, Norway



(b) BlueROV2 with Qualisys Nodes

Figure 5.3: MC-lab Setup

In Figure 5.3a, the MC-lab pool is displayed. On each long-side wall, motion-capture cameras are installed, with 3 cameras on the left side and 2 cameras on the right side. In Figure 5.3b, the BlueROV2 vehicle is displayed, where 7 reflector balls are attached to the vehicle. In order to estimate the pose, the camera's need to *see* enough reflectors at every position in the pool to be able to determine a shape between them.

Since the vehicle is underwater, the underwater environment also needs to be mapped, which is done manually by moving a pole with reflectors all around the pool until a satisfactory map of the pool, from the camera's perspective, is accomplished.

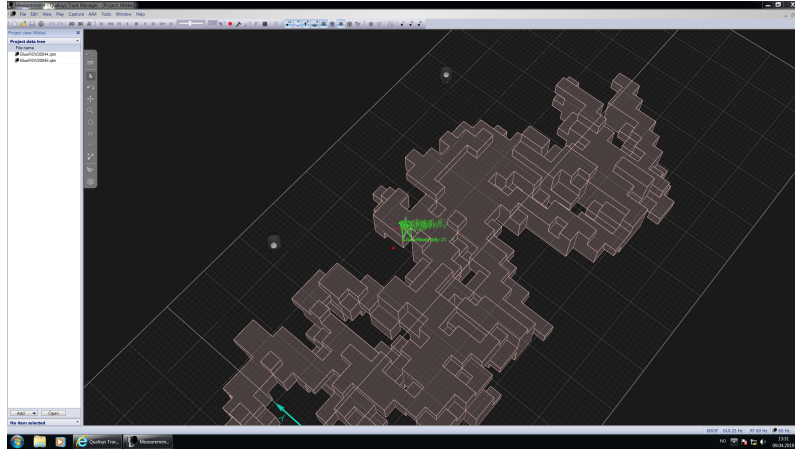


Figure 5.4: Qualisys Mapping

In Figure 5.4, the Qualisys map of the pool is displayed, where the green shape displays the vehicle. As long as the cameras manage to determine this shape, the pose estimation system is active. However, keeping this shape at every pose turned out to be very demanding, which is discussed in the results.

Chapter 6

Reward Shaping

The preliminary studies on station keeping of AUVs, with the use of a DDPG algorithm, showed that the agent managed to converge towards an optimal solution, but the reward function design was not sufficient for achieving station keeping [10]. The results showed that the agent had sufficiently learned how to approach the desired pose, but had not learned how to do station keeping at this pose. However, this was not due to the capabilities of the agent itself, but the agents' operational framework defined in programming. More specifically, wrongful design of the reward function to accomplish station keeping. Due to this, the investigation suggested to introduce a *time* constraint into the reward function, which would make sure that the terminal state was not reached until the vehicle had kept the desired pose for a sufficient time.

Furthermore, the solution presented in the preliminary work was not *universal*. A universal solution means that the vehicle is able to do station keeping at *every* pose in the state space, although it has only trained using one specific desired pose. As a result of this, these two problems are the starting point for the reward function design in this thesis.

6.1 Body-frame Error

In the preliminary DDPG algorithm design, the agent chose actions based on the error between the desired pose and the actual pose. This was defined as

$$s_{err}^{NED} = \begin{bmatrix} x_d - x_s \\ y_d - y_s \\ \psi_d - \psi_s \end{bmatrix} \quad (6.1)$$

However, the goal of the algorithm is to learn a universal policy, meaning that the vehicle should be able to do station keeping at an arbitrary pose. The state error vector defined in Equation 6.1 is defined in the (NED) frame, which is not universal. The flaw with this is that if the agent learns to do station keeping at a specific pose, it will not be able to do station keeping at any other arbitrary pose. To resolve this, s_{err}^{NED} needs to be rotated from the (NED) to (Body) frame, such that the error in x and y takes heading ψ into account. This is done according to Fossen [4] as

$$s_{err}^{Body} = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_d - x_s \\ y_d - y_s \\ \psi_d - \psi_s \end{bmatrix} \quad (6.2)$$

6.2 Reward Shaping with Time Constraint

One of the biggest flaws with the lack of a time constraint in the preliminary work was that the agent had no understanding of which actions to take when the desired pose was reached. To resolve this a time constraint is implemented in such a way that the agent has learned all the optimal actions to take close to the desired pose. This is done by introducing a clock based on the error pose in x and y . This is defined in Algorithm 2.

Algorithm 2: Reward Function with Time Constraint

```

if  $abs(X_e) < 0.2$  and  $abs(Y_e) < 0.2$  then
  r += a
  self.clock += 1
  if self.clock > b then
    r += c
    self.clock = 0
    done = True
  end
end
else
  self.clock = 0
end

```

where X_e and Y_e are defined as follows

$$X_e = \begin{bmatrix} x_e \\ x_e^{prev} \end{bmatrix} \quad (6.3)$$

$$Y_e = \begin{bmatrix} y_e \\ y_e^{prev} \end{bmatrix} \quad (6.4)$$

Here, x_e^{prev} and y_e^{prev} denotes the previous error in x and y , respectively. In Algorithm 2, the agent will check if both the error in x and y , as well as their previous error values, are within some limit, indicating that the vehicle is close to the desired pose. If this is the case, the agent receives a reward a , and a clock is initialised. If the clock exceeds some constant b , a large reward c is given, and the terminal state is reached. The constants a, b and c needs to be tuned in order to accomplish an optimal behaviour. The constant c is defined as a *large* reward compared to a , which is due to the fact that the agent should receive a larger reward for doing station keeping with some offset compared to only reaching the pose.

Furthermore, the constant b introduces a trade-off between teaching the correct policy and reaching the terminal state. If b is too large the agent might never reach the terminal state, and if b is too small the agent might not be at the desired pose for a sufficient time to learn station keeping.

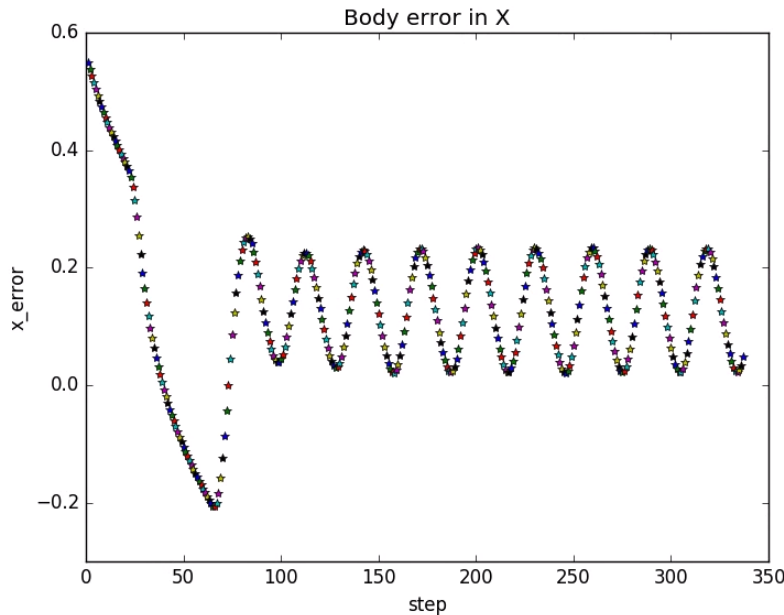


Figure 6.1: Circular Movement in x_e

The preliminary studies used the DDPG algorithm to control the states $[x, y, \psi]$, while a PD controller was used to control the states $[z, \phi, \theta]$. Introducing the time constraint to the reward function design revealed that the agent was staying within the correct quadrant in the pool, but instead of doing station keeping it performed circular movements with an offset from the desired pose, see Figure 6.1. This movement is assumed to be strongly correlated with heading, which in the preliminary

reward function design was somewhat neglected compared to the x and y states. In order to resolve this behaviour, the problem was initially simplified, resulting in investigating reward shaping *without* having the DDPG algorithm control heading first. This investigation is discussed in the following section.

6.3 DDPG without Heading

To simplify the problem, the control design was initially changed such that the DDPG algorithm controls the states $[x, y]$, while the PD controller controls the states $[z, \phi, \theta, \psi]$. Reducing the number of states in the DDPG algorithm should also simplify the learning process, since the algorithm *only* needs to learn the optimal behaviour based on 2 versus previously 3 states. Simplifying the problem showed great promise in convergence towards the optimal policy as $\epsilon \rightarrow 0$. However, when the optimal policy was evaluated, it revealed that the vehicle was following a path towards the edges of the pool, although the agent should have learned. This was defiantly not optimal, but it was a change from the previous problem, and a step further in understanding what the underlying issue was.

From evaluating the policy, it seemed to be a conflict between the heading and the $[x, y]$ states, mainly that the controller tries to satisfy $[x_e^{Body}, y_e^{Body}] \rightarrow 0$ while at the same time satisfy $\psi_e^{Body} \rightarrow 0$. Recall that the DDPG and PD are independent of each other, meaning that the DDPG algorithm does not have any information about the states in the PD controller. This means that when the agent is supposed to decide which actions to take in a given state, it does not possess any information about the heading state. The problem with this is illustrated in Figure 6.2.

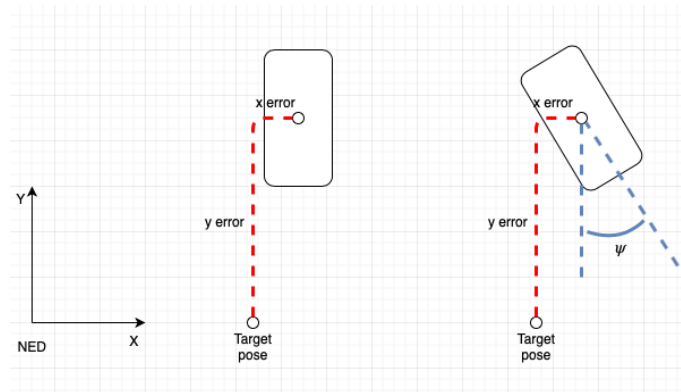


Figure 6.2: Lack of Heading Information

In Figure 6.2, two different AUV poses are illustrated, where the x_e^{Body} and y_e^{Body} are similar, *but* ψ_e^{Body} is different. For the DDPG algorithm, which only uses x_e^{Body} and y_e^{Body} , these two cases are inseparable. Recall that DDPG algorithms are closely connected to Q-learning in the way that the overall goal is to learn a Q-function and use this to learn a policy. As $\epsilon \rightarrow 0$ the agent should only exploit previous learned actions, which are stored in the Q-matrix. The Q-matrix has the form

x_e^{Body}	y_e^{Body}	a	r
0.53	2.67	[0.74, 0.9]	10
.	.	.	.
.	.	.	.

Table 6.1: Q-Matrix: without Heading

where arbitrary values are used for illustration. In Table 6.1, the state space, $[0.53, 2.67]$, and the action space, $[0.74, 0.9]$, have a dimension of 2. In a given state the agent uses the Q-matrix to choose the action which gave the largest reward in that state, which is the correct action. From Figure 6.2, it is obvious that this is a problem when the DDPG algorithm does not possess any heading information.

In order to resolve this, while still using PD control on heading, the state space of the DDPG algorithm is extended to a dimension of 3. This extends the Q-matrix into the form

x_e^{Body}	y_e^{Body}	ψ_e^{Body}	a	r
0.53	2.67	0.32	[0.74, 0.9]	10
.
.

Table 6.2: Q-Matrix: with Heading

Observe that the action space has not changed, meaning that the DDPG algorithm still only computes the thrust components for $[x, y]$. This means that the DDPG algorithm only *observes* the heading state, while the PD controller still controls it. The benefit of doing this is that the two cases in Figure 6.2 are no longer inseparable.

Unfortunately, the conflict between satisfying $[x_e^{Body}, y_e^{Body}] \rightarrow 0$ while at the same time satisfy $\psi_e^{Body} \rightarrow 0$ was not removed by doing this. Although the agent had *learned* the correct policy, the performance evaluation showed a movement pattern as illustrated in Figure 6.3. In order to explain what is happening here, three different stages of the repeating pattern are explained.

1. The agent observes $[x_e^{Body}, y_e^{Body}, \psi_e^{Body}]$, and choose the action which reduces $[x_e^{Body}, y_e^{Body}]$.
2. The action results in an increase in ψ_e^{Body} , which the DDPG algorithm does not control. Remember that it only cares about reducing $[x_e^{Body}, y_e^{Body}]$.
3. The PD controller now observes an increase in ψ_e^{Body} , and produces thrust to reduce this error. The result from this is that the vehicle is pushed back from the target pose instead.

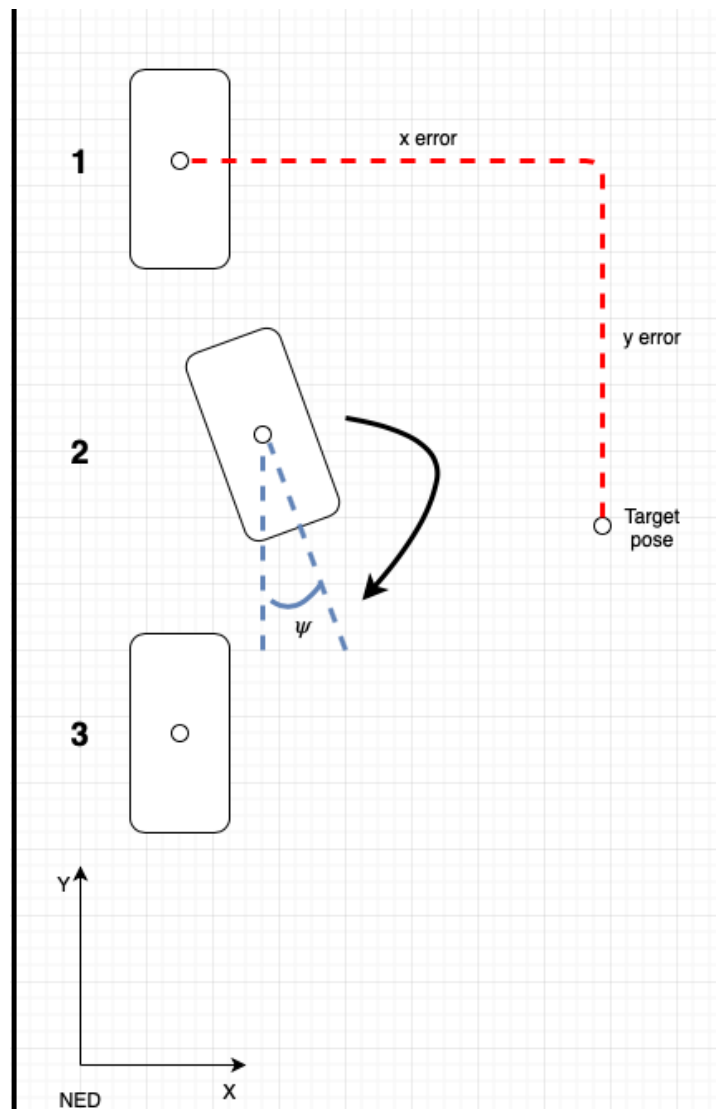


Figure 6.3: Conflict Between ψ and $[x, y]$

From Figure 6.3 it was clear that the conflict between ψ_e^{Body} and $[x_e^{Body}, y_e^{Body}]$ was still present. Recall that the error values are rotated from (NED) to (Body), which was initially done to make sure that the solution was *universal*. However, this was only needed when the DDPG algorithm was controlling all three states $[x, y, \psi]$. When ψ is no longer controlled by the DDPG algorithm rotating from (NED) to (Body) is unnecessary since x and y have the same values in the (NED) frame. Meaning that the solution is universal in (NED), and the rotation matrix can be removed. The reason for removing this is that it might be the reason for the attitude in Figure 6.3 since the x and y error are computed from a rotation about ψ , but the DDPG algorithm does not control ψ .

Removing the rotation removed the problem related to Figure 6.3 and defiantly resulted in a sufficient reward function for training the agent on the $[x, y]$ states. This result is presented in the next chapter.

Chapter 7

Results

This chapter presents the results from the investigations. The chapter divides the results into two main control designs; DDPG without heading and DDPG with heading. Both of these cases start by examining the training results and then proceeds to the validation results from the simulation and real-life experiments.

The control designs utilise a combination of a DDPG algorithm in conjunction with a PD controller and the learning parameters for the DDPG algorithm are similar in both designs. Table 7.1 shows the learning parameters for the DDPG algorithm.

number of episodes	4800
number of steps	1000
learning rate actor	10^{-4}
learning rate critic	10^{-4}
reward discount	0.9
memory capacity	10^4
batch size	16
ϵ_{max}	0.95
ϵ_{min}	0.05

Table 7.1: DDPG: Learning Parameters

Furthermore, *step* and *episode* are two parameters used throughout this thesis and are defined as follows.

- A **step** denotes one computer step in simulation, which in ROS and Gazebo approximately equals 0.02 seconds.

- An **episode** denotes one training period, which is either equal to 1000 steps, or the number of steps used to reach the *terminal state*. In each training period, the vehicle is initialised at the centre of the pool, and begin moving around. If the terminal state is not reached, the vehicle continues the training period until 1000 steps, and a new episode initialises. If the terminal state is reached, indicating that the vehicle has done station keeping at the desired pose for a sufficient time, the episode terminates and a new episode initialises.

From Table 7.1, the maximum possible number of steps during one full training session is 4.8 million. Recall that the number of steps in each episode only reaches 1000 *if* the terminal state is not reached. As discussed in Chapter 3, the DDPG algorithm has problems related to convergence, meaning that long training sessions are needed. If the algorithm does not reach convergence towards the optimal policy during this time, it is also possible to restart the training using the stored training data from the previous session.

The learning rate α for both the actor and critic neural networks are set as 10^{-4} . Recall that a small α makes sure that local minima in the loss function are not missed, but could also result in a longer time needed to reach convergence. To deal with this, α is initially set small (10^{-4}), such that if convergence becomes an issue, it can be increased. The reward discount γ , discussed in Chapter 2, makes sure that the uncertainty about the future is taken into account. This factor has to be smaller than 1, in order to prove convergence, and $\gamma \approx 0.9$ is usually used. The memory capacity deals with how much of the previous data that is stored at a time, which in this case is set as the 10 last episodes (10 000 steps). Recall from Chapter 3 that a batch is defined as a subset of all the training data, and the batch size sets the size of this batch. Here, the batch size is set as 16, which means that in each gradient update the algorithm utilises an arbitrary batch of size 16 from the training data. The last two parameters in Table 7.1 defines the upper- and lower bound for ϵ , which makes sure that the agent initially explores new actions, and gradually exploit learned actions with larger and larger probability.

7.1 DDPG without Heading

In the previous chapter it was discussed how the conflict between $[x, y]$ and ψ resulted in initially defining the DDPG algorithm with a state space of 3 and an action space of 2. This means that the DDPG algorithm observes $[x_e, y_e, \psi_e]$, but only computes actions to minimise $[x_e, y_e]$. While ψ_e and $[z_e, \phi_e, \theta_e]$ are controlled by the PD controller. Doing this results in the reward function design in Algorithm 3.

Algorithm 3: DDPG without Heading: Reward Function

```

r =  $\frac{1}{\|s_{err}\|+0.05}$ 
r += -6
if  $abs(s_{err_x}) < 0.05$  or  $abs(s_{err_y}) < 0.05$  then
  | r += 10
end
if  $abs(X_e) < 0.2$  and  $abs(Y_e) < 0.2$  then
  | r += 50
  | self.clock += 1
  | if self.clock > 100 then
  | | r += 5000
  | | self.clock = 0
  | | done = True
  | end
end
else
  | self.clock = 0
end
if  $abs(x) > 18.5$  or  $abs(y) > 2.5$  then
  | r += -100
end

```

The reward function for the DDPG algorithm without heading control is displayed in Algorithm 3. The first line in the algorithm indicates the reward for being close to the target pose. Here, the norm of the error state, s_{err} , is summed with a constant 0.05. The reason for doing this is illustrated in Figure 7.1.

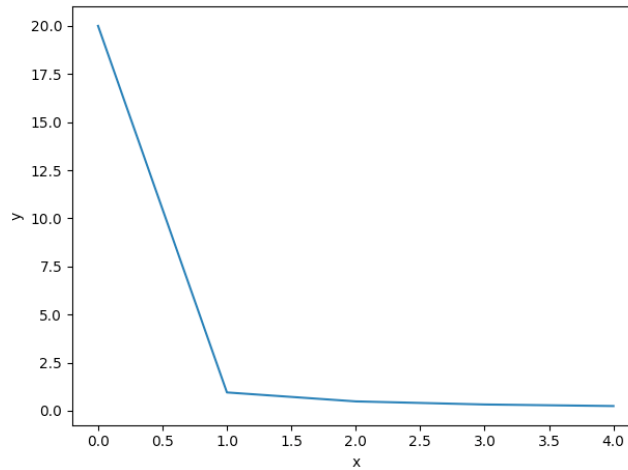


Figure 7.1: DDPG without Heading: Norm of the Error State

Taking the sum of the norm and a constant make sure that the agent receives a minimal reward when the vehicle is far from the desired pose and gives an exponential increasing reward when it is close. Since heading is not controlled by the DDPG, s_{err} only contains the error values in x and y . The second line in Algorithm 3 gives the agent a *penalty* at each time-step, which is crucial to avoid convergence to a sub-optimal policy. The following lines include the time constraint for station keeping, as well as an additional reward if the vehicle is close to either one of the desired axis. This is done to *help* the agent in the learning process. The last line in the algorithm is a penalty related to the bounds of the state space, in this case, the walls in the MC-lab pool, to make sure that the agent receives a substantial penalty if the vehicle makes contact with the wall.

7.1.1 Training Results

For each training session, three distinctive parameters are measured, these are the *total reward*, *number of steps* and *number of PD penalties*. The parameters are defined as follows

- The **total reward** is defined as a vector containing the average reward of the 100 latest episodes. Taking the average is done because of uncertainty in the measurements. The total reward can be interpreted as a *push-back* vector, where the latest reward is added at the end, while the first reward in the vector is pushed out.
- The **number of steps** is defined as the number of steps in each episode. If the *terminal state* is not reached this is equal to 1000, but if it is reached the number of steps is less than this. This is done to calculate how many episodes the agent is able to reach the terminal state, indicating that it has performed station keeping at the desired pose.
- The **number of PD penalties** is based on the parameters ϕ and θ violating some threshold. This is tracked to prevent a large pitch and roll movement, which can result in the vehicle spinning, which is not a desired policy.

In Figure 7.2, 7.3 and 7.4, the results from training are displayed. The agent was trained for approximately 600 episodes. Due to problems related to the software, Gazebo and ROS, the training session is restarted at episode 400.

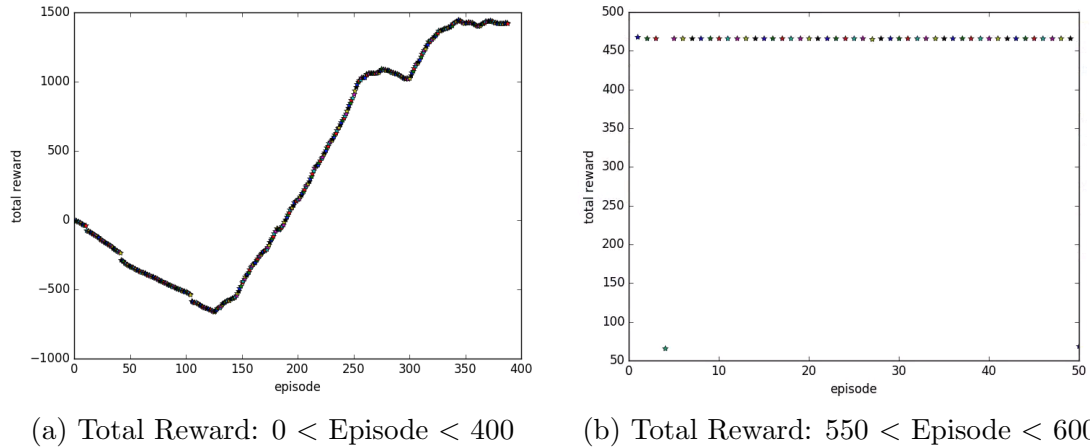


Figure 7.2: DDPG without Heading: Total Reward per Episode

In Figure 7.2a, the total reward over the first 400 episodes are visualised. Initially, the agent explores arbitrary actions in the environment, and since it also receives a penalty at each time-step, the decline during the first 125 episodes is expected. However, as $\epsilon \rightarrow 0$, the agent begins to exploit the learned best action in every state, and from episode 125 in Figure 7.2a, the total reward is increasing. By evaluating Figure 7.3a as well, one can see that the increase in total reward corresponds to the terminal state being reached. If the terminal state is reached, it means that the vehicle has successfully done station keeping at the desired pose for a sufficient time.

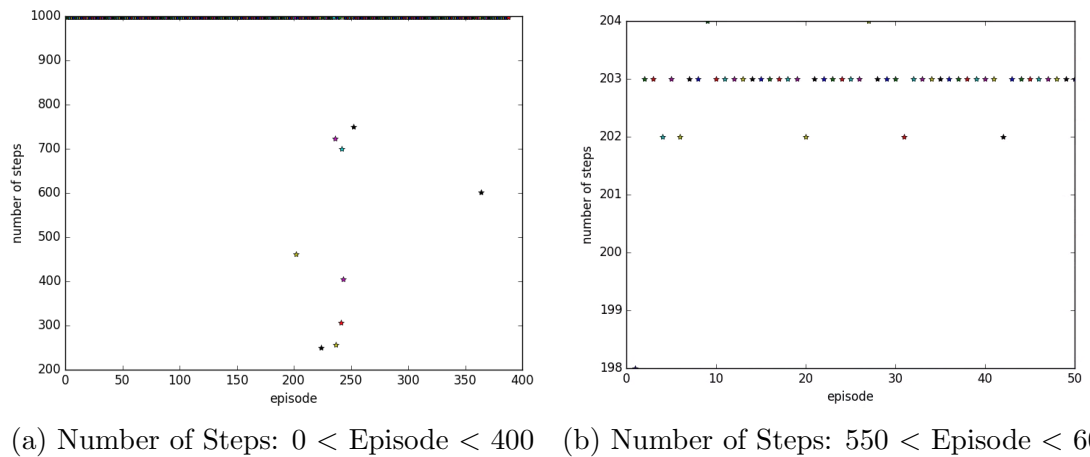


Figure 7.3: DDPG without Heading: Number of Steps per Episode

As the number of episodes are reaching 400, one can see in Figure 7.2a that the

total reward is starting to converge. However, it is not reaching the terminal state anymore in Figure 7.3a. By looking at the number of PD penalties in Figure 7.4a as well, the number of PD penalties is suddenly increasing after episode 300. This obviously impacts the training performance, and to resolve this the training session is restarted.

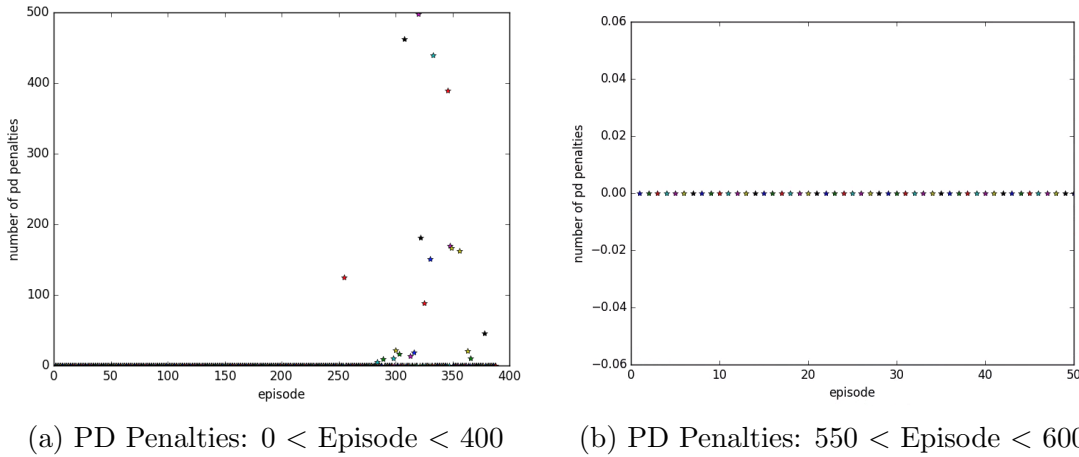


Figure 7.4: DDPG without Heading: Number of PD Penalties per Episode

By restarting the training session, while using the stored data from the previous session, the total reward eventually converges after ≈ 600 episodes. Figure 7.2b displays the last 50 episodes of the training, which shows that the total reward has stabilised at approximately 470. By looking at Figure 7.3b as well, it is shown that in each episode the agent uses approximately 203 steps to reach the terminal state, and in Figure 7.4b the number of PD penalties are reduced to zero. The agent has successfully converged to the optimal policy, and the next step is to evaluate the performance of this policy.

7.1.2 Validation Results: Simulation

In order to evaluate the policy performance, the agent should only exploit the learned actions in that policy. This means that in every state the agent will determine which action to take, based on the information stored in the Q-table. During training the agent was taught to do station keeping at the desired pose given by $[x_d, y_d, \psi_d] = [2, 0, 0]$. Since the solution should be universal it also needs to be evaluated at a different arbitrary pose, chosen as $[x_d, y_d, \psi_d] = [2, 2, 0]$. By plotting the error values in all 6-DOF, $[x_e, y_e, z_e, \phi_e, \theta_e, \psi_d]$, the following result was achieved.

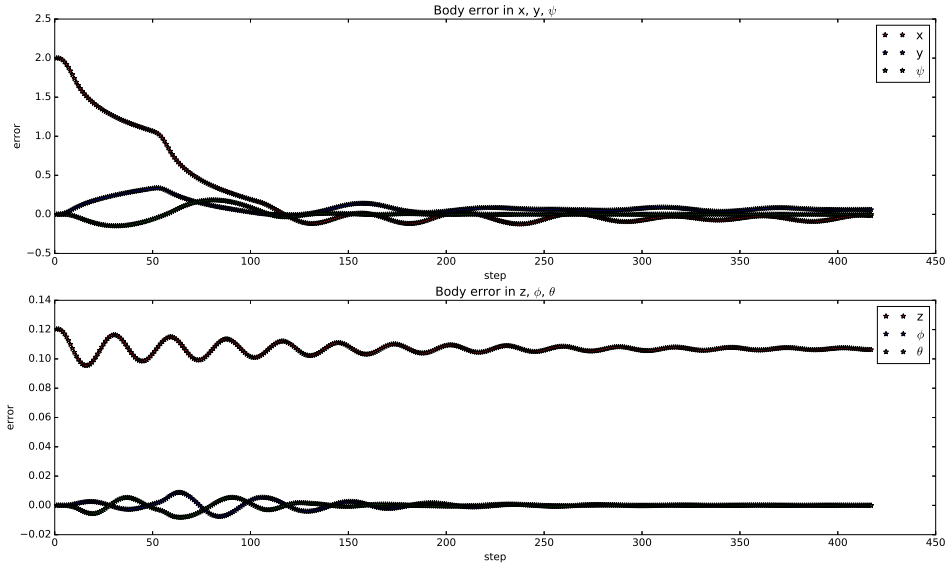


Figure 7.5: DDPG without Heading: Performance Validation, $[x_d, y_d, \psi_d]=[2,0,0]$

Figure 7.5 displays the state error values in $[x, y, \psi]$ and $[z, \phi, \theta]$. The state error values are defined in *meter* (m) for $[x, y, z]$ and *radian* (rad) for $[\phi, \theta, \psi]$. The desired pose is set as $[x_d, y_d, z_d, \phi_d, \theta_d, \psi_d] = [2, 0, 0, 0, 0, 0]$, and we see that the agent uses approximately 200 steps to stabilise at $[x_e, y_e] = [0, 0]$ which is ≈ 4 seconds. The oscillations are in the order of $10^{-2}m$, which is satisfactory.

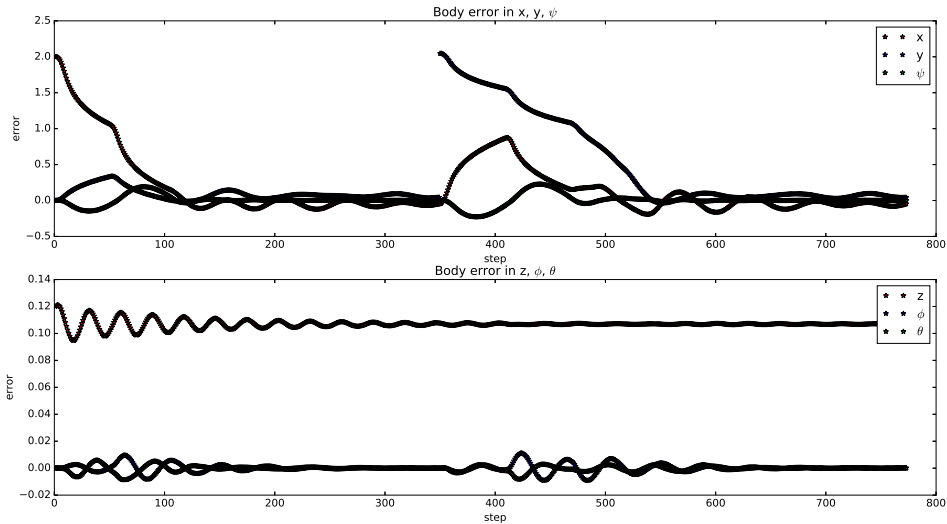


Figure 7.6: DDPG without Heading: Performance Validation, $[x_d, y_d, \psi_d]=[2,2,0]$

In Figure 7.6, the agent receives a new desired pose, $[x_d, y_d, z_d, \phi_d, \theta_d, \psi_d] = [2, 2, 0, 0, 0, 0]$, at $step = 350$. This is an arbitrary pose, and the agent is able to sufficiently reduce the error once again to satisfy $[x_e, y_e] = [0, 0]$. There is no difference in the performance between these two desired poses, meaning that the vehicle is able to do station keeping with the same precision at the arbitrary pose. Consequently, the solution is *universal*.

Evaluating the first plot in Figure 7.6 also reveals some of the conflict between satisfying $[x_e, y_e] \rightarrow [0, 0]$ at the same time as making sure that $[\psi_e] \rightarrow 0$, which was discussed in Chapter 6. Especially when looking at x after step 350, one can observe that it is 180° out of phase from ψ . This means that when ψ_e increases, the vehicle has a little bit more struggle of satisfying $[x_e, y_e] \rightarrow [0, 0]$, resulting in the not so smooth slope of especially x in this case. This is why it would be beneficial to use the DDPG algorithm on the heading state ψ as well, which is discussed later.

From the second plot in Figure 7.6 we see that the PD controller is doing a good job in controlling $[z, \phi, \theta]$, and ψ . When a new desired pose is given, it uses some time to stabilise, but the error values are in the order of $10^{-2}m$.

7.1.3 Validation Results: MC-lab Experiments

The overall reason for training the agent in a simulated environment, using Gazebo and ROS, was that in *theory* the performance of the vehicle in the real-life environment should be the same. This means that instead of training the agent on the real-life system, which is expensive and less safe, the agent is trained in simulation, and then applied to the real-life system.

This section includes the validation of the DDPG algorithm, without heading control, in the MC-lab at Tyholt with the actual BlueROV2 vehicle. Here, three experiments are conducted, with different desired pose definitions, as well as a DP 4-corner test. For every test, the following parameters are tracked.

- **DDPG state error**, which contains information about $[x_e, y_e, \psi_e]$.
- **DP state error**, which contains information about $[z_e, \phi_e, \theta_e]$.
- **Force**, which contains information about the produced force.
- **Torque**, which contains information about the produced torque.

An important note in these results is that the use of Qualisys as the positioning system means that the vehicles velocity states are not accessible, resulting in the

PD controller becoming a P (proportional) regulator.

Experiment 1

In experiment 1, presented by the results in Figure 7.7, 7.8, 7.9 and 7.10, the initial error state is defined as $[x_e, y_e] = [0.3, 0.3]$ and at $t = 15$ seconds a new error state is given to x such that $[x_e^{t=15}, y_e^{t=15}] = [-0.5, 0]$. The control objective was defined as this because it was reasonable to evaluate the performance when only changing one error state first, and then later include changes in both states. The reason for choosing $t = 15$ was to make sure that the vehicle had enough time to become stable at the desired pose.

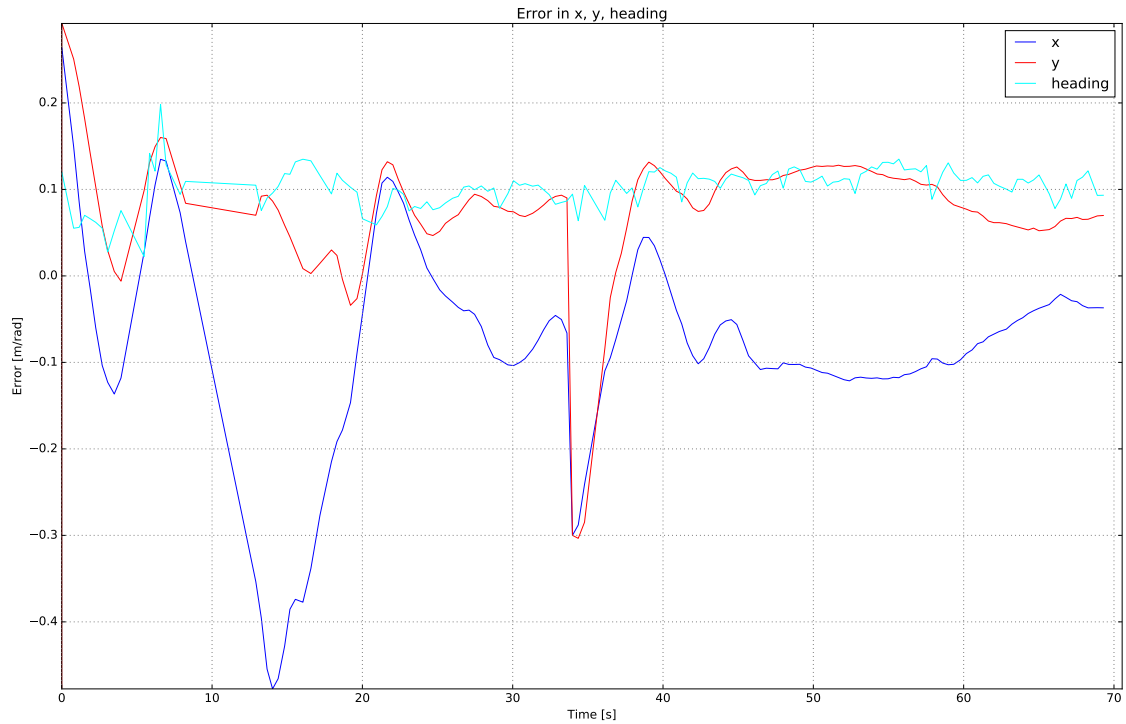


Figure 7.7: Experiment 1: DDPG State Error

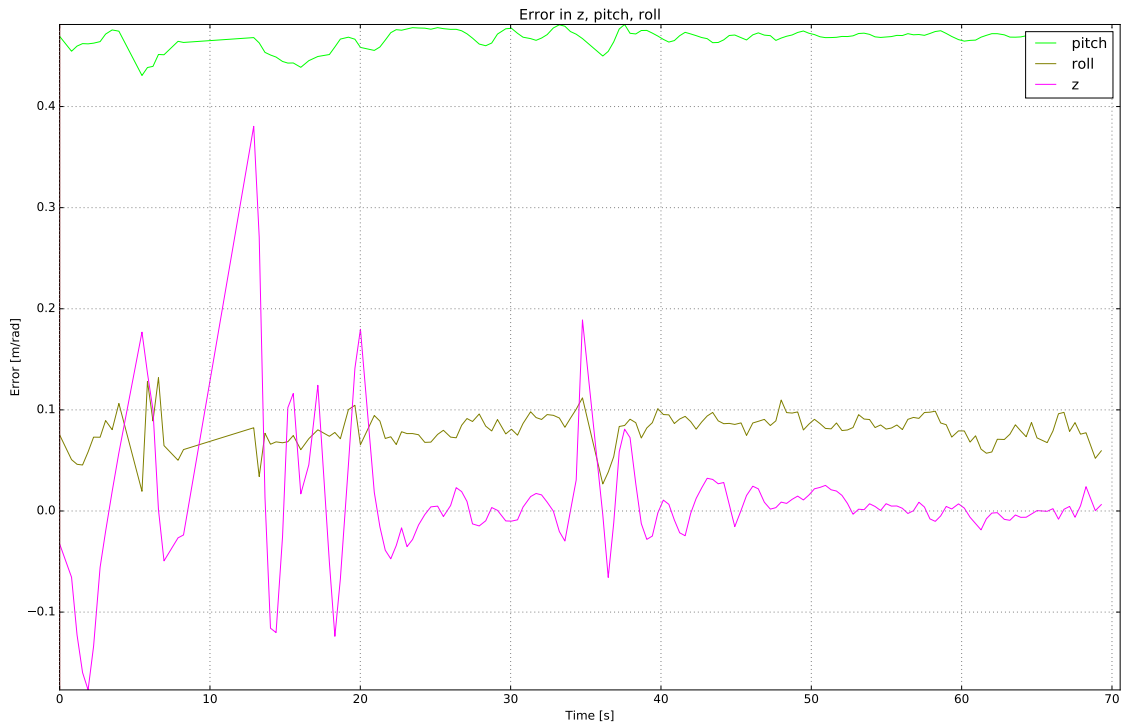


Figure 7.8: Experiment 1: PD State Error

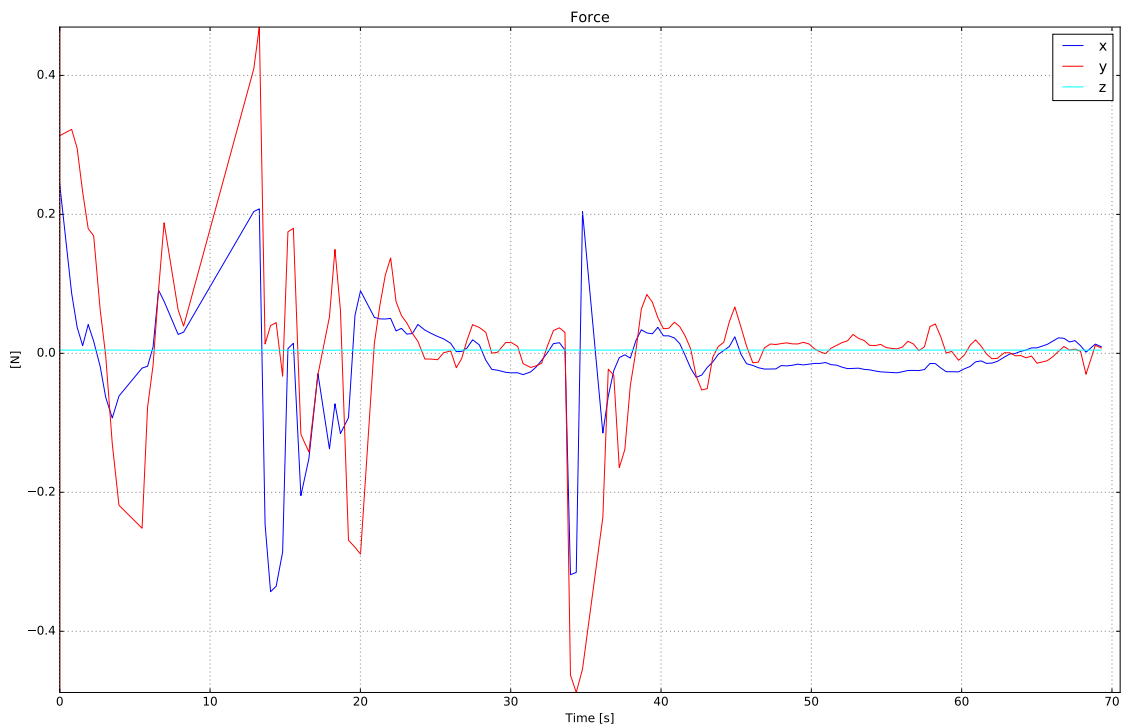


Figure 7.9: Experiment 1: Force

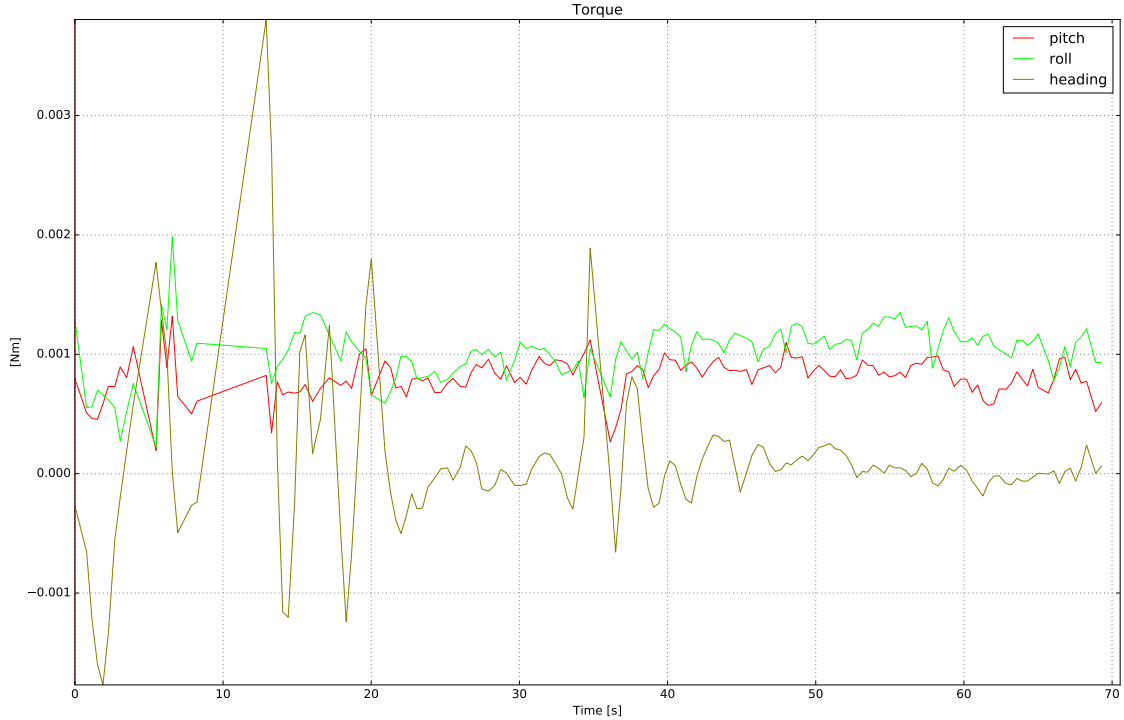


Figure 7.10: Experiment 1: Torque

In Figure 7.7, $[x_e, y_e, \psi_e]$ is tracked over 70 seconds. Starting off with the initial error state the vehicle is able to reduce the error value towards zero, with a relatively small oscillation in the order of $10^{-1}m$. At $t = 15$ seconds the new error state is given and the agent is able to reduce x_e once again towards zero. At $t = 35$ seconds a large spike in both x_e and y_e is observed, which results in x_e having a constant deviation for the remaining seconds. This spike is due to the lack of a robust pose estimation system, and clearly reveals the flaws of Qualisys. The Qualisys pose estimation system is very sensitive, and the possibility of the vehicle losing its pose is large. When this happens the agent will continue to produce thrust based on the last known pose until it *hopefully* regains the pose estimation system, and can produce the necessary thrust. However, from Figure 7.7 the agent does a good job in keeping the desired pose *when* the pose is known.

Figure 7.8 displays $[z_e, \phi_e, \theta_e]$, where the PD controller should keep these states constant, which it does with small oscillation. These states are also affected by the disadvantages of using Qualisys, which could explain why especially z_e has some larger spikes. In Figure 7.9 and 7.10, the force and torque outputs, respectively, are displayed, and these corresponds well with the results shown in Figure 7.7 and 7.8.

Experiment 2

In experiment 2, presented by the results in Figure 7.11, 7.12, 7.13 and 7.14, the initial error state is defined as $[x_e, y_e] = [0, 0]$ and at $t = 18$ seconds a new error state is given as $[x_e^{t=18}, y_e^{t=18}] = [0.5, 0.3]$. Here, the control objective was chosen such that a change in both error states at the same time could be evaluated. From the experience gained in the first experiment, where Qualisys created problems for the vehicle, t was increased to 18 seconds to make sure that the vehicle becomes stable at the desired pose.

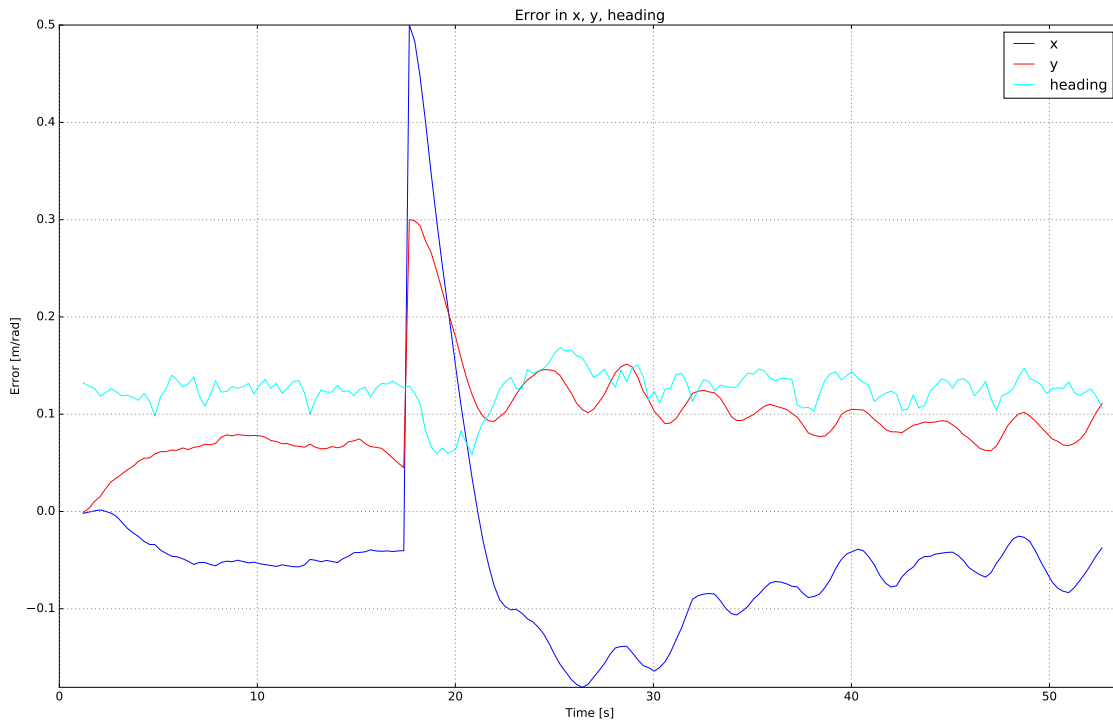


Figure 7.11: Experiment 2: DDPG State Error

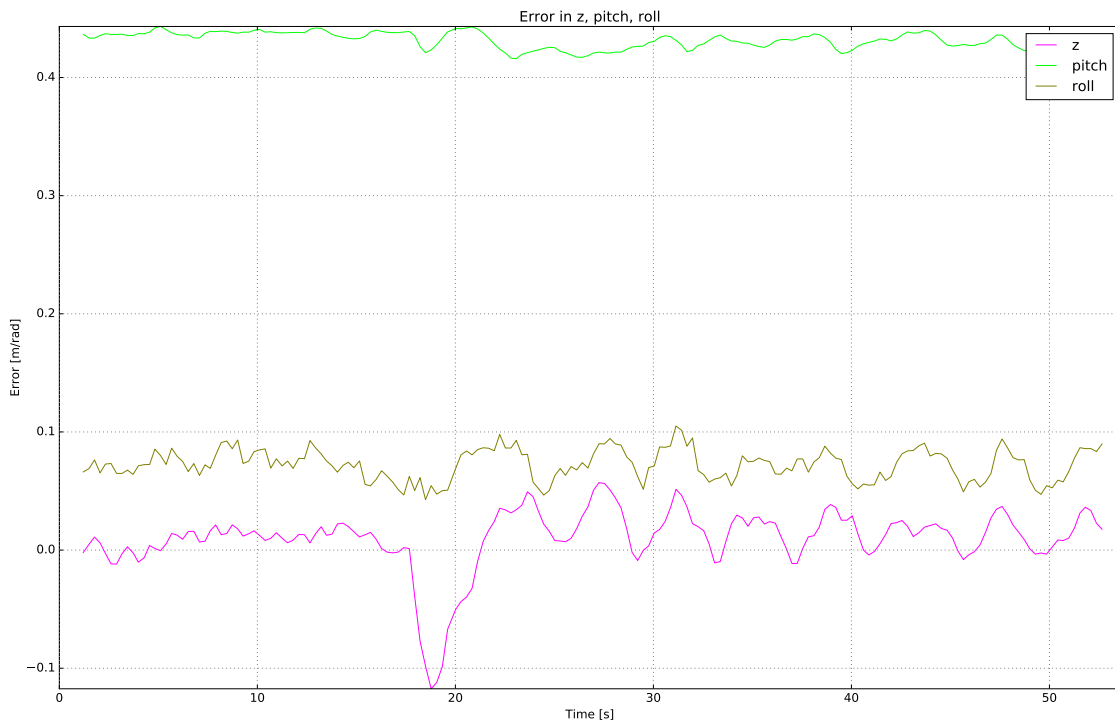


Figure 7.12: Experiment 2: PD State Error

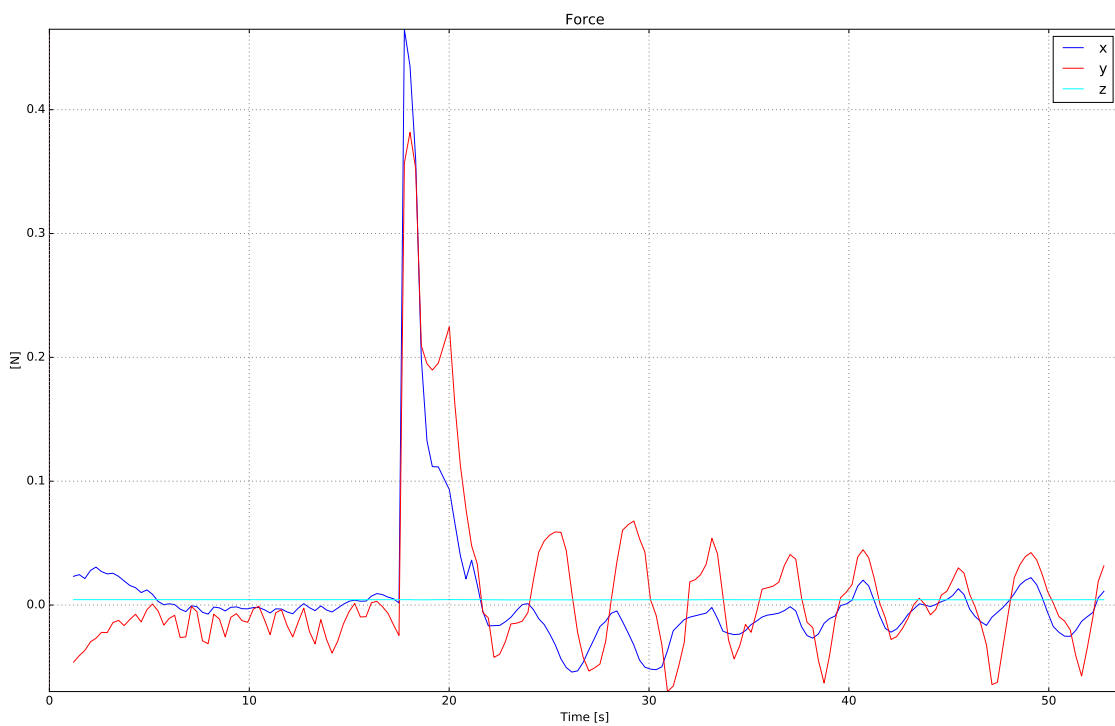


Figure 7.13: Experiment 2: Force

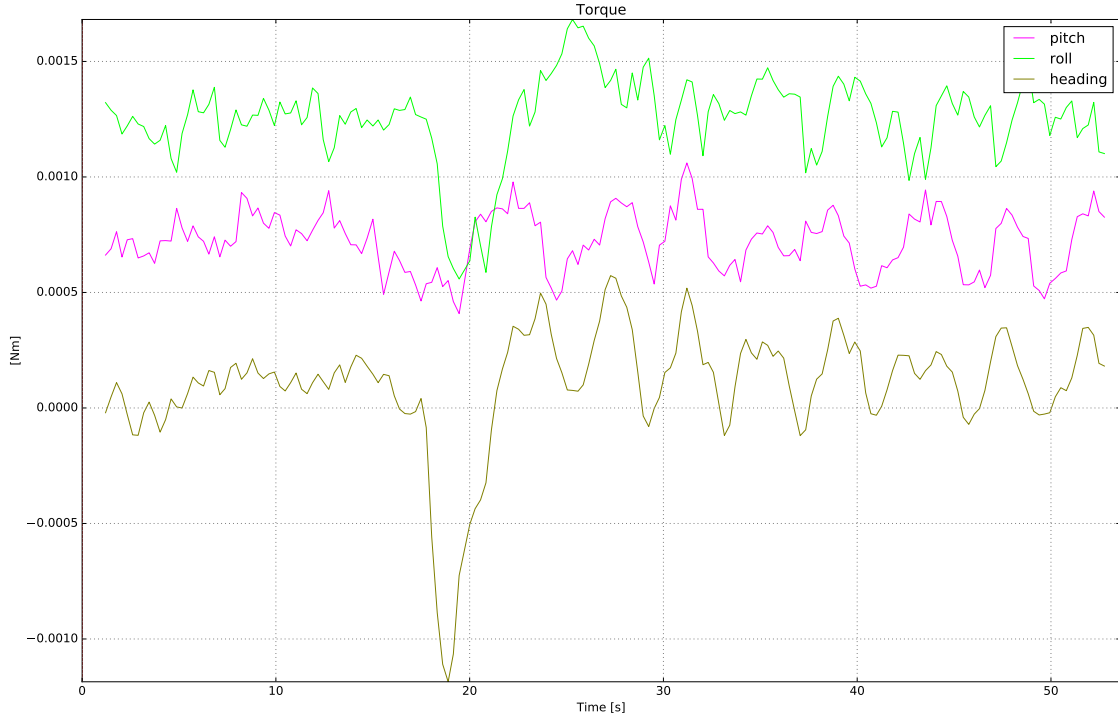


Figure 7.14: Experiment 2: Torque

In Figure 7.11, the DDPG state error does not include the unwanted spike presented in Figure 7.7, and the performance is overall more satisfactory. Again, this has to do with the positioning system which did not have the same impact on experiment 2 compared to experiment 1. The reason for this is that some areas in the MC-lab had better pose measurement capabilities, which as previously stated largely impacts the results. Better pose estimation also reflects the results in Figure 7.12, where the PD algorithm is able to keep z , ϕ and θ constant. Again, both the force and torque computations, visualised in Figure 7.13 and 7.14 respectively, coincides with what should be expected from the results in Figure 7.11 and 7.12.

Experiment 3

In experiment 3, presented by the results in Figure 7.15, 7.16, 7.17 and 7.18, a DP 4-corner test is conducted. The DP 4-corner test is a benchmark test in validation of dynamic positioning (DP) systems, and because of this it is included here. The DP 4-corner test is conducted by defining the error states as the following.

1. $[x_e, y_e] = [0, 0]$ at $t = 0s$
2. $[x_e, y_e] = [0.5, 0]$ at $t = 18s$

3. $[x_e, y_e] = [0, 0.5]$ at $t = 38s$
4. $[x_e, y_e] = [-0.5, 0]$ at $t = 70s$
5. $[x_e, y_e] = [0, -0.5]$ at $t = 90s$

Compared to the previous experiments, having the vehicle perform a squared movement results in an even larger possibility of the pose measurements being lost, which is revealed in the following results.

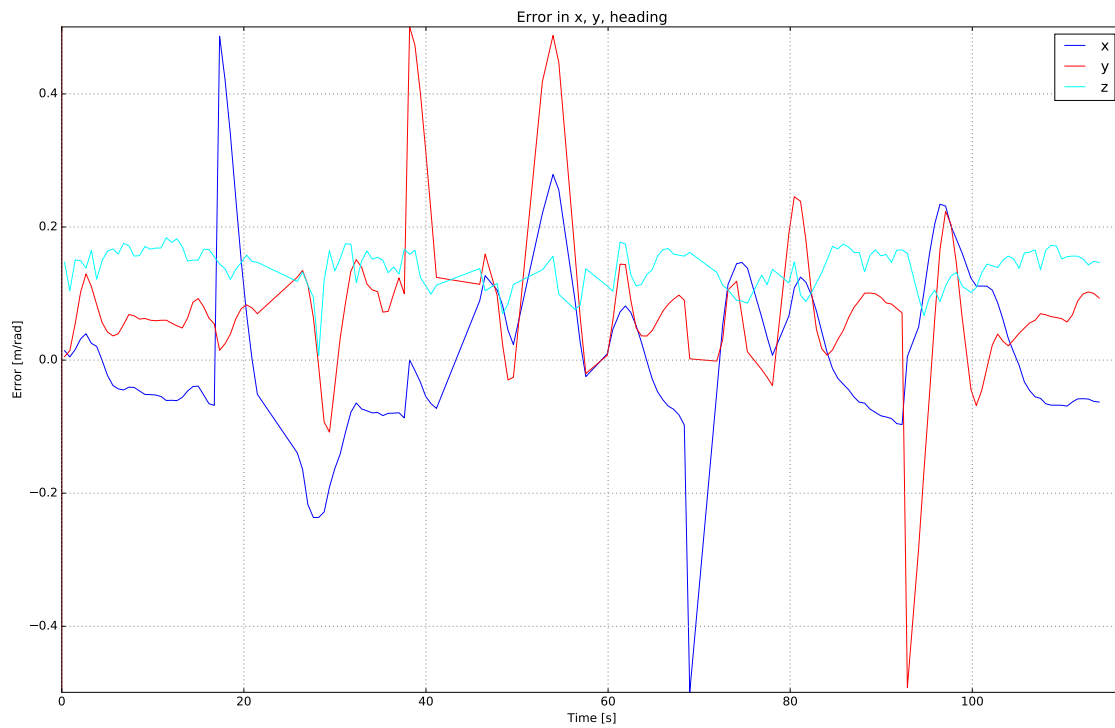


Figure 7.15: Experiment 3: DDPG State Error

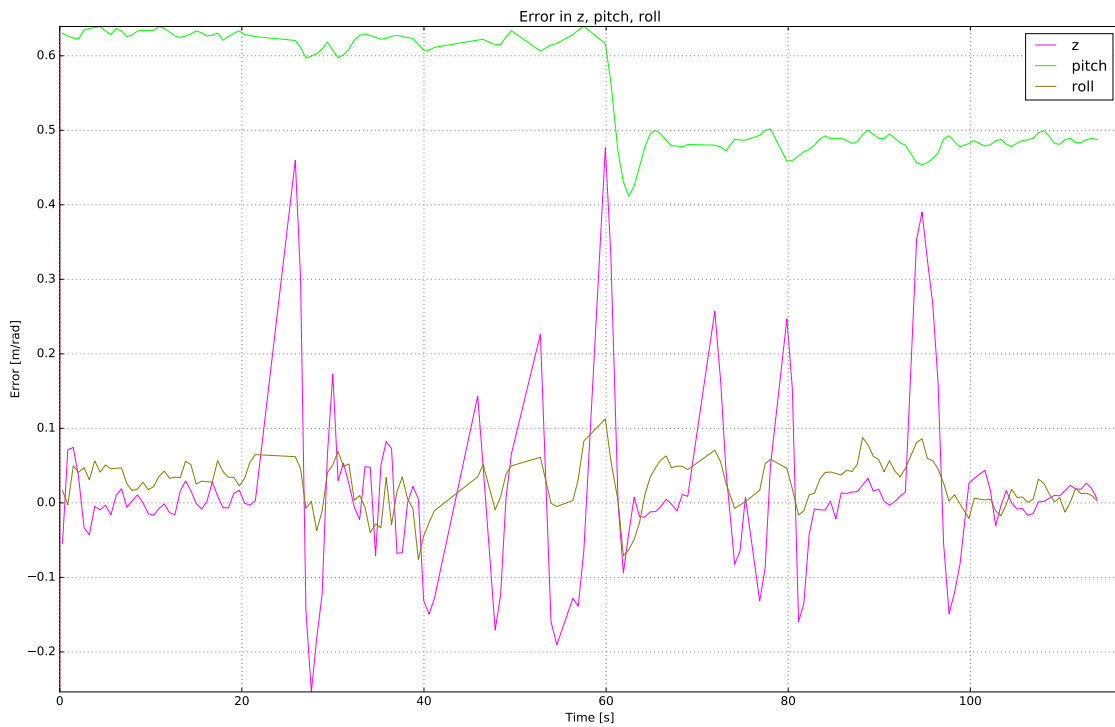


Figure 7.16: Experiment 3: PD State Error

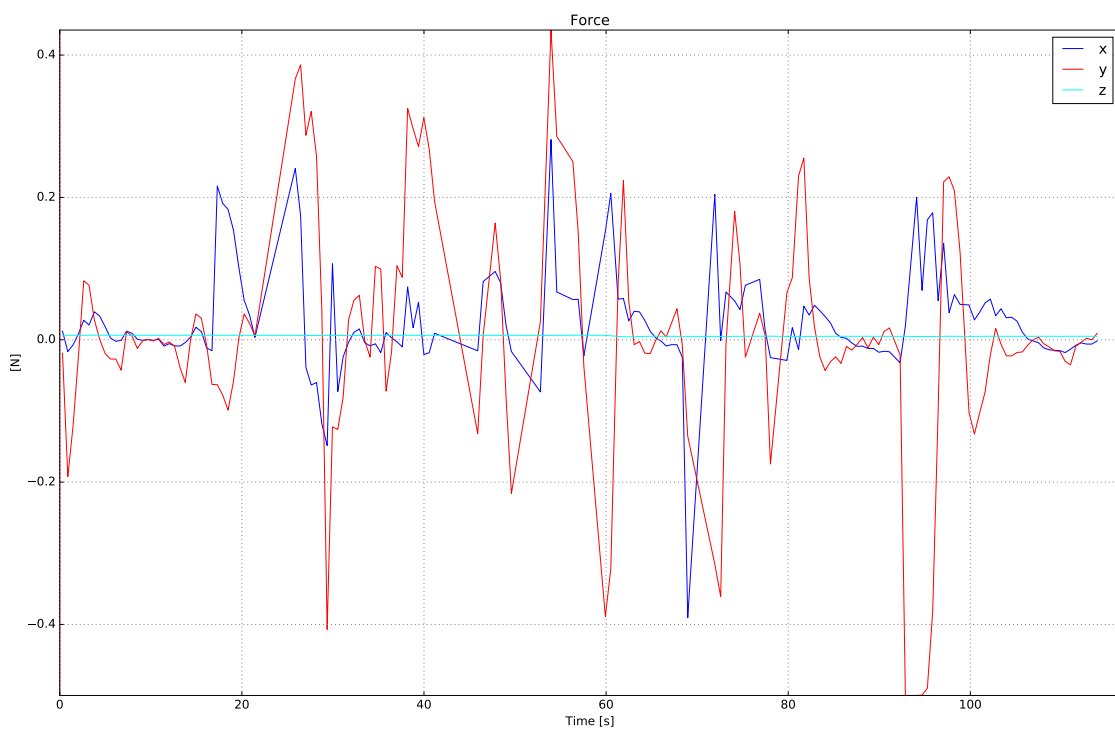


Figure 7.17: Experiment 3: Force

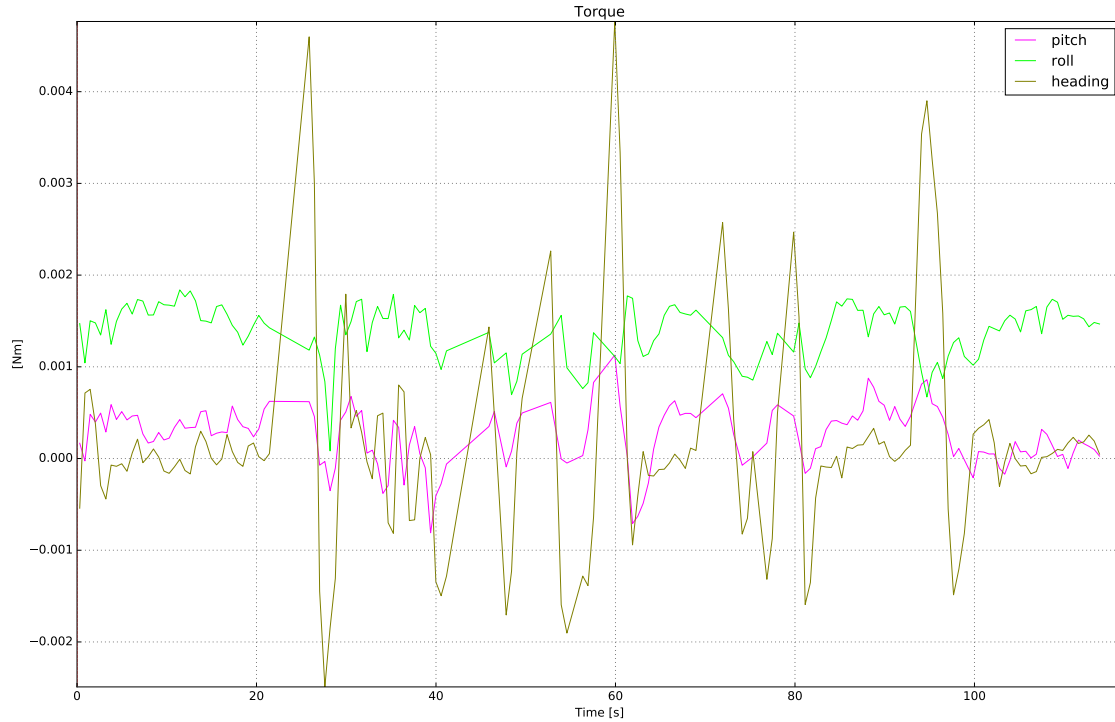


Figure 7.18: Experiment 3: Torque

From Figure 7.15, the performance in x_e and y_e are better than what would be expected based on the large possibility of losing the pose measurements. The agent does a satisfactory job in reducing the error values when a new desired pose is given, and the only larger *unwanted* spike occurs at $t = 55s$. The PD state error values in Figure 7.16 is also satisfactory, with some spikes occurring when a new desired pose in x and y is given. Once again, the force and torque measurements represented in Figure 7.17 and 7.18, respectively, coincides with the previous experiments, and obviously have larger oscillations because of the larger oscillations in Figure 7.15 and 7.16.

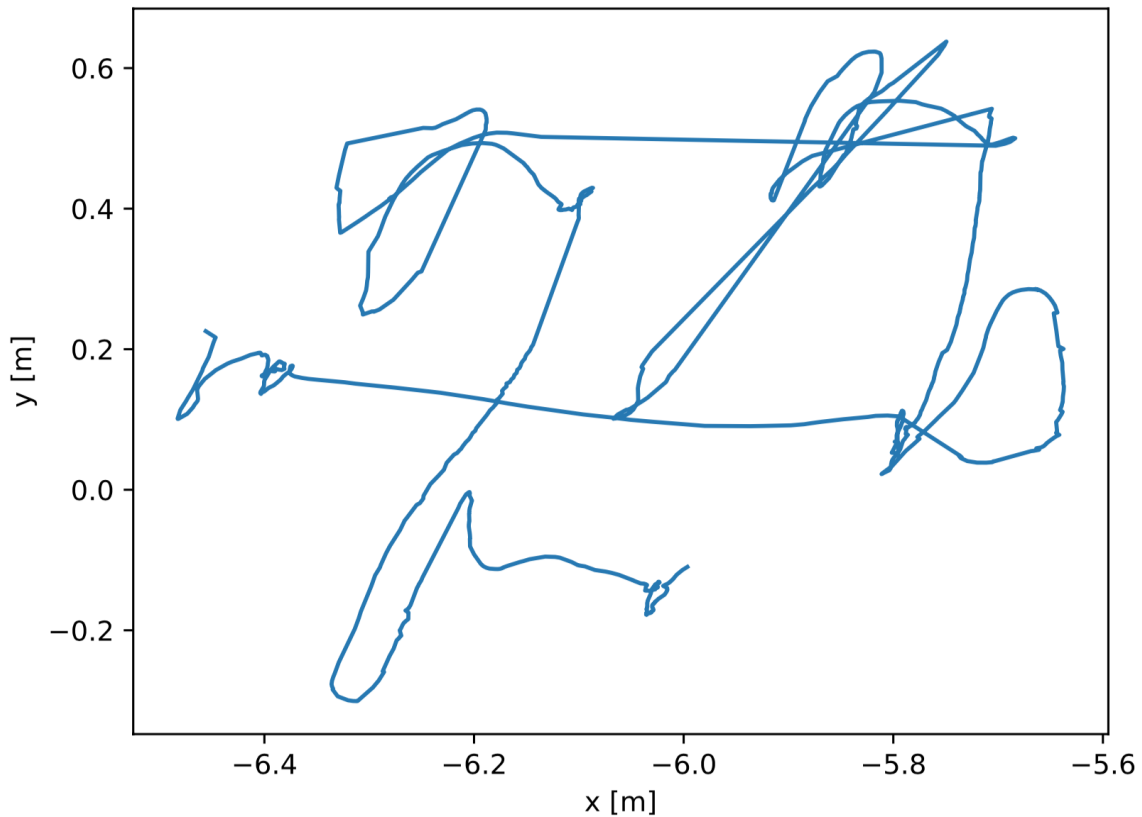


Figure 7.19: Experiment 3: DP 4-corner Test

In Figure 7.19, the performance of the DP 4-corner test is visualised, where the (NED) position in x and y are plotted against each other. The results shows that the vehicle performs a 4 corner movement, but struggles with overshooting at each turn. At the corner in $[x, y] = [-5.8, 0.5]$ this struggle is more extreme, but the reason for this is the loss of pose measurement at $t = 55s$ visualised in Figure 7.15. However, based on the initial concerns regarding the lack of pose measurements, the DP 4-corner test performance is better than what was initially expected.

Overall, the experiments conducted in the MC-lab on the BlueROV2 vehicle were satisfactory, especially based on the disadvantages of using Qualisys. In the areas where the pose estimation was present the vehicle was able to do station keeping at the desired pose with error values in the order of $10^{-1}m$, with some oscillation, in all three experiments. However, compared to the performance evaluation in the simulation environment, where the error values were in the order of $10^{-2}m$ and the oscillation was non-existent, this was defiantly worse for the real-life system.

7.2 DDPG with Heading

Controlling $[x_e^{NED}, y_e^{NED}]$ with the DDPG algorithm, and having the PD controller control ψ_e^{NED} , was a success. The agent was able to satisfactorily learn the optimal policy, which resulted in the vehicle doing station keeping within $[x_e^{NED}, y_e^{NED}] \in \pm 10^{-2}m$. However, the overall goal is still to use the DDPG algorithm to control all three states, and with increased knowledge from the previous tests, the heading is included in the DDPG algorithm. Meaning that $[x_e^{Body}, y_e^{Body}, \psi_e^{Body}]$ are controlled by the DDPG, and $[z_e^{Body}, \phi_e^{Body}, \theta_e^{Body}]$ are controlled by the PD controller. Observe that since the DDPG algorithm now controls heading as well the rotation from (NED) to (Body) needs to be included to make sure that the solution is universal.

The reward function design is the same as in Algorithm 3, the only difference being that s_{err} now includes heading as well.

7.2.1 Training Results

For each training session, three distinctive parameters are measured, these are the *total reward*, *number of steps* and *number of PD penalties*. The parameters are defined as follows

- The **total reward** is defined as a vector containing the average reward of the 100 latest episodes. Taking the average is done because of uncertainty in the measurements. The total reward can be interpreted as a *push-back* vector, where the latest reward is added at the end, while the first reward in the vector is pushed out.
- The **number of steps** is defined as the number of steps in each episode. If the *terminal state* is not reached this is equal to 1000, but if it is reached the number of steps is less than this. This is done to see how many episodes the agent is able to reach the terminal state, indicating that it has performed station keeping at the desired pose.
- The **number of PD penalties** is based on the parameters ϕ and θ violating some threshold. This is tracked to prevent a large pitch and roll movement, which can result in the vehicle spinning, which is not a desired policy.

Figure 7.20, 7.21 and 7.22 presents the results from training the algorithm over approximately 1000 episodes.

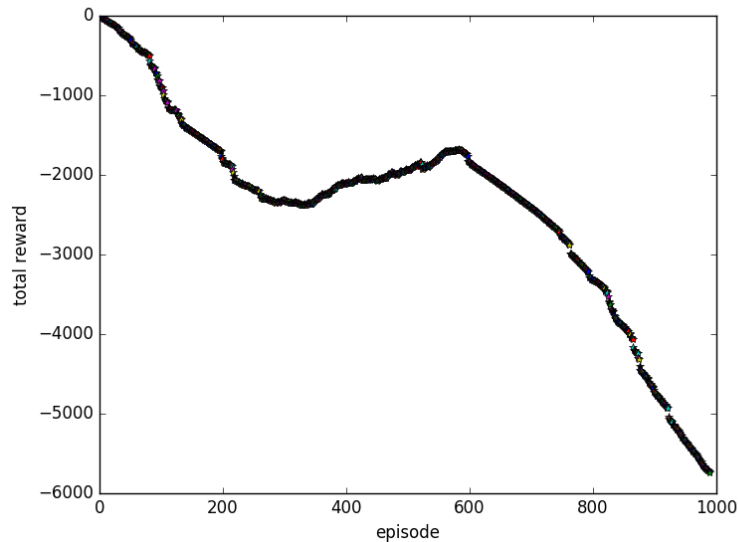


Figure 7.20: DDPG with Heading: Total Reward per Episode

When returning to the original design, an interesting behaviour was revealed. In figure 7.20 the total reward over all episodes is illustrated. Recall that the total reward was defined as the average value of the reward R from the 100 latest episodes. When training is initialised, ϵ is set equal to 0.95, which means that the probability of exploring new actions is substantial. This factor, together with the penalty the agent receives at each time-step, makes the first 200 episodes in Figure 7.20 a reasonable behaviour. The agent receives mainly negative rewards, which is revealed through this decline in total reward.

As $\epsilon \rightarrow 0$, the probability of the agent exploiting learned actions increases, and if the agent has observed the station keeping pose the total reward should increase, as is shown at episode 300 in Figure 7.20. This is further confirmed by investigating Figure 7.21, which illustrates the number of steps in each episode. If the station keeping criteria, defined in the reward function, is not satisfied the episode uses 1000 steps, but if it is satisfied, the terminal state is reached before this. From Figure 7.21 it is clear that around episode 300 the vehicle reach the terminal state prior to reaching 1000 steps, which means that it has satisfied the station keeping criteria.

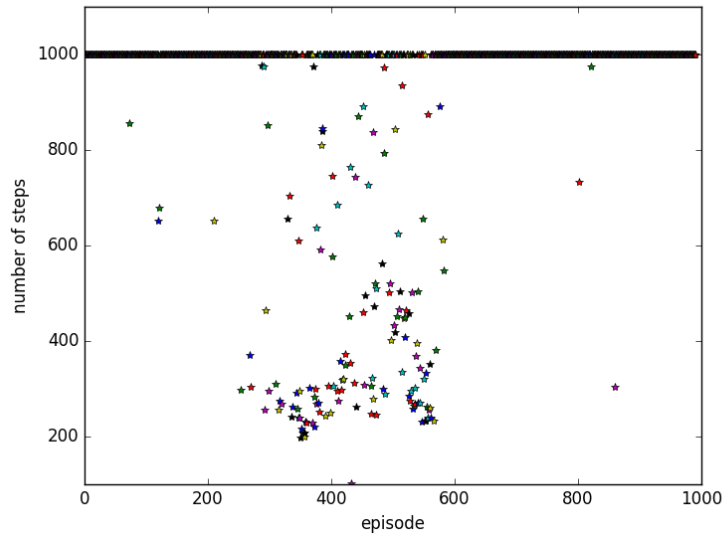


Figure 7.21: DDPG with Heading: Number of Steps per Episode

However, as the number of episodes is exceeding 600, the behaviour suddenly changes, and the total reward is declining rapidly, as shown in Figure 7.20. This is also revealed in Figure 7.21, where the number of steps is suddenly back to 1000, meaning that the terminal state is not reached.

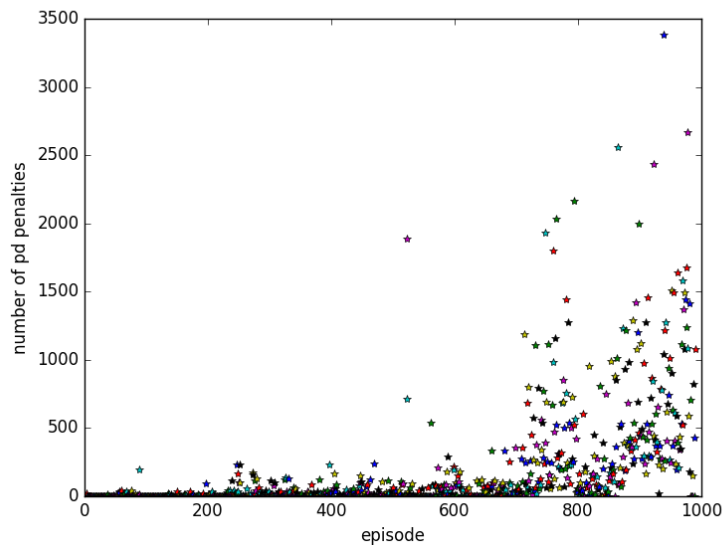


Figure 7.22: DDPG with Heading: Number of PD Penalties per Episode

From Figure 7.22 it is revealed that the number of PD penalties has a rapid increase after episode 600, which coincides with the episode where the previous plots saw a drastic change in behaviour. After inspecting the simulation through the graphical user interface as well, which allows for *viewing* the training, it was revealed that the roll and pitch movement could suddenly go spinning. This was probably the reason for the number of PD penalties suddenly increasing, and this defiantly was an unwanted behaviour. To resolve this, the training session was restarted multiple times, but unfortunately the algorithm was not able to reach convergence.

Chapter 8

Discussion

The results in Chapter 7 revealed that the dual control design was successful in doing station keeping at an arbitrary pose when the PD controller controlled heading ψ . The performance evaluation in the simulated environment revealed that the vehicle was able to satisfy station keeping capabilities with error values in the order of $10^{-2}m$. Furthermore, the agent was also successful in the real-life environment, where performance was slightly worse compared to simulation, but still satisfactory. As discussed in the previous chapter, the reasons for this were many.

8.1 Quality of Pose Measurement Sensors

First and foremost, the differences in pose measurement sensors between simulation and real-life were of a significant order. Whereas the agent had access to its pose at all times in simulation with zero flaws, the Qualisys software resulted in terrible pose measurement robustness in real-life. Nevertheless, the agent was performing well in real life as well when the pose measurement sensors were active. Having unsatisfactory real-life pose measurements also results in not having the possibility to test the ability of the DDPG algorithm to its *full* extent. As stated in the introduction to this thesis, a large part of the motivation for investigating the possibilities of AI-based controllers is due to the difficult process of doing traditional underwater control design. The reason for this is the complex underwater environments, which makes the autonomous control nonlinear since AUV motions are easily influenced by flow and hydraulic resistance. By only validating the station keeping capabilities in a *calm* real-life environment, with the vehicle restricted within a minimal area (0-0.5 *m*) before losing its pose measurements, the full extent of the agent's performance has not been tested.

The reliability of Qualisys also makes it challenging to compare the performance of the algorithm with traditional controller architecture, such as a PID controller. The areas where the pose was known were dependent on parameters such as pitch and roll angle as well, and because of this, it is impossible to reproduce the same conditions for every sea trial, which makes it irrelevant to compare performance. This limitation reveals one of the problems which is often neglected when working with developing new technology. Although one can develop a new controller, e.g. based on reinforcement learning, which performs well in a simulated environment, the liability of the sensors used in real-life is often not discussed. It doesn't matter if the controller performs excellently in a simulated environment if the real-life sensor technology is not sufficient. Furthermore, as mentioned in Chapter 7, the PD algorithm only had a proportional term P when used in experiments. The reason for this is that the BlueROV2 did not have a velocity sensor, i.e. DVL¹, meaning that the velocity states were not accessible. This is also a factor that could explain some of the discrepancy between the simulation and MC-lab results.

8.2 Dynamic Model

Another possible reason for the discrepancy between the results is the dynamic model of the BlueROV2, provided by Nielsen et al. [16]. Chapter 4 presented the dynamic model, which was the basis for the implementation of the DDPG algorithm in Chapter 5. The result from this is that the implementation has in fact used a model-based approach with a model-free algorithm, and due to assumptions used in the dynamic model it does not fully match the real-life vehicle.

One of the reasons for this mismatch is the coupling between motions. In the dynamic model of the BlueROV2 the equations of motion in surge x , sway y , heave z , pitch ϕ , roll θ and yaw ψ are assumed *decoupled*, meaning that each equation can be solved separately. However, this is not the case for a real-life system, where the equations of motion are *coupled*, at least to some extent. This means that for a real-life system the equations of motion in x, y, z, ϕ, θ and ψ will affect each other, which is not assumed in simulation, and could be the reason for worse performance.

Furthermore, in the dynamic model, the centre of mass (COG) is defined in the centre of geometry instead, due to simplicity. This simplification is also a factor that could have had some impact on the discrepancy.

¹Doppler Velocity Log

8.3 Limitations in Software

Applying the DDPG algorithm to all three states $[x, y, \psi]$ turned out to be a real challenge, which was largely due to the conflict between ψ and $[x, y]$ emphasised in Chapter 6. As discussed in Chapter 3, DDPG algorithms, in general, suffers from convergence difficulties, specifically the training time needed to reach convergence towards an optimal policy. The state- and action space considerably influence this, and increasing the action space from 2 to 3 states affects the training time needed to a great extent. Therefore, based on the theory in Chapter 3, the training time was expected to increase.

Although this was expected, the issue was made worse when the software being used also imposed a problem. In order for the algorithm to find the optimal solution, it needs a large number of simulation steps. Unfortunately, as the number of steps increased the simulation would *freeze*, resulting in the flow of messages being subscribed and published pausing for a longer period. This disadvantage further increases the necessary simulation time needed to reach convergence and it seemed to significantly reduce the quality of training as well, which can explain why the number of PD penalties is suddenly increasing in Figure 7.22. Understanding why this problem occurred was not accomplished as it was uncertain if it was due to the simulation environments, the performance of the computer used or the DDPG algorithm itself.

8.4 Difficulties in Reward Function Design

In Chapter 3, the challenges of policy gradients were discussed. Recall that in PG methods the steepest ascent (or descent) direction of the rewards is computed, defined as g in Equation 3.11. g is used to update the policy θ by taking a gradient step αg in the steepest direction. The problem with this is that the step is a first-order derivative, which essentially meant that the reward function was assumed to be flat. In Figure 8.1, the reward function defined in Algorithm 3 is visualised.

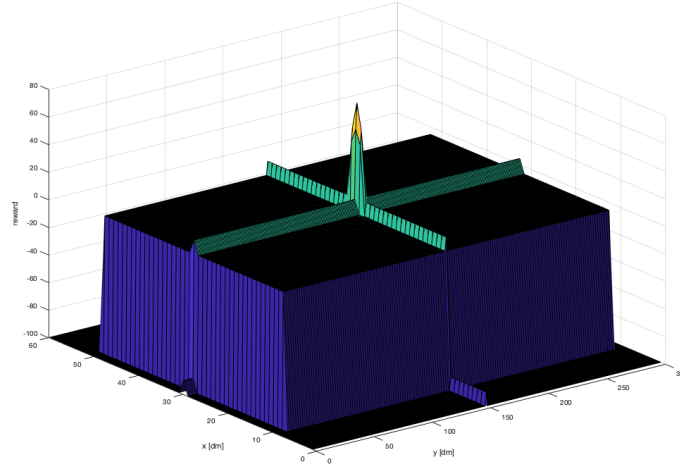


Figure 8.1: DDPG: Reward Function

From Figure 8.1, the reward function is relatively flat in the majority of states, with a large negative reward along with the states close to the edges of the pool. However, as the agent reaches the desired pose, there suddenly is a drastic increase in reward, as seen in Figure 8.1. This increase is due to the second if-sentence in Algorithm 3, where the agent receives a large reward if the current and previous error pose are smaller than some value. The issues in Chapter 6 revealed that this reward function design was needed to accomplish station keeping, but it is not flat. If the agent reaches the desired pose, indicating the top of the reward function in Figure 8.1, there is a good chance that it takes a too large step, and falls back down. When the agent resumes exploration, it does so from a worse state with a bad local policy, which means that it needs a long time to recover. This factor influence the convergence of the algorithm, and a more flat reward function would be desirable. Nevertheless, as shown by the difficulties in Chapter 6, there is a trade-off between the curvature of the reward function and accomplishing sufficient station keeping of the vehicle.

8.5 Possibilities of TRPOs and PPOs

Chapter 3 addressed how TRPOs and PPOs aims to deal with some of the challenges related to DDPGs. Especially PPO with clipped surrogate objective has been shown to have better performance than TRPO, as well as speed exceeding gradient descent

methods [22]. Due to this, developing a PPO based controller could be a possible solution to accomplish ML-based heading control as well. However, in this thesis, the PPO based controller was not implemented. First of all, the implementation of the DRL control design was a continuation of the work done during the preliminary studies, which showed promising results [10]. Because of this, continuing using a DDPG based controller was a reasonable starting point.

Furthermore, the vehicle was successful in doing station keeping at an arbitrary pose, both in simulation and in real life. Although this was accomplished by only using ML on the x and y states, it is not certain at all that the PPO would have done any better. Remember that one of the major problems was that the training would freeze when using ROS and Gazebo, and this would happen when the number of steps exceeded $\approx 10^6$. The discussion about the challenges of PG methods in Chapter 3 concluded that PG methods often need 10 million or more training steps to reach convergence [6]. However, from the results in Chapter 7, the DDPG algorithm without heading was able to converge towards an optimal policy only using $\approx 6 \cdot 10^5$ steps. Although a PPO based algorithm should need shorter training time to reach convergence, in theory, it is no guarantee that it would not exceed 10^6 number of steps and experience the freeze as well.

Furthermore, it is important to emphasise once again that the major flaw in the real-life experiments was not having sufficient pose measurement equipment, which is independent of the chosen deep reinforcement learning algorithm. Qualisys would probably not be used in a real-life application, e.g. intervention tasks at an installation site, but there are still flaws in real life sensor equipment as well, such as IMU and INS. In order to accomplish the full potential of DRL based controllers, there needs to be further development in robust and reliable sensor technology as well.

Chapter 9

Conclusions

This thesis has investigated the possibilities of using deep neural networks, trained by a deep reinforcement learning algorithm, to accomplish sufficient station keeping of AUVs in 6-DOF. From today's challenges in underwater control, in combination with the rapid developments in artificial intelligence, the possibilities of applying machine learning techniques in control design has been revealed.

A throughout introduction into the concepts of machine learning has showcased how reinforcement learning can be applied to solve problems in the underwater environment. Especially, deep reinforcement learning (DRL) techniques have shown remarkable results in control problems, and the thesis has discussed the fundamentals and advantages of three *state-of-the-art* methods within this family; Deep Deterministic Policy Gradient (DDPG), Trust Region Policy Optimisation (TRPO) and Proximal Policy Optimisation (PPO).

The investigation conducted in this thesis was a continuation of the preliminary work done on station keeping of AUVs [10]. This work showcased how a DDPG algorithm could be used to teach an agent to reach a desired pose. As a result of this, a dual control design, consisting of a DDPG algorithm in conjunction with a PD controller, was proposed to accomplish station keeping capabilities at an arbitrary pose. The key-component to accomplish sufficient implementation of DRL techniques is the reward function, and because of this, different reward function designs were evaluated, which emphasised the importance of using time constraints and penalise the agent.

The vehicle accomplished station keeping capabilities at an arbitrary pose, by using a DDPG based controller on the states $[x, y]$ and a PD controller on the states

$[z, \phi, \theta, \psi]$. The controller was successful in both simulation and real-life experiments, with error values in the order of $10^{-2}m$ and $10^{-1}m$, respectively. The reason for the discrepancy between the simulated and real-life experiments were two-fold. Firstly, the dynamic model used in training differs from the actual vehicle. Secondly, the flaws related to the real-life pose measurement equipment. However, the real-life experiments were satisfactory when the pose measurement equipment worked, and the thesis has verified that it is possible to train a DRL based controller in a simulated environment and apply it on a real-life system.

The investigation revealed that controlling heading ψ through ML-based techniques as well, was a difficult process. This was mainly due to the conflict between satisfying $[x_e, y_e] \rightarrow [0, 0]$ at the same time as making sure that $\psi_e = 0$. The main drawback with this conflict was that the time needed for the agent to reach convergence towards an optimal policy would increase significantly. Furthermore, the simulation environment, ROS and Gazebo, would also freeze as the number of steps increased, which further extended the time needed as well as aggravating training performance.

Suggestions for further work to resolve this issue, as well as on the subject in general, are presented in the next chapter.

Chapter 10

Further Work

The investigation done in this thesis was successful in teaching a DDPG algorithm to perform successful station keeping of the BlueROV2 by controlling the x and y states. However, as discussed throughout the thesis, the algorithm had difficulties in learning to control heading ψ as well. Furthermore, the real-life experiments also exposed weaknesses compared to the simulated results. In the following sections, suggestions for further work in order to resolve these issues, and improve the results, are discussed.

First and foremost, the issues related to convergence needs addressing. DDPG algorithms and PG methods, in general, suffers from the fact that it samples the whole trajectory for only one policy update, and that the policy cannot update at every time step. This design is not *sample efficient* since there probably are thousands of steps in one trajectory, which results in PG methods commonly needing 10 million or more training steps [6]. On the other hand, PPO methods enable multiple epochs of minibatch updates at each time step, which reduces the needed number of training steps significantly. Due to this, implementing a PPO based controller is a natural next step, and by changing the DRL method, it could also reveal if the problems related to freezing were due to the simulation software, Gazebo and ROS, or the DDPG algorithm itself.

Independent of the chosen DRL method, another suggestion for further work is to revisit the reward function design. As mentioned and emphasised throughout this thesis, as well as in the preliminary work, sufficient reward function design is the key-component for achieving successful implementation of reinforcement learning techniques. Chapter 6 discussed the difficulties of designing a sufficient reward function for the DDPG algorithm, but by having the design as shown in Algorithm

3 and Figure 8.1 the agent was able to learn station keeping capabilities sufficiently. However, from the discussion in Chapter 8, the trade-off between accomplishing station keeping capabilities and having the curved reward function in Figure 8.1 was revealed. DDPG methods assume a flat reward function curvature, so if it is desired to continue with this method, a redesign of the reward function could reduce the time needed for convergence.

One method that could resolve this issue, or at least reduce it, is *transfer learning* [27]. The core idea of transfer learning is that the experience gained from performing one task can help improve learning a related task. Transfer learning is usually applied to object detection tasks but could be applicable here as well. By dividing the task of accomplishing station keeping into sub-tasks, the reward function for each sub-task could be designed as a more flat function, and thereby reduce the possibility of taking a too large gradient step.

A third suggestion, which also is independent of the choice of DRL method, is to improve the real-life experiment setup. The real-life pose measurement system, specifically Qualisys, made it very difficult to measure the possibilities of the DRL controller to its full extent. Furthermore, since the areas where the system worked was not constant, it was impossible to reproduce the exact conditions for every trial. The result of this was that comparing the DRL results to, e.g. a PID controller was not of any use. Because of this, further work on the subject should defiantly try to resolve the issues related to Qualisys. One way to do this is to include an observer into the controller architecture. Observers are *state estimators*, which aims to reconstruct unmeasured states. When Qualisys does not work the pose estimate from the observer can be used instead, such that the agent can continue to compute the necessary thrust. Furthermore, an observer could also be used to estimate the velocity states, such that the derivative gain D in the PD controller is available in real-life as well.

Other suggestions for further work are given below.

- Performance is sensitive to hyper-parameter tuning, such as the learning rate which was discussed in Chapter 5. Tuning the hyper-parameters in Table 7.1 could improve performance.
- Further development of the BlueROV2's dynamic model, such that discrepancy between the simulated and real-life experiments are reduced.
- Use a DVL on the BlueROV2, such that the velocity states are accessible.
- Evaluate performance on a *better* computer, such as more CPU (GPU) and

RAM. Throughout this thesis a Dell laptop was used, and although we did not find evidence that this created the freezing problems it should be checked.

- Include *visual servoing* (VS) to estimate the pose, which was discussed in the preliminary work [10].

Bibliography

- [1] Bluerobotics.
- [2] Luis. Bermudez. Overview of Neural Networks. *Medium - machinevision*, 1(1), 2017.
- [3] Soumyajit Dasgupta. What is Remotely Operated Underwater Vehicle (ROV)? *Marine Insight*, 1(1), 2017.
- [4] Thor I. Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control*. John Wiley Sons Ltd, 2011.
- [5] Jonathan Hui. RL - Proximal Policy Optimization (PPO) Explained. *Medium - Towards Data Science*, 1(1), 2018.
- [6] Jonathan Hui. RL - Trust Region Policy Optimization (TRPO) Explained. *Medium - Towards Data Science*, 1(1), 2018.
- [7] Sanyam Kapoor. Policy Gradients in a Nutshell. *Medium - Towards Data Science*, 1(1), June 2018.
- [8] Min J. Kim. Way-point tracking of a Hovering AUV by PID Controller. *Control, Automation and Systems (ICCAS), 2015 15th International Conference*, 1(1), 2015.
- [9] Diederik. Kingma and Jimmy. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 1(1), 2014.
- [10] Kristoffer B. Knudsen. Deep Learning for Station Keeping of AUVs: A Preliminary Study. *Project Thesis, NTNU, Institute of Marine Technology*, 1(1), December 2018.
- [11] Vihar Kurama. Unsupervised learning with python. *Medium - Towards Data Science*, 2018.

- [12] Hunt Jonathan J. Pritzel Alexander. Heess Nicholas. Lillicrap, Timothy P. and Tom Erez. Continuous control with deep reinforcement learning. *arXiv:1509.02971*, 1(1), 2015.
- [13] Scherer S. Voss M. Douat L. Manhães, M. and T. Rauschenbach. UUV Simulator: A Gazebo-based package for underwater intervention and multi-robot simulation. *OCEANS 2016 MTS/IEEE Monterey*, 1(1), 2016.
- [14] Paula De. Mariona and Gerardo G. Acosta. Trajectory tracking algorithm for autonomous vehicles using adaptive reinforcement learning. *OCEANS'15 MTS/IEEE Washington*, 1(1), 2015.
- [15] Rustad Anne M. Moe, Signe. and Kristian G. Hanssen. Machine Learning in Control Systems: An Overview of the State of the art. *Bramer M., Petridis M. (eds) Artificial Intelligence XXXV. SGAI 2018. Lecture Notes in Computer Science*, 11311(1), 2018.
- [16] Eidsvik O. A. Blanke M. Nielsen, M. C. and I. Schjøberg. Constrained multi-body dynamics for modular underwater robots - Theory and experiments. *Ocean Engineering*, 358-372(1), 2018.
- [17] OpenAI. Deep Deterministic Policy Gradient. *OpenAI Spinning Up*, 1(1), 2018.
- [18] K. O'Shea. An Introduction to Convolutional Neural Networks. *Research gate, Aberystwyth University*, 1(1), 2015.
- [19] Jean Francois. Puget. What is Machine Learning. *IBM community*, 1(1), 2016.
- [20] Jeffery S. Riedel and Anthony J. Healey. Shallow Water Station Keeping of AUVs Multi-Sensor Fusion for Wave Disturbance Prediction and Compensation. *Naval Postgraduate School, Center for AUV Research*, 1(1), 1998.
- [21] Levine Sergey. Moritz Philipp. Jordan Michael. Schulman, John. and Pieter. Abbeel. Trust Region Policy Optimization. *University of California, Berkeley, Department of Electrical Engineering and Computer Sciences*, 1(1), 2017.
- [22] Wolski Filip. Dhariwal Prafulla. Radford Alec. Schulman, John. and Oleg. Klimov. Proximal Policy Optimization Algorithms. *OpenAI*, 1(1), August 2017.
- [23] Ben Shaver. A Zero-Math Introduction to Markov Chain Carlo Methods. *Medium - Towards Data Science*, 1(1), December 2017.
- [24] Badreesh Shetty. Supervised machine learning: Classification. *Medium - Towards Data Science*, 2018.

- [25] David. Silver. Deterministic policy gradient algorithms. *Proceedings of the 31st International Conference on Machine Learning*, 1(1), 2014.
- [26] David Silver. Introduction to reinforcement learning. *University of London*, 2015.
- [27] Matthew E. Taylor and Peter. Stone. Transfer Learning for Reinforcement Learning Domains: A Survey. *Journal of Machine Learning Research 10 (2009) 1633-1685*, 1(1), 2009.
- [28] Andre Violante. Simple Reinforcement Learning: Temporal Difference learning. *Medium*, 1(1), 2018.
- [29] Shi Zhenyu. Huang Chaoxing. Li Tenglong. Yu, Runsheng. and Qiongxiang Ma. Deep Reinforcement Learning Based Optimal Trajectory Tracking Control of Autonomous Underwater Vehicle. *Proceedings of the 36th Chinese Control Conference*, 1(1), 2017.
- [30] Hafidz Zulkifli. Understanding Learning Rates and How It Improves Performance in Deep Learning. *Medium - Towards Data Science*, 1(1), 2018.

Appendix A

BlueROV2 Parameters

The following matrices are extracted from *BlueRobotics* [1].

A.1 Rigid Body Mass Matrix

$$\mathbf{M}_{RB} = - \begin{bmatrix} 10.5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.156 & 0.003 & -0.006 \\ 0 & 0 & 0 & 0.003 & 0.214 & 0.004 \\ 0 & 0 & 0 & -0.006 & 0.004 & 0.127 \end{bmatrix} \quad (\text{A.1})$$

A.2 Added Mass Matrix

$$\mathbf{M}_A = - \begin{bmatrix} 7.0377 & -1.2910 & -1.6817 & 0.0954 & 0.2690 & -0.0563 \\ 0.5638 & 18.5399 & 0.9321 & 0.193 & -0.1080 & -0.1984 \\ 2.6036 & 8.6394 & 13.2816 & -0.5730 & -1.7952 & 0.2603 \\ 0.0587 & 0.2892 & 0.0834 & 0.0546 & -0.0087 & -0.0284 \\ 0.1266 & 0.1660 & 0.1468 & -0.0124 & 0.0173 & 0.0044 \\ -0.0621 & -0.2041 & -0.0711 & -0.0168 & 0.0061 & 0.2795 \end{bmatrix} \quad (\text{A.2})$$

A.3 Damping Matrices

$$\mathbf{D}_L = -diag\{0, 0.26, 0.19, 0.895, 0.287, 4.64\} \quad (\text{A.3})$$

$$\mathbf{D}_Q = -diag\{3.96|u|, 103.25|v|, 74.23|\omega|, 0.084|p|, 0.028|q|, 0.43|r|\} \quad (\text{A.4})$$

Appendix B

IEEE Oceans 2019 Seattle - Abstract

Deep Learning for Station Keeping of AUVs

1st Kristoffer Borgen Knudsen
Dept. of Marine Technology
NTNU
Trondheim, Norway
kristobk@stud.ntnu.no

2nd Mikkel Cornelius Nielsen
Dept. of Marine Technology
NTNU
Trondheim, Norway
mikkel.cornelius.nielsen@ntnu.no

3rd Ingrid Schjøberg
Dept. of Marine Technology
NTNU
Trondheim, Norway
ingrid.schjolberg@ntnu.no

Abstract—Control of underwater vehicles remains an active research topic within the literature. Multiple challenges exist for controlling an underwater vehicle, including highly nonlinear effects due to hydrodynamics. Control based models seek to model the underlying dynamics but suffer from the balance between tractable computation and performance. Machine Learning (ML) control techniques show promise as an alternative to classical model-based approaches.

This paper investigates the usage of a model-free deep reinforcement learning algorithm, Deep Deterministic Policy Gradient (DDPG), for station keeping in six degrees of freedom (DOF) for an underwater vehicle.

Index Terms—underwater robotics, station keeping, deep reinforcement learning, deep deterministic policy gradients

I. INTRODUCTION

Station keeping denotes the act of maintaining a constant position and orientation (pose), relative to a reference object [1]. Many underwater operations rely on the underwater vehicles station keeping capabilities, and thus station keeping represents a fundamental control task.

The controller design is crucial for accomplishing station keeping. Unfortunately, the underwater environment is complex, making the autonomous control nonlinear since flow and hydraulic resistance easily influences the AUVs motions [2]. In turn, the classic model-based approaches become challenging to apply. These challenges, in combination with the rapid developments in artificial intelligence (AI), have triggered the interest of applying machine learning (ML) techniques in AUV control designs.

This paper aims at applying Deep Reinforcement Learning to enable station keeping for AUVs by a dual controller design. The dual controller design encompasses a DDPG algorithm, based on the work of Silver et al. and Runsheng et al. [2] [3], in conjunction with a PD controller.

II. METHOD

Reinforcement Learning (RL) aims at solving tasks through an agent, who seek to maximize a reward signal obtained from the environment in which the agent acts. The reward signal informs the agent on how good the action taken in a particular state was. The agent progressively learns the environment through rewards and tries to maximize the accumulated rewards over all states in the environment by finding an optimal policy to follow [4]. Deep Deterministic Policy Gradients (DDPGs) is a state-of-the-art DRL algorithm, which utilizes

Artificial Neural Networks (ANNs) to learn a policy. DDPG has shown remarkable results on both benchmark computer games [3] and trajectory tracking control of AUVs [2]. This paper applies DDPG to the station keeping problem on a BlueROV2 platform.



Fig. 1: Left to right: The simulation environment Gazebo, the actual BlueROV2 experiment setup, the location of the experiments at MCLab NTNU.

The BlueROV2 is a small and low-cost remotely-operated-vehicle (ROV) shown in Figure 1.

A DRL algorithm requires many training samples for the algorithm to converge. Therefore, simulation-based training provided the foundation for the algorithm. The Gazebo simulator with the UUV_SIMULATION [5] plugin and a dynamic model of the BlueROV2 [6] provided a simulation environment for the training. Real-life experiments conducted in Marine Cybernetics Laboratory (MCLab), in Trondheim, provided the validation for the training. The dual controller design splits the fully actuated BlueROV2 by using a DDPG algorithm to control the (NED) position in surge x and sway y , and a PD algorithm to control the (NED) position in heave z and the (NED) orientations in pitch ϕ , roll θ and yaw ψ .

III. RESULTS AND DISCUSSION

A. Training results

The training utilizes a simulated environment, which is safer and can be completed at a faster rate than in a real-life environment. Two parameters are measured; the total reward and number of steps. The total reward is a vector containing the average reward of the last 100 episodes, and each episode consists of a number of steps. A step denotes one computer step, and an episode denotes one training interval, which ends when the number of steps reaches 1000 or the agent reach the terminal state. The terminal state indicates that the agent has sufficiently done station keeping.

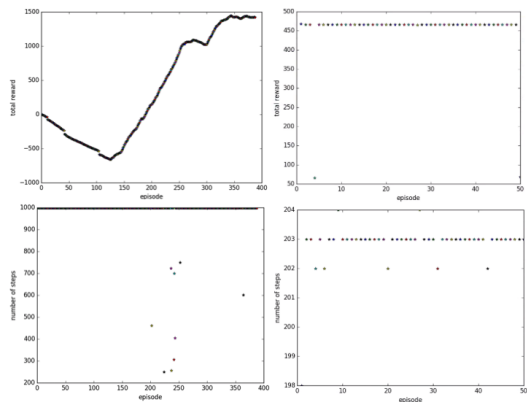


Fig. 2: Training results: left figures: $0 < \text{episode} < 400$, right figures: $550 < \text{episode} < 600$.

The agent trained for approximately 600 episodes. The left figures in Figure 2 shows the result from the first 400 episodes. ROS and Gazebo experienced difficulties when the number of episodes increased. Therefore, the training simulator restarted after 400 episodes. The total reward converges towards 450 as the agent reaches approximately 600 episodes. The right figures in Figure 2 shows the results where the last 50 episodes remain approximately 450 in the returned reward. From the bottom right Figure, the agent uses approximately 203 steps in each episode, meaning that it has sufficiently satisfied the station keeping criteria in each episode.

B. Validation results

The validation of the training involves both simulation and real-life testing on the actual vehicle.

1) *Simulation*: Figure 3 visualizes the body-fixed error for each DOF. The agent received an initial desired pose of $[x_d, y_d, z_d, \phi_d, \theta_d, \psi_d] = [2, 0, 0, 0, 0, 0]$, and at $t = 350$ a new desired pose was given as $[x_d, y_d, z_d, \phi_d, \theta_d, \psi_d] = [2, 2, 0, 0, 0, 0]$. The desired pose change was done to evaluate how the agent responds to change in pose, as well as to validate that the solution was universal.

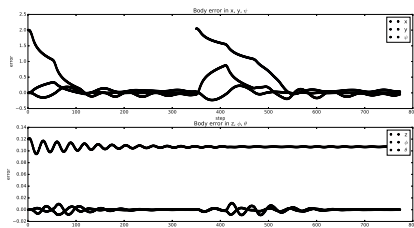


Fig. 3: Simulated validation results showing convergence of the ROV to the desired states.

As shown in Figure 3, the agent has sufficiently accomplished station keeping capabilities, with error values in the order of $10^{-2}m$.

2) *Real-life experiments*: The real-life experiments were conducted on the actual BlueROV2 in the MCLab at NTNU,

Trondheim. Here, the initial error state was defined as $[x_e, y_e] = [0.3, 0.3]$ and at $t = 18$ seconds a new error state was given to x such that $[x_e^{t=18}, y_e^{t=18}] = [-0.5, 0]$.

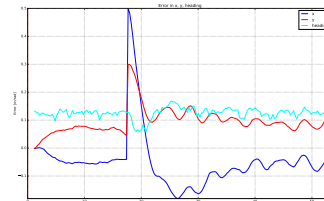


Fig. 4: Real-life validation results, showing the error states for $[x, y, \psi]$.

From the results in Figure 4, the agent is sufficient in making sure that $[x_e, y_e] \rightarrow [0, 0]$ when receiving a new error state, but the performance is slightly worse than simulation. The reason for the discrepancy between the simulated and real-life trials are two-fold. Firstly, the dynamic model used in training does not exactly match the actual vehicle. Secondly, the flaws related to the real-life pose measurement equipment. Unfortunately, these flaws made it very difficult to test the ability of the DDPG based controller design to its full extent.

IV. CONCLUSION

This paper applied a deep reinforcement learning algorithm to the station keeping problem for underwater vehicles. The results revealed that the DDPG algorithm was capable of conducting station-keeping for an AUV. Both simulation and real-life experiments validated the presented results. In the simulated environment station keeping capabilities achieved an error in the order of $10^{-2}m$, and likewise $10^{-1}m$ in real-life. The discrepancy between the real and simulated case is two-fold. Firstly, the dynamics of the simulated vehicle did not match the actual vehicle exactly. Secondly, the pose measurement system in the laboratory often lost tracking, which was detrimental to the results. Further work on the topic should try to resolve this, as well as including heading control, ψ , into the DRL controller design.

REFERENCES

- [1] J. S. Riedel, and A. J. Healey, "Shallow Water Station Keeping of AUVs Multi-Sensor Fusion for Wave Disturbance Prediction and Compensation," Naval Postgraduate School, Center for AUV Research, vol. 1, 1998.
- [2] R. Yu, Z. Shi, C. Huang, T. Li, and Q. Ma, "Deep Reinforcement Learning Based Optimal Trajectory Tracking Control of Autonomous Underwater Vehicle," Proceedings of the 36th Chinese Control Conference, vol. 1, 2017.
- [3] D. Silver, "Deterministic policy gradient algorithms," Proceedings of the 31st International Conference on Machine Learning, vol. 1, 2014.
- [4] D. Silver, "Introduction to Reinforcement Learning," University of London | lecture notes, vol. 1, 2015.
- [5] M. Manhães, S. Scherer, M. Voss, L. Douat, and T. Rauschenbach, "UUV Simulator: A Gazebo-based package for underwater intervention and multi-robot simulation," OCEANS 2016 MTS/IEEE Monterey, vol. 1, 2016.
- [6] M. C. Nielsen, O. A. Eidsvik, M. Blanke, and I. Schjøberg, "Constrained multi-body dynamics for modular underwater robots - Theory and experiments," Ocean Engineering, 358-372, 2018.

