Learning Maritime Surface Ship by Imitation Learning

Clément LANDAIS 2002153
Master’s thesis in computer science
ÅAU Supervisors: Sepinoud Azimi, Sébastien Lafond
INSA Supervisor: Muriel Pressigout
Faculty of Science and Engineering
Double Degree INSA Rennes - Åbo Akademi University
2021

Abstract

Autonomy is being developed further and further in the maritime navigation area. The wish to create such autonomous vessels has several logistic, economic and environmental origins. Safety can be improved by applying automatic adequate responses to risk-prone situations. Efficiency in the ship's route, which implies both fewer costs and a smaller carbon footprint, can be optimized with computer-level precision. The issue is to develop sufficiently reliable solutions for this critical area.

This master's thesis aims at giving an answer to this problem by using Imitation Learning. The goal is to build an autonomous agent that performs well in surface maritime navigation. The algorithmic approach is based on three main steps. The first one consists of isolating the best human behaviours from a provided dataset. The second one is to identify, in this reduced "expert" dataset, which navigation principles and rules are followed. Finally, the agent learns to imitate these expert actions. This agent is implemented in Python and assessed in a simulated maritime environment.

Keywords: Autonomy, Surface Maritime Navigation, Imitation Learning, Inverse Reinforcement Learning, Safety and Efficiency.

Preface

First, I would like to thank my supervisors Sepinoud Azimi and Sébastien Lafond for offering me the opportunity to work on this project and for the help provided during our regular meetings. In addition, I want to express my satisfaction on how the master's thesis has been conducted. Working in a student team was very motivating, and I thank all the members for their advice and feedback on my work. Furthermore, I am grateful to the staffs of Åbo Akademi University and INSA Rennes for making this master's thesis possible. Finally, I want to thank my relatives for their emotional support.

Clément Landais
Turku, 7th June 2021

Contents

List of Abbreviations i

List of Figures iii

List of Tables v

Glossary vi

1 Introduction 1

2 Imitation Learning 3

2.1 Background . . . 3
2.1.1 Reinforcement learning . . . 3
2.1.2 Imitation learning . . . 5
2.2 Elements of reinforcement learning . . . 6
2.2.1 Agent . . . 6
2.2.2 Environment . . . 7
2.2.3 Policy . . . 8
2.2.4 Reward function . . . 9
2.2.5 Value functions . . . 9
2.2.6 Model . . . 10
2.3 Mathematical basis of reinforcement learning . . . 10
2.4 Imitation learning algorithms . . . 13
2.4.1 Behavioural cloning . . . 13
2.4.2 Direct policy learning . . . 14
2.4.3 Inverse reinforcement learning . . . 14
2.4.3.1 Non-linear inverse reinforcement learning . . . 15
2.4.3.2 Deep inverse reinforcement learning . . . 16
2.5 Deadlocks . . . 17
2.5.1 Exploration vs Exploitation . . . 17
2.5.2 Discount factor . . . 18
2.5.3 On-policy vs Off-policy . . . 18

3 **Resources and design** **19**
3.1 Dataset . . . 19
3.1.1 Aboa Mare Dataset . . . 19
3.1.2 Reduced Expert Dataset . . . 20
3.1.3 Environment Data . . . 21
3.1.4 Data Augmentation . . . 21
3.2 Simulated environment . . . 22
3.2.1 Simple Ship Sim . . . 22
3.2.2 Rotterdam map . . . 24
3.3 Modelling . . . 24
3.3.1 Navigation . . . 26
3.3.2 Reach the destination . . . 27
3.3.3 Avoid collisions . . . 28
3.3.4 Synthesis . . . 30

4 **Implementation** **32**
4.1 Libraries, modules and computing resources . . . 32
4.1.1 Simulator's libraries . . . 32
4.1.2 Agent's libraries . . . 33
4.1.3 Computation time . . . 34
4.2 Program structure . . . 35
4.2.1 get_state() method . . . 36
4.2.2 get_reward() method . . . 37
4.2.3 train() method . . . 38
4.2.4 get_transition_probability() method . . . 38
4.2.5 get_trajectories() method . . . 39
4.2.6 get_reward_function() method . . . 40

5 **Results and evaluation** **41**
5.1 Expert trajectories . . . 41
5.2 Reward function . . . 43
5.3 Navigating agent . . . 45
5.3.1 Simple situations . . . 45
5.3.2 Complete scenario . . . 46
5.4 Improvements over state-of-the-art agent . . . 47

6 **Conclusion** **49**
6.1 Future work . . . 50

Appendices 51

A Simple Ship Sim program structure 52

B IRL agent's main methods' flowcharts 53

List of Abbreviations

AI Artificial Intelligence.

AIS Automatic Identification System.

API Application Programming Interface.

ASCs Autonomous Surface Crafts.

COG Course Over Ground.

CPA Closest Point of Approach.

CPU Central Processing Unit.

CSV Comma-Separated Values.

D_{CPA} Distance to Closest Point of Approach.

DQN Deep Q-Network.

FCNNs Fully Convolutional Neural Networks.

GPIRL Gaussian Process Inverse Reinforcement Learning.

GUI Graphical User Interface.

IaaS Infrastructure as a Service.

IL Imitation Learning.

IRL Inverse Reinforcement Learning.

MDP Markov Decision Process.

ML Machine Learning.

NPB-FIRL Bayesian Nonparametric Feature Construction for Inverse Reinforcement Learning.

RL Reinforcement Learning.

SOG Speed Over Ground.

SSH Secure Shell.

STW Speed Through Water.

T_{CPA} Time to Closest Point of Approach.

UML Unified Modeling Language.

USVs Unmanned Surface Vehicles.

YAML YAML Ain't Markup Language.

List of Figures

2.1 Hussein's illustration of the Imitation Learning flowchart. [17] 6
2.2 Sutton's illustration of the agent–environment interaction in a MDP. [13] . 7
2.3 Example of three different policies, in a state without traffic, where the destination is to the right of the agent *S1*. 8

3.1 Visualisation of one sample of the dataset. [27] 20
3.2 Map of the sea off Rotterdam, with navigation channels, restricted areas and sea marks. [28] 24
3.3 Example of state space variables. 25
3.4 Comparison of vessels *S1* and *S2*, facing the same situation with different absolute coordinates, headings and destinations. Blue pentagons represent foreign vessels. 25
3.5 Discrete *rudder angle* state space. 27
3.6 Comparison of vessels *S1* and *S2*, facing different situations with the same distance to the destination (a), and facing the same situation with different distances to the destination (b). Blue pentagons represent foreign vessels. 28
3.7 Discrete *destination angle* state space. 29
3.8 An example of a crossing situation, where the salmon-coloured vessel represents the agent and the yellow vessels the traffic (a), with the corresponding D_{CPA} *trace* (b). The D_{CPA} *trace* range is represented in grey in the simulation. 31

4.1 UML diagram of the *Agent* interface and the *IRL* class, without any information about methods' parameters and return values, for readability purposes. 36

5.1 Expert trajectories' states, with the *rudder angle* in function of the *destination angle*, in number of occurrences in the dataset. 41
5.2 Expert trajectories' states, with the *rudder angle* in function of the *maximum D_{CPA} angle*, in number of occurrences in the dataset. 42
5.3 Expert trajectories' states, with the *rudder angle* in function of the current D_{CPA} value, in number of occurrences in the dataset. 43

5.4 *Reward matrix* representations, with the *rudder angle* in function of the *destination angle* (a), of the current D_{CPA} value (b), and of the *maximum D_{CPA} angle* (c). 44
5.5 Simulation of the IRL agent in simple situations, i.e., reaching the destination (a), overtaking and being overtaken (b), head-on situation (c), and crossing (d). 46
5.6 Simulation of the IRL agent in a complex scenario, where the salmon-coloured vessel is the agent and yellow ones are vessels simply navigating towards their respective destinations. The agent's goal is to navigate in the Rotterdam sea channel from west to east while avoiding other vessels. It first needs to overtake a group of ships (a, b), before crossing a flow a vessels going from south to north (c, d), and finally reaching its destination while avoiding stationary ships (e, f). Pairs of images represent the beginning and the end of each situation. 48

A.1 Penttinen's illustration of the *Simple Ship Sim* program structure. [10] . . 52

B.1 *train()* method's flowchart. 53
B.2 *get_transition_probability()* method's flowchart. 54
B.3 *get_transition_probability_slice()* method's flowchart. 55
B.4 *get_trajectories()* method's flowchart. 56
B.5 *get_reward_function()* method's flowchart. 57

List of Tables

3.1 State space variables. 30

3.2 Action space variables. 30

Glossary

Aboa Mare Is a maritime academy and training centre for professional navigators.

Automatic Identification System Is a worldwide used system that collects tracking data about maritime vessels.

COLREGs For Collision Regulations, is a set of rules adopted during the 1972 Convention on the International Regulations for preventing Collisions at Sea. Their purpose is to prevent collisions or risk-prone situations at sea.

Course Over Ground Is the direction of a vessel's motion, with respect to the ground and immobile items. It is influenced by natural forces, such as the tide, the wind and ocean currents.

MAST! Institute Is a research partnership between Aboa Mare and Åbo Akademi University, working on the modernization of maritime transports and especially autonomous navigation.

One Sea Is a consortium of companies and universities that works on autonomous maritime navigation.

RAAS For Rethinking Autonomy And Safety, is an alliance of Finnish universities that works on autonomous transports.

Simple Ship Sim Is a simulated environment created by the MAST! Institute team, whose goal is to train and test Reinforcement Learning agents in a maritime environment.

Speed Over Ground Is the speed of a vessel, with respect to the ground and immobile items. It is influenced by natural forces, such as the tide, the wind and ocean currents.

Speed Through Water Is the speed of a vessel, with respect to the water and floating items.

1. Introduction

Automation techniques are used in numerous fields as a way to improve the efficiency of systems and processes. One of these fields is transportation, where autonomy allows to enhance safety, logistics, and reduce both costs and environmental impacts [1, 2]. The development of autonomous vehicles is a current challenge, since it has recently been enabled by the apparition of new technologies, such as efficient Artificial Intelligence (AI) algorithms, complex sensing systems or solid wireless communication networks [1, 2]. The leading area in this domain is the automotive industry, as worldwide studies allow the first autonomous cars to appear on the roads. In parallel, research partnerships, such as the RAAS alliance between several Finnish universities [3] or the One Sea consortium [4], focus on the maritime navigation industry and try to build autonomous vessels. However, due to the differences in the physical behaviours of boats and cars, it is impossible to apply the advanced results of the automotive area on ships [2]. Therefore, it is necessary to build new solutions for the specific domain of maritime navigation.

For decades, maritime safety has been improved through the creation of maritime organisations, such as the International Maritime Organisation [5], and the adoption of many navigation rules. The *Convention on the International Regulations for Preventing Collisions at Sea* introduced the COLREGs rules in 1972 [6]. This set of navigation guidelines defines how to behave in collision-prone situations, in order to prevent accidents. However, these conventions are not sufficient to ensure a secured navigation, since maritime transport still suffers numerous casualties every year, mostly caused by human errors [7]. This justify the will to remove humans from the cockpit, by developing Unmanned Surface Vehicles (USVs), also called Autonomous Surface Crafts (ASCs). In this context, Machine Learning (ML) is widely used, due to the high performances it shows on such unpredictable environments [2]. The goal is, therefore, to increase the ships' level of autonomy. Many techniques are based on dividing the vessels' data and control flow into a series of steps that can be automated separately, such as the data acquisition, the analysis, the decision and the action [2]. While the last decades saw many prototypes emerge [8], more recent studies produced advanced vessels, such as the Mayflower Autonomous Ship, which is expected to cross the Atlantic Ocean in June 2021 [9].

Among all existing studies on autonomous vessels, the focus is turned on the work of Penttinen [10] because of the similarities it shows to this master's thesis. In his own mas-

ter's thesis, Penttinen implemented an autonomous agent for surface maritime navigation, using a Deep Q-Network (DQN) as the main framework [11]. The objective was to handle different collision-prone situations, by following the COLREGs rules. This agent, trained by Reinforcement Learning (RL) and using a manually designed reward function, shows promising results. However, the following limitations can be highlighted. Firstly, the autonomous vessel is controlled by the Artificial Intelligence (AI) only during risk-prone situations, while an autopilot function guides it towards the destination over the remaining time. Consequently, the agent is just partially piloted by the AI. Secondly, the agent faces a bottleneck of RL, which is the difficulty to model a well-suited reward function. By manually designing it, Penttinen introduced variances between its agent's behaviours and the goal he wanted to achieve. Thirdly, the task of navigating a ship at sea is highly complex and cannot be summed up to following the COLREGs rules. Thus, Penttinen missed a whole part of good navigation principles that are not described in conventions.

This master's thesis aims at continuing Penttinen's work, by answering the previous limitations. The objective is to develop, as a proof of concept, a safe and autonomous agent which is trained through Inverse Reinforcement Learning (IRL). This technique, which is part of the more global method of Imitation Learning (IL), is closely related to RL and uses supervised learning. To summarize, the learning process consists of showing the agent a set of good behaviour examples. Then, the agent tries to imitate these demonstrations by reproducing the good actions it has seen. As explained in the following chapters, this method, associated with a wise modelling, has shown good results both for autonomously reaching the destination and for collision avoidance. For the purpose of this master's thesis, the dataset collected from the Aboa Mare navigation simulator. In addition, the agent is modelled, trained and simulated in the same environment as used by Penttinen, which is integrated in the simulator developed by K. Hupponen [12].

The master's thesis is planned as follows. A first chapter will explain what is IL and under which form this method is used to solve the problem of maritime navigation. The history and generalities about IL and RL will be described. The different possible algorithmic approaches will be listed and the one that has been chosen, i.e. Inverse Reinforcement Learning (IRL), will be explained in detail. Moreover, the possible deadlocks and the solutions to avoid them will be presented. The second part of the master's thesis will detail the design phase. It includes explanations of the dataset processing, the simulator and the problem modelling. The third chapter deals with the implementation, which includes the used libraries and the description of the program's structure. Finally, the last part will present how the agent performs on several scenarios. These results will then be evaluated and discussed to highlight possible improvements and to explain how the produced agent can be generalised in a real-world environment.

2. Imitation Learning

Imitation Learning (IL), also known as apprenticeship learning, is a set of methods aimed at solving problems standing in the more general domain of Reinforcement Learning (RL), which is itself included in Machine Learning (ML). On the one hand, RL is described by Sutton and Barto as the most natural way to learn and answer problems, because it is based on the interaction with the environment [13]. On the other hand, IL consists of observing and imitating an expert showing demonstrations of the specific tasks that must be accomplished [14].

This chapter will explain the theoretical knowledge behind RL and IL. In addition to the mathematical basis, it will describe the main elements of RL problems and the different algorithms standing under the term of IL. Finally, recurrent deadlocks appearing in RL problems will be presented.

2.1 Background

Although RL and IL are today naturally linked, they both have different origins and emerged at different times. This section aims at providing historical knowledge about the emergence of both techniques and an explanation of the main ideas expressed behind these terms.

2.1.1 Reinforcement learning

The domain of RL as it is known today finds its origin in the late 1980s, with the reunion of three fields elaborated several decades before. These threads, described in Sutton's book [13], are called *trial-and-error*, *optimal control* and *temporal-difference*.

The oldest parent of modern RL is the concept of *trial-and-error* learning. According to Sutton, it has its origin in the second half of the 1800s, before the term is reused in 1911 by Thorndike to explain animals' behaviours and how they can learn from reward and punishment. Thorndike formalized this idea under his "Law of Effect". A more recent definition of trial-and-error learning is proposed in Bei's work, as a methodology consisting of "posing a sequence of candidate solutions" to a problem, and "observing their validity" [15]. The algorithm then decides, for each solution whether it is valid. If

not, an error is raised to inform the learned that this solution is not suited for answering the current task.

The second ancestor of RL is, according to Sutton, issued from the notion of *optimal control*. This term, which appeared in the late 1950s, is used to refer to "the problem of designing a controller to minimize or maximize a measure of a dynamical system's behaviour over time" [13]. In other words, this problem consists of modelling an entity that is able to control and optimize a dynamic system's metric. A solution to this question was proposed by Bellmann in the 1950s. He elaborated a formula known today as the Bellmann equation, which relies on the system's state and a value function, both discussed later in this chapter. Answering a problem of optimal control with the Bellmann equation is today classified under the term *dynamic programming*.

Finally, the last but less obvious parent of RL is called *temporal-difference*. Temporal-difference learning also emerged in the 1950s and relies on the comparison of successive values of the same metrics. Sutton gives the example of the probability to win a game of Tic-Tac-Toe. The successive values of this probability can be used to learn whether performed actions are efficient enough to lead to victory.

From what was previously explained, modern RL can be seen as a method which tries to optimize one or several metrics of a dynamic system's state. The main technique used to achieve good behaviours consists of trying an action and evaluating the benefit resulting from it thanks to the *reinforcer* returned by the environment. This reinforcer can be positive if the action is good in the current state of the system, or negative if it is inadequate. Generally, the reinforcer, or *reward*, is delivered through a *reward function*, presented later in this chapter.

To illustrate how an entity can learn specific behaviours through RL, Penttinen presented in his master's thesis [10] the following concrete example imagined by Harmon in 1997 [16]. Here, the learning entity is a bicycle which can either turn the handlebars to the left or to the right and can have an inclination from 0° to 45°. Over the inclination of 45°, the bike crashes as in real life and it results in giving the learner a negative reward. If now the bicycle trains by itself through trial-and-error, it reaches one day or another an inclination of 45° to the right. In this state, the learner chooses one of the two possible actions. If it decides to turn to the left, the bicycle crashes and receives a negative reward. The learner now associates the action of turning to the left in the state of 45° tilt to the right with a bad reward. The second time it reaches this state, it naturally takes the action of turning to the right. However, this also results in a crash and provides a negative reinforcement. Now the learner can associate the whole inclination of 45° to the right to a negative reward. Therefore, because this state is now linked with a negative reinforcement, the learner can understand that the action of turning to the left when being inclined

at 40° to the right is bad, because it will lead the bicycle to reach an inclination of 45°. As a result, with a sufficiently long training, the learner will associate every state-action pair with a profitability value and, consequently, will adopt a behaviour that allows it to never crash.

2.1.2 Imitation learning

IL, which has more recent origins, is issued from neuroscience. In his survey on Imitation Learning methods, Hussein defines IL as the "acquisition of skills or behaviours by observing a teacher demonstrating a given task" [17]. This definition can be reformulated and summarized as learning from demonstrations. The idea of using the knowledge of experts as a starting point introduces new techniques to answer optimization problems and brings solutions for the issues faced with classic RL. Hussein and others before him claim that copying expert demonstrations is faster and more efficient compared to establishing a solution from scratch. Furthermore, it prevents both finding sub-optimal solutions and reaching unnatural behaviours. In addition, Hussein states that the process of trial-and-error requires to manually design how rewards are delivered for every specific task that needs to be performed. However, designing a suitable reward function can be very difficult and time-consuming. Therefore, deducing it from expert demonstrations allows at the same time to speed up the development of an intelligent system and increase its efficiency and robustness. [17]

According to Hussein, the typical work flow of an IL process begins with the acquisition of expert demonstrations through various sensing systems. Those examples are then processed to extract the useful features, which depend on the tasks that must be performed. For instance, in the context of autonomous navigation, a feature could be the distance between the vessel and its destination. After this treatment, expert demonstrations are stored in *trajectories* consisting of series of state-action pairs (s_t, a), i.e., a mapping between the action a taken by the expert in the state s_t. Those trajectories are then used to learn an initial behaviour that imitates the expert demonstrations. This behaviour, also called *policy*, can later be refined by the learner through other processes, such as RL. This refining is sometimes necessary to overcome the differences between the expert's environment and the learner's one. As a result, an IL process may contain a RL part, depending on the chosen algorithm. This is especially the case when dealing with Inverse Reinforcement Learning (IRL), discussed later in this chapter. The overall process of IL presented above is illustrated by the flowchart in Figure 2.1. [17]

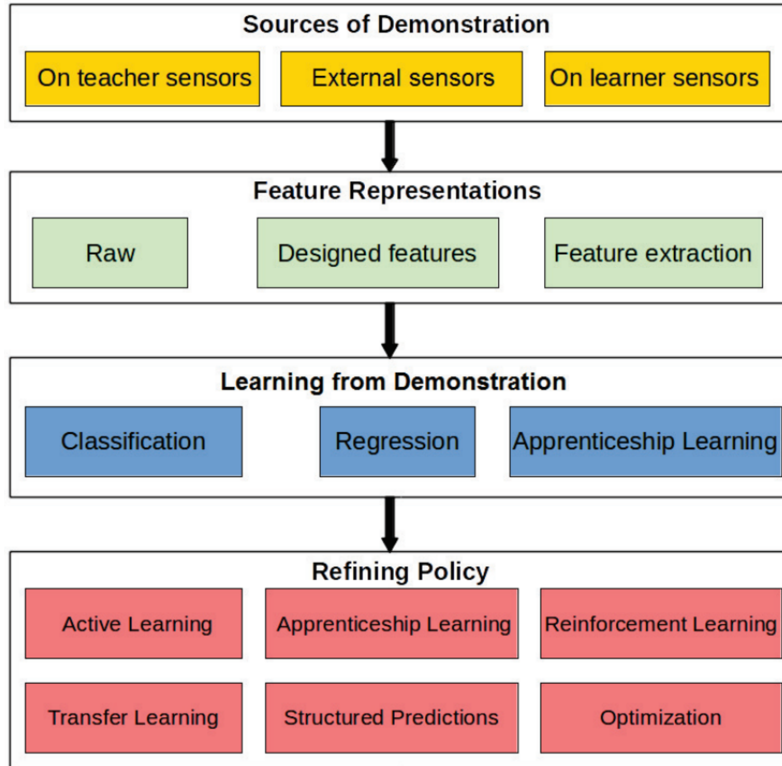


Figure 2.1: Hussein's illustration of the Imitation Learning flowchart. [17]

2.2 Elements of reinforcement learning

According to the previous explanations, IL and RL both aim at answering optimization problems. As a result, they share the same root elements. These elements are named the *agent*, the *environment*, the *reward function*, the *value functions*, the *policy* and the *model*. Their descriptions, presented in this section, are inspired by the works of Sutton [13] and Hussein [17].

2.2.1 Agent

The *agent* is one of the two basic elements of RL or IL. It is the autonomous entity controlled by the AI and aims at optimizing or achieving an objective. It is able to interact with its *environment*, i.e., to acquire information of its surroundings from sensors, and to perform, through actuators, actions which have repercussions on the environment. An important characteristic of an agent is its ability to learn, by itself from previous experiences or from demonstrations. This training alters the future agent's behaviours and allows it to take the decisions it judges the best in the states it encounters. As a result of the training process, the agent is able to identify the state-action pairs from which it can benefit the most.

When modelling an agent, defining its state and action spaces is required. The *state space* S of an agent consists of all the possible states it can reach. Similarly, the *action space* A is the set of all possible actions it can perform. Defining those spaces is a critical part when modelling a problem for the two following reasons. Firstly, too poor state and action spaces with a reduced number of variables can lead to missing substantial data and, therefore, making the agent unable to answer the task it is designed for. Secondly, if those spaces are oversized, the agent will struggle to identify the relevant features and the training time will be severely increased. Generally, the modeller will try to use non-redundant variables providing useful data for the specific problem he wants to answer.

In the context of this master's thesis, an agent can be illustrated as an autonomous vessel, which is able, on one hand, to sense its environment and, on the other hand, to perform actions such as rotating the rudder or accelerating and decelerating the speed of the propeller. This vessel will be controlled by a RL *policy*, discussed below, and its goal will be to reach its destination and avoid collisions with land or other boats on the sea.

2.2.2 Environment

The *environment* is the second basic element of RL and IL. It includes all the entities and elements the agent interacts with. As illustrated in Figure 2.2, it responds to the agent's action A_t by providing a new situation and a feedback. The new state S_{t+1} results from the last action and the intrinsic behaviour of the environment. The feedback R_{t+1} is the reward evaluating the previous agent's action and consists of a numerical value. Both S_{t+1} and R_{t+1} allows the agent to learn and to decide its next action A_{t+1}.

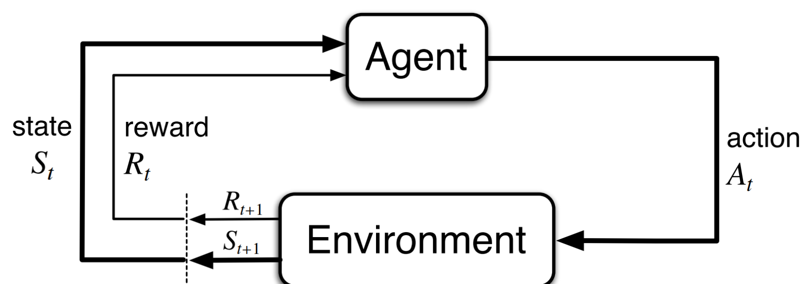


Figure 2.2: Sutton's illustration of the agent–environment interaction in a MDP. [13]

For the problem of autonomous navigation, the environment can be represented by the sea with the traffic evolving around the agent, the coast and reefs, the navigation channels, the sea signs, as well as meteorological and natural forces such as wind or ocean currents.

2.2.3 Policy

The *policy* is a function that guides the agent behaviour in its environment. It maps the current state of the agent to the action to be taken. It may be simple (e.g. basic function or lookup table) or more complex (e.g. search process through a neural network). Generally, the policy is stochastic and deals with probabilities, i.e., the probability to take any action a in any state s, as shown in Equation (2.1) [18]. Because the policy is learned through RL, these probabilities are continuously altered by the feedback given during the trial-and-error process. The goal of the training is to reach an optimal policy which best answers the problem.

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{2.1}$$

Moreover, the policy can be *stationary* or *non-stationary*. A stationary policy does not include the temporal parameter t when determining the next action. Conversely, a non-stationary policy considers the current stage of the task. Therefore, stationary policies are suited for long or infinite scenarios, while non-stationary ones are adapted for temporary tasks. However, the last category suffers difficulties when facing unseen scenarios.

To give an example of policies, the situation illustrated in Figure 2.3 is considered. There, the destination is to the right of the agent named *S1*. Furthermore, the agent's action space includes three different actions : turning to the right, turning to the left, or going straight. In this configuration, if the goal of the agent is to reach its destination, the optimal policy would choose to turn to the right (policy number one). Conversely, bad policies would choose to either go straightforward or turn to the left (policies number two and three).

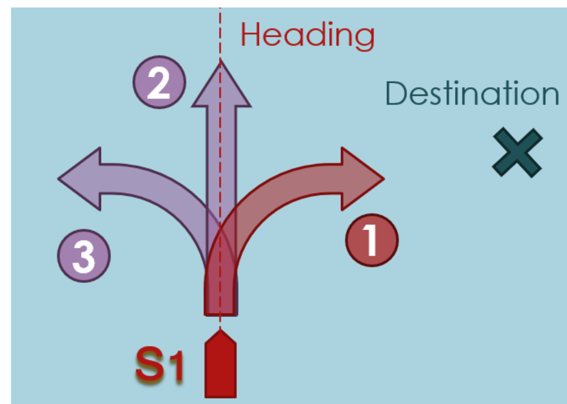


Figure 2.3: Example of three different policies, in a state without traffic, where the destination is to the right of the agent *S1*.

2.2.4 Reward function

The *reward function* maps every state or state-action pair with a *reward*. As explained above when presenting the environment, the reward function sends this numerical feedback after every agent's action. A high reward is synonym to good behaviours, while a low reward is sent in unprofitable states or after poor actions. In other words, the reward defines how desirable an action is in the short term and allows the agent to distinguish the immediate preferred action in each state. The reward of taking action a in state s is defined in Equation (2.2) [18].

$$R_s^a = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] \tag{2.2}$$

Often, the reward is calculated with a mathematical or logical operation over the state variables. It can be either manually designed when using classic RL or computed with IL methods. The reward function is only used during the training process and serves to converge on the most optimized policy. After, the reward function has no more influence on the agent's actions, since the agent's behaviour is guided by the learned policy.

A simple example of a reward function is given in the following Equation (2.3). The reward R_t is equal to the negative distance between the agent and its destination. Thus, because the main goal of the agent is to maximize the total reward it receives over the whole scenario, it would try to come closer to its destination.

$$R_t = -\ distance\ to\ the\ destination \tag{2.3}$$

During this master's thesis, the term *reward matrix* will be used to refer to the matrix which associates a quantity of reward to every possible state. This matrix will be the result of the IRL algorithm, discussed later in this chapter.

2.2.5 Value functions

The *state-value function* and *action-value function* are similar to the reward function, with the exception that they calculate the desirability of state or an action in the long run. They return, for every state or every action, a quantity called a *value*. The value of a state $v_\pi(s)$ is the maximum amount of reward that can be collected in the future starting from that state, when following the policy π. In the same way, the value of an action $q_\pi(s,a)$, also called its Q-value, is the cumulative amount of reward that can be obtained after taking this action. Thus, it enables the agent to distinguish which immediate action

allows to accumulate the maximum total reward in the future. The agent's decisions are often based on the value rather than the reward, because most solutions to optimization problems aim for long-term efficiency.

To illustrate the difference between reward and value, Sutton provides a human analogy in his book. On the one hand, the reward is similar to the direct pleasure or pain that can be felt immediately. On the other hand, the value is like the deeper and more complex feeling of satisfaction. For instance, an athlete running a marathon who suffers during the race can still be satisfied with his performance or motivation.

2.2.6 Model

The *model*, used in some RL systems, is a simulation of the environment that is used to predict the possible future states before they are experienced. Therefore, by inferring the behaviour of the environment, it allows the agent to know what is likely to happened in the different available paths. When the model is used to decide the next agent's action, the decision process is called planning. This method is opposed to the classic trial-and-error learning.

2.3 Mathematical basis of reinforcement learning

This section aims at explaining the mathematical basis of RL and all formulas within the theory. The following explanations are inspired from Silver's work [18] and Sutton's book [13]. The first step when formulating a RL problem is to design a *Markov Decision Process (MDP)*. This last, introduced by Bellman in 1957, can be seen as the discrete and stochastic representation of the optimal control problem. It models the agent's environment and can be formulated with the tuple in Equation (2.4) [18]. While S and A are respectively finite sets of states and actions, P is the state transition matrix of the environment, R the reward function and γ the discount factor. These elements are described in more detail below. In IL, MDPs are widely used because they ease the representation of expert demonstrations.

$$< S, A, P, R, \gamma > \tag{2.4}$$

As indicated by its name, a MDP respects the *Markov* property, that asserts the past trajectory of the agent, except its current state, has no influence on its future positions. In other words, the current state holds all the necessary information the agent needs, to choose an action and progress on its task. This statement is described in Equation (2.5)

[18], where the probability to go from state S_t to S_{t+1} is not influenced by the states preceding S_t. This implies that all policies, in a MDP, are independent of the time parameter t and, therefore, stationary.

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, ..., S_t] \tag{2.5}$$

The environment of a MDP is, in addition, governed by a three-dimensional *state transition probability matrix* P, which holds the probability to go from state s to state s' under action a, for every state and every action. The mathematical representation of this matrix is given in Equation (2.6) [18].

$$P^a_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a] \tag{2.6}$$

Now the MDP is defined, the objective is to find the path that will return the maximum amount of reward. In order to make the agent focus on a long term efficiency, the *discounted rewards* are introduced. Discounting is the agent's ability to evaluate the importance of immediate rewards against the more distant ones in the future. The *discount rate* γ is a value between 0 and 1 which is used to calculate the *total discounted reward* G_t, as shown in Equation (2.7) [13], with R_{t+k} being the reward obtained in the kth state in the future. Therefore, a smaller γ means a low consideration of distant rewards, while a higher γ means a high consideration.

$$G_t = R_{t+1} + \gamma.R_{t+2} + \gamma^2.R_{t+3} + \ldots = \sum_{k=0}^{+\infty} \gamma^k.R_{t+k+1} \tag{2.7}$$

From the total discounted reward, it is possible to mathematically define the value functions of the MDP in Equations (2.8) and (2.9) [18]. The state-value function $v_\pi(s)$ returns the cumulative discounted rewards the agent is expected to obtain in the future, when starting from state s and following the policy π. The action-value function $q_\pi(s,a)$ returns the expected total discounted reward when taking action a in state s. Both these values depends on the chosen policy and the discount factor.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.8}$$

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.9}$$

Another representation of these formulas is know as the Bellman equation. Its first form, shown in Equation (2.10) [18], links the value of a state to the values all possible actions that can be perform in this state. The state-value $v_\pi(s)$ is then equal to the sum, over all possible actions, of the probability to take action a under policy π, multiplied by the value of this action. The second form of the Bellman equation, shown in Equation (2.11) [18], draws the relationship between the value of an action and the value of future possible states. The action-value $q_\pi(s,a)$ can now be computed by adding the immediate reward to the discounted values of potential future states, weighted by the probability of ending up in these states. By developing these equations, Bellman obtained the formulas appearing in Equations (2.12) and (2.13) [18]. On one hand, the value of a state is now associated to the values of possible subsequent states and, on the other hand, the value of an action is connected to the values of possible future actions.

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s,a) \tag{2.10}$$

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \tag{2.11}$$

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) [R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')] \tag{2.12}$$

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s',a') \tag{2.13}$$

The Bellman equation is widely used in RL since it allows to divide the whole problem into smaller ones, where the goal is to compute the optimal value for every state or every action. This is done by iteratively adjusting these functions during the trial-and-error process, so they better represent the real reward structure of the environment. Solving a MDP means therefore finding the optimal state-value or action-value functions, presented in Equations (2.14) and (2.15) [18]. The latter return, for every state or every state-action pair, the value given by the best existing policy in this state.

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.14}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.15}$$

Once the MDP is solved, the optimal policy can be computed. This policy respects the condition stated in Equation (2.16) [18], when compared with all other policies. In short, it is the policy that returns the highest value in all states. An simple method to find the optimal policy is, in every state, to set at one the probability of taking the action that

returns the maximum optimal value. This procedure is described in Equation (2.17) [18]. Once the optimal policy is determined, the whole RL process is finished.

$$\pi \geq \pi' \quad if \quad v_\pi(s) \geq v_{\pi'}(s),\ \forall\ s \tag{2.16}$$

$$\pi_*(a,s) = \begin{cases} 1 & if \quad a = \underset{a \in A}{\operatorname{argmax}}\ q_*(s,a) \\ 0 & otherwise \end{cases} \tag{2.17}$$

2.4 Imitation learning algorithms

IL differentiates from classic RL in the sense that the agent is learned from previously collected data. The goal is to achieve an efficient policy given by one or several human experts. To reach this objective, three main approaches exist and are summarized in Lőrincz's work [19]. These techniques, named *Behavioural Cloning*, *Direct Policy Learning* and *Inverse Reinforcement Learning*, are presented below.

2.4.1 Behavioural cloning

Behavioural Cloning is the simplest method and consists of learning the expert policy through supervised learning on the expert dataset. Expert trajectories are first stored into state-action pairs. These pairs are then directly used to learn the agent's policy. For example, if the action a is recurrently performed by the expert in the state s, the agent will map this action with this state, i.e., learn that this action is good and should be taken in state s.

Despite its apparent simplicity and the positive results it shows in small applications, this technique suffers strong issues when it comes to resolve large and highly complex problems. The origin of these issues is the assumption made in supervised learning presuming that the state-action pairs are independent and uniformly distributed over the state and action spaces. However, this is not true since every action from the expert trajectories has an influence on the next state. As a result, many possible states, and especially critical ones, will never be visited by the expert. Therefore, the agent lacks examples of how to recover from problematic states if it comes to deviate from expert trajectories. A second issue, which is correlated to the previous one, is the difficulty to obtain a sufficient amount of data for applications owning a large state space. The larger the state space is, the bigger the number of examples needs to be. Thus, an efficient training with behavioural cloning would be highly data- and time-consuming.

In addition, Hussein states in his survey that an efficient training with IL requires the

agent to re-optimize, by itself, the learned expert policy through a classic RL process. However, this is not the case with behavioural cloning. Hussein explains that insufficient or erroneous expert demonstrations and light variations between the agent's and expert's environments can lead the optimal expert policy to not work for answering the agent's task. Therefore, by re-optimizing the policy, the agent learns to perform the task, which is more robust than strictly copying the expert. [17]

To conclude, due to the complexity of the maritime navigation area and the differences between the provided expert and the agent's reduced model, both presented in Chapter 3, the method of behavioural cloning has not been retained to solve the problem expressed in this master's thesis.

2.4.2 Direct policy learning

Direct Policy Learning is an approach starting in the same way as behavioural cloning. First, some expert demonstrations are collected to learn an initial policy. Then, the algorithm aims at converging this policy towards an expert-equivalent one, through an iterative loop executed in three steps. Firstly, new trajectories are collected by letting the agent evolve in its environment while following the previously learned policy. Secondly, for every encountered state, an expert gives to the agent the action it should have taken. Finally, based on these new data, a new policy is learned through supervised learning. Two techniques exist to learn the new policy: *data aggregation* and *policy aggregation*. Data aggregation learns the new policy on all the previously collected demonstrations. In this way, it remembers past iterations. Policy aggregation learns a new policy only on the trajectories obtained in the current loop iteration. This policy is then merged with previously learned policies through geometric blending.

Direct policy learning shows good results both on small and large-scale problems. Nevertheless, it requires an available expert during the whole training. This requirement is not compatible with the resources and data provided for this master's thesis. As a consequence, it is impossible to use this method in that context.

2.4.3 Inverse reinforcement learning

Inverse Reinforcement Learning is the last approach of IL and consists of learning the reward function that corresponds to the expert behaviours. The goal is to extract the underlying motivations guiding the expert's decisions. These motivations can be computationally illustrated as a reward function. After obtaining this function, learning the optimal policy through RL becomes possible. The general process of IRL can be summarized as follows. It is first initialized by collecting expert trajectories and learning an

original reward function based on these demonstrations. Then follows an iterative loop which is made of three steps. Firstly, a policy is learned on the previously calculated reward function. Secondly, this policy is compared to the expert's one, to be evaluated. Thirdly, if the policy is satisfactory, the process is terminated. On the contrary, if it is not, the reward function is updated to better represent the expert's motivations.

IRL is divided into two main methods : *model-given* and *model-free*. The first one is suited for simple problems with small state spaces and linear reward functions. It defers from the other one by solving the full RL problem when learning a new policy. To use it, knowing the state transition probability matrix P, i.e. the probability to go from every state s_x to every state s_y under every action a_z, is necessary. The second approach is made for more complex problems running on simulators and does not need to know the state transition probabilities in advance. Because of the large state space, only one step of the RL problem is solved at each iteration of the loop.

As explained in the two previous subsections, the approaches of behavioural cloning and direct policy learning are not suited to answer the problem expressed in this master's thesis. However, the method of IRL is both adapted to the problem's high complexity and to the available resources. This justifies the choice of using IRL. Therefore, a deeper presentation of the state-of-the-art techniques is required and presented below.

2.4.3.1 Non-linear inverse reinforcement learning

Among all studies on IRL, the focus is put on four of them. The first one, proposed by Ziebart et al., is a framework able to approximate linear reward functions [20]. This framework relies on the maximum entropy paradigm. This principle states that the probability distribution, which best represents a stochastic system, is the one with the maximum entropy, or in other words, the one which makes the fewer assumptions. Ziebart uses this paradigm over the distribution of possible paths to not end up with a sub-optimal reward function, where the distribution prefers one path to others. However, a strong weakness of Ziebart's work, and others before him, is that the reward function is represented as a linear combination of state features. According to Wulfmeier, linear functions present inherent limitations because they can only solve linear MDP [21]. Consequently, Ziebart's algorithm is strongly inadequate to resolve complex real-world problems, such as autonomous navigation. Choi also raises another issue of linear models, which is their strong dependency on the manually defined features [22]. As a result of these issues, many later studies focused on introducing non-linearities in their approximated reward functions.

Levine et al. introduced in their algorithm *Gaussian Process Inverse Reinforcement Learning (GPIRL)* the use of non-parametric functions, such as Gaussian Processes [23]. This technique enables to represent the reward as a non-linear combination of features.

The principle of this framework relies on a method which computes the hyper-parameters of the Gaussian process's kernel function. As a consequence, the structure of the unknown reward function is determined. However, a limitation of this algorithm, raised by Wulfmeier, is its high computational complexity. Another method, called *Bayesian Nonparametric Feature Construction for Inverse Reinforcement Learning (NPB-FIRL)*, is presented by Choi and Kim, and relies on associating atomic state features through logical conjunctions to create composite features [22]. Consequently, non-linearities in the logical conjunctions are transferred to the reward function. However, according to Wulfmeier, non-linearities are limited in this case to the set of composite features manually designed [21].

2.4.3.2 Deep inverse reinforcement learning

To enhance results shown by previous works in the field of IRL, Wulfmeier et al. present and evaluate a new framework in their article "Maximum Entropy Deep Inverse Reinforcement Learning" [21]. This framework introduces the use of deep neural networks, and more precisely Fully Convolutional Neural Networks (FCNNs), to approximate complex non-linear reward functions. According to Wulfmeier, deep structures are suited for IRL for the two following reasons. First, assuming the network is large and deep enough, it can represent any possible binary or piecewise-linear function. Second, it naturally relies on and generalises the maximum entropy paradigm explained in Ziebart's work [20]. The reward function is, therefore, approximated through an iterative process. Every step first resolves the MDP with the reward function calculated during the previous iteration, then computes the maximum entropy loss and gradients and, finally, applies back-propagation on the network's gradients.

Wulfmeier argues that his new technique answers the defects of the previous methods. Using a deep structure allows to handle non-linearities well and to show a good computational complexity. In addition, the neural network architecture allows to automatically model the environment and to generalise complex state spaces by extracting atomic features. Thus, it answers another issue of prior studies, which was the need to manually model the relevant features. To assess the accuracy and efficiency of their new framework, Wulfmeier et al. perform tests on two benchmarks against the previously cited techniques. The evaluation metric is the expected value difference, i.e., the difference between the optimal value functions obtained on both the true and the approximated reward functions. On the first benchmark, where the true reward structure remains simple, Ziebart's Maximum Entropy framework is not able to extract non-linearities while all three Deep IRL, GPIRL and NPB-FIRL show accurate predictions. However, Deep IRL is less robust to noise and requires more expert data to reach equivalent results. On

the second benchmark, which has a more complex true reward structure, all techniques except Deep IRL fail to approximate an accurate reward function. Moreover, a last test is performed to evaluate the ability of the FCNNs to extract features from raw input. This shows that, with enough training data, the results match the ones obtained with optimal manually designed features.

To conclude, this framework provides improvements for complex problems with large state spaces. Furthermore, it allows to get rid of preprocessing by automatically extracting optimal features. Finally, it is well suited for continuous learning scenarios, because its algorithmic complexity is independent of the size of the dataset. In the context of this master's thesis, training an agent with this framework is a wise choice due to the high dimensionality of the involved state space and the complex task of driving USVs. In addition, introducing non-linearities in the reward function is essential, since navigating cannot be reduced to maximizing or minimizing individual variables.

2.5 Deadlocks

When dealing with IRL, and therefore with RL, it is likely to face numerous deadlocks while modelling or implementing an agent. This section aims at describing possible deadlocks and explaining how they can be answered. The following descriptions come from Sutton and Barto's book [13].

2.5.1 Exploration vs Exploitation

The trade-off between *exploration* and *exploitation* is known to be a strong issue when it comes to RL. The agent, whose goal is to obtain the maximum possible reward, must make a compromise, during the training process, between exploiting what it has previously learned and exploring new states. In other words, it can either reproduce actions it has previously tried and categorized as good ones or explore new state-action pairs to discover more reward-productive decisions and exploit them in the future. Consequently, an agent focusing on reproducing the best actions it has learned could end up being stuck in a sub-optimal policy, while it will never reach the absolute-optimal one.

A common method to answer this deadlock consists of introducing randomness when choosing the action. This can be performed by using different exploration strategies. One of the best-known is called the ε-greedy policy, with ε being a parameter between 0 and 1. With this strategy, the agent tries a random action with the probability ε. The rest of the time, it chooses the action with the highest Q-value in its current state. Thus, with sufficiently long training, all possible states would be tested and evaluated. However, a weakness of the ε-greedy approach is that all actions have the same probability to be

chosen randomly. As a result, the agent does not distinguish good actions from bad ones, except for the best one. A solution of this issue is given by the Boltzmann (or softmax) exploration strategy. This approach introduces the Boltzmann softmax Equation (2.18) [24], which computes for every action, in a given state, the probability to take this action. This probability relies on the Q-value of the targeted action a in the current state s_t, as well as the Q-value of the m other actions a_i. Thus, the best actions are more likely to be chosen. In addition, the positive parameter T allows to gradually transit during the training from a fully random mode ($T \to +\inf$) to a mode where the best action is always chosen ($T = 0$). [24]

$$\pi(s_t, a) = \frac{e^{Q_t(s_t,a)/T}}{\sum_{i=0}^{m} e^{Q_t(s_t,a_i)/T}} \tag{2.18}$$

2.5.2 Discount factor

As presented in Section 2.3, the *discount factor* allows to distinguish immediate rewards against more distant ones in the future. The issue here is to decide which γ value is best suited for the problem, since it has a huge impact on the agent behaviours. A simple method to answer it is to train the agent with different γ values and evaluate which one shows the best performances.

In the case of this master's thesis, two different discount factors must be chosen. The first one concerns the discount value used in the IRL algorithm to compute the reward function from the expert trajectories. It is possible to estimate this value to be high, since driving a vessel is a task that requires anticipation. The second discount value concerns the RL agent learning by itself in its environment. Since the task is similar to the one of the expert, anticipation and therefore a high discount value is needed.

2.5.3 On-policy vs Off-policy

A distinction must first be made between *behaviour* policy and *estimation* policy. The behaviour policy is the one used to decide which actions are taken by the agent. The estimation policy is the one that is improved through the learning process.

When improving a policy, two approaches can be used. The *on-policy* approach consists of using the same policy as behaviour and estimation policies. The *off-policy* approach consists of having two different behaviour and estimation policies. This allows, on the one hand, to use a very versatile behaviour policy to explore all possible states and, on the other hand, to improve another policy. The "exploration vs exploitation" deadlock can then be avoided. For this master's thesis, the *on-policy* approach will be used with the Boltzmann softmax exploration strategy.

3. Resources and design

Designing a solution to a problem is a process comprising a series of steps. From clearly defining the problem to improving the results, it requires both collecting the data on which the solution will be based and modelling the system that will solve the problem. [25]

This chapter will present how the author designed a solution to the problem expressed in this master's thesis. It will first describe the form of the raw dataset and will provide details on how these data have been processed. In addition, the simulator and the environment where the agent is developed will be presented. This agent will be named the *IRL agent*, since it is trained with IRL. Finally, an explanation of how the autonomous navigation problem has been modelled will be provided.

3.1 Dataset

The first step when resolving most ML problems is to process the learning data. This is especially the case in this master's thesis, because IRL relies on showing expert demonstrations to the agent. Processing the raw data means transforming and adapting them to be used in a specific algorithm. In this master's thesis, the dataset will first be reduced to answer the requirements of IL, before being duplicated through data augmentation.

3.1.1 Aboa Mare Dataset

Aboa Mare is a maritime academy that trains seafarers in a professional context. It offers both theoretical and practical courses and provides the possibility to learn safely in realistic simulators. In addition, Aboa Mare is involved in numerous research partnerships on maritime navigation, due to the large possibility of developments and tests in its simulated environments. Among these partnerships is the MAST! Institute, which is an alliance with the Software Technology Research Lab of Åbo Akademi University. [26]

Based on this joint project, Aboa Mare provides the dataset that is used in this master's thesis. It consists of 135 simulated navigation scenarios performed by students during their training. These simulations take place in an environment representing the south part of the North Sea, off Rotterdam. For equality between students, depicted in red in Figure 3.1, they all start approximatively in the same configurations. Their goal is to navigate from west to east in the navigation channel. In addition, they need to avoid collisions. The

two main situations they encounter are overtaking vessels and crossing a flow of ships that crosses the maritime channel from south to north. The results of these simulations have been discretised with steps of one second and stored in CSV files.

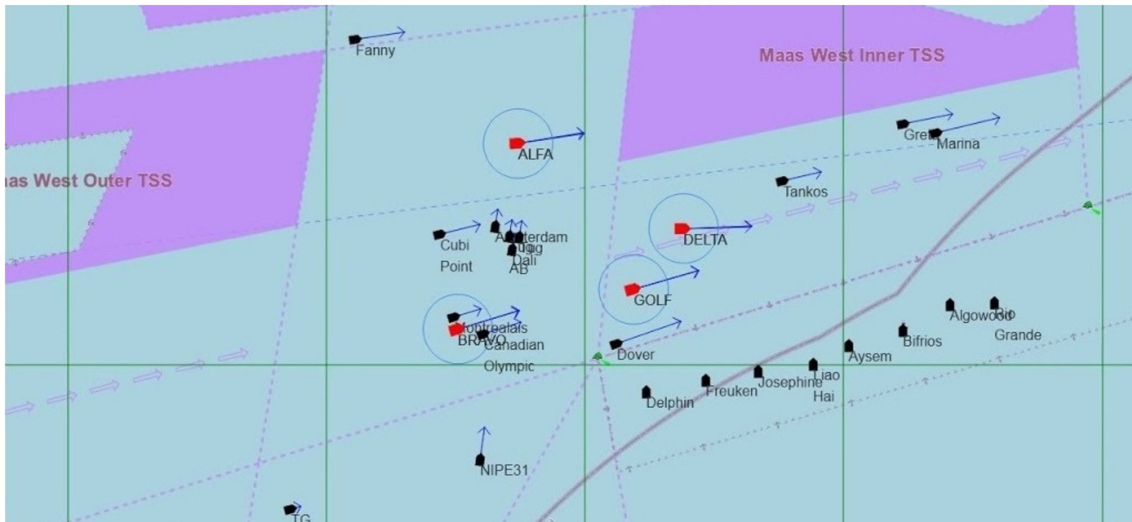


Figure 3.1: Visualisation of one sample of the dataset. [27]

Variables are divided into three types of files, called *ShipDynamics*, *Traffic* and *Log*. The focus is put on the two first types, which store information about the student's vessel and the traffic, i.e., other ships on the water. This allows to quickly eliminate redundant and unnecessary parameters in *Log* files, and retain the four following types of data for every scenario. First, information about the vessel driven by the student, such as its position in latitude and longitude, Course Over Ground (COG), Speed Over Ground (SOG) and Automatic Identification System (AIS) data. Second, the actions taken by the operator, namely turning the rudder, accelerating or decelerating. Third, information about the traffic, including the position and direction of other vessels on the water. Finally, two grades given by teachers evaluating the quality of the navigation over the whole scenario, on a scale from zero to ten.

3.1.2 Reduced Expert Dataset

The first characteristic that can be observed on the dataset is the heterogeneous grades given to all scenarios. While some of them have perfect grades of ten, others have medium or poor grades. As explained in Chapter 2, the algorithmic approach of IL needs to train the agent only with the best possible demonstrations. Consequently, the dataset must be sorted in order to eliminate all scenarios that have a grade under a lower bound limit. However, since the dataset is quite small, it is important to consider the size of the learning data, which has an impact on the agent accuracy and robustness. Thus, it is impossible to

eliminate too many scenarios.

This analysis is performed as follows. The average of the two grades given to each scenario is considered as its final grade and the lower bound limit is fixed at seven. This allows to keep only relatively good grades and retain 68 scenarios out of the 135 original ones, i.e., 50% of the dataset.

3.1.3 Environment Data

In addition to information about the student's vessel, environment data need to be collected in order to perform an effective training of the agent. Environment data include all information not directly related to the agent, such as the state of other vessels on the water, the position of the coast and reefs, the navigation channels or the sea signs.

Some of this information is provided in the Aboa Mare dataset, such as the data concerning the traffic. In addition, the coast and reefs are handled by the simulator that will be used to model and test the agent. However, some elements, such as the navigation channels and sea signs, which are important for safe navigation, are not yet implemented into the simulator. It is, therefore, necessary to search for resources about the sea infrastructures of Rotterdam's waters and to extract from them the suitable data. To accomplish this task, the tool OpenSeaMap is used [28]. This web application shows existing maritime signage all over the world. Thus, it is possible to collect Rotterdam's maritime infrastructures from it.

3.1.4 Data Augmentation

Data augmentation is a set of techniques whose purpose is to increase the amount of available data to train an agent or a neural network. This is used when the amount of original data is too low compared to the complexity of the problem to be solved. Instead of performing a repetitive training on the exact same examples and, therefore, be subject to overfitting, it is possible to apply small transformations on the original data to create similar ones which are different. A majority of existing works on data augmentation deal with image data, due to the facility to modify an image without removing the main information it contains. For instance, the size or the orientation of an image can be modified by cropping, flipping or rotating it. Its colours can also be changed with convolutions or filters. [29]

Due to the reduced size of the dataset involved in this master's thesis, in contrast with the complexity of autonomous navigation, data augmentation techniques are considered. However, since the dataset consists of CSV files containing various variables, it is impossible to apply directly the usual transformations used on image data. Therefore, two

concepts are investigated here. On the one hand, it could be possible to multiply the data by rotating or flipping both the environment and the student's vessel trajectory. On the other hand, scenarios can re-sampled by increasing or decreasing the time step between samples.

As explained later in Section 3.3, the agent's state space is modelled with relative variables, such as the current rudder angle or the angle between the vessel's heading and the straight line between the agent and its destination. Therefore, by rotating the environment, these variables remain the same. It is, therefore, impossible to multiply the data with rotational transformations. However, flipping the environment allows to reverse the value of every variable. For instance, in a state where the rudder angle is 15° and the angle towards the destination is -10°, the reversed situation would show a rudder angle of -15° and an angle towards the destination of 10°. This transformation allows to multiply the data by a factor of 2. Nevertheless, using this transformation can lead to adding new expert behaviours, since international navigation rules differentiate these two situations. According to COLREGs rules [6], different actions are theoretically required when crossing the path of a boat by its right or by its left. However, when observing expert demonstrations [27], it is apparent that students are not strictly following COLREGs rules and find sometimes paths that are more beneficial. As a result, using flipping transformations makes sense, since the developed agent will try to find the safest paths and not rigorously follow navigation rules.

The second way to perform data augmentation is to re-sample the data, such as suppressing one sample of data every n samples. Despite the possibility to obtain numerous additional trajectories, this may lead to losing the uniformity of the timestamps among the dataset or may break the physic of vessels, if the same time step is conserved for the whole re-sampled data. Therefore, this approach has not been retained.

3.2 Simulated environment

The agent designed in this master's thesis is trained and evaluated in a simulated environment provided by the MAST! Institute. This simulator, named *Simple Ship Sim*, has been partially developed by Penttinen and Hupponen in their respective master's theses [10, 12].

3.2.1 Simple Ship Sim

Simple Ship Sim is a simulated environment whose goal is to train and simulate RL agents in a maritime environment. This simulator, implemented in Python, allows to easily integrate new RL agents, to train them in various situations and to display the simulation

through a GUI. Despite its ability to represent complex scenarios, this simulator is a simplified version of the real world. For instance, it does not include natural forces, such as the wind or ocean currents. Therefore, a ship is unable to drift, which means a vessel's heading is similar to its COG, and its STW is equal to its SOG. However, *Simple Ship Sim* is a very helpful tool when it comes to designing and testing proofs of concept, which is the case in this master's thesis. [10, 12]

The full structure of the *Simple Ship Sim* program is presented in Appendix A. It contains five main parts: the scenario configuration, the ship object, the RL agent, the core that handles the simulation and the GUI.

The *configuration module* allows to set different scenarios by using YAML configuration files. This standard allows to easily serialize data in a user-friendly format [30]. These configuration files select the scenario's map and set the initial characteristics of all vessels, such as their sizes, their original positions and the coordinates of their target destinations. This configuration is then loaded into the simulator at the start of every training or simulation.

The *ship* element is the Python object that represents a vessel in the simulator. It contains attributes that are defined in the configuration files. Attributes can be constant, such as the vessel's size, mass or propeller diameter. They can also be variable, namely the vessel's position, speed or heading. Finally, some attributes are used to save the state of the vessel, i.e., if it has collided or if it has reached its destination. The ship object also handles the vessel's physic.

The *RL agent* is the entity that controls a ship. It takes actions based on the current vessel's state and the prediction return by the associated neural network. The author's practical part of the master's thesis consists of modelling and implementing a new agent which learns first through IRL and then with RL.

The *core* of the program contains two main modules. The *environment* module holds the Open AI Gym environment, discussed later in Section 4.1.1. The *simulation* module handles all vessels and lands during the simulation. The combination of those two modules returns environment information, such as the vessel's state, to the RL agent. An important characteristic of the environment is its ability to know every attribute of any ships at a given time. Therefore, the implemented RL agents have an omniscient view of their surroundings, without the need to use sensing systems. As in the dataset, simulations are run with discrete time steps of one second.

Finally, the *GUI* provides a graphic interface to the user in order to visualise ships' paths during the simulations. It also allows to modify the ships' trajectories with a set of buttons, but this functionality will not be used in the context of this master's thesis.

3.2.2 Rotterdam map

The simulation environment consists of a two-dimensional plan, where objects are located with their (x, y) coordinates. Additionally, it allows to display custom maps in the background. The IRL agent is trained on the same map as used in the dataset, i.e., in the waters off Rotterdam. To draw the outline of the map, the author chose to use the maximum and minimum latitude and longitude positions of experts in the dataset as the outline values. In addition, the author added a five-kilometre margin on all sides of the map. The resulting map, downloaded from the OpenSeaMap website [28], is shown in Figure 3.2. In addition to restricted areas in purple and sea signs, it contains one of the main navigation channel of the North Sea. This channel is represented in Figure 3.2 with purple arrows. Like the experts, the agent mission will be to navigate in the channel from west to east, while avoiding other vessels.

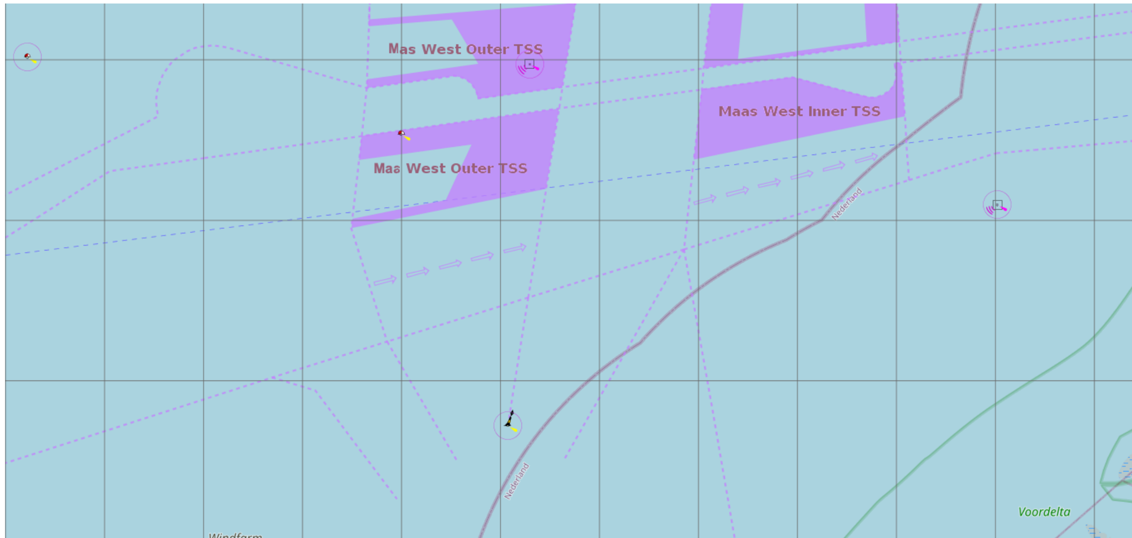


Figure 3.2: Map of the sea off Rotterdam, with navigation channels, restricted areas and sea marks. [28]

3.3 Modelling

To implement and compute this ML problem, it must be modelled. The goal here is to define how the agent perceives its environment and which actions it is able to perform. In other words, this section defines the agent's state and action spaces. However, as stated in Section 2.2.1, the difficulty lies in selecting the suited variables and ignoring unnecessary parameters, in order to obtain small state and action spaces without losing important information. This will enable a less data- and time-consuming training.

The initial step of the modelling process is to list all available variables that can be used to describe the agent's state. These variables can be issued or calculated from the following elements: the agent, its destination, the traffic, the infrastructures and the shore. Firstly, absolute variables, such as (x, y) coordinates or vessels' heading and speed, are available in the simulator without processing needs. Secondly, it is possible to compute new parameters from the raw variables in the simulator. For instance, relative distances and angles can be calculated as shown in Figure 3.3. It includes the distance d to the destination, the angle θ between the agent's heading and the straight line connecting it to its destination, the distances d_2 to other vessels or angles θ_2 towards them.

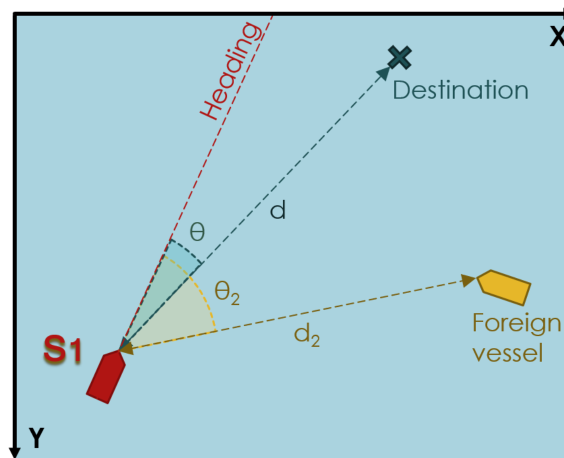


Figure 3.3: Example of state space variables.

For the rest of this section, only relative parameters will be considered to answer the problem expressed in this master's thesis. As shown in Figure 3.4, using absolute variables, such as the (x, y) coordinates, can not lead to the creation of a robust agent able to navigate in scenarios it has never seen. In the illustration, agents *S1* and *S2* are expected to perform similar behaviours, even though they do not have the same absolute coordinates, heading and destination.

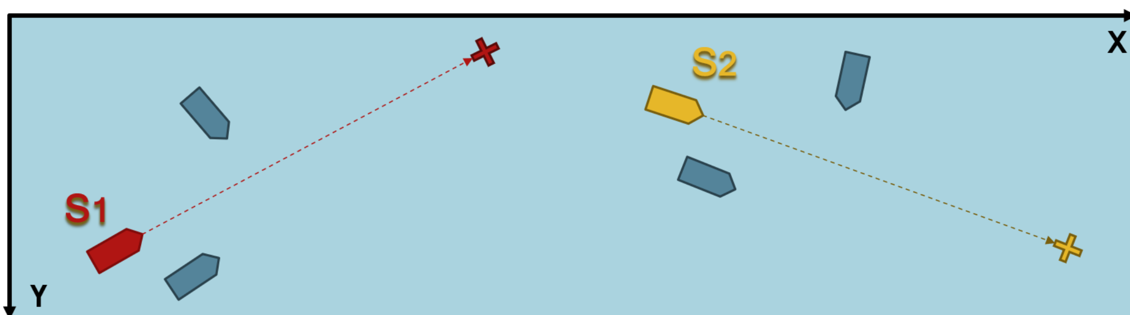


Figure 3.4: Comparison of vessels *S1* and *S2*, facing the same situation with different absolute coordinates, headings and destinations. Blue pentagons represent foreign vessels.

The last statement introduced the second step of the modelling process, which is to select the best action and state variables for the specific tasks the agent must perform. In this master's thesis, the focus is on the three following tasks. First, the agent should be able to navigate in its environment. Second, it has to reach its destination. Third, it also needs to avoid collisions.

3.3.1 Navigation

To drive a vessel, two main attributes can be controlled : the SOG and the COG. The speed is managed by the engine, which can accelerate or decelerate the vessel by varying the rotation speed of the propellers. The heading is directed by the rudder, which can either turn the ship to port or to starboard.

In order to minimize the number of possible actions, the agent's speed is fixed at the average value used by experts in the dataset, i.e., 9.26 metres per second. Thus, the agent's action space considers only the rudder. In his master's thesis, Penttinen suggested using fifteen actions, each one setting the rudder at $-35 + i * 5$ degrees, i being the index of the action between zero and fourteen [10]. A benefit of this design was to have a direct link between the taken action and the targeted rudder angle. However, it shows an important drawback, which is the huge number of actions. Minimizing the action space can succeed in strongly reducing the corresponding problem's complexity. Undoubtedly, using two different actions to set the rudder at 30° and 35° is redundant, because both these actions aim, in most states, at turning the rudder to the right. Therefore, they both show similar results.

The author proposes an amelioration of Penttinen's design by reducing the action space to three alternatives. The latter are turning the rudder to the immediate angle on the right, turning the rudder to the immediate angle on the left, and keeping the same rudder value. The actions are now relative and their resulting behaviours do not depend on the current rudder angle. In addition, the rudder angle range between -35° and 35° has been discretised. While it is possible to handle a continuous state space during the RL process, limiting the number states is required to have reasonable computation time with IRL. As a result, the rudder can now take seven possible values, namely -27°, -15°, -6°, 0°, 6°, 15°, and 27°. The various intervals between the chosen values aim at refining the agent's trajectory. To make this design work, the author increased the rotation rate of the rudder, so each transition between two angles is instant. For example, turning the rudder to the left when the current value is 15° will lead the next rudder angle to be 6°. The agent is now able to fully control its heading by using its rudder.

However, the rudder values in the dataset covers the continuous range from -35° to 35°. Therefore, to map every data sample to a state, the full angle range is divided into

seven windows, as illustrated in Figure 3.5. Every rudder angle in the dataset can then be adjusted to the actual value of the window it belongs to.

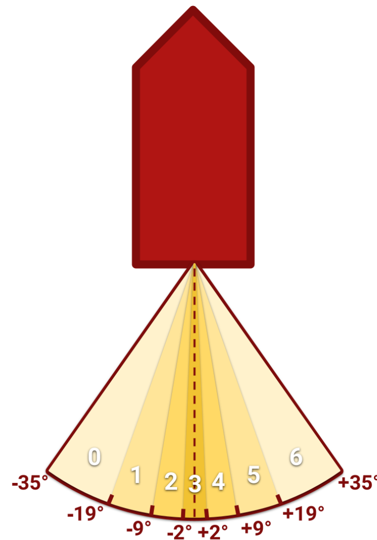


Figure 3.5: Discrete *rudder angle* state space.

3.3.2 Reach the destination

The second but not the least task given to the agent is to reach its destination. In his work, Penttinen suggested using the distance separating the agent to its destination as the main variable to perform this task [10]. The agent would then try to minimize this distance, i.e., to converge towards the destination, in order to maximize its reward. However, this parameter is not sufficient to distinguish different situations requiring different behaviours. On the one hand, as illustrated in Figure 3.6 (a), two vessels having the same distance to their respective destination could have this last on the opposite side of the boat. In the example, the agents *S1* and *S2* need to turn respectively to starboard and to port. On the other hand, as illustrated in Figure 3.6 (b), two vessels facing approximatively the same situation with different distances to their destination should perform similar actions. As a result, it seems impossible to use the distance to the destination alone to guide an agent to its destination.

Furthermore, it is often easier during the learning process to have a small and constant time between the actions and the rewards the agent receives. This helps the agent to link a state-action pair to its corresponding benefit. Thus, variables are divided in two classes: the ones depending on the agent's heading and the others relying on the agent's position. When turning the rudder, the agent's direction is quickly modified, while it takes more time to have significant variations of the agent position. For instance, when using the distance to the destination, which depends on the agent's position, a significant amount

of time can separate the action of turning the rudder and the fact of getting closer to the destination.

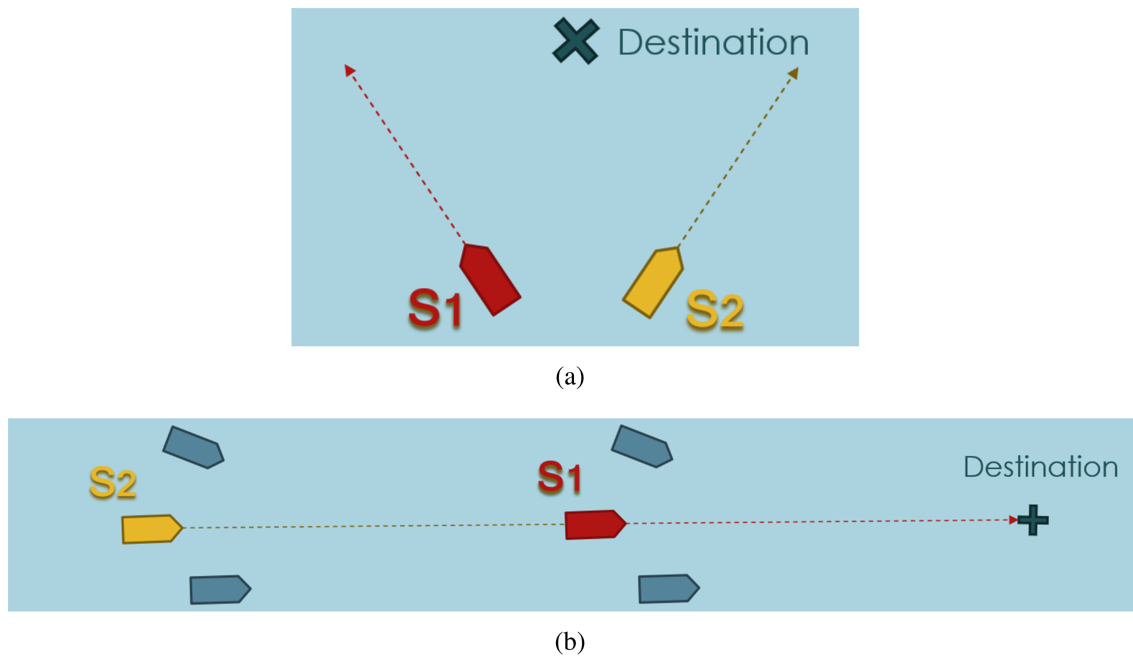


Figure 3.6: Comparison of vessels *S1* and *S2*, facing different situations with the same distance to the destination (a), and facing the same situation with different distances to the destination (b). Blue pentagons represent foreign vessels.

Based on the previous arguments, the author decides to use the angle between the agent's heading and the strait line between it and its destination as the main variable for reaching the destination. This parameter, illustrated as the angle θ in Figure 3.3, will be called the *destination angle* from now on. Like the *rudder angle*, the *destination angle* is discretised, as illustrated in Figure 3.7. It can take seven possible values, namely -67°, -32°, -12°, 0°, 12°, 32°, and 67°, each of them belonging to one window. The agent has now enough information to navigate towards its destination. By minimizing the absolute value of the *destination angle*, the agent can align its trajectory towards the destination.

3.3.3 Avoid collisions

The last task considered in this master's thesis is avoiding collisions. The nature of a collision can be of two types: with the shore or with another vessel. Since the location of the scenarios in the dataset is deep into the North Sea, no demonstration exists on how to avoid reefs or land. Therefore, collisions with land are not considered in this model. The focus is then turned to collisions with other vessels. Among all metrics describing the state of a vessel compared to other ones, the author chooses the Closest Point of Approach (CPA) parameter.

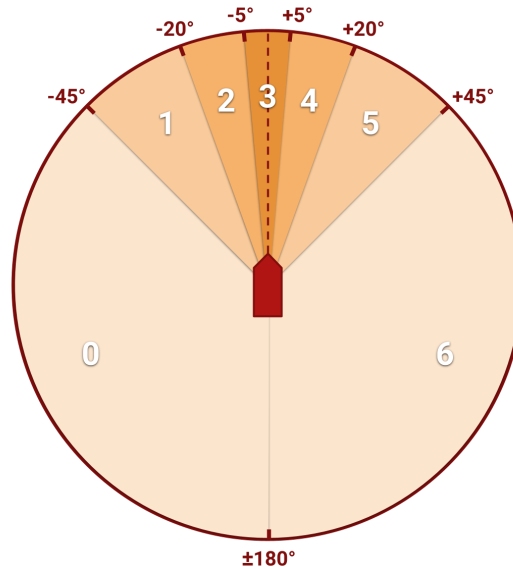


Figure 3.7: Discrete *destination angle* state space.

The CPA is a commonly used metric in maritime navigation. It corresponds to the closest vessel the agent will approach in the future. Two sub-metrics exist to describe the danger of a potential collision. The Distance to Closest Point of Approach ($\mathrm{D_{CPA}}$) is the minimum distance between the agent and another vessel in the future. The acronym CPA is often used as a shortcut to refer to this metric. A $\mathrm{D_{CPA}}$ of zero means a collision will occur if nothing is performed to avoid it. The Time to Closest Point of Approach ($\mathrm{T_{CPA}}$) is the second sub-metric and corresponds to the time until the agent reaches the moment of the CPA. Both $\mathrm{D_{CPA}}$ and $\mathrm{T_{CPA}}$ can be controlled by the agent if it changes its trajectory to avoid the potential collision. Furthermore, it is possible with these metrics to identify a future risk-prone situation before it occurs. [31]

The current D_{CPA} is included as the third variable of the agent's state space. This variable describes the need to modify the trajectory against an imminent collision. Lower is the $\mathrm{D_{CPA}}$, bigger is the need to alter the agent's path. This metric presents several advantages, because it considers the speed and heading of the agent, as well as the ones of the other vessel. This is not the case of simpler metrics, such as the relative angle θ_2 towards foreign vessels as illustrated in Figure 3.3. The major benefit of $\mathrm{D_{CPA}}$ is its ability to see into the future. Therefore, it can detect possible collisions before ending up in dangerous situations. Like previous state parameters, this variable is discretised in seven possible values between 0 and 4200 metres, namely 150, 500, 950, 1500, 2150, 2900 and 3750 metres. However, the $\mathrm{D_{CPA}}$ is not sufficient since it only indicates the need to modify the trajectory. No information is given about the preferable path, i.e., turning to port or to starboard.

The fourth and last parameter of the state space is called the *maximum D_{CPA} an-*

gle. This variable represents the relative angle of the safest path compared to the current agent's heading. It is computed by calculating the D_{CPA} for every path between -20° and 20°, 0° being the agent's heading. The trace obtained from this calculation is illustrated in the example given in Figure 3.8. After finding local maximums, depicted with green dots in the figure, the safest path can be deducted by comparing their D_{CPA} values and selecting the highest one. By taking this path, the agent will stay the furthest away from the other vessels. Finally, to compute the IRL process, this state feature is also discretised. Thus, the *maximum* D_{CPA} *angle* is adjusted to one of the following values: -15°, -6°, 0°, 6° and 15°. The example below shows the salmon-coloured agent facing a flow of vessels going from its starboard to its port. A gap in this flow can be seen as the safest path for the agent. As exposed on the D_{CPA} *trace*, the agent will cross the flow through the middle of the gap if it turns 14° to starboard.

3.3.4 Synthesis

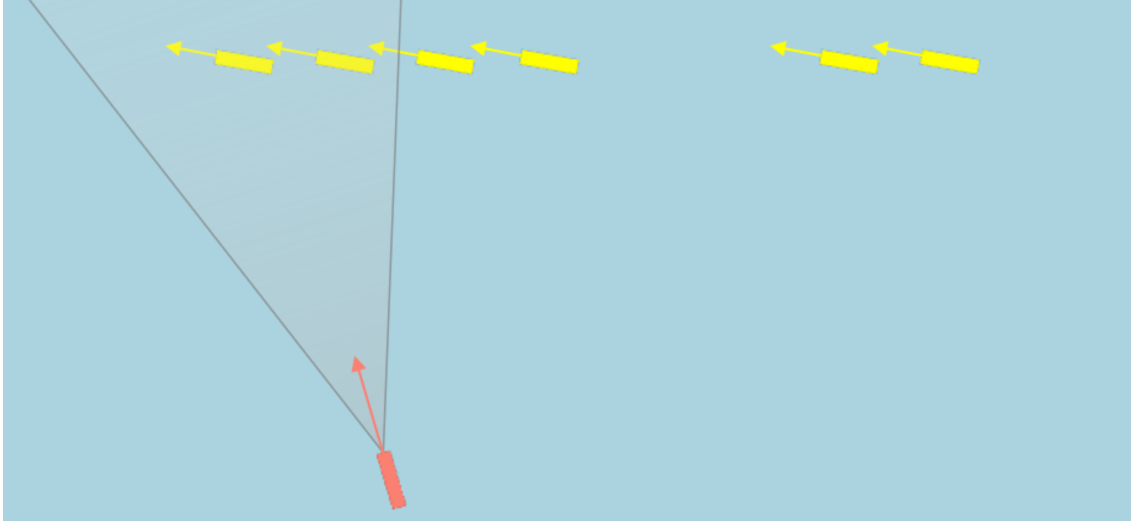
To conclude this chapter, a synthesis of the agent's state and action spaces is given in Tables 3.1 and 3.2. The state space involved in the IRL process contains 1715 possible states. Thanks to a wise discretisation, this reduced state space allows to strongly facilitate and speed up the computation of the *reward matrix* through IRL. Nevertheless, the RL part of the training still runs on the continuous state space. This justifies the *destination angle* range between -180° and 180°, the D_{CPA} value between 0 and 4200 metres and the *maximum* D_{CPA} *angle* between -20° and 20°.

Table 3.1: State space variables.

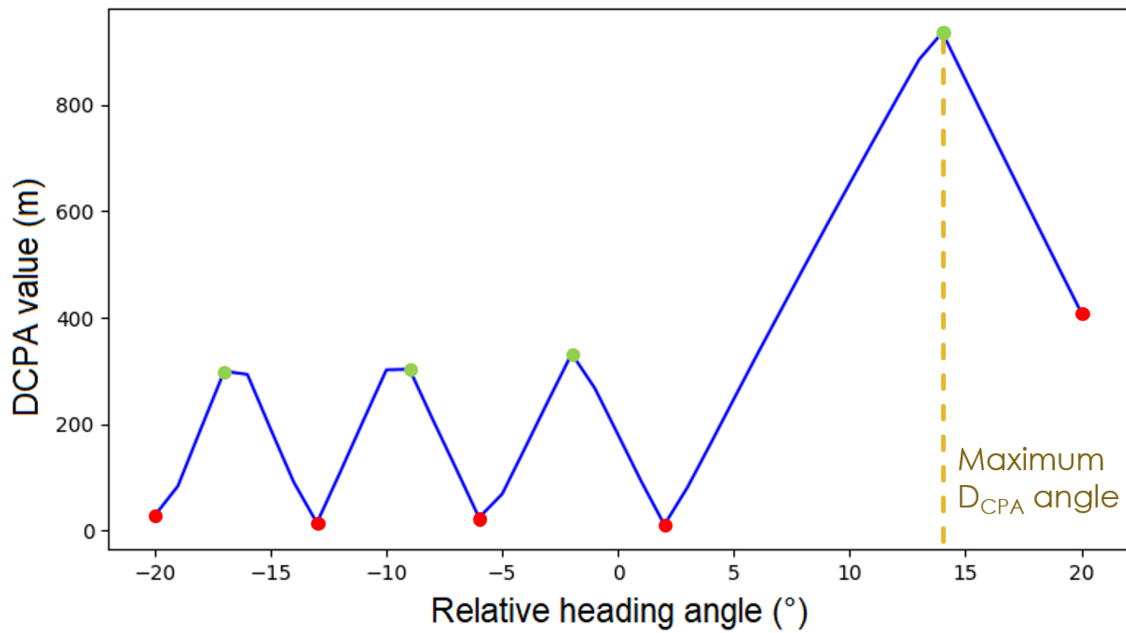
Task	**Variable**	**Unit**	**Minimum**	**Maximum**
Navigation	Rudder angle	Degree (°)	-27	27
Reaching destination	Destination angle	Degree (°)	-180	180
Avoiding collisions	D_{CPA}	Metre (m)	0	4200
	Max D_{CPA} angle	Degree (°)	-20	20

Table 3.2: Action space variables.

Task	**Action**	**Unit**	**Minimum**	**Maximum**
Navigation	Turn the rudder to port Do not turn the rudder Turn the rudder to starboard	Degree (°)	-27	27



(a)



(b)

Figure 3.8: An example of a crossing situation, where the salmon-coloured vessel represents the agent and the yellow vessels the traffic (a), with the corresponding D_{CPA} *trace* (b). The D_{CPA} *trace* range is represented in grey in the simulation.

4. Implementation

Implementation is the usual word in the software development field for the process of producing a system or an application. It involves to develop, in any computer language, the solution previously designed to solve the problem. In the context of this master's thesis, it mainly refers to writing the Python code of the developed agent and finding solutions to optimise it.

This chapter will first present the different Python libraries, modules and computing resources used when implementing the proof of concept agent. In addition, it will explain the major functions of the IRL agent, before giving details on the computation of the *reward matrix* and the necessary inputs.

4.1 Libraries, modules and computing resources

Libraries and modules are a way for developers to easily create new applications, by using already optimized and efficient tools or algorithms. Python is known to be a programming language which provides many useful libraries, such as Numpy [32] for the computation of large matrices or Tensorflow [33] and Keras [34] when dealing with AI and ML. This section describes the Python libraries and modules used during the implementation of the autonomous agent. Furthermore, an explanation of how the author speed up the execution of the training process will be given.

4.1.1 Simulator's libraries

PySide 2. The *Simple Ship Sim* simulator requires a set of libraries to run simulations and display results on a GUI. *PySide 2* is the Python library holding the *Qt5 framework*, which is a commonly used tool when developing GUIs in Python applications. This API, developed in C++, provides graphical components and widgets. It was used to implement the simulator's interface, which is discussed in more detail in Penttinen and Hupponen master's theses. [10, 12, 35]

Open Ai Gym. To implement its maritime environment, the *Simple Ship Sim* program uses the *Open Ai Gym* package. This library was created to evaluate and compare state-of-the-art RL algorithms. To achieve this goal, it provides a set of benchmarks and envi-

ronments, such as Atari and board games, that show a common interface. In addition, it allows the implementation of new environments, that can be added to its growing collection. *Open Ai Gym* was chosen to build the *Simple Ship Sim* simulator, due to the industry standard it provides and the ease of implementing a new environment. [10, 12, 36]

4.1.2 Agent's libraries

Tensorflow. To ease the implementation of the agent, many libraries related to ML are used. Among them stands *Tensorflow*. This library, developed by Google, provides an interface for modelling ML solutions on large-scale problems. Today, *Tensorflow* is a standard in the industry, due to its good performances, flexibility and ability to run computations on a wide panel of platforms. Firstly used exclusively by Google for research and development purposes, the library was made open-source in 2015. [33]

Theano. This open-source software performs strong optimisations on mathematical expressions at compilation time. These expressions are memorised as graphs of variables and operations, which can be modified to eliminate unnecessary computations. The optimisations performed by *Theano* depend on the target platform and aim at saving computational and memory resources. The goal of this software is to overcome a limitation of Python, namely its inefficient interpreter for mathematical computations. [37]

Keras. This open-source Python framework provides APIs that ease the development of deep neural-networks and ML solutions. It is widely used among scientific organizations and universities, because of its powerful tools and its flexibility. It can be run with other libraries, such as *Theano* and *Tensorflow*, this last being well integrated with *Keras*. This framework was chosen by Penttinen to build the first agents in the *Simple Ship Sim* simulator, due to the fast prototyping it allows and the large existing documentation. [10, 34, 38]

Keras-rl. This open-source library provides support on deep RL for the *Keras* framework. Furthermore, it integrates well with *Open Ai Gym* environments. *Keras-rl* implements many methods, such as the DQN algorithm used by Penttinen. This algorithm, that has shown good results in many applications, combines RL with deep neural networks. Since it resulted in Penttinen's promising agent, it has been reused to solve the RL part of this master's thesis. [10, 39]

NumPy. This widely used library is a standard when dealing with large arrays in Python. *NumPy* provides array structures that can store large and multi-dimensional matrices.

These arrays, also called *tensors*, can be accessed efficiently and saved in specific *NumPy* files. Furthermore, the library implements a large number of methods to manipulate and perform optimised operations on the matrices. It is used during the implementation of this master's thesis, due to the use of large vectors and arrays, both in the RL and IRL algorithms. [32, 40]

IRL. As explained in Section 2.4.3, the method chosen to extract expert intentions, namely IRL, requires to compute the reward function of the environment. To achieve this, the library developed by Alger [41] is used to ease the implementation. This library contains different IRL algorithm implementations, including Wulfmeier's *Deep IRL* [21]. As discussed in Section 2.4.3.2, this last is the one used in this master's thesis. An advantage of this library is the fact it is not linked to any environment. Instead of having a direct connection between the IRL algorithm and the environment, it indexes states and actions, and refers to them with numbers. As a result, using this library with a new environment, such as the *Simple Ship Sim* simulator, requires less development costs. Concerning its implementation, the library works on a *Theano* backend and provides an interface function for every algorithm. This function requires various inputs, such as the *trajectories matrix*, the *state features matrix* and the *transition probability matrix P* of the state space. These *NumPy* matrices contains respectively the expert demonstrations, the state feature values associated to every state and the probability to go from every state to every other state under each action. The function returns another *NumPy* matrix, which associates a reward value to every state.

Matplotlib. This Python package is a tool used to generate figures, including histograms, maps or diagrams. It was created to provide a cross-platform and high-quality support for graphic generation in the Python language. In the context of this master's thesis, *Matplotlib* is used to display and visualise the intermediate results of the IRL process, which are stored under the form of *NumPy* matrices. [42]

4.1.3 Computation time

Multiprocessing. This Python module is used to run in parallel the execution of a program on the different CPU cores of the target machine. It provides an API able to create processes by instantiating its *Process* class. Those processes can then be launched and synchronized at the end of the execution. The *multiprocessing* package also implements shared memory objects to communicate data between processes. The author used this module to reduce the computation time of the *transition probability matrix*, because it is the most resource-consuming part of the training process. [43]

CSC services To minimise the computation time of the *reward matrix* and all the inputs of the IRL library, the execution is run on distant resources provided by CSC. This company provides cloud computing services, such as *cPouta*, which is the one used during this master's thesis. This specific service is an IaaS, which allows the user to manage all the stages of the infrastructure from the application down to the middleware and the operating system. A virtual machine is created on the server and an SSH connection is used to communicate. The whole *Simple Ship Sim* simulator and the IRL agent can then be transferred on the virtual machine, for training purposes. [44]

4.2 Program structure

Since libraries and computing resources have now been presented, it is possible to explain how the agent developed in this master's thesis has been implemented. Based on the agents previously developed by Penttinen and their corresponding Python classes, the author created an interface named *Agent*. This interface declares all the methods required by an agent to run in the *Simple Ship Sim* environment. The author later implemented the class *IRL*, which inherits from the previous interface and implements all the required methods. Furthermore, the *IRL* class defines new methods to compute the *reward matrix*. A UML diagram of this class is illustrated in Figure 4.1.

The methods declared in the *Agent* interface allow to initialise (*init_model*) or load (*load_from_file*) the agent's neural network, to perform one step in the environment (*step*), to learn a new policy (*train*), to define the state (*get_observation_space*) and action (*get_action_space*) spaces, to compute the current state (*get_state*), to obtain its corresponding reward (*get_reward*) and to perform actions on the actuators based on the prediction of the neural network (*take_action*). The only attribute of the interface is the path of the agent's neural network file.

The inheriting class *IRL* owns a larger list of private attributes. These attributes aim at defining the discrete state space (*sta_dest*, *sta_rudder*, *sta_cpa*, *sta_maxcpa_angle*), giving the number of states, actions and state dimensions (*n_sta_dest*, *n_sta_rudder*, *n_sta_cpa*, *n_sta_maxcpa_angle*, *n_states*, *n_actions*, *n_dimensions*), and holding the computed *reward matrix* (*reward_matrix*). In addition to the required methods, it implements new ones to compute the *reward matrix* through IRL (*get_transition_probability*, *get_trajectories*, *get_reward_function*), to identify actions or states by their numbers (*get_num_from_state*, *get_state_from_num*, *get_action_num*), and to duplicate the learning data (*get_reversed_action_num*, *get_reversed_state_num*). In addition, a short method shows a use example of the IRL library (*irl_test*). The most important methods of both classes are further discussed below and some are illustrated by flowcharts in Appendix B.

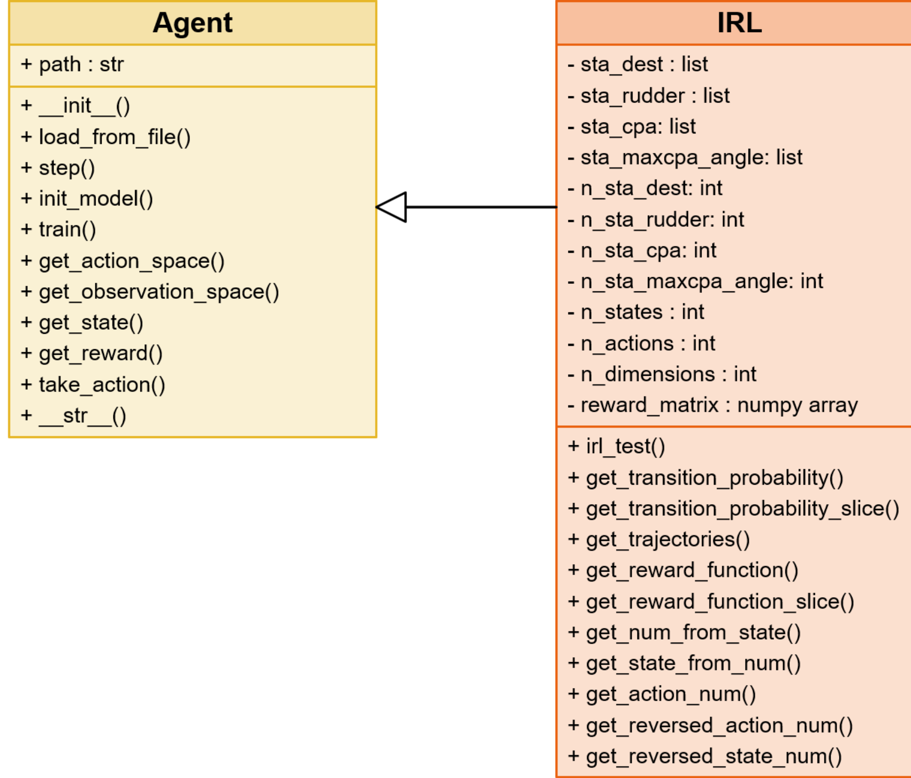


Figure 4.1: UML diagram of the *Agent* interface and the *IRL* class, without any information about methods' parameters and return values, for readability purposes.

4.2.1 get_state() method

The *get_state()* method is required because it computes and returns the current state of the agent. Therefore, it gives the input of subsequent functions, including the reward computation and the neural network prediction. The state is returned as a tuple, which holds the current value of all state features. For the IRL agent, these features are the *destination angle*, the *rudder angle*, the D_{CPA} and the *maximum* D_{CPA} *angle*.

While the *rudder angle* is simply copied from the *Ship* object's attributes, the *destination angle* θ is calculated with the formula in Equation (4.1). First, the absolute angle between the environment's horizontal axis and the vector connecting the agent to its destination is calculated. Then, the agent's absolute heading is subtracted from the result. Finally, this angle is converted in degrees and adjusted between -180° and +180°.

$$\theta = atan2(y_{destination} - y_{agent}, x_{destination} - x_{agent}) - heading_{agent} \tag{4.1}$$

The computation of the two last state features, i.e., the D_{CPA} and *the maximum* D_{CPA} *angle*, is more complex. First, the minimum distance in the future between the agent and

every foreign ship is calculated. The future (x,y) coordinates of all vessels are therefore computed, to determine where and when two ships will be the closest to each other. The estimated coordinates of a vessel depend on its current position, heading and speed at time t, as shown in Equations (4.2) and (4.3). The parameter α is gradually increased to compute the successive positions of both vessels and, therefore, the successive distances between them. The shortest distance is the last value before the distance begins to increase. After determining the future minimum distance between the agent and every other vessel, the smallest one is taken as the D_{CPA} value.

To reduce the computation time, positions are calculated every *increment* seconds. This variable is decreased during the process, until the minimum distance is found. This allows to both have a precise minimum distance and avoid computing unnecessary positions. Furthermore, the parameter α in Equations (4.2) and (4.3) is limited to 900 seconds, in order to exclude risk-prone situations that are too far in the future.

$$x_{t+\alpha} = x_t + cos(heading_t) * speed_t * \alpha \tag{4.2}$$

$$y_{t+\alpha} = y_t + sin(heading_t) * speed_t * \alpha \tag{4.3}$$

The same computation is performed for every agent's relative heading between -20° and +20°, 0° being the original heading. This enables to obtain the D_{CPA} *trace*, as illustrated in Figure 3.8. The local maximums, depicted with green dots in the figure, are then determined. A last function chooses the safest relative heading angle from the D_{CPA} *trace*. All local maximums lower than 0.8 times the highest one are eliminated and, among the remaining ones, the closest to the current heading, i.e., 0°, is designated as the *maximum* D_{CPA} *angle*. This allows to avoid unnecessary turns.

4.2.2 get_reward() method

The *get_reward()* method maps a state to its corresponding reward in the *reward_matrix* attribute. However, a requirement of the IRL library is to identify all possible states and actions with a unique number. These numbers are used as indexes in the *reward_matrix* array. Consequently, the author implemented three methods, namely *get_num_from_state()*, *get_state_from_num()* and *get_action_num()*. The first one converts a state tuple into an integer index. The second one performs the reversed operation. The last method converts an action, i.e., turning the rudder, into an integer index. The *get_reward()* method is now able to determine the index of a state tuple and to return the corresponding reward value.

In addition, a conditional statement was added to check if the agent is in the Rotterdam's maritime channel, illustrated in Figure 3.2. If not, the agent receives an additional

negative reward and a collision happens. This may lead the agent to learn primitive behaviours on how to stay in the sea channel, on the specific map of Rotterdam waters.

4.2.3 train() method

The *train()* method aims at learning an efficient policy that will control the agent's decisions. When training classic RL agents, only the policy needs to be calculated. Nevertheless, the algorithm used in this master's thesis requires to compute more elements, such as the *reward matrix*. As explained in Section 4.1.2, the IRL library requires three matrices as input. The *feature matrix* can be computed easily, because it is simply a mapping between the state index and the corresponding feature values. However, the two others, namely the *transition probability matrix* and the *trajectories matrix*, need time-consuming calculations. Therefore, it is preferable to implement separated methods, which compute and save the matrices before returning them to the main *train()* method. The computation of the *reward matrix* through IRL suffers the same issue and has also a dedicated method. These methods are explained in detail in the following sections.

As a consequence, the computations of all intermediate results are divided into their respective methods. This separation allows to run only some parts of the training process, with the intention of using the results later. Thus, a parameter, specified when launching the process in command line, was added to the *train()* method. It consists of a string, that can contain the five following characters: *t*, *p*, *a*, *i* and *r*. Each letter, acting like a flag, enables the computation of an intermediate result or the display of debug information.

4.2.4 get_transition_probability() method

The *transition probability matrix* P is the first input matrix needed by the IRL library. As deduced from Equation (2.6), the size of this three-dimensional matrix is (n_s, n_a, n_s), with n_s and n_a being respectively the number of states and actions.

To compute such probabilities from the simulator, the author implemented the method *get_transition_probability()*, which is called by the method *train()*. This method is based on an iterative process where randomness is used to obtain unbiased probabilities. Each iteration consists of letting the agent performing a random action in a random state, before noting in which state it ends up. By repeating this iteration a sufficient number of times, it is possible to draw all transition probabilities.

However, three difficulties raise when implementing this method. First, the random situations need to resemble to the ones in the dataset. Second, the computation time of each iteration needs to be as short as possible, since tens of millions of repetitions are required to ensure accurate probabilities. Third, the distribution of states must be uniform,

since the most inaccurate probabilities would be the ones describing the state that appears the less during the process.

To ensure situations are similar to the ones in the dataset, the agent is first placed in the centre of a ten-kilometre square. While its speed is fixed at the value chosen in Section 3.3.1, i.e., 9.26 metres per second, its destination coordinates, heading angle and rudder angle are set randomly. In addition, three other vessels are randomly placed in the square. Because foreign vessels are on average slower than the experts in the dataset, these ships have a lower speed, i.e., 6.0 metres per second. Furthermore, to make the distribution of states uniform, all ships are converging towards the agent's destination. This enables to increase the probability of collisions with the agent and, therefore, reduces the number of required ships. Thus, the computation time is decreased, because the simulation runs faster. After placing all vessels, the current state number i and action number a are computed with the methods *get_state()*, *get_num_from_state()* and *get_action_num()*. One simulation step is then performed in the simulator, before computing the new state number k. As a result, the value in $P[i][a][k]$ is incremented. When all iterations are completed, the matrix is normalized to obtain probability values between zero and one. The matrix is finally saved in a NumPy format.

To speed up even more the computation, the author decided to divide iterations between all CPU cores of the computer. The package *multiprocessing*, discussed in Section 4.1.3, is used to launch one process per core. When started, processes call the method *get_transition_probability_slice()*, where all the previous computation has been moved. The result matrix, shared among processes, allows each process to increment it. The matrix is normalized after all processes are completed and synchronized. As a consequence, the computation time is nearly divided by the number of CPU cores.

4.2.5 get_trajectories() method

The *trajectories matrix*, which contains the expert demonstrations, is the second input needed by the IRL library. The goal of the method *get_trajectories()* is to compute and save the state-action pairs from the raw dataset. The size of the resulting three-dimensional matrix is $(n_t, l_t, 2)$, with n_t and l_t being respectively the number of trajectories and their length in time steps. The last dimension holds both the index of the state and the one of the action in the current sample. The approach used to compute this matrix consists of, first, incorporating every expert situation in the simulator and, second, retrieving the state and action indexes with the IRL class methods.

Among the variables present in the dataset and selected in Section 3.1.1, the author focussed on the latitude and longitude values. From these variables, it is possible to replace every vessel in the (x, y) coordinate system of the simulator. Furthermore, by using

the very next position coordinates, it is possible to deduce the vessels' absolute SOG and COG, this last being equivalent to the absolute heading in the *Simple Ship Sim* simulator. The author justifies his choice to recompute these variables, available in the raw dataset, by answering the problem of map projection. While latitude and longitude are spherical coordinates, the simulator works with Cartesian coordinates in a two-dimensional environment. Therefore, distortions occur during the conversion. Even if they may be small when working on small-scale areas, they can still appear, especially between the south and the north of the map. For instance, when considering the map chosen in Section 3.2.2, the distance between the upper right and left corners is actually shorter than the distance between the lower right and left corners. In addition to the position, the author uses the successive rudder angles to set the vessels in the environment and to determine the actions taken by the operators. Finally, the last position of every scenario is used as the destination, for all the samples in this scenario.

Based on the size of the matrix, another requirement of the IRL library can be deducted: all expert trajectories must be the same length. However, scenarios show variable lengths between 3,300 and 5,400 time steps. Therefore, they are divided into reduced trajectories of one hundred samples, to lose only a few samples at the end of every scenario.

Finally, to implement the data augmentation technique defined in Section 3.1.4, the methods *get_reversed_action_num()* and *get_reversed_state_num()* were added. The latter take respectively as input the index of an action or a state and return the index of the reversed action or state. For instance, the state index is first converted into a state tuple with the method *get_state_from_num()*. The opposite values of the *rudder angle*, *destination angle* and *max CPA angle* are then calculated. Finally, the index of the new state is calculated with the method *get_num_from_state()*.

4.2.6 get_reward_function() method

When all required matrices are computed or loaded, it is possible to calculate the *reward matrix* with the IRL library. The method *get_reward_function()* starts by creating the *feature matrix*, which associates to each state the corresponding feature values. Because the computation time of the *reward matrix* severely increases with large state spaces, the author tried to run in parallel this function by launching different processes that call the method *get_reward_function_slice()*. Every execution of this method computes a *reward matrix*. These matrices are then merged into one unique matrix, which is the one returned by the method *get_reward_function()*. However, due to the lack of consistency in merged matrices, the author finally decided to compute a single matrix in one unique process . Nevertheless, the code structure for parallel computation is preserved for further work.

5. Results and evaluation

Now both the IRL agent's design and implementation have been explained, it is possible to present and evaluate the resulting behaviours against various situations.

This chapter will present the main intermediate and final results of the IRL process. Firstly, several visualisations of the *trajectories matrix* will be shown. Secondly, the shape of the *reward matrix* will be discussed. Thirdly, the final autonomous agent will be evaluated through simulations on different types of situation. Finally, improvements over the previous state-of-the-art agent in the simulator will be presented.

5.1 Expert trajectories

The first interesting result is the *trajectories matrix*. This matrix stores the expert demonstrations under the format designed by the state and action spaces. In this section, two-dimensional histograms are used to visualise this matrix under different point of view. The goal is to uncover some patterns, in order to validate the state space features designed in Section 3.3.

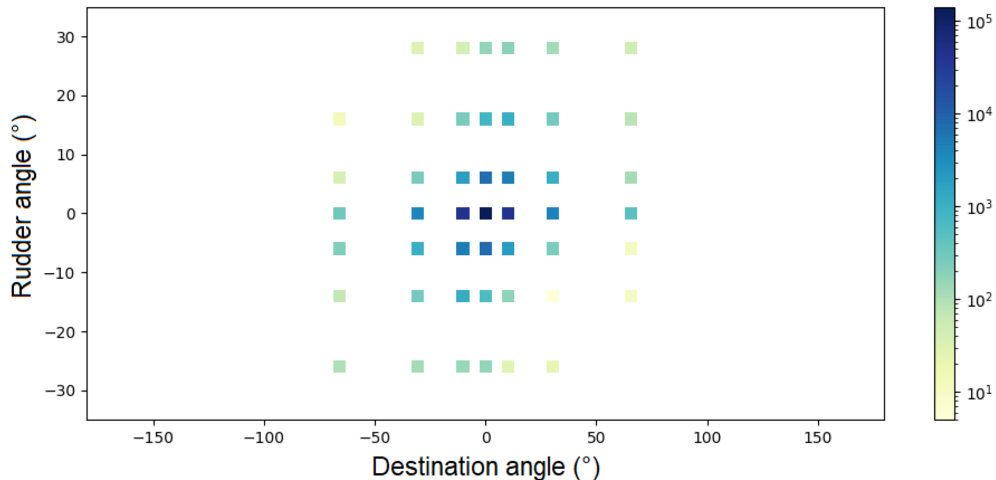


Figure 5.1: Expert trajectories' states, with the *rudder angle* in function of the *destination angle*, in number of occurrences in the dataset.

The first visualisation, appearing in Figure 5.1, draws the experts' *rudder angle* in function of their *destination angle*. The metric is the number of occurrences in the dataset, of each combination of these angles. As it can be seen in the figure, the experts are more

likely to have a positive *rudder angle* when having a positive *destination angle*. In other words, they prefer turning to the right when their destination is also on the right. Conversely, they turn more often to the left when their destination is on their left. As a result, the experts converge towards a *destination angle* of zero degree, i.e., they navigate towards their destination. This is confirmed by the high occurrence of the central state, where both angles are null. The pattern observed in this histogram proves that the combination of both features, namely the *rudder angle* and the *destination angle*, can be suited to navigate the agent to its destination.

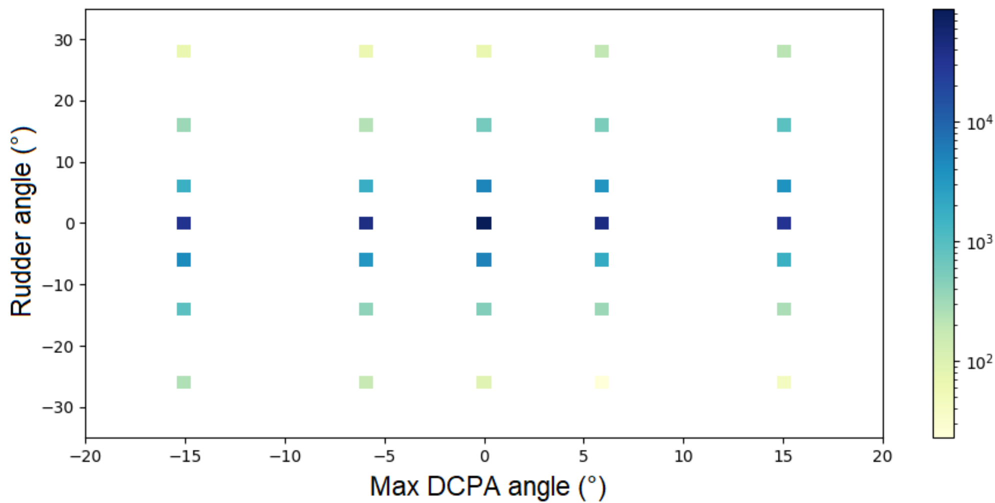


Figure 5.2: Expert trajectories' states, with the *rudder angle* in function of the *maximum D_{CPA} angle*, in number of occurrences in the dataset.

The second visualisation, illustrated in Figure 5.2, plots the *rudder angle* in function of the *maximum D_{CPA} angle*. Like in the last figure, a pattern can be identified in this histogram. When the *maximum D_{CPA} angle* is positive, i.e., when the estimated safest path is on the right, the experts prefer turning to the right. On the contrary, they turn more often to the left when the safest path is also on the left. Moreover, no preference is shown when the *maximum D_{CPA} angle* is null. This pattern confirms that this pair of features can be used to answer the task of collision avoidance. These results also demonstrate that the implementation of the *maximum D_{CPA} angle* allows to find the safest path most of the time.

The third and last visualisation, shown in Figure 5.3, plots the *rudder angle* in function of the current D_{CPA} value. As it was designed for, this feature informs the agent when altering the trajectory is required. On the one hand, the experts rarely modify their heading when being in safe situations, i.e., when the D_{CPA} value is high. On the other hand, they deviate more often from their trajectory when this value is low. Therefore, this metric can be considered as a valuable threshold for collision avoidance.

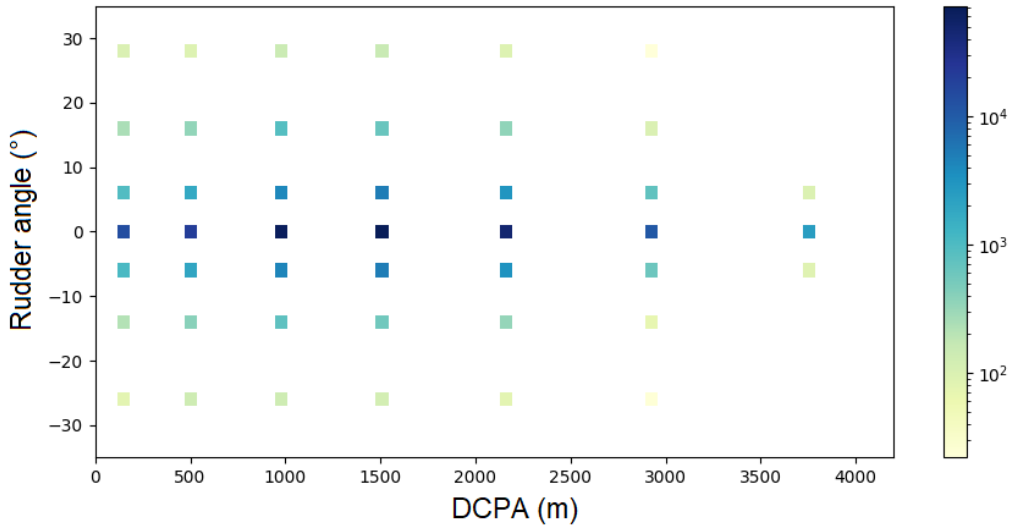
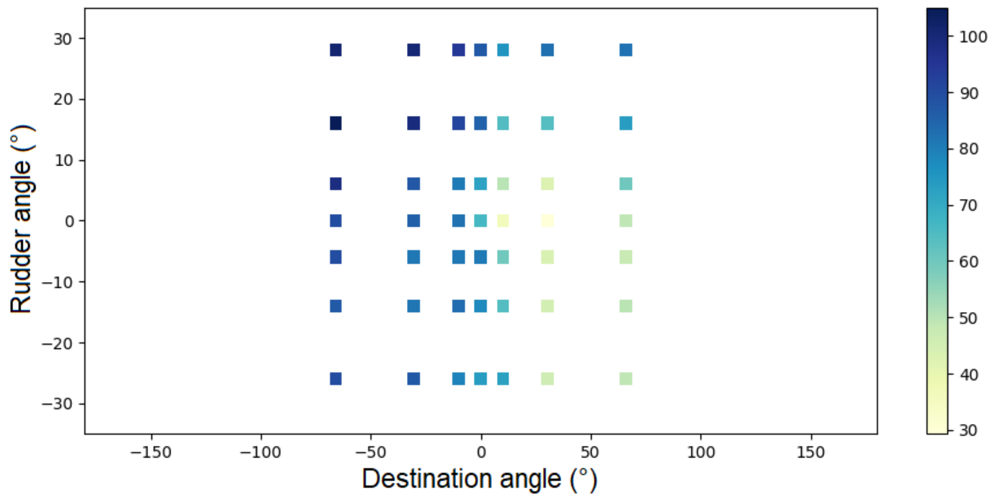


Figure 5.3: Expert trajectories' states, with the *rudder angle* in function of the current D_{CPA} value, in number of occurrences in the dataset.

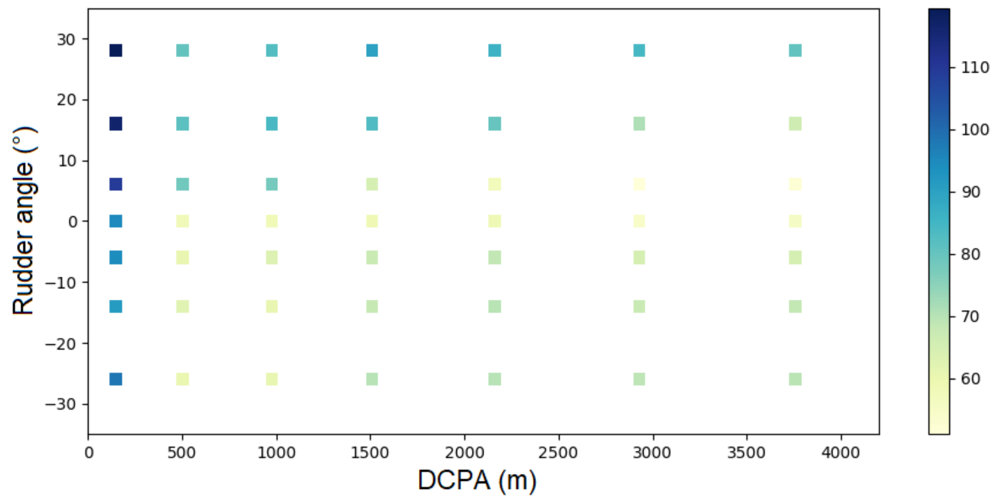
5.2 Reward function

The second intermediate result presented in this chapter is the *reward matrix*. This matrix is a representation of the expert intentions, when navigating in the demonstration scenarios. As explained in Section 2.2.4, this matrix holds a reward for every state and is later used to train the agent through RL. To visualise this matrix, the same three previous representations are used and illustrated in Figure 5.4.

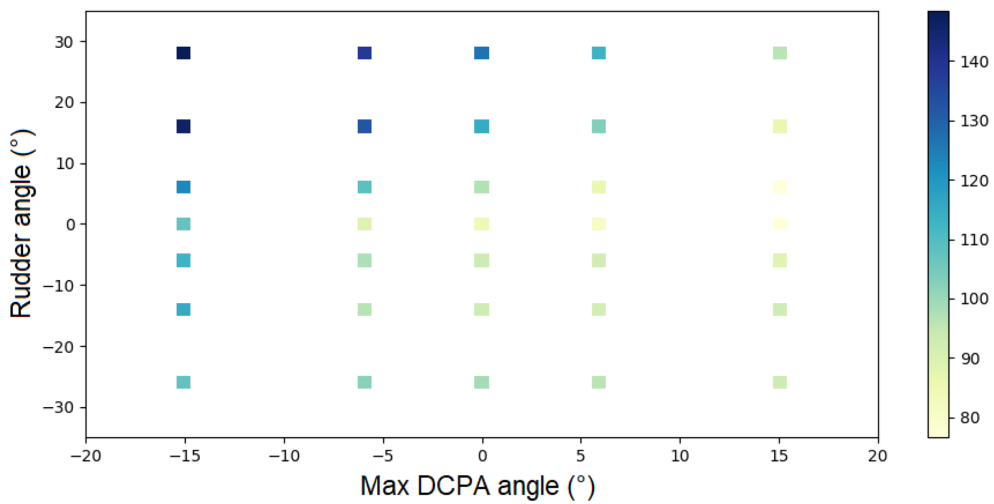
In these visualisations, blue states are beneficial while green and yellow ones are unfavourable. Therefore, when looking at Figure 5.4, some logical patterns can be seen. For instance, when the destination is on the right of the agent, this last obtains more reward by turning to the right. Nevertheless, this type of logical pattern does not fulfil the whole matrix and many incoherent or even senseless values appear. For example, the agent acquires more reward when turning to the right while its destination is on its left. Another example is the high reward obtained when the current D_{CPA} value is low, i.e, when there is a risk of collision. These unexpected results can be seen, with human eyes, as unwanted behaviours. However, two significant elements are usually not considered when looking at this matrix. Firstly, when learning through RL, the agent takes into account the discount factor discussed in Section 2.5.2. This factor has a major influence during the training, because it forces the agent to focus on a long term efficiency. As a result, it will not always choose the action which gives the maximum immediate reward. Secondly, the author integrated in the state space the current agent's *rudder angle*, which has a direct effect on the agent's trajectory. Therefore, modifying the value of this feature, by turning



(a)



(b)



(c)

Figure 5.4: *Reward matrix* representations, with the *rudder angle* in function of the *destination angle* (a), of the current D_{CPA} value (b), and of the *maximum* D_{CPA} *angle* (c).

the rudder, has medium- or long-term consequences on the other variables. Thus, tasks, such as reaching the destination, are caught between pairs of features. The agent is then able to converge towards the best combination of feature values.

In addition, these results highlight the purpose of IRL. It is commonly stated that the most critical part when solving a RL problem is to design the reward function. Moreover, it is sometimes impossible to define it manually, since it is a complex abstract representation of expert intentions, that does not consider the discount factor. The unexpected reward function obtained in this master's thesis perfectly illustrates this difficulty, since the author would not have designed such a function if he did it manually. [14, 17]

5.3 Navigating agent

From the reward function was learned, through classic RL, a policy that controls the agent's behaviours. This section aims at evaluating this policy, which is the final result of this master's thesis. However, no evaluation benchmark exists for the specific problem of navigation in the *Simple Ship Sim* simulator. In addition, there is no available metric to evaluate the agent, since the experts, i.e., Aboa Mare students, were evaluated by the overall appreciations of their teachers over their performances. Due to this lack of evaluation metrics, the author opts for a qualitative evaluation rather than a quantitative one. The success criterion is the capacity to answer the three tasks the agent was built for, namely navigating in its environment, reaching the destination and avoiding collisions.

The policy is tested in scenarios the agent never experienced during its training. Firstly, simple scenarios involving few vessels verify the agent's ability to answer classic situations. Secondly, a complete and more complex scenario demonstrates the agent's skills to navigate in complex situations involving many vessels.

5.3.1 Simple situations

The four scenarios illustrated in Figure 5.5 are used to evaluate the IRL agent on simple situations requiring little manoeuvre. In the figure, the agent's paths are represented by the grey lines and the arrows indicate the current heading and speed of vessels. Longer is the arrow, faster is the agent and vice versa. Tests lead to the conclusion that the agent is able to reach its destination (a), to overtake a slower vessel or to be overtaken (b), to answer a head-on situation (c), and to cross the trajectory of another vessel without colliding with it (d). The agent learned both to alter its path to avoid an obstacle and to recover the most direct trajectory towards its destination. The three navigation tasks are then completed in simple situations. Furthermore, since some of the test situations were not present in the

expert demonstrations, such as the head-on confrontation, these simulations demonstrate the high generalisation performed by the RL algorithm and the designed state space.

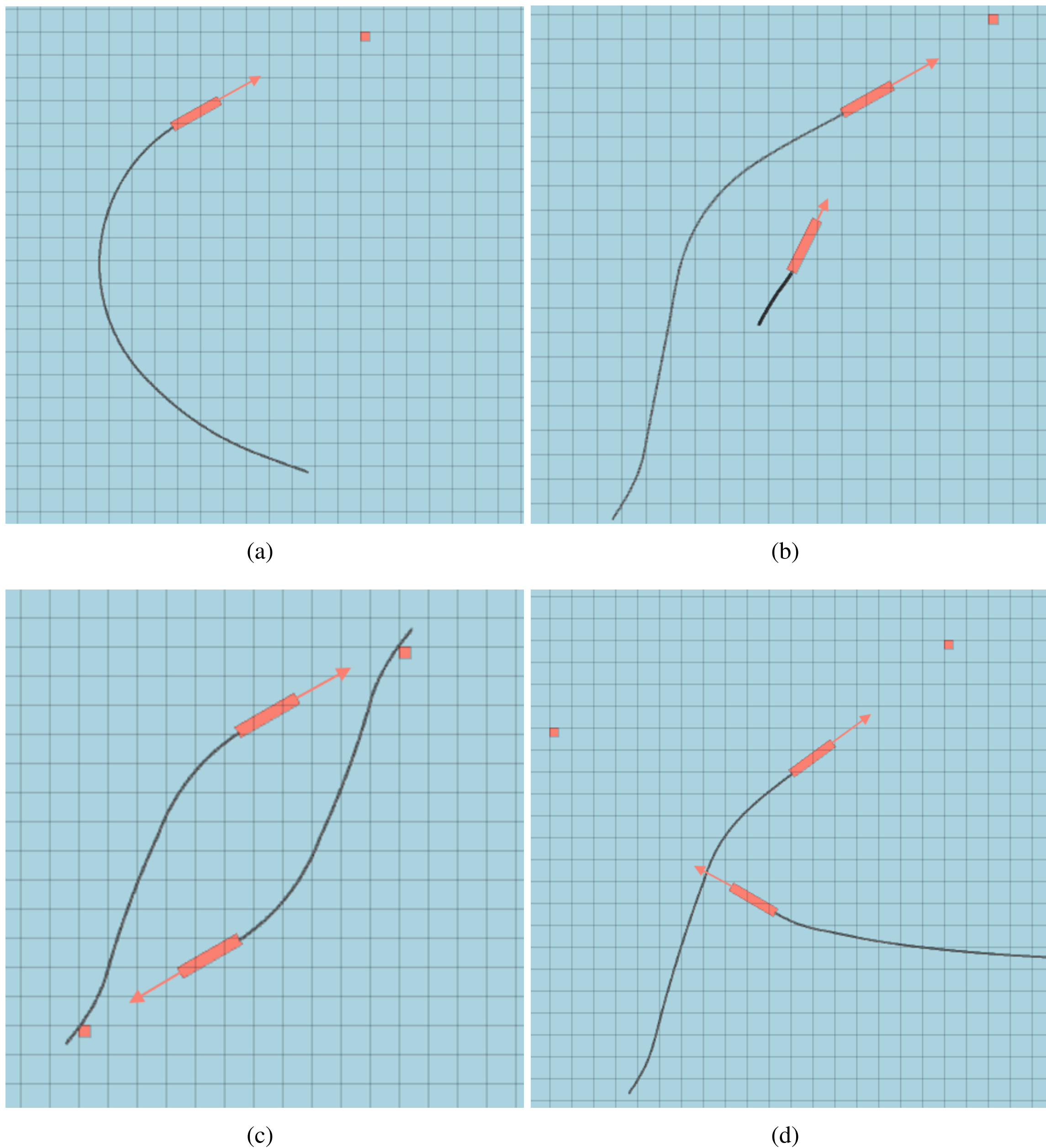


Figure 5.5: Simulation of the IRL agent in simple situations, i.e., reaching the destination (a), overtaking and being overtaken (b), head-on situation (c), and crossing (d).

5.3.2 Complete scenario

In addition to the previous scenarios, a more complex one is elaborated to test the agent's ability to answer the three tasks when navigating around many vessels. This scenario, illustrated in Figure 5.6, consists of the agent navigating in the Rotterdam maritime chan-

nel, from west to east, while finding its way through different flows of vessels. In the figure, the arrows illustrate again the vessels' heading and speed. The path of the agent is indicated by the gray line, while the ones of other vessels have been disabled for readability purposes.

This test highlights the agent's capability to answer multi-vessel situations and its ability to find the safest and most efficient path among all available options. When overtaking a group of vessels (a, b), the agent chooses the path that will leave the other vessels as far away as possible. Additionally, it anticipates the other vessels' trajectories when crossing through the gap in the perpendicular flow (c, d). Finally, it is able to avoid stationary obstacles without unnecessary detours (e, f).

5.4 Improvements over state-of-the-art agent

To complete the agent's evaluation, a comparison is made with the previous state-of-the-art agent in the simulator, that is, the DQN agent implemented by Penttinen [10]. Thus, it is possible to highlight the improvements brought during this master's thesis.

Firstly, the IRL agent is totally controlled by the AI, while this is not the case of the DQN agent. This last was piloted by the neural network only during critical situations, where another vessel is close to the agent. An autopilot function guides, the rest of the time, the agent towards its destination. As a consequence, the IRL agent is more autonomous that the one presented by Penttinen. Moreover, the neural network answers the new task of reaching the destination, which was completed by the autopilot function in Penttinen's agent.

Secondly, the IRL agent is able to answer complex situations, unlike the DQN agent. Since this last focusses on the closest vessel when avoiding collisions, it is blind to every other vessel. Therefore, it struggles to answer multi-ship situations. On the contrary, the IRL agent can find the safest path in a flow of vessels, as shown in Figure 5.6 (b, d).

Thirdly, while the DQN agent was built to strictly follow the COLREGs rules, the IRL agent is able to identify and take more efficient paths, like real navigators would. This can be seen in Figure 5.5 (d), where the agent coming from the lower left corner prefers crossing in front of the other vessel, because it knows this manoeuvre will require less effort. In this situation, COLREGs rules would recommend to cross behind the vessel coming from the right. The IRL agent also considers, through the D_{CPA} and *maximum* D_{CPA} *angle* features, the COG and SOG of other vessels when choosing its actions. This was not the case with the previous agent, which only considered the current distance to the closest vessel and its relative angle of approach.

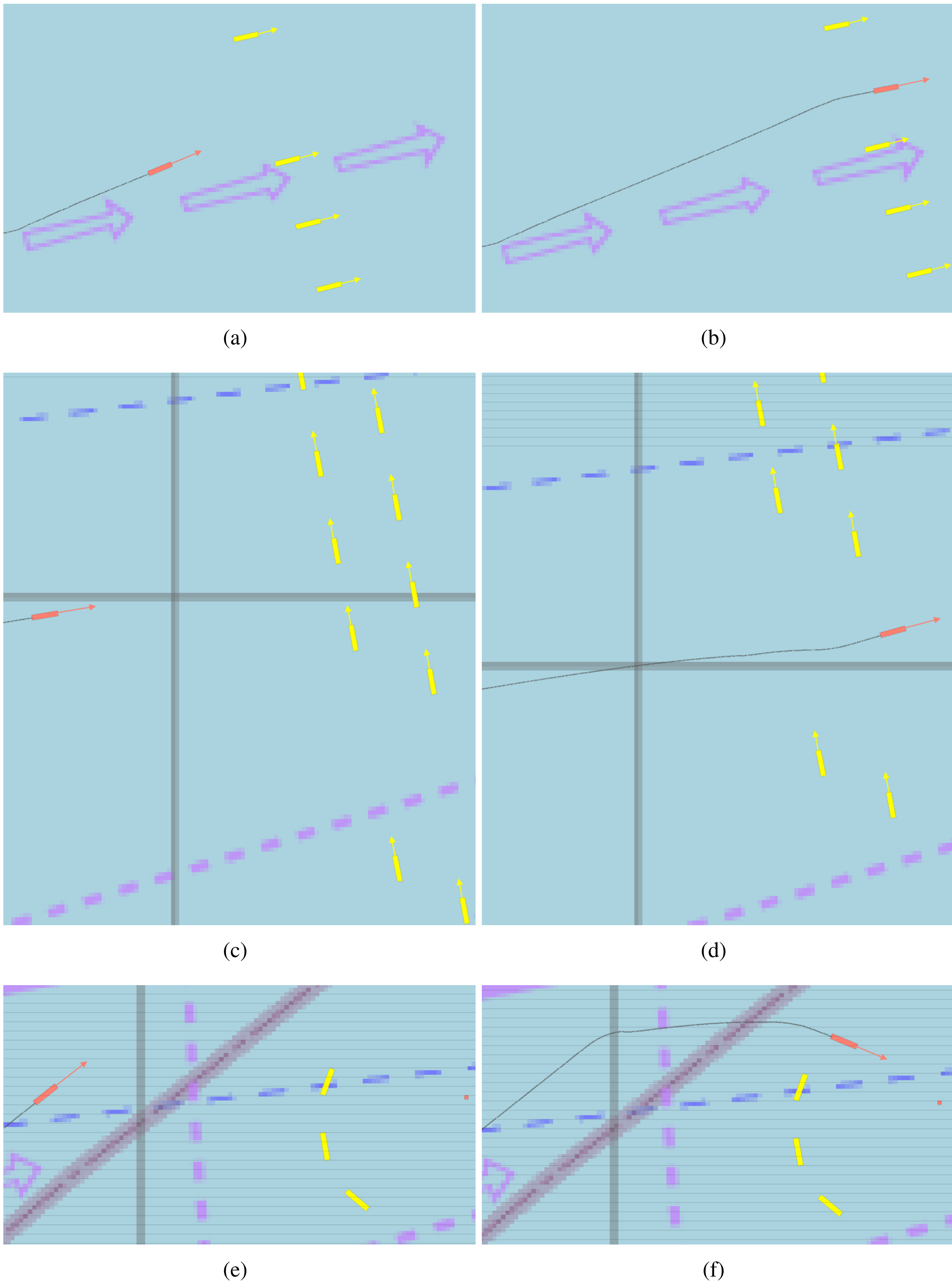


Figure 5.6: Simulation of the IRL agent in a complex scenario, where the salmon-coloured vessel is the agent and yellow ones are vessels simply navigating towards their respective destinations. The agent's goal is to navigate in the Rotterdam sea channel from west to east while avoiding other vessels. It first needs to overtake a group of ships (a, b), before crossing a flow a vessels going from south to north (c, d), and finally reaching its destination while avoiding stationary ships (e, f). Pairs of images represent the beginning and the end of each situation.

6. Conclusion

In this master's thesis, the method of IL was used and tested in the context of autonomous navigation. Furthermore, it has been shown that the algorithm of IRL can bring many improvements in this complex domain, where the reward function of the environment is difficult to manually design. The resulting agent, built as a proof of concept, is able to answer the three main tasks of maritime navigation, namely moving in its environment, heading towards its destination and avoiding collisions. The resulting simulations also demonstrated how the designed state space and the IRL process enable a high generalisation of the problem.

The proof of concept agent was implemented and evaluated in the *Simple Ship Sim* simulator, provided by the MAST! Institute team and Åbo Akademi University. Furthermore, it was trained with student navigator demonstrations, issued from the Aboa Mare maritime academy. The agent's performances were assessed by the author appreciation on the performed simulations. The latter contained both simple and complex situations, requiring various behaviours. However, due to the absence of evaluation metric or formal proof, the resulting agent's performances could be questioned. Nevertheless, the goal of this master's thesis has been reached, since the used evaluation technique is sufficient to demonstrate the agent's ability to answer situations similar to the ones performed by the experts in the dataset. Furthermore, it is able to handle unknown scenarios or even ones it never had demonstrations on.

The simulator used to implement and test the developed agent remains very simple and does not represent all the aspects of the real-world maritime navigation. Therefore, this agent would be unable to navigate and control a real vessel, where the forces applied on the ship are much diverse and complex. However, this proof of concept shows promising results with limited resources, and introduces the use of IRL in such environments. Because the *Simple Ship Sim* simulator has the potential to be upgraded, in order to better match with the outside world, an improved version of this agent, more robust and efficient, can be expected in the near future. Furthermore, implementing a new IRL agent in a more powerful simulator, such as AILiveSim [45], could be a continuation of this master's thesis.

6.1 Future work

The promising results shown in this master's thesis open the door to future improvements. To answer the given objectives, the author made many compromises and simplifications. For instance, the agent's state space was severely reduced and discretised to allow reasonable computation times. A starting point for ameliorations would be to run trainings on more powerful computational resources. This would enable to design a much bigger state space and use continuous state features with wider ranges. In addition, the full D_{CPA} *trace* could be given as an input to the neural network without the prior calculation of the *maximum* D_{CPA} *angle*. The neural network would then analyse the trace and determine, by itself, where is the safest path. These modifications could help to refine the agent's trajectory, in order to have faultless behaviours. In addition, more actions, such as accelerating and decelerating, could be added to the action space.

A second basis for improvements would be to use a better and larger dataset. Due to the lack of data, it was impossible to select only excellent demonstrations. As a result, some expert trajectories were suboptimal. However, the principle of IL requires to show only perfect behaviours to the agent. Therefore, having a more complete dataset would enable to retain only the best possible examples. In addition, the dataset used in this master's thesis shows a poor distribution, with mostly overtaking and crossing situations. Using a more distributed dataset could help to better answer all types of scenarios.

A third amelioration, linked to the second one, would be to abandon the data augmentation process. Despite the benefits it gives on small datasets, mirroring the data brings inconsistencies in maritime navigation, since COLREGs rules respect priorities, such as the priority to the right in crossing situations. This issue is illustrated by the IRL agent, which does not systematically follow these priorities, even when an available path that respect them is equivalent in terms of efficiency. Using a larger dataset could, therefore, allow to get rid of data augmentation.

Finally, because the experimentation takes place deep in the sea off Rotterdam, the agent is unable to learn how to avoid lands. Therefore, including lands, both in the dataset and in the state space, could help the agent to navigate near the coast or in ports.

Appendices

A. Simple Ship Sim program structure

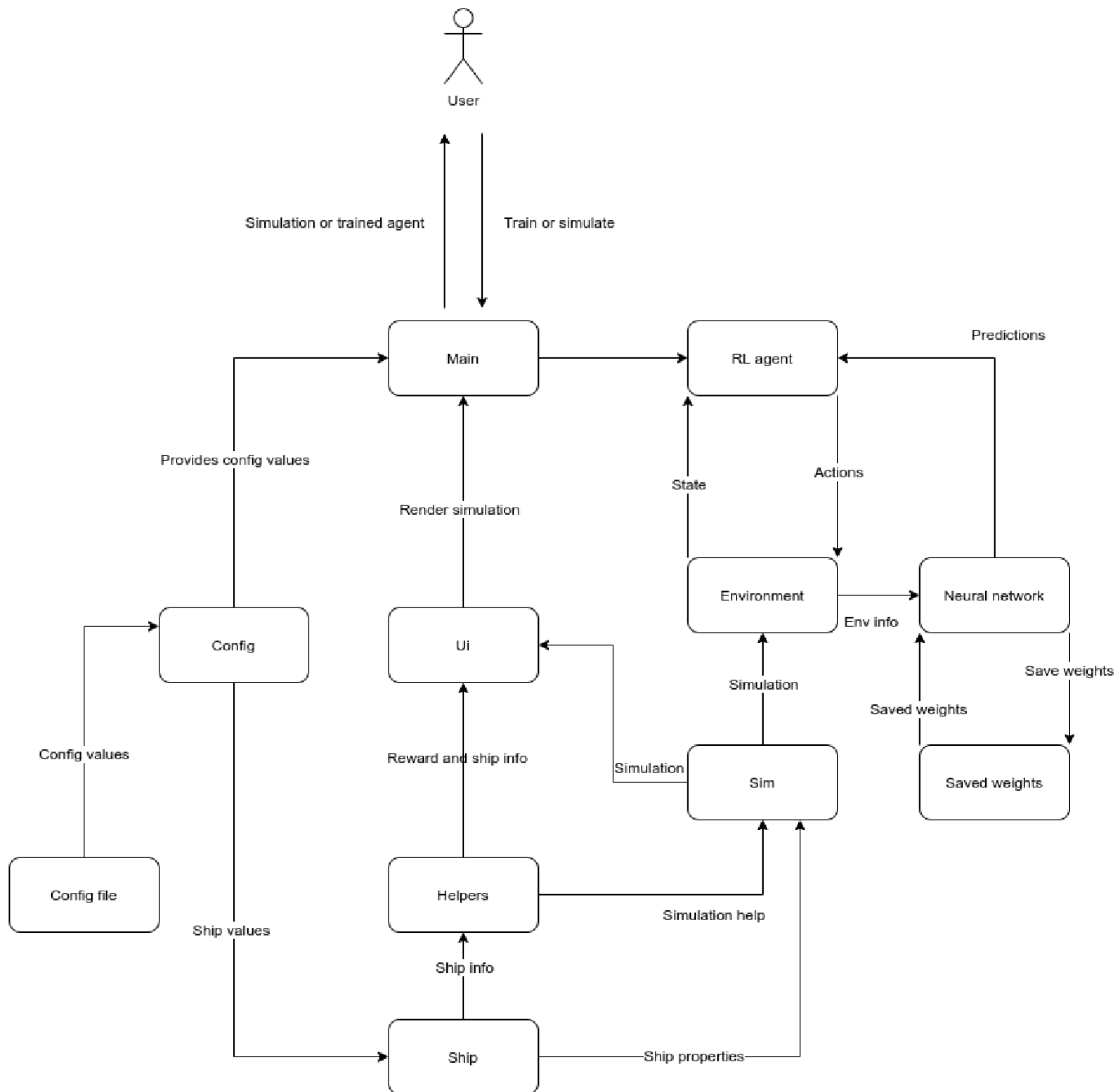


Figure A.1: Penttinen's illustration of the *Simple Ship Sim* program structure. [10]

B. IRL agent's main methods' flowcharts

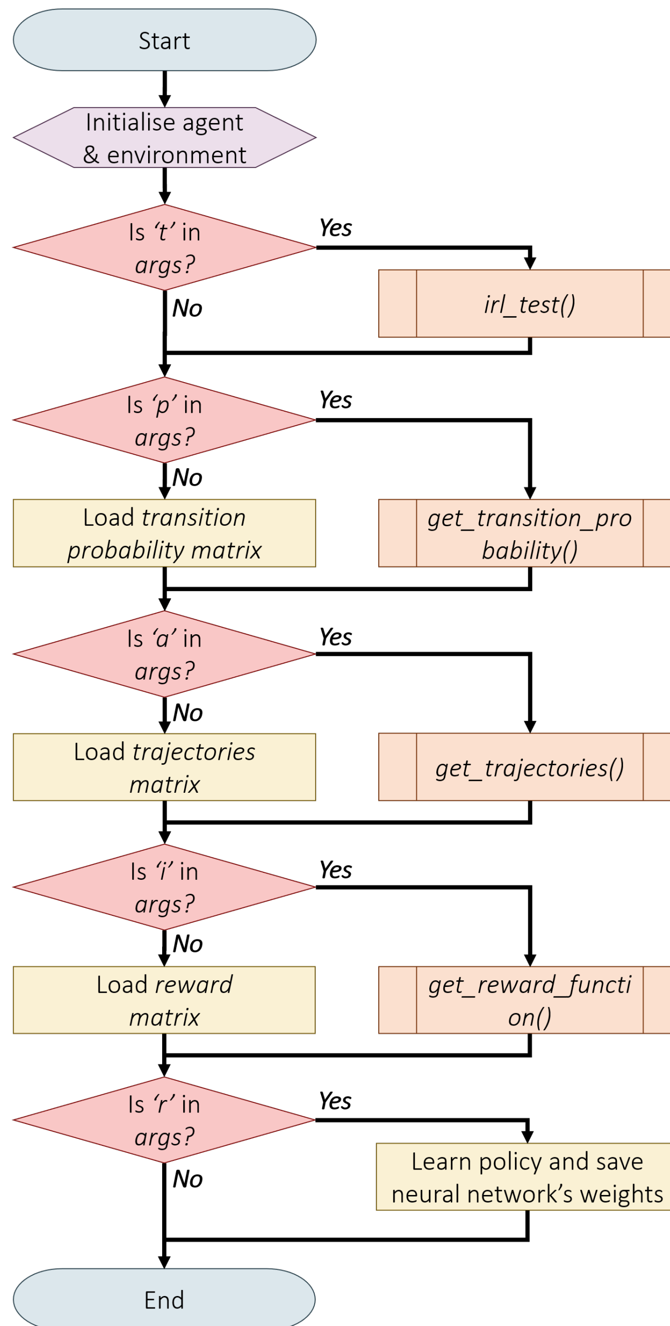


Figure B.1: *train()* method's flowchart.

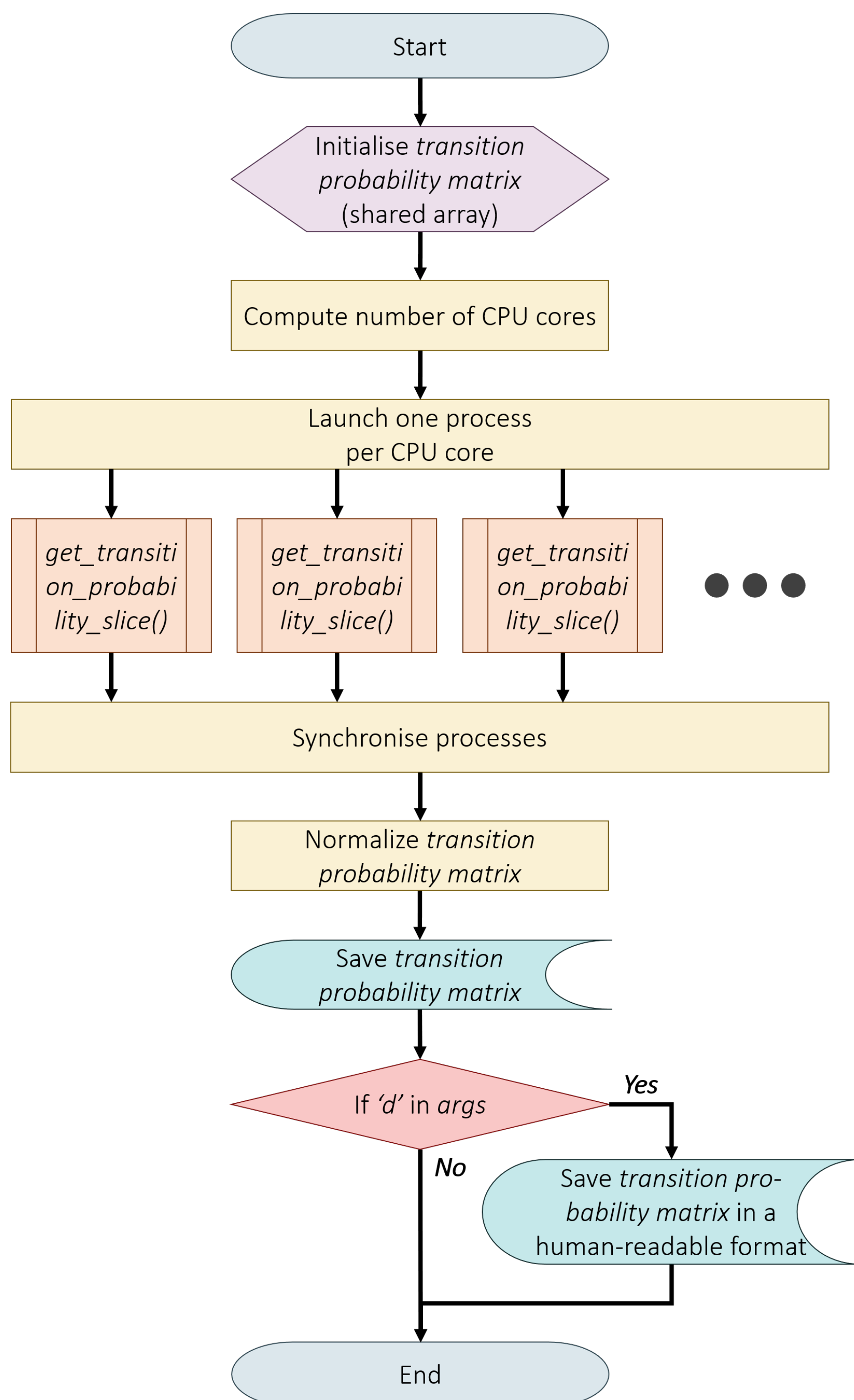


Figure B.2: *get_transition_probability()* method's flowchart.

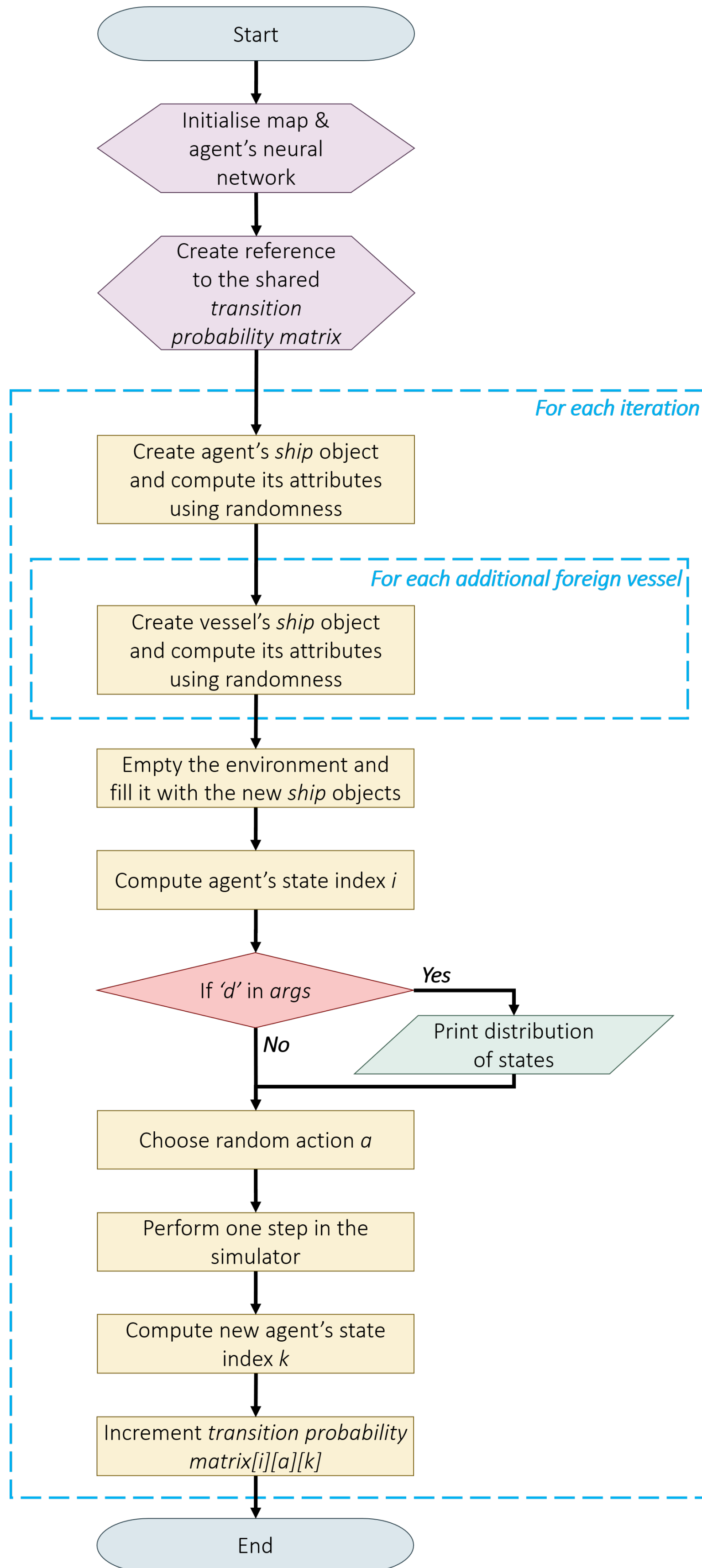


Figure B.3: *get_transition_probability_slice()* method's flowchart.

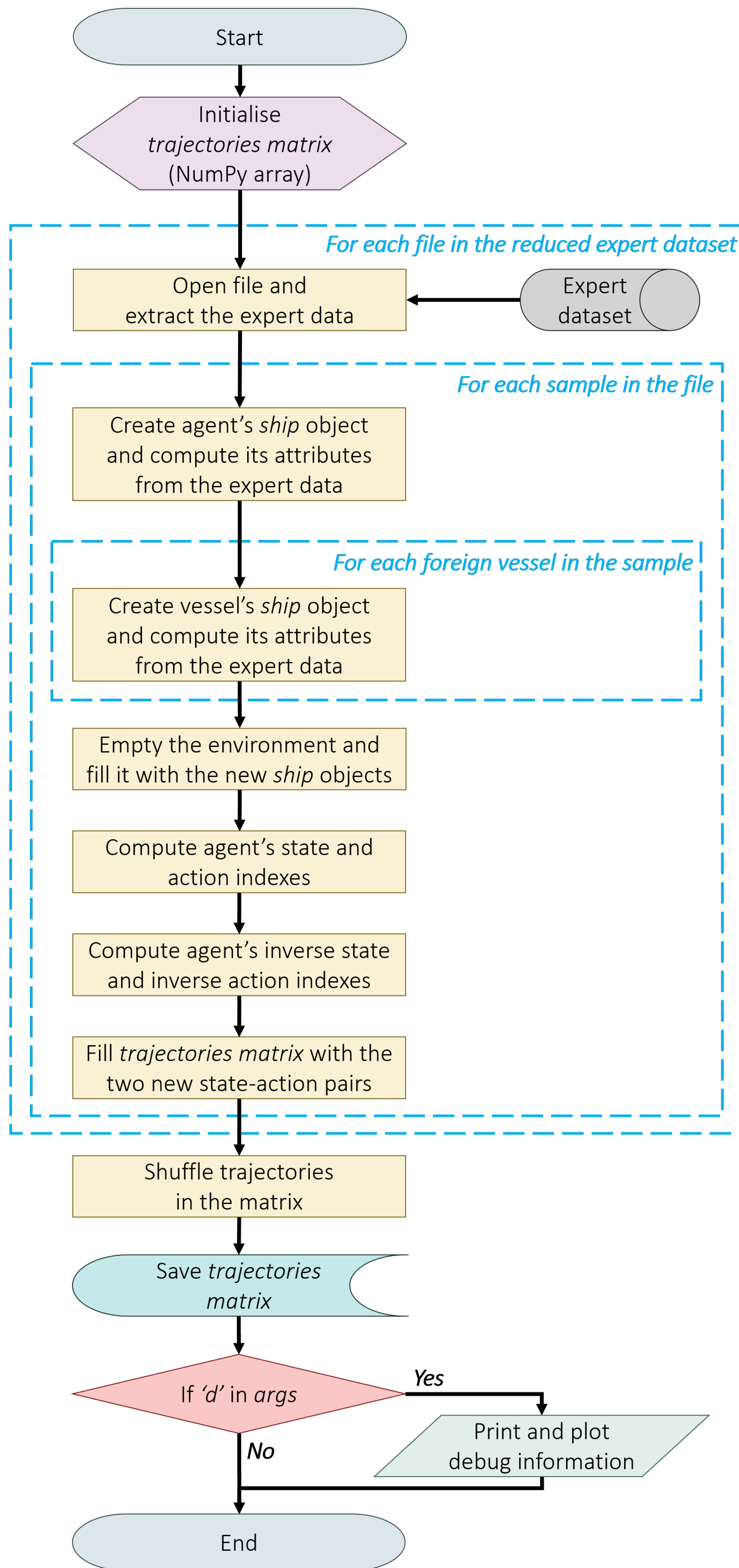


Figure B.4: *get_trajectories()* method's flowchart.

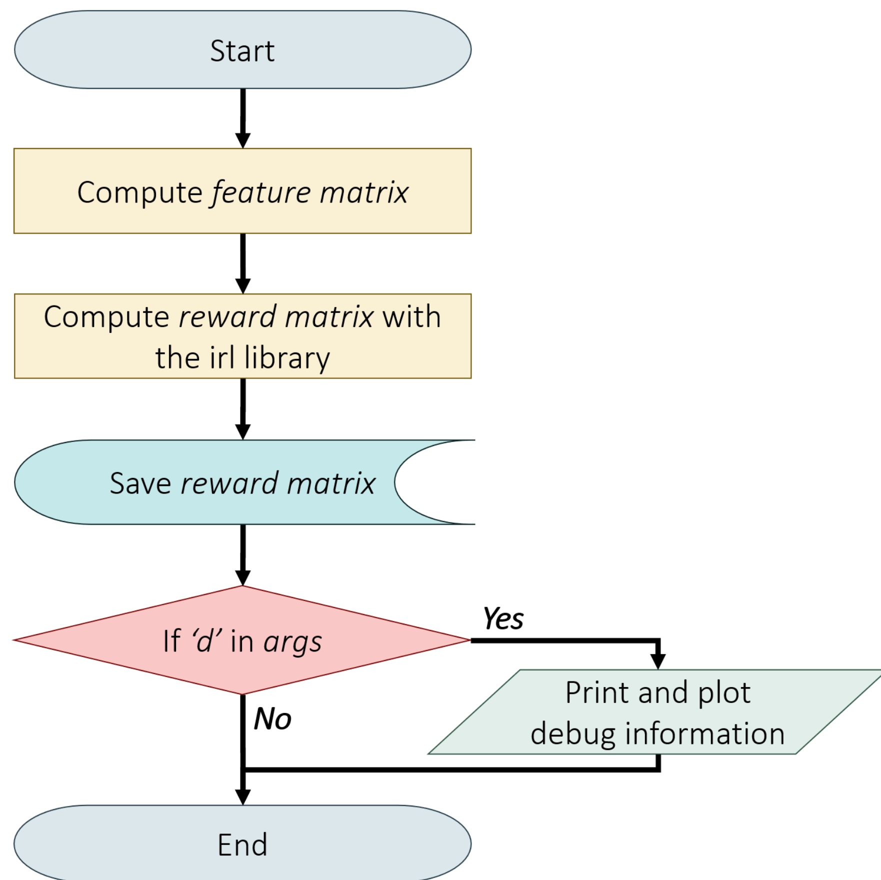


Figure B.5: *get_reward_function()* method's flowchart.

References

[1] Z. Liu, Y. Zhang, X. Yu, and C. Yuan, “Unmanned surface vehicles: An overview of developments and challenges,” *Annual Reviews in Control*, vol. 41, pp. 71–93, 2016.

[2] S. Azimi, J. Salokannel, S. Lafond, J. Lilius, M. Salokorpi, and I. Porres, “A survey of machine learning approaches for surface maritime navigation,” in *Maritime Transport VIII: proceedings of the 8th International Conference on Maritime Transport: Technology, Innovation and Research: Maritime Transport’20*, Universitat Politècnica de Catalunya. Departament de Ciència i Enginyeria ..., 2020, pp. 103–117.

[3] *Rethinking Autonomy And Safety*. [Online]. Available: `https://autonomous.fi/`.

[4] *One Sea*. [Online]. Available: `https://www.oneseaecosystem.net/`.

[5] *International Maritime Organisation*. [Online]. Available: `https://www.imo.org/en`.

[6] International Maritime Organization, *Colreg rules*, 1972. [Online]. Available: `https://www.imo.org/en/About/Conventions/Pages/COLREG.aspx`.

[7] A. M. Rothblum, “Human error and marine safety,” in *National Safety Council Congress and Expo, Orlando, FL*, 2000.

[8] J. E. Manley, “Unmanned surface vehicles, 15 years of development,” in *OCEANS 2008*, Ieee, 2008, pp. 1–4.

[9] *Mayflower Autonomous Ship*. [Online]. Available: `https://mas400.com/`.

[10] S. Penttinen, “Colreg compliant collision avoidance using reinforcement learning,” M.S. thesis, Åbo Akademi University, 2020.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[12] K. Hupponen, “A simulator for evaluating machine-learning algorithms for autonomous ships,” M.S. thesis, Åbo Akademi University, 2020.

[13] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[14] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 1.

[15] X. Bei, N. Chen, and S. Zhang, "On the complexity of trial and error," in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, 2013, pp. 31–40.

[16] M. E. Harmon and S. S. Harmon, "Reinforcement learning: A tutorial," 1997.

[17] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning methods," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–35, 2017.

[18] D. Silver, "Lecture 2: Markov decision processes," *UCL. Retrieved from www0. cs. ucl. ac. uk/staff/d. silver/web/Teaching_files/MDP. pdf*, 2015.

[19] Z. Lőrincz, *A brief overview of imitation learning*, 2019. [Online]. Available: `https://medium.com/@SmartLabAI/a-brief-overview-of-imitation-learning-8a8a75c44a9c`.

[20] B. D. Ziebart, A. L. Maas, J. A. Bagnell, and A. K. Dey, "Maximum entropy inverse reinforcement learning.," in *Aaai*, Chicago, IL, USA, vol. 8, 2008, pp. 1433–1438.

[21] M. Wulfmeier, P. Ondruska, and I. Posner, "Maximum entropy deep inverse reinforcement learning," *arXiv preprint arXiv:1507.04888*, 2015.

[22] J. Choi and K.-E. Kim, "Bayesian nonparametric feature construction for inverse reinforcement learning," in *Twenty-Third International Joint Conference on Artificial Intelligence*, Citeseer, 2013.

[23] S. Levine, Z. Popovic, and V. Koltun, "Nonlinear inverse reinforcement learning with gaussian processes," *Advances in neural information processing systems*, vol. 24, pp. 19–27, 2011.

[24] A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for q-learning in random stochastic mazes," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, 2016, pp. 1–8.

[25] *The design process*, Chicago Architecture Center. [Online]. Available: `https://discoverdesign.org/handbook`.

[26] *Aboa mare*. [Online]. Available: `https://www.aboamare.fi/`.

[27] *Visualisation of the expert dataset*. [Online]. Available: https://youtu.be/JqK4TiYQW1g?list=PLr_seq4zUPcMlG9wB63AiifJm1TrVIyWu.

[28] *OpenSeaMap*. [Online]. Available: http://map.openseamap.org/.

[29] M. Laskin, K. Lee, A. Stooke, L. Pinto, P. Abbeel, and A. Srinivas, "Reinforcement learning with augmented data," *arXiv preprint arXiv:2004.14990*, 2020. [Online]. Available: https://www.github.com/MishaLaskin/rad.

[30] *YAML Ain't Markup Language*. [Online]. Available: http://yaml.org/.

[31] A. S. Lenart, "Manoeuvring to required approach parameters-cpa distance and time," *Annual of Navigation*, pp. 99–108, 1999.

[32] *Numpy*. [Online]. Available: https://numpy.org/.

[33] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous distributed systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[34] F. Chollet *et al.*, *Keras*, 2015. [Online]. Available: https://github.com/fchollet/keras.

[35] *PySide2*. [Online]. Available: https://wiki.qt.io/Qt_for_Python.

[36] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[37] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688.

[38] *Keras*. [Online]. Available: https://keras.io/.

[39] M. Plappert, *Keras-rl*, 2016. [Online]. Available: https://github.com/keras-rl/keras-rl.

[40] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: `https://doi.org/10.1038/s41586-020-2649-2`.

[41] M. Alger, *Inverse reinforcement learning*, 2016. [Online]. Available: `https://doi.org/10.5281/zenodo.555999`.

[42] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[43] *Multiprocessing python module*. [Online]. Available: `https://docs.python.org/3/library/multiprocessing.html`.

[44] *Docs csc*. [Online]. Available: `https://docs.csc.fi/`.

[45] J. Leudet, F. Christophe, T. Mikkonen, and T. Männistö, "Ailivesim: An extensible virtual environment for training autonomous vehicles," in *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, IEEE, vol. 1, 2019, pp. 479–488.