Master's thesis

2020

Eivind Meyer

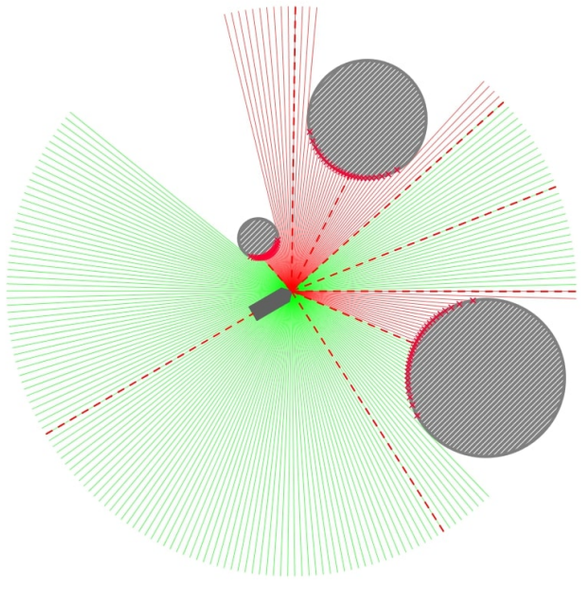
NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Engineering Cybernetics

Eivind Meyer

On Course Towards Model-Free Guidance

A Self-Learning Approach To Dynamic Collision Avoidance for Autonomous Surface Vehicles

May 2020



NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

On Course Towards Model-Free Guidance

A Self-Learning Approach To Dynamic Collision Avoidance for Autonomous Surface Vehicles

Eivind Meyer

MTTK
Submission date: May 2020
Supervisor: Adil Rasheed

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Samandrag

I denne masteroppgåva vart det demonstrert at djup forsterkande læring (engelsk: Deep Reinforcement Learning / DRL) kan nyttast for å trena eit reaktivt, autonomt fartøy utstyrt med påmonterte avstandssensorar til å navigera ukjent farvatn, kva omfattar ikkje berre ei utfordring om å unngå stranding medan ein går framover i samsvar med den ønskte ruta, men også dynamisk kollisjonsunngåelse, altså styringsstrategiar som minimerer risikoen for sammentreff i situasjonar der fartøyet er på kollisjonskurs med andre imøtekommande eller kryssande skip.

For dette formålet vart læringsalgoritmen PPO (engelsk Proximal Policy Optimication / PPO) nytta, som er sett på som ein leiande DRL-metode for anvendelser innan reguleringsteknikk av kontinuerleg natur. Den lærande agenten, som gjennom treningsprosessen vart rettleidd av ein belønningsfunksjon konstuert for å, på numerisk vis, gjenspegla preferansane våre for styringsåtferda til fartøyet, vart så evaluert basert på prestasjonen sin i eit virtuelt simuleringsmiljø som vart rekonstruert frå terreng- og maritime trafikkdata frå Trondheimsfjorden.

Summary

In this project, we show that Deep Reinforcement Learning (DRL) is applicable to the problem of training a reactive, autonomous vessel to navigate unknown waters, which entails not only the challenge of avoiding running ashore while efficiently making progress along the desired path, but also dynamic obstacle avoidance, i.e. control that mitigates collision risk upon ship encounters. A rangefinder sensor suite attached to the vessel, whose output is fed to the agent's control policy network, is designed, implemented in software and efficiently pre-processed to reduce the dimensionality of the perception vector while maintaining sensing integrity.

The contribution of this work is two-fold: First, we outline the design, implementation and training of the perception-based guidance agent, with the goal of making it capable of following priori known trajectories while avoiding collisions with other vessels. The reinforcement learning agent is trained to control the vessel's actuators, which include both thrusters as well as rudder control surfaces. A carefully constructed reward function, which balances the prioritization of path adherence versus that of collision avoidance (which can be considered competing objectives), is used to guide the agent's learning process. Then, the state-of-the-art Proximal Policy Optimization (PPO) DRL algorithm is utilized for training the agent's policy such that it, in the end, yields optimal actions with regards to maximizing the reward that the agent receives by the environment over time. Finally, we evaluate the trained agent's performance in challenging, dynamic test scenarios, including ones that are reconstructed from real-world terrain and maritime traffic data from the Trondheim Fjord, an inlet of the Norwegian sea.

Furthermore, The Python simulation framework **gym-auv**, which was developed to facilitate this research, has a vast potential to enable further research in the field, and is thus covered extensively in this thesis. It provides not only a software foundation that can be easily expanded by new environments, reward function designs and vessel models, but also access to high-quality plotting and reporting functionality as well as access to real-time (and recorded) video rendering in both 2D and 3D.

Preface

This thesis marks the finalization of my Master's degree in Cybernetics and Robotics at the Norwegian University of Technology (NTNU), and is written under the supervision of Professor Adil Rasheed, who did an exemplary job supporting me along the way - not only in terms of technical guidance, but also by encouraging me to publish my work in scientific journals.

I would also like to thank the anonymous reviewers of my first journal article[1] whose constructive criticism laid the foundation of this work which is once again under consideration in a reputed journal.

Furthermore, I am also grateful to Haakon Robinson for providing great constructive feedback, and to Anders Haver Vagle and Håkon Eckholdt for serving as excellent discussion partners, during the early phases of my work.

Finally, I am thankful to Amalie Heiberg for taking an active part in, and advancing the state of my research, to the point where she is co-authoring my second research paper[2].

29.05.2020, Trondheim

Eivind Meyer

[1] Meyer E; Robinson H; Rasheed A; San O, Taming an autonomous surface vehicle for path following and collision avoidance using deep reinforcement learning, IEEE, February 2020

[2] Meyer E; Heiberg A; Rasheed A and San O, COLREG-Compliant Collision Avoidance for Unmanned Surface Vehicle using Deep Reinforcement Learning, In preparation

Abstract

In this study, we explore the feasibility of applying Proximal Policy Optimization (PPO), a state-of-the-art Deep Reinforcement Learning (DRL) algorithm for continuous control tasks, on the dual-objective problem of controlling an underactuated Autonomous Surface Vehicle (ASV) to follow an a priori known path while avoiding collisions with dynamic obstacles, particularly other vessels, along the way. With no a priori knowledge of the environment except for the waypoints of its desired path, the agent makes reactive control decisions based on rangefinder sensors measuring the distance to nearby obstacles, be it static obstacles such as the shoreline or dynamic obstacles such as other vessels.

Furthermore, a software framework based on the OpenAI gym Python toolkit, in which AI-based ASVs can be simulated, trained and evaluated in a challenging, stochastically generated virtual environment, is developed. For the sake of demonstrating the potential of deploying the algorithm on a real-world vessel, the Trondheim Fjord area in Norway is reconstructed virtually based on high-fidelity terrain data. Furthermore, a sample of marine tracking data in the area is used to simulate realistic vessel encounters, so that the agent can be evaluated on its performance in real-world-like scenarios.

The excellent (i.e. collision-free) results that were obtained through software simulations clearly show that, without any prior knowledge about the dynamics governing the motion of a marine surface vehicle, a DRL-based agent is capable of learning how to navigate unknown waters, facing not only the danger of running ashore or stranding, but also challenging encounters with other moving vessels. By rewarding the agent for its performance in two separate, competing problem domains, namely those of path following and collision avoidance, we achieve an excellent level of guidance performance from the trained agent.

Contents

List of Tables

List of Figures

Abbreviations

AI	=	Artificial Intelligence
ASV	=	Autonomous Surface Vehicle
COLAV	=	Collision Avoidance
DL	=	Deep Learning
DNN	=	Deep Neural Network
DOF	=	Degrees Of Freedom
DRL	=	Deep Reinforcement Learning
GAE	=	Generalized Advantage Estimation
MDP	=	Markov Decision Process
ML	=	Machine Learning
NED	=	North-East-Down
PPO	=	Proximal Policy Optimization
RL	=	Reinforcement Learning
RNN	=	Recursive Neural Network
SNAME	=	Society of Naval Architects and Marine Engineers

Chapter 1

Introduction

Autonomy, i.e. the capacity of self-governance, offers an opportunity to improve the efficiency of transportation while reducing the frequency of maritime casualties. In order to realize safe and reliable autonomous surface vehicles (ASVs), however, effective path planning is a critical prerequisite. This is a highly complex challenge, as it requires not only the ability to follow an a prior known path, but also to make dynamic adjustments to the projected vessel path when encountering unforeseen obstacles, such as other vessels.

Deep Reinforcement Learning (DRL), a machine learning paradigm concerned with using deep learning for iteratively approximating optimal behavior policies in unknown environments, has gained a lot of traction in recent years following highly successful applications on similar problems, but is a yet largely unexplored approach to the task.

Autonomous vehicles is one of the most interesting prospects associated with the rise of artificial intelligence and machine learning in recent years. In particular, the success of DRL-based applications, in an ever-increasing number of domains, has contributed to putting the former pie-in-the-sky proposal of self-driving vehicles on the horizon of technological development.

While automated path following, at least in the maritime domain, has been a relatively trivial endeavor in the light of classical control theory and is a well-established field of research (4; 7; 46; 19; 23; 1; 52; 50), considerably more advanced capabilities are required to navigate unknown, dynamic environments; characteristics that, generally speaking, apply to the real world. Reactive collision avoidance, i.e. the ability to perform evasive maneuvers that mitigate collision risk based on a sensor-based perception of the local environment, remains a challenging undertaking.

This is not to say that the topic is not well-researched; a wide variety of approaches have

been proposed, including especially (but not exhaustively) artificial potential field methods (29; 3; 51), dynamic window methods (20; 5; 11), velocity obstacle methods (16; 31) and optimal control-based methods (8; 43; 12; 25; 2). Yet, it appears from a literature review that, when applied to autonomous vehicles with non-holonomic and real-time constraints, the approaches suggested so far suffer from one or more of the following drawbacks (69; 30; 47; 48):

- Unrealistic assumptions or neglect of the vessel dynamics.
- Inability to scale to environments of non-trivial complexity (e.g. multi-obstacle scenarios).
- Excessive computation time requirements.
- Disregard for desirable output trajectory properties, including smoothness, continuity, feasibility and safety.
- Incompatibility with external environmental forces such as wind, currents and waves.
- Stability issues caused by singularities.
- Sub-optimal outputs due to local minima.
- Requirement of a precise mathematical model of the controlled vessel.

Focusing on the maritime domain, this project will explore how a DRL-based approach can be used for training an end-to-end autopilot mechanism capable of avoiding collisions at sea. Given the potential of deep neural networks to generalize over the observation space, this is a particularly promising approach to vessel control.

In the simpler problem where path following and collision avoidance are treated as separate challenges, DRL-based methods have already demonstrated remarkable potential, yielding promising results in a multitude of studies, including especially (39; 38; 70; 68; 40) for the former problem domain and (24; 34; 71; 35) for the latter.

1.1 Motivation

Arguably, the most promising aspect of autonomous vessels is not the obvious economic impact resulting from increased efficiency and the replacement of costly human labor, but rather the potential to eliminate injuries and material damage caused by collisions. According to the European Maritime Safety Agency, which annually publishes statistics on maritime accidents related to the EU member states, maritime collisions account for hundreds of injuries each year in the EU alone, as shown in Figure 1.1. Furthermore, almost half of all casualties at sea are “navigational in nature, including contact, collision and grounding or stranding” (14), and 65.8% of the accidents can be attributed to human error (13), highlighting the value that autonomy can bring in this domain.

DRL-based methods, whose potential as a guidance system for ASVs is explored in this work, are fundamentally distinct from classical, model-based guidance approaches to autonomy. While providing highly desirable theoretical guarantees in terms of stability and robustness, existing methods typically require full knowledge of the non-linear dynamics governing the motion of the controlled vessel. As this often relies on costly system identification procedures for estimating the vessel parameters, a demonstration of the potential DRL-based guidance offers as an alternative will be valuable to the field of autonomous vessel guidance.

Validating a DRL-based approach to vessel guidance in a simulated, real-world-like environment can pave the way for applying the technology on a real, physical vessel. A positive result could be a preliminary step on the important path towards the adoption of AI systems for autonomous vessel guidance. Due to the limitations of existing methods, this has yet to take place on a larger scale.

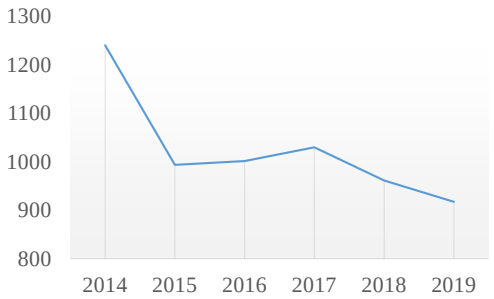


Figure 1.1: Human injuries per year according to maritime accident statistics published by the European Maritime Safety Agency. (13)

1.2 Report outline

In **Chapter 2**, we introduce the relevant background topics for this project. This will not only cover the vessel dynamics and existing approaches to collision avoidance, but also give a comprehensive introduction to DRL, starting with the fundamentals and, in a piece by piece fashion, describing the defining premises and concepts for the Proximal Policy Optimization (PPO) algorithm which is utilized in this work. Next, **Chapter 3** will outline the specifics of our methodology, i.e. describe and justify the design choices of our solution for a RL-based path following and collision avoidance agent, as well as the training and evaluation methodology. In **Chapter 4**, the Python software framework which facilitated the research will be presented in a detailed fashion, such that it can function as a reference for future research. The empirical test results obtained in this project will then

be presented and discussed in **Chapter 5** together with the project's conclusion, which also outlines interesting directions that future work based on this project might take.

Chapter 2

Background

In this chapter, relevant background topics for this project are introduced, most notably the theoretical foundation underlying the dynamics of a marine vessel, as well as deep reinforcement learning, which is the solution approach to our research problem.

2.1 Dynamics of a marine vessel

In this section, a brief description of the dynamics governing the motion of a marine vessel will be provided. For a more comprehensive overview of topic, the reader is referred to (18).

2.1.1 Coordinate frames

In order to model the dynamics of marine vessels, one must first define the coordinate frames. Two coordinate frames typically used in vehicle control applications are of particular interest: The geographical North-East-Down (NED) and body frames. The NED reference frame $\{n\} = (x_n, y_n, z_n)$ forms a tangent plane to the Earth's surface, making it useful for terrestrial navigation. Here, the x_n-axis is directed north, the y_n-axis is directed east and the z_n-axis is directed towards the center of the earth.

The origin of the body-fixed reference frame $\{b\} = (x_b, y_b, z_b)$ is fixed to the current position of the vessel in the NED-frame, and its axes are aligned with the heading of the vessel such that x_b is the longitudinal axis, y_b is the transversal axis and z_b is the normal axis pointing downwards. However, as the vessel is restricted to surface level motion in our application, only the North and East components are of interest.

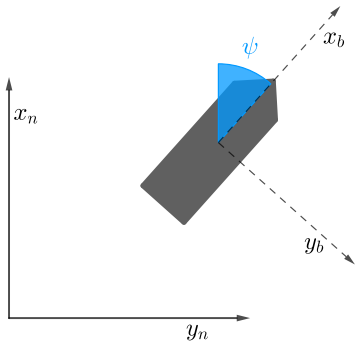


Figure 2.1: Illustration of the NED and body coordinate frames.

2.1.2 State variables

Assumption 1 (State space restriction)**.** *The vessel is always located on the surface, with no fluctuations in pitch and roll angle.*

Following Assumption 1, the state vector consists of the generalized coordinates $\boldsymbol{\eta} = [x^n, y^n, \psi]^T$, where x^n and y^n are the North and East positions, respectively, in the reference frame $\{n\}$, and ψ is the yaw angle, i.e. the current angle between the vessel's longitudinal axis x_b and the North axis x_n (59). Correspondingly, the translational and angular velocity vector $\boldsymbol{\nu} = [u, v, r]^T$ consists of the surge (i.e. forward) velocity u, the sway (i.e. sideways) velocity v as well as yaw rate r.

Symbol	Variable
x^n	North position in reference frame $\{n\}$
y^n	East position in reference frame $\{n\}$
ψ	Yaw corresponding to a Euler angle *zyx* convention from $\{n\}$ to $\{b\}$

Table 2.1: Generalized vessel coordinates (SNAME notation)

Symbol	Variable
u	Surge (i.e. speed along the x_b-axis)
v	Sway (i.e. speed along the y_b-axis
r	Yaw rate measured along the z_b-axis

Table 2.2: Body-frame velocities (SNAME notation)

2.1.3 Vessel model

We base the vessel dynamics on CyberShip II, a 1:70 scale replica of a supply ship which has a length of 1.255 m and mass of 23.8 kg (57). Training the DRL agent on a small vessel, such as CyberShip II, would allow for a relatively straight-forward deployment on a real-world model ship for further testing of the algorithm. However, the symbolic representation of the dynamics of a surface vessel, which is obtained from well-researched ship maneuvering theory, is the same regardless of the vessel - the distinctions lie solely in the numerical model parameters. Thus, if it can be demonstrated that an DRL agent can control a small-sized model ship in an intelligent manner, there is reason to believe that controlling a full-sized ship would be within its reach.

As it is equipped with rudders and propellers aft, as well as one bow thruster fore, Cybership II is a fully actuated ship. This means that it could, in principle, be commanded to follow an arbitrary trajectory in the state space, as it is able to accelerate independently in every relevant degree of freedom simultaneously. However, for the purpose of simplifying the DRL agent's action space, we disregard the bow thruster in this study and allow only the aft thrusters and control surfaces to be applied by the DRL agent as control signals. This omission is further motivated by the fact that bow thrusters have limited effectiveness at higher speeds (63). Thus, the control vector can be modelled as $\boldsymbol{f} = [T_u, T_r]^T$, where T_u represents the force input in surge and T_r represents the moment input in yaw.

Assumption 2 (Calm sea)**.** *There are no external disturbances to the vessel such as wind, ocean currents or waves.*

Given Assumption 2, the 3-DOF vessel dynamics can be expressed in a compact matrix-vector form as

$$\begin{aligned} \dot{\boldsymbol{\eta}} &= \mathbf{R}_{z,\psi}(\boldsymbol{\eta})\boldsymbol{\nu} \\ \mathbf{M}\dot{\boldsymbol{\nu}} + \mathbf{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \mathbf{D}(\boldsymbol{\nu})\boldsymbol{\nu} &= \mathbf{B}\boldsymbol{f} \end{aligned} \tag{2.1}$$

where $\mathbf{R}_{z,\psi}$ represents a rotation of ψ radians around the z_n-axis as defined by

$$\mathbf{R}_{z,\psi} = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Furthermore, $\mathbf{M} \in \mathbb{R}^{3\times3}$ is the mass matrix and includes the effects of both rigid-body and added mass, $\mathbf{C}(\boldsymbol{\nu}) \in \mathbb{R}^{3\times3}$ incorporates centripetal and Coriolis effects and $\mathbf{D}(\boldsymbol{\nu}) \in \mathbb{R}^{3\times3}$ is the damping matrix. Finally, $\mathbf{B} \in \mathbb{R}^{3\times2}$ is the actuator configuration matrix. The numerical values of the matrices are taken from (58), where the model parameters were estimated experimentally for CyberShip II in a marine control laboratory.

2.2 Deep reinforcement learning

In this section, we introduce the machine learning paradigm of Deep Reinforcement Learning, and outline the specific technique that our method builds on. For a more comprehensive coverage of the topic, the reader is advised to consult the book by Sutton and Barto (61), as well as the (55), where the PPO algorithm was first suggested.

On a very basic level, reinforcement learning (RL) establishes a framework for letting autonomous (i.e. self-governing) agents figure out how to ideally behave in their surroundings. Here, ”let learn”, as opposed to ”teach” is not an incidental word choice; a characterizing feature of reinforcement learning is that the learning is not instructive, as opposed to the related field of supervised learning. Instead, learning is accomplished through a combination of exploration and evaluative feedback, which, to some extent, resembles the manner by which people and other mammals learn (61) as they grow up; they become progressively more intelligent through experimentation with their surroundings, or stated otherwise, by virtue of trial-and-error.

Applications of RL on high-dimensional, continuous control tasks heavily rely on function approximators to generalize over the state space. Even if classical, tabular solution methods such as Q-learning can be made to work (provided a discretizing of the continuous action space), this is not considered an efficient approach for control applications (33). In recent years, given their remarkable generalization ability over high-dimensional input spaces, the dominant approach has been the application of deep neural networks which are optimized by means of gradient methods. There are, however, different approaches to how the networks are utilized, and thus their semantic interpretation in the context of the learning agent differs. In Q-Learning-based methods such as Deep Q-Learning (DQN) (45), a deep neural network is used to predict the expected value (i.e. long-term, cumulative reward) of state-action pairs, which reduces the policy to an optimization problem over the set of available actions given the current state. In gradient-based policy methods, on the other hand, the policy itself is implemented as a deep neural network whose weights are optimized by means of gradient ascent (or approximations thereof). Lately, several algorithms built on this principle have gained a large traction in the RL research community, most notably Deep Deterministic Policy Gradient (DDPG) (33), Asynchronous Advantage Actor Critic (A3C) (44) and Proximal Policy Optimization (PPO) (55). For continuous control tasks, this family of DRL methods is commonly considered to be the more efficient approach (64). Based on previous work, where the PPO algorithm significantly outperformed other methods on a learning problem similar to the one covered in this study (42), we focus our efforts on this method. PPO strikes a balance between data efficiency and ease of implementation, and is likely to be applicable in a wide range of continuous, high-dimensional control scenarios with relatively minor needs for hyperparameter tuning (55).

2.2.1 RL Preliminaries

First, we model the interplay between the agent and the environment as an infinite-horizon discounted Markov Decision Process (MDP), formally defined by the 6-tuple $(\mathcal{S}, \mathcal{A}, p, p_0, r, \Omega, o, \gamma)$

where

- $\mathcal{S}$ is the state space,
- $\mathcal{A}$ is the action space,
- $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ defines the conditional transition probabilities for the next state s' such that $p(s'|s, a) = Pr(S_{t+1} = s'|S_t = s, A_t = a)$,
- $p_0 : \mathcal{S} \to [0, 1]$ is initial state distribution, i.e. $p_o(s) = Pr(S_0 = s)$,
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ returns the numeric reward at each time-step as function of the current state and applied action,
- $\gamma \in [0, 1]$ is the discount factor for future rewards.

The agent draws its actions from its policy π. The policy may be a deterministic function (as in DDPG), but in the context of PPO, it is modelled as a stochastic function. The conditional action distribution given the current state s is given by $\pi(a|s) : \mathcal{S} \times \mathcal{A} \to\to [0, 1] = Pr(A_t = a|S_t = s)$. Specifically, we assume that the agent is drawing actions from a non-uniform multivariate Gaussian distribution whose mean is outputted by a neural network parametrized by the weights θ. Formally, this translates to $a_t \sim \pi(s_t)$, where t is the current time-step.

Next, we introduce the state-value function $V^\pi(s)$ and the action-value function $Q^\pi(s, a)$. $V^\pi(s)$ is the expected return from time t onwards given an initial state s, whereas $Q^\pi(s, a)$ is the expected return from time t onwards, but conditioned on the current action a_t. Formally, we have that

$$V^\pi(s_t) = \mathbb{E}_{s_{i\geq t}, a_{i\geq t} \sim \pi} [R_t|s_t] \tag{2.2a}$$

$$Q^\pi(s_t, a_t) = \mathbb{E}_{s_{i\geq t}, a_{i\geq t} \sim \pi} [R_t|s_t, a_t] \tag{2.2b}$$

where the random variable R_t represents the reward at time-step t.

2.2.2 Policy gradients

The stochasticity of the policy enables us to translate the RL problem, i.e. the search for the optimal policy, into the problem of optimizing the expectation

$$J(\theta) = \mathbb{E}_{s_i, a_i \sim \pi(\boldsymbol{\theta})} [R_0] \tag{2.3}$$

The family of policy gradient methods, to which PPO belongs, approach gradient ascent by updating the parameter vector θ according to the approximation $\theta_{t+1} \leftarrow \alpha\theta_t + \widehat{\nabla_\theta J(\theta)}$, where $\widehat{\nabla_\theta J(\theta)}$ is a stochastic estimate of $\nabla_\theta J(\theta)$ satisfying $\mathbb{E}\left[\widehat{\nabla_\theta J(\theta)}\right] = \nabla_\theta J(\theta)$. From the policy gradient theorem (60) we have that the policy gradient $\nabla_\theta J(\theta)$ satisfies

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a \nabla_\theta \pi(a|s) Q^\pi(s, a) \tag{2.4}$$

where μ is the steady state distribution under π such that $\mu(s) = \lim_{t\to\infty} Pr\{S_t = s|A_{0:t-1} \sim \pi\}$. Following the steps outlined in (62), this can be algebraically transformed to

$$\nabla_\theta J(\theta) \propto \mathbb{E}_\pi[\nabla_\theta \ln \pi(A_t|S_t) Q^\pi(S_t, A_t)] \tag{2.5}$$

Also, it can be shown that one can greatly reduce the variance of this expression by replacing the state-action value function $Q^\pi(s,a)$ in Equation 2.4 by $Q^\pi(s,a) - b(s)$, where the **baseline** function $b(s)$ can be an arbitrary function not depending on the action a, without introducing a bias in the estimate. Commonly, $b(s)$ is set to be the state value function V^π, which yields the **advantage** function

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{2.6}$$

which represents the expected improvement obtained by an action compared to the default behavior. This leads to

$$\nabla_\theta J(\theta) \propto \mathbb{E}_\pi[\nabla_\theta \log \pi(A_t|S_t) A^\pi(s,a)] \tag{2.7}$$

Thus, an unbiased empirical estimate based on N episodic policy rollouts of the policy gradient $\nabla_\theta J(\theta)$ is

$$\widehat{\nabla_\theta J(\theta)} = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=0}^{\infty} \hat{A}_t^n \nabla_\theta \log \pi(a_t^n|s_t^n) \tag{2.8}$$

$A^\pi(s,a)$ is, like $Q^\pi(s,a)$ and $V^\pi(s)$, unknown, and must thus be estimated by the function approximator $\hat{A}(s)$. Generalized Advantage Estimation (GAE), as proposed in (54), is the most notable approach. GAE makes use of a function approximator (commonly a neural network) $\hat{V}(s)$ to approximate the actual value function $V(s)$. A common approach is to use an artificial neural network, which is trained on the discounted empirical returns.

2.2.3 Proximal policy optimization

PPO, as well as its predecessor (Trust Region Policy Optimization (53)) do not, even though it is feasible, optimize the policy directly via the expression in Equation 2.8. TRPO instead optimizes the surrogate objective function

$$J^{CPI}(\theta') = \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \hat{A}_t^{\pi_\theta} \right] \tag{2.9}$$

which provides theoretical guarantees for policy improvement. However, as this relies on an approximation that is valid only in the local neighborhood, carefully choosing the step size is critical to avoid instabilities. Unlike in TRPO, where this is achieved by imposing a hard constraint on the relative entropy between the current and next policy, PPO elegantly incorporates the preference for a modest step-size in the optimization target, yielding a more efficient algorithm (55). Specifically, it instead focuses on maximizing

$$\begin{aligned} J^{CLIP}(\theta') &= \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta) \hat{A}_t^{\pi_\theta}, \text{clip}_\epsilon \left(r_t(\theta) \right) \hat{A}_t^{\pi_\theta} \right) \right] \\ \text{clip}_\epsilon(x) &= \text{clip}\left(x, 1-\epsilon, 1+\epsilon\right) \end{aligned} \tag{2.10}$$

where $r_t(\theta)$ is a shorthand for the probability ratio $\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}$.

The training process, which is written in pseudocode format in Algorithm 1, can then be summarized as follows: At each iteration, PPO first collects batches of Markov trajectories from concurrent rollouts of the current policy. Next, the policy is updated according to a stochastic gradient descent update scheme.

Algorithm 1 Proximal Policy Optimisation

for iteration = $1, 2, ...$ **do**
  for actor = $1, 2, ...N$ **do**
    For T time-steps, execute policy π_θ.
    Compute advantage estimates $\hat{A}_1, \dots \hat{A}_T$
  for epoch = $1, 2, ...N_E$ **do**
    Obtain mini batch of N_{MB} samples from the $N_A T$ simulated time-steps.
    Perform gradient descent update from minibatch $(\mathbf{X}_{MB}, \mathbf{Y}_{MB})$.
    $\theta \leftarrow \theta'$

2.3 Tools and libraries

The code implementation of our solution make use of the RL framework provided by the Python library **OpenAI Gym** (6), which was created for the purpose of standardizing the benchmarks used in RL research. It provides a easy-to-use framework for creating RL environments in which custom RL agents can be deployed and trained with minimal overhead.

Stable Baselines (27), another Python package, provides a large set of state-of-the-art parallelizable RL algorithms compatible with the OpenAI gym framework, including PPO. The algorithms are based on the original versions found in OpenAI Baselines (10), but Stable Baselines provides several improvements, including algorithm standardization and exhaustive documentation.

The most challenging aspect of the simulation, which is the calculation of the intersection points between the sensor rays and the boundaries of the nearby obstacles, is handled efficiently by the **shapely** Python library (56), which offers an easy-to-use interface to a wide range of geometric analysis-related operations.

The generation of continuous, smooth parameterized path is done using 1D Piecewise Cubic Hermite Interpolator (PCHIP), which is provided by the Python library **SciPy** (67), which offers a wide range of scientific computing methods.

2.3.1 Terrain data

Our maritime simulation environment is made from a digital reconstruction of the Trondheim Fjord (Figure 2.2), an inlet of the Norwegian sea. Specifically, it is based on a digital

Figure 2.2: Map of the Norwegian mainland highlighting the area of interest.[1]

terrain model (DTM) provided by the Norwegian Mapping Authority (Kartverket). The data set, which is called DTM10, is generated from airborne laser scanning, and has a horizontal resolution of 10x10 meters with coverage of the entire Norwegian mainland (49). The coordinates are given according to the Universal Transverse Mercator (UTM) rectangular projection system, which partitions the Earth into 60 north-south zones, each of which has a 6 degree longitudinal span. Within each zone, which is indexed consecutively from zone 1 (180°W to 174°W) to zone 60 (174°E to 180°E), a mapping from latitude/longitude coordinates to a Cartesian x-y coordinate system is performed based on a local flat earth-assumption. Given the vast number of zones used in the UTM projection system, the approximated coordinates, which of course have inherent distortions because of the spherical shape of the Earth, are of relatively high accuracy. The DTM10 data set is given with respect to zone 33.

2.3.2 Tracking data

We obtain a sample of historical vessel tracking data in the Trondheim Fjord area from a query of the Norwegian Coastal Administration's AIS Norway data service. The automatic identification system (AIS) is an automatic tracking system which provides both static (e.g.

[1] Original image source: NordNordWest (`https://commons.wikimedia.org/wiki/File:Norway_location_map.svg`), "Norway location map"

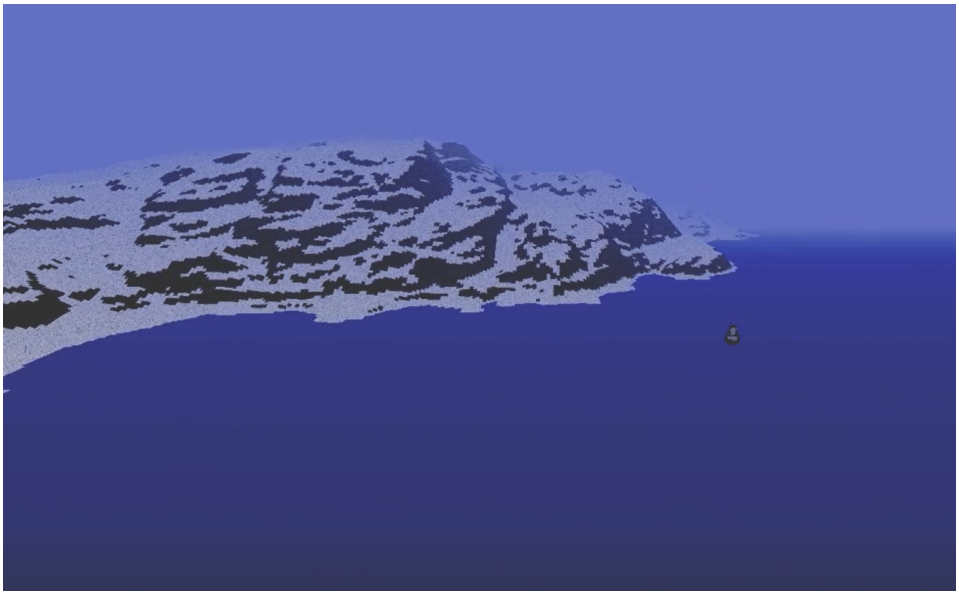


Figure 2.3: Digital terrain reconstructed from DTM10 (Norwegian Mapping Authority) rendered in 3D for debugging and showcasing purposes. Specifically, this shows a view of the Bymarka area, a nature reserve on the west side of Trondheim.

vessel dimensions) and dynamic (e.g. vessel position, heading and speed) information based on vessel transmissions. Within the field of autonomous surface vehicle guidance, AIS information is often used as a supplementary data source that is, by method of sensor fusion, combined with marine radar in collision avoidance algorithms. Additionally, given a large enough sample time within the area of interest, it provides a historical model of the marine traffic in the area. In our case, our historical data results from a 10 day data query ranging from January 26, 2020 to February 6, 2020 of all recorded traffic (Figure 2.4) within a rectangular area around the Trondheim Fjord.

Depending on the transmitter characteristics for each individual vessel, the resulting tracking data resolution varies from 2-20 seconds, facilitating a high-accuracy reconstruction of each vessel's trajectory in our simulation. As the AIS tracking data represents vessel position by latitude/longitude coordinates, a conversion to the zone 33 UTM x-y coordinate system is called for. To do the conversion, we utilize the *from_latlon* method provided by the Python package **utm** (65).

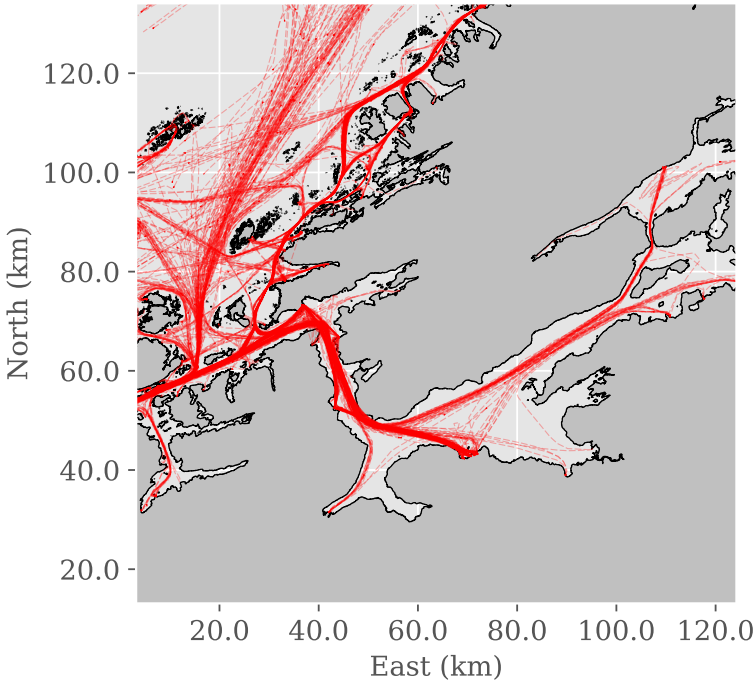


Figure 2.4: Snapshot of the marine traffic from January 26, 2020 to February 6, 2020 in the Trondheim Fjord area based on AIS tracking data. Each red line represents one recorded travel.

Chapter 3

Methodology

In this chapter, the design and implementation of our DRL controller will be covered. First, we outline the procedure for generating the training environment in which the DRL agent optimises its policy - a randomly created obstacle environment whose purpose is to prepare the agent for real-world-like testing. Second, we, outline in detail how the observation vector, which is the agent's perception of the environment, is engineered. Notably, this entails both path information as well as sensor-based distance measurements of the local obstacle neighborhood. Also, we justify and present our reward function design. As for all RL applications, engineering the reward function is critical to achieving the desired behavior from the trained agent.

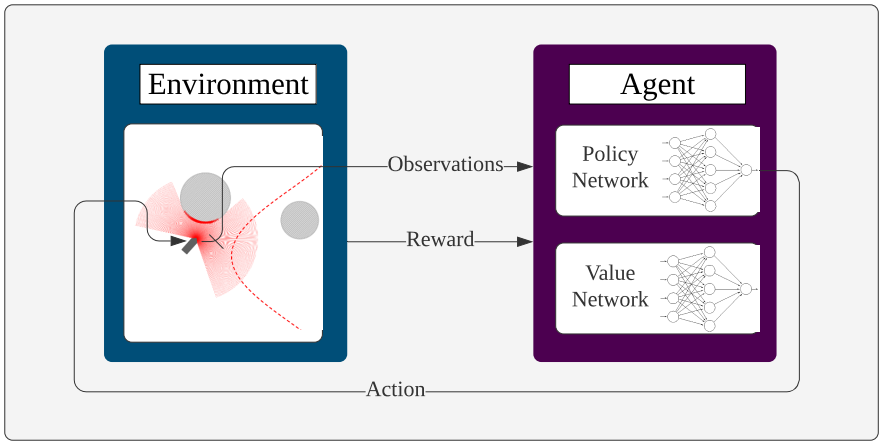


Figure 3.1: Flowchart outlining the structure of the guidance system explored in this study. At each time-step, the agent receives an observation vector, and then, according to its policy, which is implemented as a neural network, outputs an action (i.e. control vector), influencing the state of the simulated environment. During training, the agent's policy is continuously improved by means of gradient ascent based on the reward signal that it receives at each time-step. This constant feedback enables the agent, whose policy is initially nothing more than a clean slate with no intelligent characteristics, to improve its capabilities through a trial-and-error based approach. Its learning objective is simple: Find the policy that yields the highest expectation of the agent's long-term future reward.

3.1 Training environment

DRL extends the utility-maximizing self-improvement concept of reinforcement learning by the generalization potential of deep neural networks. For that reason, DRL-based autonomous agents have a remarkable ability to generalize their policy over the observation space, including the domain of unseen observations. And given the complexity and heterogeneity of the Trondheim Fjord environment, with archipelagos, shorelines and skerries (as seen in Figure 2.4), this ability will be fundamental to the agent's performance. However, the training environment, in which the agent is supposed to evolve from a blank slate to an intelligent vessel controller, must be both representative, challenging and unpredictable to facilitate the generalization.

Of course, the most representative choice for a training scenario would be the Trondheim Fjord itself, which would, if it was not for the generalization issues associated with this approach (9), also allow for training the agent via behavior cloning based on the available vessel tracking data. However, given the resolution of our terrain data, the resulting obstacle geometry is typically very complex, leading to overly high computational demands for simulating the functioning of the agent's visual perception system.

Thus, the better choice is to carefully craft an artificial training scenario with simple obsta-

cle geometries. To reflect the dynamics of a real marine environment, we let the stochastic initialization method of the training scenario spawn other target vessels with deterministic, linear trajectories. Additionally, circular obstacles, which are scattered around the environment, are used as a substitute for real-world terrain. Specifically, we generate new environments according to Algorithm 3 (found in the Appendix), a random output sample of which is shown in Figure 3.2. The parameters used for generating the scenario are listed in Table 3.1.

Parameter	Description	Initialization
$N_{o,stat}$	Number of static obstacles	20
$N_{o,dyn}$	Number of dynamic obstacles	35
N_w	Number of path waypoints	$\sim$ *Uniform*$(2, 5)$
L_p	Path length	8000 m
$\mu_{r.stat}$	Mean static obstacle radius	300 m
$\mu_{r.dyn}$	Mean moving obstacle radius	100 m
σ_d	Obstacle displacement distance standard deviation	3000 m

Table 3.1: Parameters used for generating training environment with moving obstacles.

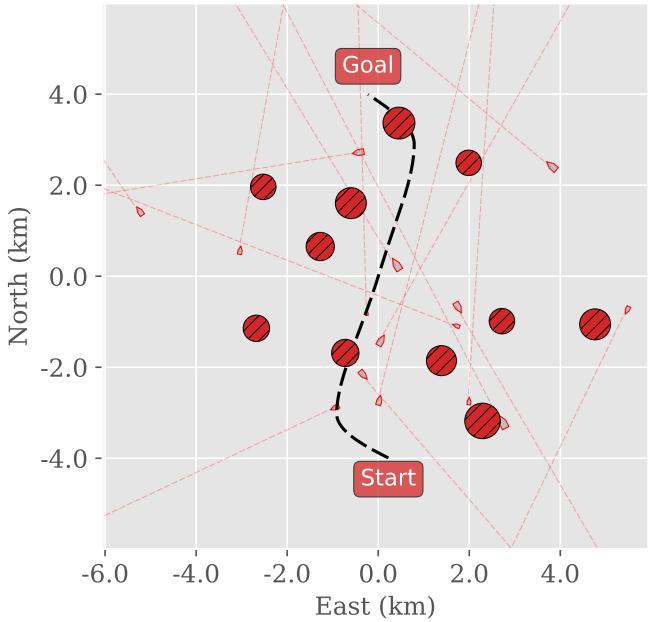


Figure 3.2: Random sample of the stochastically generated path following training scenario with moving obstacles. The circles are static obstacles, whereas the vessel-shaped objects are moving according to the trajectory lines.

3.2 Observation vector

Here, the goal is to engineer an observation vector s containing sufficient information about the vessel's state relative to the path, as well as information from the sensors. To achieve this, the full observation vector is constructed by concatenating navigation-based and perception-based features, which formally translates to $s = [s_n, s_p]^T$. In the context of this paper, we consider the term *navigation* as the characterization of the vessel's state, i.e. its position, orientation and velocity, with respect to the desired path. On the other hand, *perception* refers to the observations made via the rangefinder sensor measurements. In the following, the path navigation feature vector s_n and the elements culminating in the perception-based feature vector s_p are covered in detail.

3.2.1 Path navigation

A sufficiently information-rich path navigation feature vector would be such that it, on its own, could facilitate a satisfactory path-following controller (without any consideration for obstacle avoidance). A few concepts often used in the field of vessel guidance and control are useful in order to formalize this.

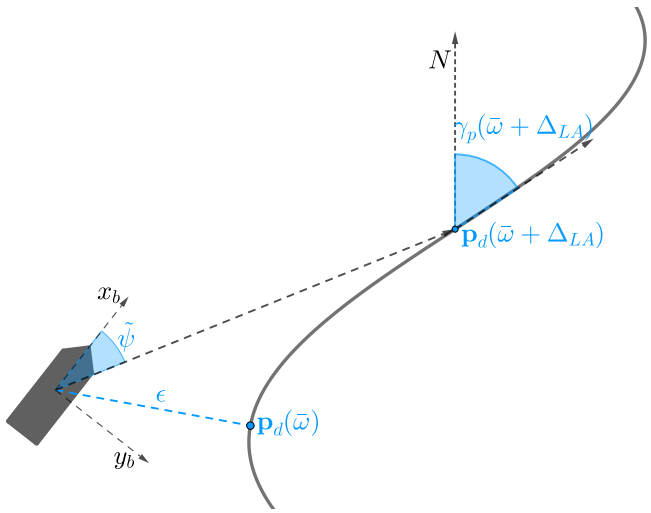


Figure 3.3: Illustration of key concepts for navigation with respect to path following. The path reference point $\boldsymbol{p}_d(\omega)$, i.e. point yielding the closest Euclidean distance to the vessel, is here located right of the vessel, while the look-ahead reference point $\boldsymbol{p}_d(\bar{\omega} + \Delta_{LA})$ is located a distance Δ_{LA} further along the path.

First, we introduce the mathematical representation of the parameterized path, which is

expressed as

$$\boldsymbol{p}_d(\omega) = [x_d(\omega), y_d(\omega)]^T \tag{3.1}$$

where $x_d(\omega)$ and $y_d(\omega)$ are given in the NED-frame. Navigation with respect to the path necessitates a reference point on the path which is continuously updated based on the current vessel position. Even though other approaches exist, this reference point is best thought of as the point on the path that has the closest Euclidean distance to the vessel, given its current position, as visualised in the example illustration shown in Figure 3.3. To find this, we calculate the corresponding value of the path variable $\bar{\omega}$ at each time-step. This is an equivalent problem formulation because the path is defined implicitly by the value of ω. Formally, this translates to the optimization problem

$$\bar{\omega} =_{\omega} (x^n - x_d(\omega))^2 + (y^n - y_d(\omega))^2 \tag{3.2}$$

Which, using the Newton–Raphson method, can be calculated accurately and efficiently at each time-step. Here, the fact that the Newton–Raphson method only guarantees a local optimum is a useful feature, as it prevents sudden path variable jumps given that the previous path variable value is used as the initial guess (37).

Accordingly, we define the corresponding Euclidean distance to the path, i.e. the deviation between the desired path and the current track, as the cross-track error (CTE) ϵ. Formally, we thus have that

$$\epsilon = \left\| [x^n, y^n]^T - \boldsymbol{p}_d(\bar{\omega}) \right\| \tag{3.3}$$

Next, we consider the look-ahead point $\boldsymbol{p}_d(\bar{\omega} + \Delta_{LA})$ to be the point which lies a constant distance further along the path from the reference point $\boldsymbol{p}_d(\bar{\omega})$. The parameter Δ_{LA}, the look-ahead distance, is set by the user and controls how aggressively the vessel should reduce the distance to the path. Look-ahead based steering, i.e. setting the look-ahead point direction as the desired course angle, is a commonly used guidance principle (18).

We then define the heading error $\tilde{\psi}$ as the change in heading needed for the vessel to navigate straight towards the look-ahead point from its current position, as illustrated in Figure 3.3. This is calculated from

$$\tilde{\psi} = \text{atan2}\left(\frac{y_d(\bar{\omega} + \Delta_{LA}) - y^n}{x_d(\bar{\omega} + \Delta_{LA}) - x^n}\right) - \psi \tag{3.4}$$

where ψ is the vessel's current heading and x^n, y^n are the current NED-frame vessel coordinates as defined earlier.

However, even if minimizing the heading error will yield good path adherence, taking into account the path direction at the look-ahead point might improve the smoothness of the resulting vessel trajectory. Referring to the first order path derivatives as $x'_p(\bar{\omega})$ and $y'_p(\bar{\omega})$, we have that the path angle γ_p, in general, can be expressed as a function of arc-length ω such that

$$\gamma_p(\bar{\omega}) = \text{atan2}\left(y'_p(\bar{\omega}), x'_d(\bar{\omega})\right) \tag{3.5}$$

As visualized in Figure 3.3, the path direction at the look-ahead point is then given by $\gamma_p(\bar{\omega} + \Delta_{LA})$. Accordingly, we can then define the look-ahead heading error, which is zero in the case when the vessel is heading in a direction that is parallel to the path direction at the look-ahead point, as

$$\tilde{\psi}_{LA} = \gamma_p(\bar{\omega} + \Delta_{LA}) - \psi \tag{3.6}$$

Our assumption is then that the navigation feature vector s_n, defined as outlined in Table 3.2, should provide a sufficient basis for the agent to intelligently adhere to the desired path.

Feature	Definition
Surge velocity	$u^{(t)}$
Sway velocity	$v^{(t)}$
Yaw rate	$r^{(t)}$
Cross-track error	$\epsilon^{(t)}$
Heading error	$\tilde{\psi}^{(t)}$
Look-ahead heading error	$\tilde{\psi}_{LA}^{(t)}$

Table 3.2: Path-following feature vector s_n at timestep t.

Formally, we thus have that

$$s_n^{(t)} = \left[u^{(t)}, v^{(t)}, r^{(t)}, \epsilon^{(t)}, \tilde{\psi}^{(t)}, \tilde{\psi}_{LA}^{(t)}\right]^T \tag{3.7}$$

3.2.2 Sensing

Using a set of rangefinder sensors (i.e. a distance-measuring sensors) as the basis for obstacle avoidance is a natural choice, as it yields a comprehensive, easily interpretable representation of the neighbouring obstacle environment. This should also enable a relatively straightforward transition from the simulated environment to a real-world one, given the availability of common rangefinder sensors, be it lidars, radars, sonars or depth cameras. The set of distance sensors, commonly referred to as the *sensor suite*, has one particularly desirable characteristic: At close distances, where high perception precision is needed for making swift and safe guidance decisions, the spot resolution, i.e. the distance between the sensor beams, is less than for higher distances. The spot resolution $\sigma_{res} : \mathbb{R} \mapsto \mathbb{R}$ increases as a linear function of distance according to

$$\sigma_{res}(d) = \frac{2\pi d}{N} \tag{3.8}$$

which is plotted from $d = 0$ to $d = S_r$ in Figure 3.4. Practically speaking, this allows for high-fidelity optic sensing of the nearby obstacle environment, while the observations at larger distances are more scattered.

In our setup, the vessel is equipped with N distance sensors with a maximum detection range of S_r, which are distributed uniformly with 360 degree coverage, as illustrated in

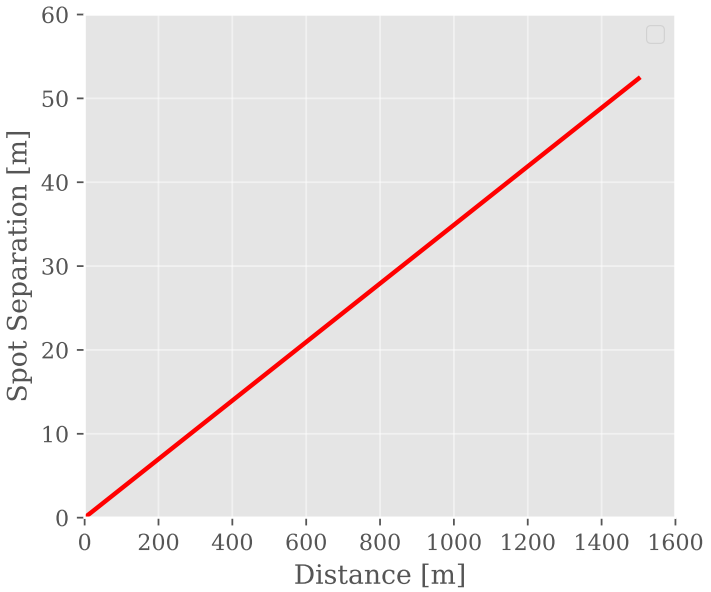


Figure 3.4: Spot separation resolution of rangefinder sensor suite.

Figure 3.5. While the area behind the vessel is obviously of lesser importance, and not necessary to consider for navigating purely static terrain, the possibility of overtaking situations where the agent must react to another vessel approaching from behind makes full sensor coverage a necessity.

3.2.3 Sensor partitioning

The most natural approach to constructing the final observation vector would then be to concatenate the path information feature vector with the array of sensor outputs. However, initial experiments with this approach were aborted as it became apparent that the training process had stagnated - at a very dissatisfactory agent performance level. A likely explanation for this failure is the size of the observation vector which was fed to the agent's policy and value networks; as it becomes overly large, the agent suffers from the well-known *curse of dimensionality*. Due to the resulting network complexity, as well as the exponential relationship between the dimensionality and volume of the observation space, the agent fails to generalize new, unseen observations in an intelligent manner (21). This calls for a significant dimensionality reduction. This can, of course, be achieved simply by reducing the number of sensors, something which would also have the fortunate side effect of reducing the simulation's computational needs. Unfortunately, this approach also turned out unsuccessful, even after testing a wide range of smaller sensor setups. Clearly, when the sensor count becomes too low, the agent's perception of the neighboring obstacle environment is simply too scattered to facilitate satisfactory obstacle-avoiding behavior in

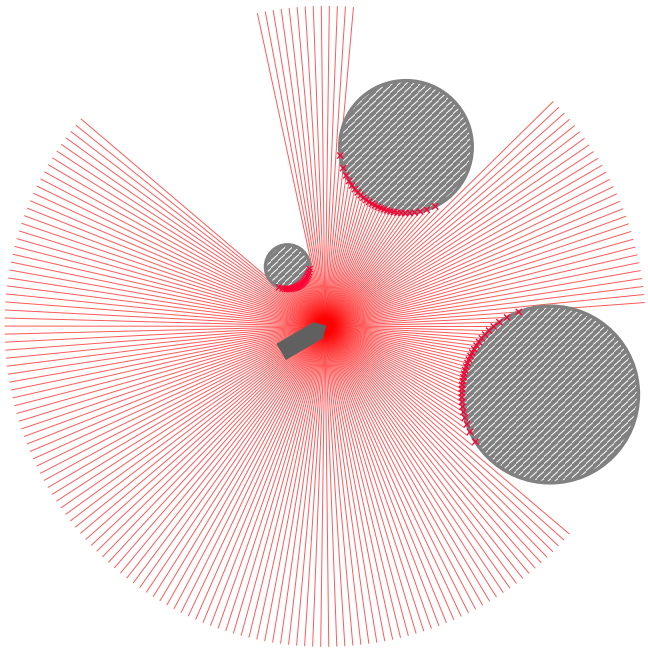


Figure 3.5: Rangefinder sensor suite attached to autonomous surface vessel.

challenging scenarios such as the ones used for training the agent. As balancing the trade-off between sensor resolution and observation dimensionality appears intractable, this calls for a more involved approach.

A natural approach is to partition the sensor suite into D sectors, each of which produces a scalar measurement which is included in the final observation vector, effectively summarizing the local sensor readings within the sector. However, given our desire to minimize its dimensionality, dividing the sensors into sectors of uniform size is likely sub-optimal, as obstacles located in front of the vessel are significantly more critical and thus require a higher degree of perception accuracy than those that are located at its rear. In order to realize such a non-uniform partitioning, we use a logistic function - a choice that also fulfills our general preference for symmetry. Assuming a counter-clockwise ordering of sensors and sectors starting at the rear of the vessel, we map a given sensor index $i \in N$ to sector index $k \in D$ according to

$$\kappa : i \mapsto \kappa(i) = \left\lfloor \underbrace{D\sigma\left(\frac{\gamma_C i}{N} - \frac{\gamma_C}{2}\right)}_{\text{Non-linear mapping}} - \underbrace{D\sigma\left(-\frac{\gamma_C}{2}\right)}_{\text{Constant offset}} \right\rfloor \tag{3.9}$$

where σ is the logistic sigmoid function and γ_C is a scaling parameter controlling the density of the sector distribution such that decreasing it will yield a more evenly distributed

partitioning. In Figure 3.6, the practical output of this sensor mapping procedure is visualised, with the sectors being the narrowest near the front of the vessel. We can then formally define the distance measurement vector for the k^{th} sector, which we denote by $\boldsymbol{w}_k$, according to

$$\boldsymbol{w}_{k,i} = x_i \qquad \text{for } i \in N \text{ such that } \kappa(i) = k$$

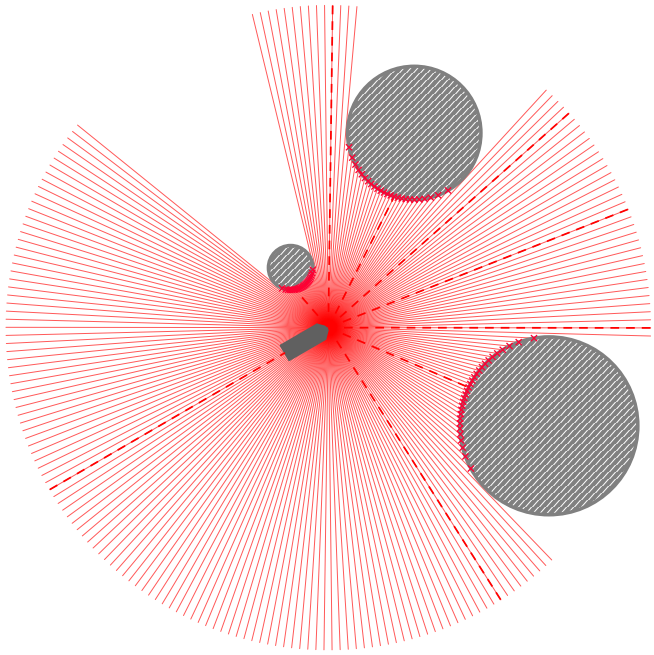
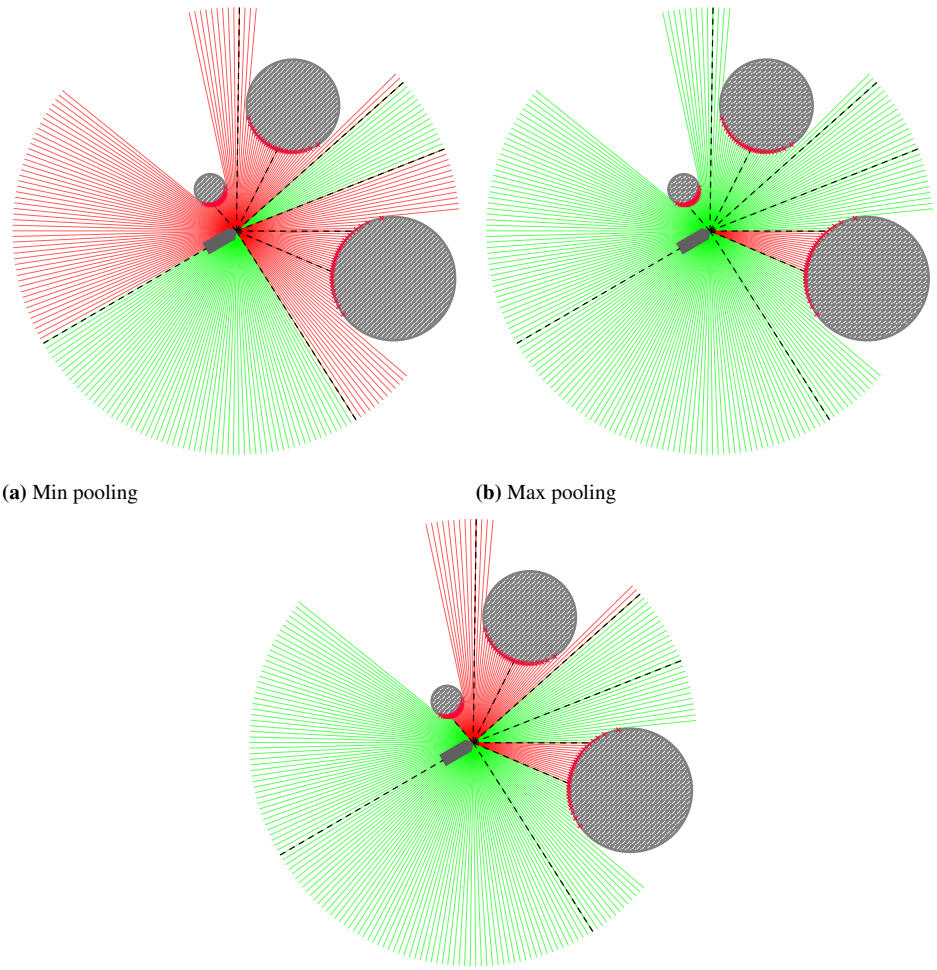


Figure 3.6: Rangefinder sensor suite partitioned into $D = 9$ sectors according to the the mapping function κ with the scale parameter $\gamma_C = 0.13$.

Next, we seek a mapping $f : \mathbb{R}^n \mapsto \mathbb{R}$, which takes the vector of distance measurements $\boldsymbol{w}_k$, for an arbitrary sector index k, as input, and outputs a scalar value based on the current sensor readings within the sector. Always returning the smallest measured obstacle distance within the sector, i.e. $f = \min$ (in the following referred to as *min pooling*), is a natural approach which yields a conservative and thereby safe observation vector. As can be seen in Figure 3.7a, however, this approach might be overly restrictive in certain obstacle scenarios, where feasible openings in between obstacles are inappropriately overlooked. However, even if the opposite approach (*max pooling*, i.e. $f = \max$) solves this problem, it is straight-forward to see, e.g. in Figure 3.7b by considering the fact that the presence of the small obstacle near the vessel is ignored, that it might lead to dangerous navigation strategies.



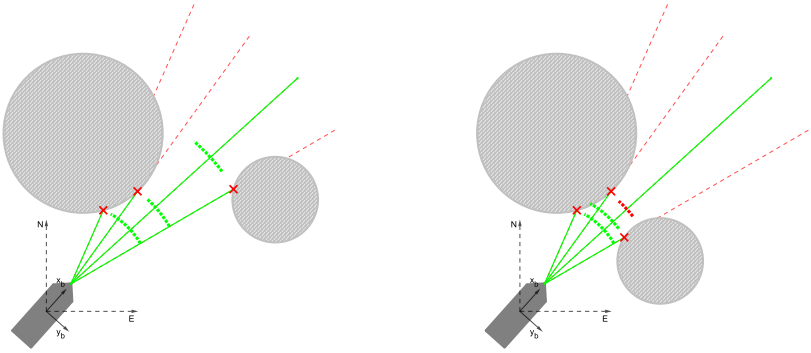
(a) Min pooling

(b) Max pooling

(c) Feasibility pooling

Figure 3.7: Pooling techniques for sensor dimensionality reduction. For the sectors colored green, the maximum distance S_r was outputted, implying that the sector is clear of any obstacles. It is obvious that min-pooling yields an overly restrictive observation vector, effectively telling the agent that a majority of the travel directions are blocked. On the other hand, max pooling yields overly optimistic estimates, potentially leading to dangerous situations. The feasibility pooling algorithm, however, mirrors an intuitive reasoning about the reachability within each sector, producing a more intelligent estimate.

In order to alleviate the problems associated with min and max pooling mentioned above, a new approach is required. The *feasibility pooling* procedure calculates the the maximum reachable distance within each sector, taking into account the location of the obstacle sensor readings as well as the width of the vessel. This method requires us to iterate over the



(a) Full distance is reachable.

(b) Less than half the distance is reachable.

Figure 3.8: Illustration of the feasibility algorithm for two different scenarios. After sorting the sensor indices according to the corresponding distance measurements, the algorithm iterates over them in ascending order, and, at each step, decides if the vessel can feasibly continue past this point. In the scenario displayed in the figure on the right, the opening is deemed too narrow for the full distance to be reachable.

sensor reading in ascending order corresponding to the distance measurements, and for each resulting distance level check whether it is feasible for the vessel to advance beyond this level. As soon as the widest opening available within a distance level is deemed too narrow given the width of the vessel, the maximum reachable distance has been reached. Formally, we define f to be the algorithm outlined in Algorithm 2.

Having a quadratic runtime complexity, the feasibility pooling algorithm is slower than simple max or min pooling, which both can be executed in a linear fashion. In Figure 3.9, empirical runtime estimates are reported for $n = 9$. Given our modest sensor setup, however, this is not a major concern, as the increased computation requirements are far from rivaling those of simulating the functioning of the rangefinder sensors.
Another interesting aspect to consider when comparing the pooling methods, is the sensitivity to sensor noise. A compelling metric for this is the degree to which the pooling output differs from the original noise-free output when normally distributed noise with standard deviation σ_w is applied to the sensors. Specifically, we report the root mean square of the differences between the original pooling outputs and the outputs obtained from the noise-affected measurements. The results for $\sigma_w \in \{1, \ldots, 30\}$ are presented in Figure 3.10. Evidently, the proposed feasibility method for pooling is slightly more robust than the other variants.

Algorithm 2 Feasibility pooling for rangefinder sensors.

Require:
Vessel width $W \in \mathbb{R}^+$
Angle between neighboring sensors θ
Sensor rangefinder measurements for current sector $\boldsymbol{x} = \{x_1, \ldots, x_n\}$
procedure FEASIBILITYPOOLING($\boldsymbol{x}$)
Initialize $\mathcal{I}$ to be the indices of $\boldsymbol{x}$ sorted in ascending order according to the measurements x_i
for $i \in \mathcal{I}$ **do**
Arc-length $d_i \leftarrow \theta x_i$
Opening-width $y \leftarrow d_i/2$
Opening was found $s_i \leftarrow false$
for $j \leftarrow 0$ to n **do**
if $x_j > x_i$ **then**
$y \leftarrow y + d_i$
if $y > W$ **then**
$s_i \leftarrow true$
break
else
$y \leftarrow y + d_i/2$
if $y > W$ **then**
$s_i \leftarrow true$
break
$y \leftarrow 0$
if s_i is $false$ **then return** x_i

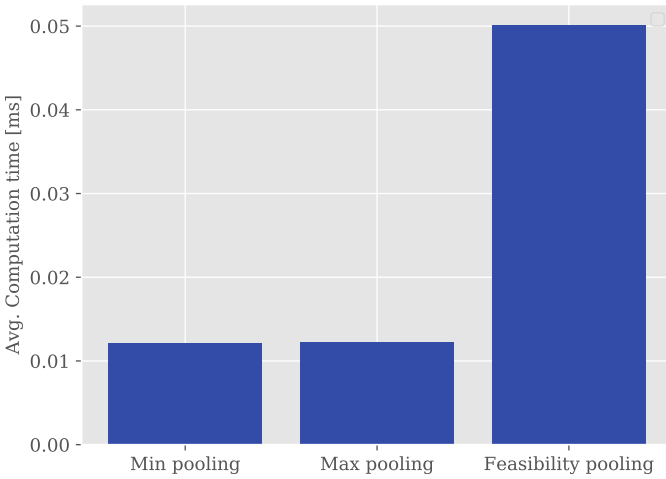


Figure 3.9: Avg. computation time for feasibility pooling compared to max and min pooling for $n = 9$, calculated based on sensor measurements that were extracted from vessel simulations.

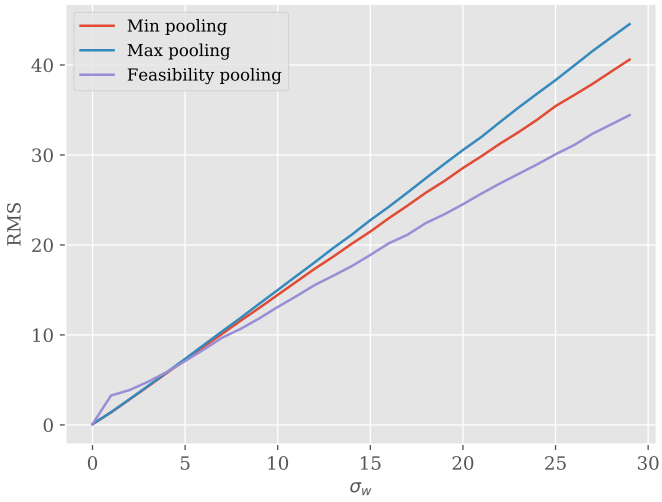


Figure 3.10: Robustness metric for pooling methods for $\sigma_w \in \{1, \ldots, 30\}$ estimated based on measurements extracted from repeated simulations. The noise-affected measurements are clipped at zero to avoid negative values.

3.2.4 Motion detection

Simply feeding the pooled current rangefinder sensor readings to the agent's policy network, will, without any doubt, be insufficient for the agent to learn a policy for intelligently avoiding moving obstacles. A continuous snapshot of the environment can facilitate a purely reactive (but still intelligent (42)) agent in a static environment, but without explicit or implicit knowledge of the nearby obstacles' velocities, such an agent will invariantly fail when placed in a dynamic environment, as it will be unable to distinguish between stationary and moving obstacles.

An implicit approach worth mentioning is to process the sensor readings sequentially using a Recurrent Neural Network (RNN). In recent years, RNN architectures, such as Long Short-Term Memory LSTM, have gained a lot of traction in the ML research community (36) and been successfully applied to sequential RL problems. An example of this is the LSTM-based AlphaStar agent, which reached grandmaster level in the popular real-time strategy game StarCraft II (66). It is therefore possible that a high-performing collision avoidance policy could be found by feeding a recurrent agent with sensor readings. If such an implementation was shown to be successful, it would facilitate a very straight-forward transition to an implementation on a physical vessel, as no specialized sensor equipment for measuring object velocities would be needed.

However, even if sequentially feeding sensor readings to a recurrent network might sound relatively trivial, the motion of the vessel would induce rotations of the observed environment, complicating the situation. Initial experimentation with an off-the-shelf recurrent policy compatible with our simulation environment confirmed the difficulties with this approach. Even with a purely static environment, the recurrent agent was incapable of learning how to avoid collisions.

Thus, this preliminary study will focus on the explicit approach, i.e. providing the obstacles' velocities as features in the agent's observation vector. Admittedly, while the implementation of this is trivial in a simulated environment, as obstacle velocities can simply be accessed as object attributes, a real-world implementation will necessitate a reliable way of estimating obstacle velocities based on sensor data. However, even if this can be challenging due to uncertainty in the sensor readings, object tracking is a well-researched computer vision discipline. We reserve the implementation of such a method to future research, but refer the reader to (22) for a comprehensive overview of the current state of the field.

For each sector, we provide the decomposed velocity of the closest moving obstacle within the sector as features for the agent's observation vector. Specifically, the decomposition, which yields the x and y component of the obstacle velocity, is done with respect to the coordinate frame in which the y-axis is parallel to the center line of the sensor sector in which the obstacle is detected. This is illustrated in Figure 3.11. For each sector k, we denote the corresponding decomposed x and y velocities as $v_{x,k}$ and $v_{y,k}$, respectively. Naturally, if there are no moving obstacles present within the sector, both components are zero.

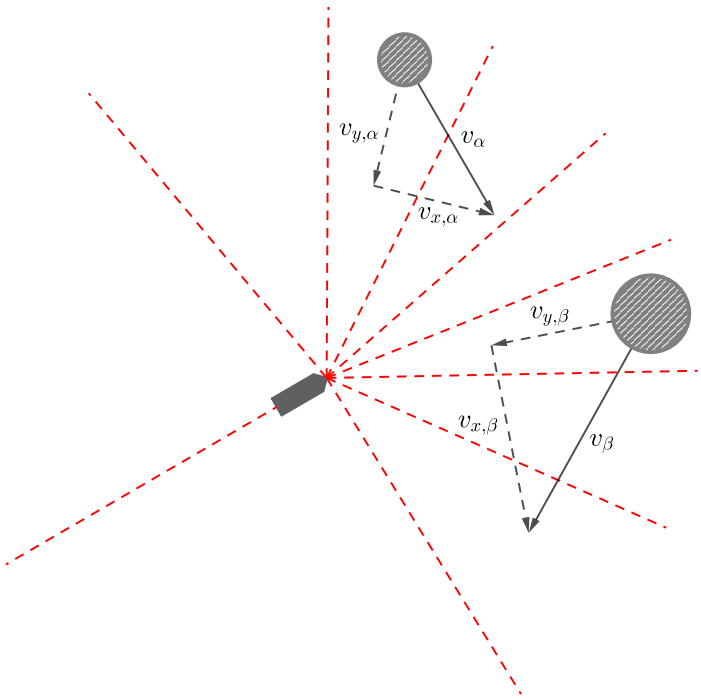


Figure 3.11: Velocity decomposition for two moving obstacles, α and β. For each obstacle, its velocity vector is decomposed into x and y components relative to the obstacle sector, such that the decomposed y-component is parallel to the center line of the corresponding sector, and has a positive value if it is moving *towards* the vessel.

3.2.5 Perception state vector

As having access to both obstacle distances and obstacles velocities is critical to achieve satisfactory obstacle-avoiding agent behavior, we include both in the perception state vector. To avoid discontinuities in the obstacle distance features caused by the sudden transition from 0 to S_r at the point of detection, we introduce the concept of obstacle *closeness*. The *closeness* to an obstacle is such that it is 0 if the obstacle is undetected, i.e. further away from the vessel than the maximum range of the distance sensors, and 1 if the vessel has collided with the obstacle. Furthermore, within this range, is it reasonable to map distance to closeness in a logarithmic fashion, such that, in accordance with human intuition, the difference between 1m and 10m is more significant than the difference between, for instance, 51m and 60m. Formally, we have that a distance d maps to closeness $c(d) : \mathbb{R} \mapsto [0, 1]$ according to

$$c(d) = \text{clip}\left(1 - \frac{\log(d+1)}{\log(S_r+1)},\ 0,\ 1\right) \tag{3.10}$$

By concatenating the reachable distance and the decomposed obstacle velocity from every sector, we then define the perception state vector s_p as

$$s_p^{(t)} = \Bigg[\underbrace{c\left(\left(\boldsymbol{w}_1^{(t)}\right)\right),\ v_{x,1}^{(t)},\ v_{y,1}^{(t)}}_{\text{First sector}},\ \dots \Bigg]^T \tag{3.11}$$

3.3 Reward function

Any RL agent is motivated by the pursuit of maximizing its reward. and consequently, engineering the reward function is a crucial part of to achieving satisfactory performance from the trained agent. The simplest, and thus highly sought-after approach to rewarding RL agents is to reward it at the end of each episodes - at that point, one already knows if the agent succeeded or failed. However, given the length of a full episode, such a reward function turns out extremely sparse, leaving the agent with a near impossible learning task. This calls for a continuous reward signal, rewarding the agent based on its current adherence to its objectives, i.e. how well it is currently doing with respect to both path following and obstacle avoidance. Given the complexity of the dual-objective learning problem focused on in this study, as well as the general tendency of RL agents' to exploit the reward function in any way possible (e.g. standing still, going in circles), designing an appropriate rewards function $r^{(t)}$ is paramount to the agent exhibiting the desired behavior after training.

For instance, consider the case where the agent receives a significant penalty whenever it collides with an obstacle. As the easiest way to avoid collisions is to move in an endless loop or to move in the opposite direction of the path, where there are no obstacles, it is likely that the agent will learn such behavior unless sufficiently penalized. As was demonstrated in (26), scaling the reward function by a constant scalar can have a large effect on the agent's performance, even in the vanilla single-objective Open-AI gym environments commonly used as benchmarks for new RL algorithms. In a dual objective scenario, such as the one tackled in this paper, it is obvious that the reward trade-off between avoiding collisions and following the path can be critical: If the relative penalty for collisions is too low, the agent will frequently collide. If the relative penalty is too high, however, we risk that the agent will learn a policy that avoids making progress along the desired path at all costs.

It is natural to reward the agent separately for its performance in the two relevant domains: path following and collision avoidance. Thus, we introduce the independent reward terms $r_{path}^{(t)}$ and $r_{colav}^{(t)}$, representing the path-following and the obstacle-avoiding reward components, respectively, at time t. Furthermore, we introduce the weighting coefficient $\lambda \in [0, 1]$ to regulate the trade-off between the two competing objectives. In addition, as it is crucial to penalize the agent whenever it collides with an obstacle, we represent this by the negative reward term $r_{collision}$, which is activated upon collision. This leads to the

preliminary reward function

$$r^{(t)} = \begin{cases} r_{collision}, & \text{if collision} \\ \lambda r_{path}^{(t)} + (1-\lambda)\, r_{colav}^{(t)}, & \text{otherwise} \end{cases} \tag{3.12}$$

3.3.1 Path following performance

A natural approach to incentivize path adherence is to reward the agent for minimizing the current absolute cross-track error $\left|\epsilon^{(t)}\right|$. In (37), a Gaussian reward function centered at $\epsilon = 0$ with standard deviation σ_e was suggested. However we argue that the absolute exponential reward function $\exp\left(-\gamma_\epsilon \left|\epsilon^{(t)}\right|\right)$ has more desirable characteristics due to its fatter tails, as seen in Figure 3.12. By avoiding the vanishing improvement gradient of the Gaussian reward occurring at large absolute cross-track errors, the absolute exponential reward function ensures that the agent is rewarded even for a slight improvement to a very unsatisfactory state.

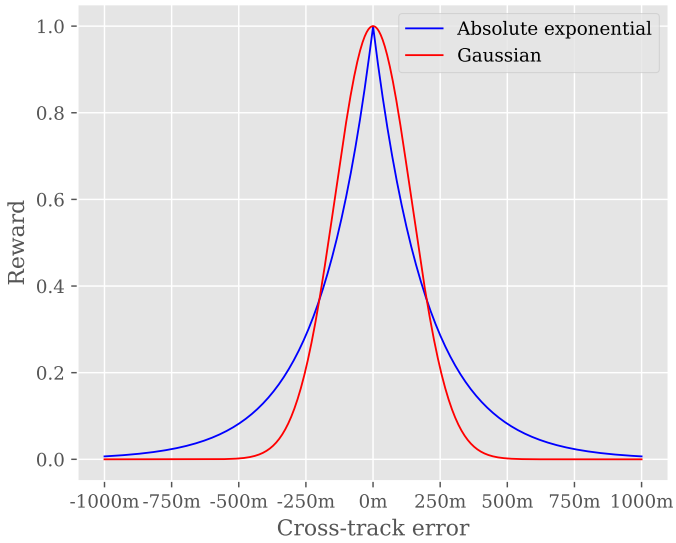


Figure 3.12: Cross-section of the path-following reward landscape for $\gamma_\epsilon = 0.05$ assuming path-tangential full-speed motion visualized for both Gaussian and absolute exponential kernels for cross-track error rewarding.

However, this alone does not reflect our desire for the agent to actually make progress along the path - and thus, the RL agent, greedy as it is, will eventually develop a policy of standing still indefinitely after closing the gap to the path. Thus, the reward signal must be expanded upon so that it incorporates the incentivisation of motion - and not just arbitrary

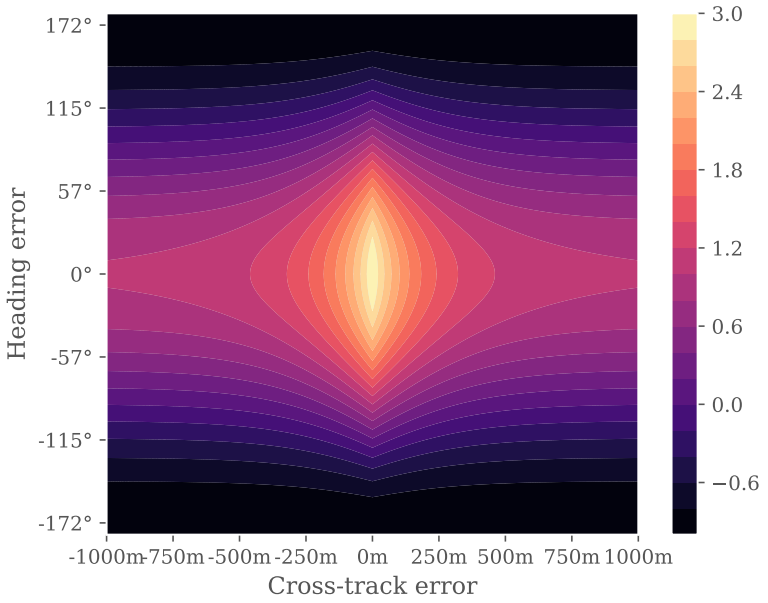


Figure 3.13: Path-following reward function assuming full-speed motion for $\gamma_\epsilon = 0.05$. The level curves represent constant reward. Zero cross-track error together with zero heading error yields maximum reward, as expected.

motion, but movement in the right direction.

The already defined look-ahead heading error term $\tilde{\psi}$ is a natural basis for formalizing this. Specifically, we consider the term $\frac{u^{(t)}}{U_{max}} \cos \tilde{\psi}^{(t)}$, with U_{max} being the maximum vessel speed, which effectively yields zero reward if the vessel is heading in a direction perpendicular to the path, and a negative reward if the agent is tracking backwards. Multiplying this with the cross-track error reward component defined earlier is a natural choice, and yields the provisional reward function

$$r_{path}^{(t)} = \underbrace{\frac{u^{(t)}}{U_{max}} \cos \tilde{\psi}^{(t)}}_{\text{Velocity-based reward}} \underbrace{\exp\left(-\gamma_\epsilon |\epsilon^{(t)}|\right)}_{\text{CTE-based reward}}$$

Given this reward function, however, we note that, if the vessel is standing still (i.e. $u^{(t)} = 0$), or if it is heading in a direction perpendicular to the path (i.e. $\tilde{\psi}^{(t)} = \pm\frac{\pi}{2}$), the agent will receive zero reward regardless of the cross-track error, which is undesired. Similarly, if the cross-track error grows very large, i.e. $\exp\left(-\gamma_\epsilon \left|\epsilon^{(t)}\right|\right) \to 0$, the reward signal will be zero regardless of the vessel velocity and heading. Thus, we add constant multiplier terms γ_r to both reward components, yielding the following expression for the final path-

following reward function

$$r_{path}^{(t)} = \underbrace{\left(\frac{u^{(t)}}{U_{max}} \cos \tilde{\psi}^{(t)} + \gamma_r\right)}_{\text{Velocity-based reward}} \underbrace{\left(\exp\left(-\gamma_\epsilon |\epsilon^{(t)}|\right) + \gamma_r\right)}_{\text{CTE-based reward}} - \gamma_r^2 \tag{3.13}$$

where the $-\gamma_r^2$ term is added to remove the constant reward bias implied by the function choice.

3.3.2 Obstacle avoidance performance

Collision avoidance involves both collisions with other vessels as well as avoiding running ashore (or colliding with some other static obstacle). However, the two aspects should be treated separately, as would any human sailor. In the following, we refer to the former as dynamic, and the latter as static obstacle avoidance.

In order to encourage obstacle-avoiding guidance behavior, penalizing the agent for the closeness of nearby terrain in a strictly increasing manner seems reasonable. However, we note that the severity of closeness intuitively does not increase linearly with distance, but instead increases in some quasi-exponential fashion. Furthermore, given the presence of a nearby obstacle, it seems clear that the penalty given to the agent must depend on the orientation of the vessel with regards to the obstacle in such a manner that obstacles located near the stern of the vessel are of significantly lower importance than obstacles that are currently right in front of the it. Thus, disregarding moving obstacle, we propose the following penalty function for a the presence of a static obstacle located at distance x from the vessel at the angle θ (measured with respect to the centerline of the vessel):

$$r_{obst,stat}(\theta, x) = -\underbrace{\frac{1}{1 + \gamma_\theta |\theta|}}_{\text{Weighting term}} \underbrace{\exp\left(-\gamma_x x\right)}_{\text{Raw closeness penalty}} \tag{3.14}$$

which is visualised on a logarithmic scale in Figure 3.14.

It is critical that the vessel stays out of collision course with other ships - even more so than with terrain, as other vessels are, by virtue of begin dynamic objects outside of the agent's control, unpredictable. To achieve this, a penalty term proportional to the target vessel's velocity towards the own-ship is added in the penalty exponential. As movement towards the vessel is equivalent with $v_y > 0$, we clip values below zero, since ships moving away from the vessel should be considered as a collision threat for the agent. For a single dynamic obstacle, the penalty thus becomes

$$r_{obst,dyn}(\theta, v_y, x) = -\underbrace{\frac{1}{1 + \gamma_\theta |\theta|}}_{\text{Weighting term}} \underbrace{\exp\left(\gamma_v \text{max}(0, v_y) - \gamma_x x\right)}_{\text{Raw penalty}} \tag{3.15}$$

where x is the distance to the obstacle, θ is the vessel-relative angle and v_y is the velocity component in the direction towards the vessel. This penalty landscape is visualised in Figure 3.15, where θ is held constant at 0.

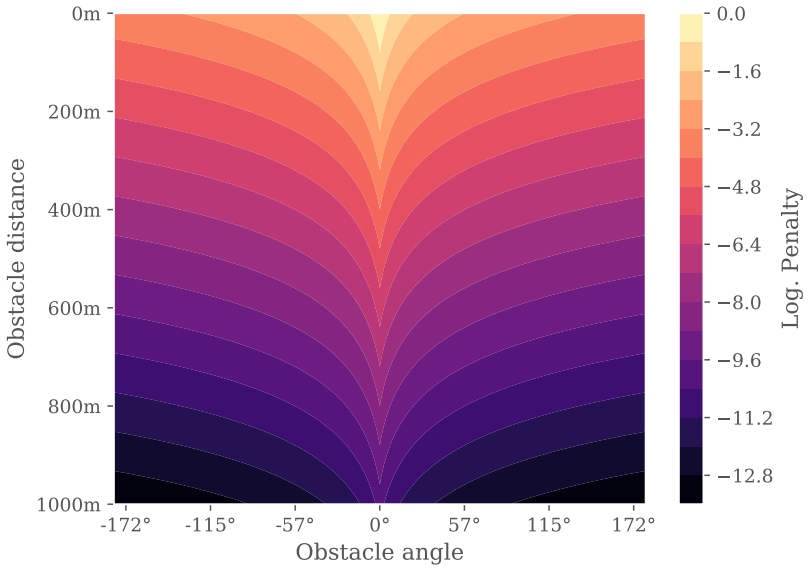


Figure 3.14: Static obstacle closeness penalty landscape as a function of obstacle distance and angle relative to the vessel with the scale parameters $\gamma_\theta = 10.0$, $\gamma_x = 0.1$. The maximum penalty is imposed for obstacles located right in front of the vessel.

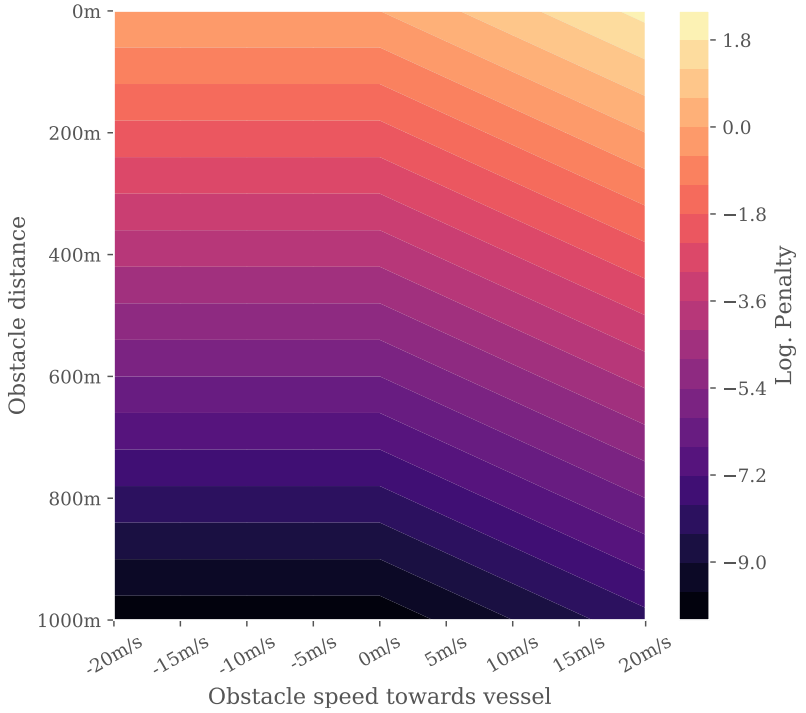


Figure 3.15: Dynamic obstacle closeness penalty landscape as a function of obstacle distance and obstacle velocity with scale parameters $\gamma_{v_y} = 1.0$, $\gamma_x = 0.1$. As should be expected, the maximum penalty is imposed for obstacles located 0m from the vessel, with full speed towards it.

For practical reasons, we use the measurements from the rangefinder sensor suite as surrogates for obstacle closeness, and penalize each sensor reading according to $r_{obst,dyn}(x_i, v_{yi}, \theta_i)$, where x_i is the i^{th} distance sensor measurement, v_{yi} is the y component of i^{th} velocity measurement and θ_i is the vessel-relative angle of the corresponding sensor ray. In order to to cancel the dependency on the specific sensor suite configuration, i.e. the number of sensors and their vessel-relative angles, that arises when this penalty term is summed over all sensors, we compute the overall obstacle-avoidance reward according to the weighted average

$$r_{colav}^{(t)} = -\frac{\sum_{i=1}^{N} \frac{\zeta(\theta_i)}{1+\gamma_\theta|\theta_i|} \exp\left(\gamma_v \text{max}(0, v_y^i) - \gamma_x x_i\right)}{\sum_{i=1}^{N} \frac{\zeta(\theta_i)}{1+\gamma_\theta|\theta_i|}} \tag{3.16}$$

3.3.3 Total reward

Furthermore, in order to discourage the agent from simply standing still at a safe location, which would yield a reward of zero given the preliminary reward function defined in Equation 3.12, we impose a constant living penalty $r_{exists} < 0$ to the overall reward function. A simple way of setting this parameter is to assume that, given a total absence of nearby obstacles and perfect vessel alignment with the path, the agent should receive a zero reward when moving at a certain slow speed $\alpha_r U_{max}$, where $\alpha_r \in (0, 1)$ is a constant parameter. This gives us

$$\begin{aligned} &r_{exists} + \lambda \left(\left(\tfrac{\alpha_r U_{max}}{U_{max}} + 1 \right) (1+1) - 1 \right) = 0 \\ &r_{exists} = -\lambda(2\alpha_r + 1) \end{aligned} \tag{3.17}$$

The expression for the final overall reward function then becomes

$$r^{(t)} = \begin{cases} r_{collision}, & \text{if collision} \\ \lambda r_{path}^{(t)} + (1-\lambda)\, r_{colav}^{(t)} + r_{exists}, & \text{otherwise} \end{cases} \tag{3.18}$$

3.4 Simulation parameters

In our solution, both the policy network as well as the value network used in the PPO algorithm's advantage estimation have two hidden layers with 64 units each, and use the *tanh* activation function across the networks. Furthermore, the following hyperparameter values were used for the PPO algorithm:

Parameter	Interpretation	Value
γ	Discount factor	0.999
T	Timesteps per training iteration	1024
N_A	Number of parallel actors	8
K	Training epochs	10^6
η	Learning rate	0.0002
N_{MB}	Number of minibatches	32
λ_{PPO}	Bias vs. variance parameter	0.95
c_1	Value function coefficient	0.5
c_2	Entropy coefficient	0.01
ϵ	Clipping parameter	0.2

Table 3.3: Hyperparameters for PPO algorithm.

In terms of the vessel setup, the following values were chosen:

Parameter	Interpretation	Value
U_{max}	Maximum vessel speed	2 m/s
N	Number of sensors	180
S_r	Sensor distance	1.5 km
d	Number of sensor sectors	9
Δ_{LA}	Look-ahead distance	3 km

Table 3.4: Vessel configuration.

Finally, the parameters in Table 4.1 were used for customizing the reward function. This choice of reward function parameters stems from intuitive reasoning about the desired characteristics of the agent's guidance behavior and how it relates to the parameters, but experimentation with other values have suggested that the sensitivity to these parameters is rather low; for any reasonable parameter choice, the agent seems to develop intelligent guidance behavior. However, it should be noted that λ should be chosen such that neither the path-following nor the collision avoidance objectives are neglected (here, 0.5 is a reasonable compromise).

Parameter	Interpretation	Value
γ_e	Cross-track error scaling	0.05
γ_θ	Sensor angle scaling	4.0
γ_x	Obstacle distance scaling	0.005
α_r	Zero-reward relative speed	0.1
r_{coll}	Collision reward	-2000
λ	Objective trade-off coefficient	0.5

Table 3.5: Reward configuration.

3.5 Evaluation

To provide a comprehensive basis for evaluating the agent's performance, i.e. the degree to which the agent is avoiding collisions, as well as the degree to which it adheres to its path following objective, we test the trained agent in various test environments. First, we simulate its behavior in new (i.e. unseen) permutations of the training scenario. As described, the training environment is challenging, with a dense scattering of both static and dynamic obstacles (as shown in Figure 3.2). Furthermore, based on combining high-fidelity terrain data with AIS tracking data from the Trondheim Fjord area, we construct three digital real-world environments in which the vessel's performance can be evaluated in a realistic manner. Here, even if the terrain (i.e. static obstacles) is pre-determined by the elevation data, and thus, unlike in the training scenarios, always identical, the scenario traffic (i.e. other vessels) is sampled as a random subset of the total recorded AIS data in the area. This allows for quantitative statistical testing by means of repeated trials.

3.5.1 Ørland-Agdenes

This scenario takes place in the heavily trafficked entrance region of the fjord: The region between the municipalities Ørland and Agdenes. After spawning near the coastline, the vessel must blend into two-way traffic and follow the path until it reaches the opening of the fjord. In particular, the agent will be tested on its ability to handle head-on and overtaking situations.

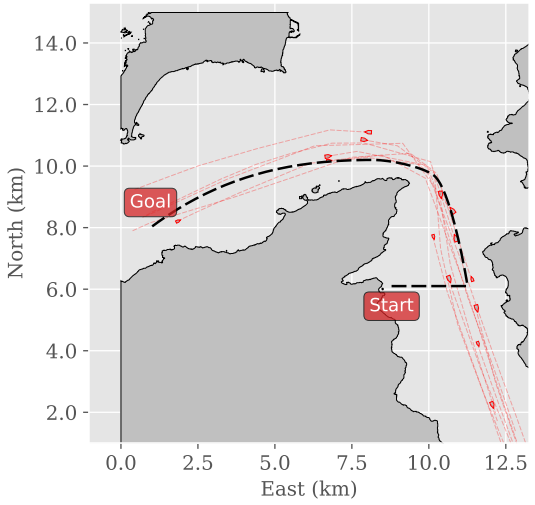


Figure 3.16: Map of the Ørland-Agdenes test scenario. The dashed black line represents the desired vessel trajectory. Each other vessel is drawn at its initial position. Also, each other vessel's trajectory is drawn as a transparent and dotted red line.

3.5.2 Trondheim

Spawning next to the Trondheim city center, the agent is expected to cross the fjord end and up at the village Vanvikan. In order to succeed in this scenario, the agent must avoid collisions with the crossing traffic, which is dominated by larger ships.

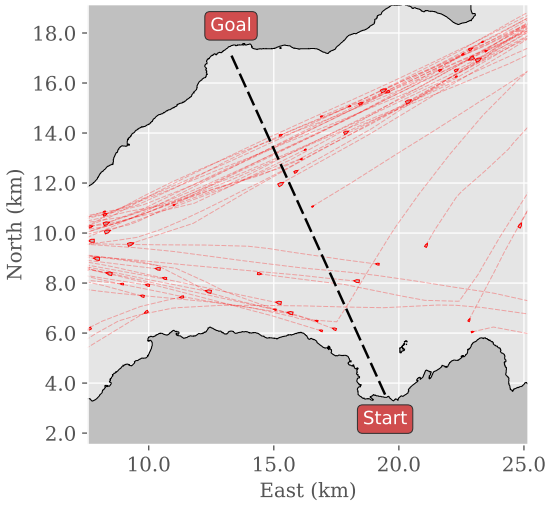


Figure 3.17: Map of the Trondheim test scenario.

3.5.3 Froan

Froan, which is located off the Trøndelag coast, is an archipelago encompassing hundreds of small, rocky islands. For this reason, it offers uniquely challenging terrain. In this scenario, the agent must carefully navigate through a cluster of small islands, before merging into traffic going to and from Sørburøy, the most populated island in the area. The challenging terrain will test the agent's ability to navigate static obstacles, whereas the traffic, comprised of smaller, fast-moving vessels, will lead to challenging head-on situations, especially in the narrow strait in which the goal is located.

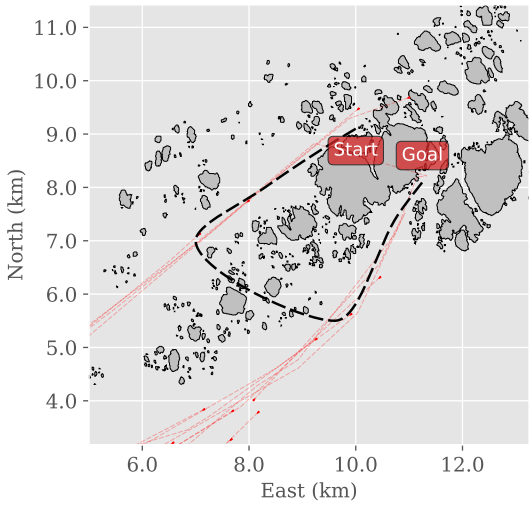


Figure 3.18: Map of the Froan test scenario.

For the purpose of evaluating the agent's vessel guidance ability is a statistically significant manner, we simulate the agent in 100 independently sampled scenarios for each of the the three real-world based training environments, as well as the artificial training scenario. Here, the main quantitative test metric is the percentage of episodes in which the agent reached the goal successfully (i.e. without collisions), which in the following is referred to as the *success rate*. Also, we report the episodic average for *cross-track error*, i.e. the average deviation from path in meters, to quantify the degree of path following adherence in the various scenarios.

Chapter 4

Software Framework

In this chapter, we outline the design and implementation of a Python-based software framework allowing researchers to train and evaluate RL-based Autonomous Surface Vessels. It serves as an interface to a simulated marine environment, facilitating not only artificially generated collision avoidance scenarios, but also reconstructions of real-world scenarios based on local terrain and marine tracking data. The framework is provided as a Python package named **gym-auv**, available at the GitHub repository found at (41). **gym-auv** heavily relies on the OpenAI **gym** API (6), a popular toolkit for developing and comparing RL algorithms. **gym** is widely used in research, and has emerged as the de facto standard programming interface for deep RL applications. As our framework, **gym-auv**, is an extension of **gym**, it inherits **gym**'s plug-and-play compatibility with numerous powerful Python libraries for training RL agents, including OpenAI Baselines (10), Ray RLlib (32) and Stable Baselines (27).

4.1 Structure

The core component of **gym** is the environment abstraction `Env`, which represents the generalized RL environment . Notably, **gym** does not include a built-in `Agent` class of any kind. Instead, all the fundamental functionality required for an RL application, i.e. agent perception, reward calculation and action execution / environment updates are handled by the `Env` instance. Fundamentally, extensions of **gym**, including our **gym-auv** package, implement a subclass of `gym.Env` which overrides the core abstract methods: `__init__`, which defines the environment's action and observation space; `step`, which simulates the environment for one timestep after an action has been performed and returns the observation vector and reward; `reset`, which resets the environment state to the initial state; and `render`, which renders the environment to the screen. In our case, this class is named `BaseEnvironment`.

Furthermore, our framework uses three other classes, namely `Vessel`, `Path`, `BaseObstacle` and `BaseRewarder`. This provides a clear modular structure for the software and allows us to abstract away tedious function implementations. Also, it facili-

tates further extensions, such as adding a new vessel type with other dynamics, adding new obstacle shapes or introducing a new reward function to achieve different vessel behaviors. In the following, we will outline the details of these classes and how they are related.

4.1.1 Path

The `Path` class represents an a priori available trajectory which is intended to be followed by a `Vessel` instance. It provides not only a lookup method mapping from a specified arc-length value to the corresponding coordinate point, but also helper methods that facilitate a vessel's navigation with the respect to the path. In the default behavior, a smooth trajectory parameterized by arc length is generated using 1D Piecewise Cubic Hermite Interpolator (PCHIP) provided by SciPy (67) based on the `waypoints` argument required by the constructor method. Optionally, by calling the constructor with the keyword `smooth=False`, the user can also create a path made of linear line segments connecting the specified way-points.

```
class Path():
    def __init__(self, waypoints:list, smooth:bool=True) -> None:
        """Initializes path based on specified waypoints."""

    @property
    def length(self) -> float:
        """Length of path."""

    @property
    def start(self) -> np.ndarray:
        """Coordinates of the path's starting point."""

    @property
    def end(self) -> np.ndarray:
        """Coordinates of the path's end point."""

    def __call__(self, arclength:float) -> np.ndarray:
        """Returns the (x,y) point corresponding to the
        specified arclength."""

    def get_direction(self, arclength:float) -> float:
        """Returns the direction in radians with respect to the
        positive x-axis."""

    def get_closest_arclength(self, position:np.ndarray) -> float:
        """Calculates the arc length value corresponding to the point
        on the path which is closest to the specified position."""
```

4.1.2 BaseObstacle

The `BaseObstacle` class is an abstract class that represents physical obstacles that a `Vessel` instance can collide with and should avoid. Due to the vast variety of obstacles the user might be interested in using in a scenario, both in terms of shape and dynamic properties, it is designed as an abstract class which is intended to be implemented by its sub-classes. As will be discussed later in this chapter, the **gym-auv** package relies upon the

Python package **Shapely** for the geometric operations required to simulate a rangefinder sensor suite. Thus, the `BaseObstacle` class' public `boundary` attribute, which has the type of a `shapely.geometry.Polygon`, is a critical feature of the class as it is required for simulating the rangefinder sensors' detection of the obstacle. Also, the `BaseObstacle` class includes an update wrapper method for updating the obstacle's position given its dynamic properties - for instance given its speed and heading if the obstacle represents another vessel. The specific update behavior must be implemented in extensions of `BaseObstacle`, and can be left blank in the case of static obstacles.

```
class BaseObstacle(ABC):
    def __init__(self, *args, **kwargs) -> None:
        """Initializes obstacle instance by calling private setup method
        ↪  implemented by subclasses of BaseObstacle and calculating
        ↪  obstacle boundary."""
        self._setup(*args, **kwargs)
        self._boundary = self._calculate_boundary()

    @property
    def boundary(self) -> shapely.geometry.Polygon:
        """The obstacle boundary represented as a
        ↪  shapely.geometry.Polygon object. Used by the 'Vessel' class
        ↪  for simulating the sensors' detection of the obstacle
        ↪  instance."""
        return self._boundary

    def update(self, dt:float) -> None:
        """Updates the obstacle according to its dynamic behavior, e.g.
        a ship model and recalculates the boundary."""
        has_changed = self._update(dt)
        if has_changed:
            self._boundary = self._calculate_boundary()

    @abstractmethod
    def _calculate_boundary(self) -> shapely.geometry.Polygon:
        """Returns a shapely.geometry.Polygon instance representing the
        ↪  obstacle
        given its current state."""

    @abstractmethod
    def _setup(self, *args, **kwargs) -> None:
        """Initializes the obstacle given the constructor parameters
        ↪  provided to
        the specific BaseObstacle extension."""

    def _update(self, dt:float) -> bool:
        """Performs the specific update routine associated with the
        ↪  obstacle class and returns a boolean flag representing
        ↪  whether something changed or not.

        Returns
        -------
        has_changed : bool
        """
        return False
```

4.1.3 Vessel

The `Vessel` class represents a physical vessel placed in an environment. This should not be confused with the *agent* as thought of in an RL-context - an autonomous entity directing its actions towards achieving its goals within its environment. In our **gym-auv** framework, the `Vessel` class is simply responsible for updating the vessel state according to the ship dynamics as well as simulating the sensor suite attached to the vessel. This functionality logically belongs to the environment module, but is implemented as a separate module to facilitate a possible multi-agent use-case with several vessels interacting within the same environment.

```
class Vessel():
    def __init__(self, config:dict, init_state:np.ndarray, width:float)
    ↪  -> None:
        """Initializes and resets the vessel."""

    @property
    def width(self) -> float:
        """Returns the width of vessel."""

    @property
    def position(self) -> np.ndarray:
        """Returns an array holding the position of the AUV in cartesian
        coordinates."""

    @property
    def path_taken(self) -> np.ndarray:
        """Returns an array holding the path of the AUV in cartesian
        coordinates."""

    @property
    def heading(self) -> float:
        """Returns the heading of the AUV with respect to true north."""

    @property
    def velocity(self) -> np.ndarray:
        """Returns the surge and sway velocity of the AUV."""

    @property
    def speed(self) -> float:
        """Returns the speed of the AUV."""

    @property
    def yaw_rate(self) -> float:
        """Returns the rate of rotation about the z-axis."""

    @property
    def max_speed(self) -> float:
        """Returns the maximum speed of the AUV."""

    @property
    def course(self) -> float:
        """Returns the course angle of the AUV with respect to true
        ↪  north."""
        crab_angle = np.arctan2(self.velocity[1], self.velocity[0])
```

```
    @property
    def sensor_angles(self) -> np.ndarray:
        """Returns numpy array containg the angles each sensor ray
        ↪  relative to the vessel heading."""

    @property
    def sector_angles(self) -> np.ndarray:
        """Returns numpy array containg the angles of the center line of
        ↪  each sensor sector relative to the vessel heading."""

    def reset(self, init_state:np.ndarray) -> None:
        """Resets the vessel to the specified initial state."""

    def step(self, action:list) -> None:
        """Simulates the vessel one step forward after applying the
        ↪  given action."""

    def perceive(self, obstacles:list) -> (np.ndarray, np.ndarray):
        """Simulates the sensor suite and returns observation arrays of
        ↪  the environment.

        Returns
        -------
        sector_closenesses : np.ndarray
        sector_velocities : np.ndarray
        """

    def navigate(self, path:Path) -> np.ndarray:
        """Calculates and returns navigation states representing the
        ↪  vessel's attitude with respect to the desired path.

        Returns
        -------
        navigation_states : np.ndarray
        """

    def req_latest_data(self) -> dict:
        """Returns dictionary containing the most recent perception and
        ↪  navigation states."""
```

4.1.4 BaseRewarder

The `BaseRewarder` class is responsible for calculating the reward received by an agent at each time step. It is designed as an abstract class which is intended to be implemented by its sub-classes. As the reward, in the general case, depends on a vessel's adherence to its desired path as well as its distance from obstacles in its proximity, a `BaseRewarder` instance gets a `Vessel` instance assigned to it in the constructor which it accesses upon calculating the reward. As was the case for the `Vessel` class, the motivation for detaching this functionality from the `BaseEnvironment` is to facilitate multi-agent extensions with possible nonuniform agent objectives, necessitating a one-to-many relationship between environment and rewarder.

```
class BaseRewarder(ABC):

    @property
    def vessel(self) -> Vessel:
        """Vessel instance that the reward is calculated with respect
        ↪  to."""

    @abstractmethod
    def calculate(self) -> float:
        """
        Calculates the step reward and decides whether the episode
        should be ended.

        Returns
        -------
        reward : float
            The reward for performing action at this timestep.
        """

    def insight(self) -> np.ndarray:
        """
        Returns a numpy array with reward parameters for the agent to
↪   have an insight into its reward function.

        Returns
        -------
        insight : np.array
            The reward insight array at this timestep.
        """
        return np.array([])
```

4.1.5 BaseEnvironment

Extending the `gym.Env` base environment class, `BaseEnvironment` is the access point for using third-party RL algorithms to train agents in our environment. It implements the core abstract `gym.Env` methods `__init__`, `reset`, `step` and `render`. Notably, it also specifies its own abstract method to be implemented by specific scenario implementations, namely that of `_generate`, which, as the name suggests, (randomly) creates a new obstacle environment and is called each time the environment resets.

```
class BaseEnvironment(gym.Env, ABC):
    @property
    def action_space(self) -> gym.spaces.Box:
        """Array defining the shape and bounds of the agent's action."""

    @property
    def observation_space(self) -> gym.spaces.Box:
        """Array defining the shape and bounds of the agent's
        ↪  observations."""

    def reset(self) -> np.ndarray:
        """Resets the environment."""
        ...
```

```
        self._generate()
        self.rewarder = ColavRewarder(self.vessel)
        obs = self.observe()
        return obs

    def observe(self) -> np.ndarray:
        """Returns the array of observations at the current
        ↪  time-step."""
        reward_insight = self.rewarder.insight()
        navigation_states = self.vessel.navigate(self.path)
        sector_closenesses, sector_velocities =
        ↪  self.vessel.perceive(self.obstacles)

        obs = np.concatenate([reward_insight, navigation_states,
        ↪  sector_closenesses, sector_velocities])
        return obs

    def step(self, action:list) -> (np.ndarray, float, bool, dict):
        """Steps the environment by one timestep. Returns observation,
        ↪  reward, done, info."""

        self._update()
        self.vessel.step(action)
        obs = self.observe()
        reward = self.rewarder.calculate()
        done = self._isdone()
        info = ...
        return (obs, reward, done, info)

    def _update(self) -> None:
        """Updates the environment at each time-step. Can be customized
        ↪  in sub-classes."""
        [obst.update(dt=self.config["t_step_size"]) for obst in
        ↪  self.obstacles if not obst.static]

    @abstractmethod
    def _generate(self) -> None:
        """Create new, stochastically genereated scenario.
        To be implemented in extensions of BaseEnvironment. Must set the
        'vessel', 'path' and 'obstacles' attributes.
        """

    def render(self, mode='human'):
        """Render one frame of the environment.
        The default mode will do something human friendly, such as pop
↪   up a window."""
```

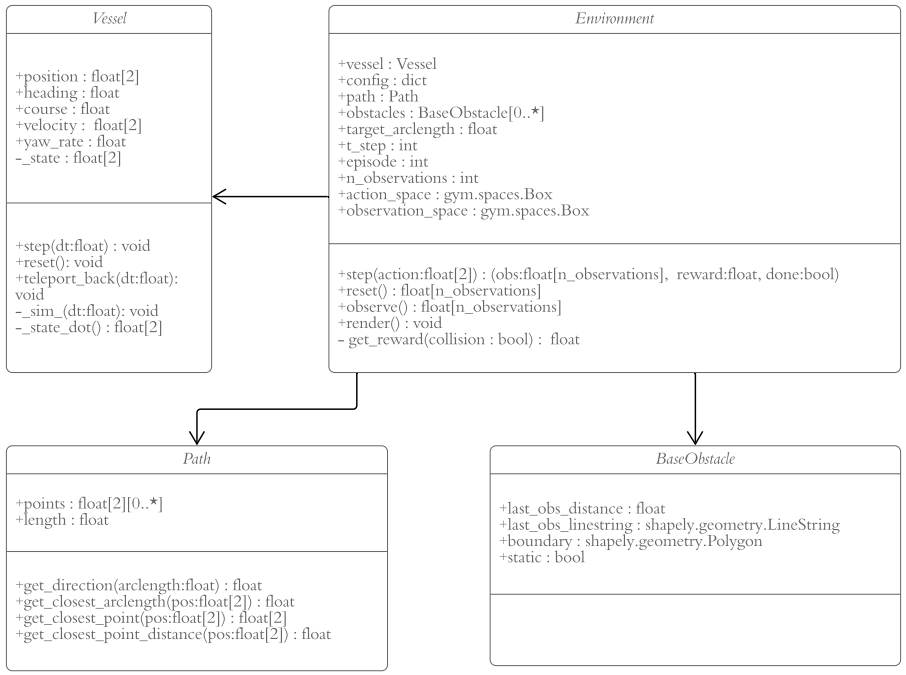


Figure 4.1: UML class diagram illustrating the interplay between the modules.

4.2 Rendering

Visualizing the agent's behavior is helpful for identifying the weaknesses of a trained agent. Even if plotting the vessel's trajectory in the environment can give valuable qualitative insight into whether the algorithm is behaving as intended, being able to see how the scenario unfolded, either in real-time or in the format of a video replay, is paramount to debugging the agent in a dynamic environment where not only the agent's vessel, but also the obstacles, are moving. For this purpose, as well as the purpose of showcasing the agent's performance, **gym-auv** includes built-in 2D and 3D rendering capabilities. 2D rendering, as seen in Figure 4.2, gives a clear visual overview of the current state of the environment and vessel, and is the preferred rendering mode for debugging and validation purposes. The 3D rendering module, which is showcased in Figure 4.3, is compatible with real-world elevation data, a provides a game-like real-time visualisation of the environment with a third person point of view. Both rendering modules are built using the **Pyglet** Python library, which provides an object-oriented application programming interface for the rendering of 2D and 3D vector graphics.

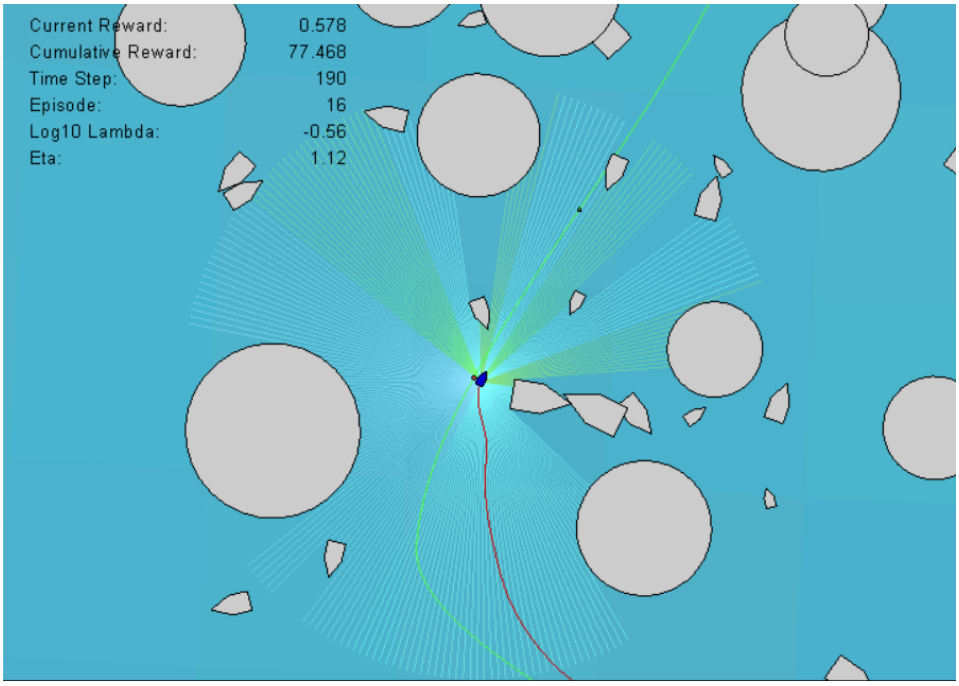


Figure 4.2: Screenshot of an environment rendered with the 2D rendering module.

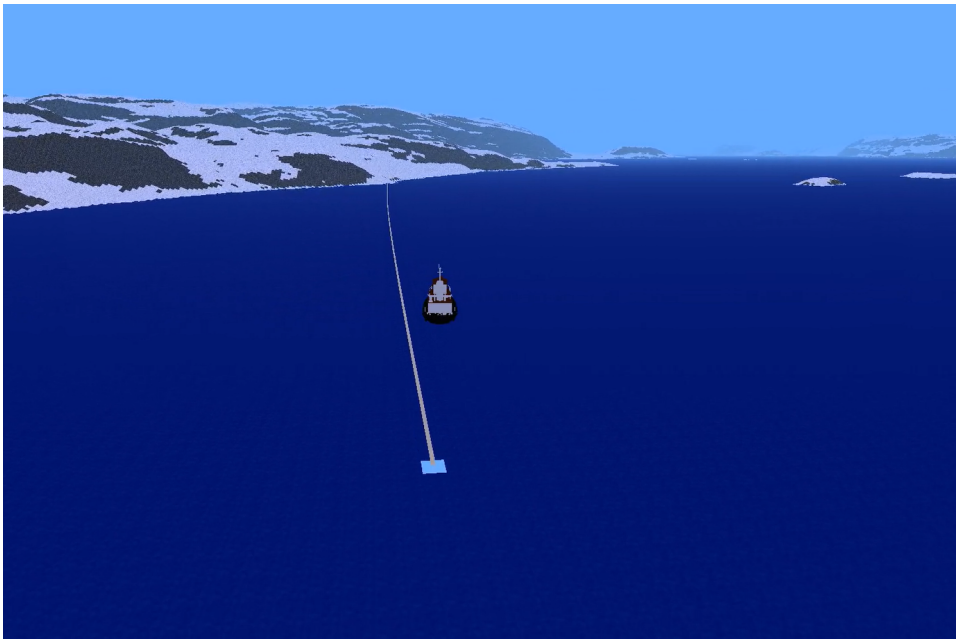


Figure 4.3: Screenshot of an environment rendered with the 3D rendering module.

4.3 Usage

A Python run script *run.py* is included along with the **gym-auv** package. This script provides a command-line interface to the relevant run modes when executing the simulation for research purposes. The first positional argument, `mode`, dictates the run mode of the executed program, and is of particular importance. In the following, the purpose and utilization of the most relevant run modes are covered, as well as the keyword arguments and flags associated with the respective run modes. The second positional argument, `env`, specifies the RL environment, and must be provided at execution. Most notably, the available environments include `MovingObstacles-v0`, i.e. the training environment for dynamic obstacle avoidance, but also test environments such as `Trondheim-v0` and `Agdenes-v0`. For a more comprehensive overview, the reader is advised to execute the run script with the -h flag.

4.3.1 Run mode *play*

The `play` run mode allows the user to control the vessel actuators by use of the arrow keys. As it provides real-time rendering with debug information on-screen, this mode is a great option for brief, manual testing of new reward function designs or other software innovations.

Example: `python run.py play Trondheim-v0 --render 3d`

4.3.2 Run mode *train*

The *train* run mode trains an RL agent on the chosen environment (i.e `env`). The user can override the default choice of RL algorithm (i.e. PPO) by specifying the `--algo` parameter. The training process will, at a regular interval, save the latest iteration of the agent (as *pkl* files) in the */logs/agents* folder. Furthermore, each random training scenario generation, as well as the agent's trajectory in the environment, will be saved in the */logs/plots*. Here, here will also be numerous plots showing the progress of the training. Additionally, videos of the agent will be recorded to the */logs/videos* folder from new, parallel environment processes that spawn at fixed intervals using the `enjoy` run mode. The training will run in parallel over multiple CPU cores unless the `--nomp` flag is given. Also, using the named argument `--agent 'path/to/agent.pkl'` will initialize the training with the network weights of the specified agent, allowing the user to restart training from an earlier iteration.

Example: `python run.py train MovingObstacles-v0`

4.3.3 Run mode *enjoy*

The *enjoy* run mode is meant for visualizing and recording the agent specified by the `--agent` parameter. Depending on the value of the `--render` argument, whose available options are `'2d'` and `'3d'`, this run mode will spawn a real-time recording process

that will output a video to the */logs/videos* folder.

Example: `python run.py enjoy TestScenario1-v0 --agent ./agent.pkl`

4.3.4 Run mode *test*

The *test* run mode is intended for qualitative and quantitative testing. It will simulate the agent given as `--agent` the number of times specified by the `--episodes` parameter. The testing will, by default, take place in the background (i.e. without rendering). Quantitative test metrics (e.g. avg. cross-track error), as well as trajectory plots, will be outputted to */logs/tests* at the end of each episode. Statistics that are aggregated over all the test episodes will also be found in the same folder.

Example: `python run.py test TestScenario2-v0 --agent ./agent.pkl --episodes 100`

4.4 Configuration

The software offers a wide range of customizable parameters found in the `__init__.py` file, the most important of which are listed below:

Parameter	Description	Default value
min_cumulative_reward	Minimum cumulative reward received before episode ends	-2000
max_timesteps	Maximum amount of timesteps before episode ends	10000
min_goal_distance	Minimum aboslute distance to the goal position before episode ends	5
min_path_progress	Minimum path progress before scenario is considered successful	0.99
t_step_size	Length of simulation timestep [s]	1.0
sensor_frequency	Sensor execution frequency	1.0
observe_frequency	Frequency of using actual obstacles instead of virtual	1.0
look_ahead_distance	Path look-ahead distance for vessel [m]	300
n_sensors_per_sector	Number of rangefinder sensors within each sector	20
n_sectors	Number of sensor sectors	9
sensor_range	Range of rangefinder sensors [m]	150

Table 4.1: Reward configuration

4.5 Optimization

Depending on the number of rangefinder sensors attached to the vessel, simulating the sensor suite can be slow. Although the **shapely** library offers efficient procedures for geometric calculations, most relevantly the `intersects` method for computing intersection points between geometric shapes, these computations are the major bottleneck of the computation. Given the resolution of the elevation data used in this research, the complexity of the polygons representing real-world terrain leads to a further worsening of computational efficiency. Thus, concrete approaches to speed up computation without sacrificing the integrity of the simulation are of large value. In the following, we present a few selected approaches to mitigating this problem.

4.5.1 Reduced sensor activation

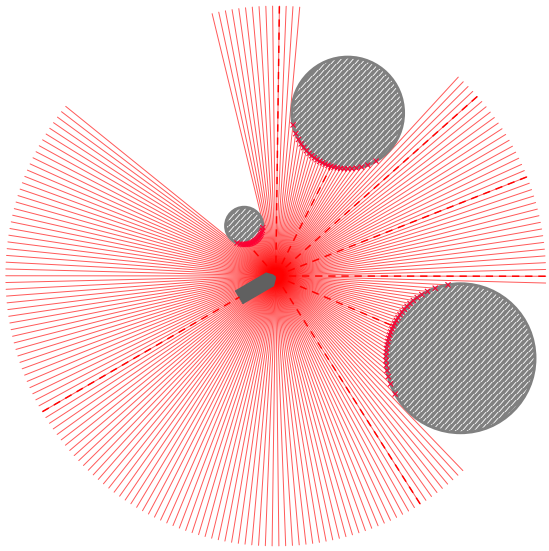
Depending on the time-step size, one can lower the activation frequency of the sensors while maintaining a perception capability sufficient of navigating in an intelligent manner. This relies on the assumption

$$x_i^{(t)} \approx x_i^{(t-1)} \quad \forall i, t \tag{4.1}$$

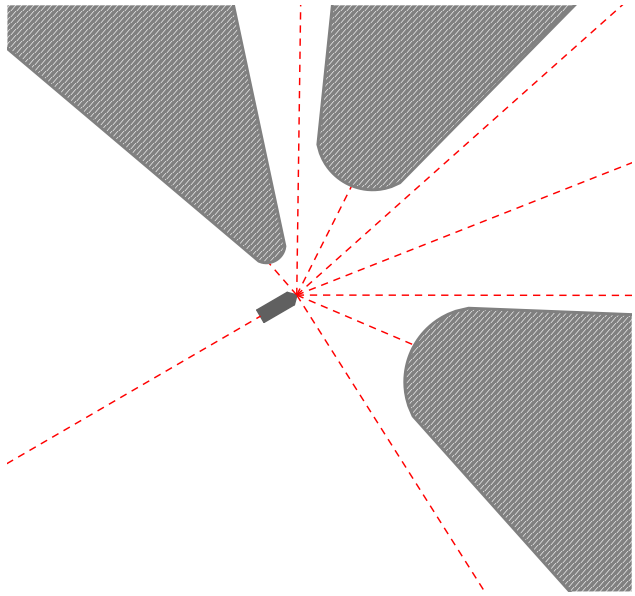
which can be controlled in the software through varying the configuration parameter `sensor_frequency`, the value of which decides the percentage of time-steps where the sensors should be executed.

4.5.2 Obstacle virtualization

Especially in test scenarios derived from real-world elevation data, the geometric complexity of the obstacles, as well as the sheer number of them, simulating the vessel is computationally expensive, and, depending on the computer, quite slow. However, a significant speed-up can be achieved by replacing the actual obstacle environment with a virtual one - one that is generated based on the last available sensor readings. Here, the procedure is simply to create artificial obstacle polygons based on the intersection points computed upon the actual execution of the sensor. On a regular interval, defined by the value of the configuration parameter `observe_frequency`, the sensor suite will be executed on the actual obstacle environment, and thus update the virtual one. Until the next update, the vessel will use the virtual obstacles as an approximation of the nearby obstacle environment, yielding a huge gain in performance. In Figure 4.4b, the virtual obstacle environment derived from the sensor measurements in Figure 4.4a is displayed.



(a) Sensor scan of actual obstacle environment



(b) Corresponding virtual obstacle environment

Figure 4.4: Illustration of the procedure yielding a virtual obstacle environment intended for speeding up computation time. When simulating the vessel in environments reconstructed from real terrain-data, the simplification of the obstacle geometry resulting from this approach leads to significant reduction in computation time for calculating the intersection points between sensor rays and obstacles.

4.5.3 Higher-order ODE solver

Using a more complex Runga-Kutta method than the forward Euler method increases the stability margin of the simulation for larger time-steps, which allows for faster simulation. In the software framework, the default numerical solver for simulating the vessel dynamics is the fifth order Runge-Kutta-Fahlberg method with the Butcher tableau shown in Table 4.2.

0					
1/4	1/4				
3/8	3/32	9/32			
12/13	1932/2197	-7200/2197	7296/2197		
1	439/216	-8	3680/513	-845/4104	
1/2	-8/27	2	-3544/2565	1859/4104	-11/40
25/216	0	1408/2565	2197/4104	-1/5	0
16/135	0	6656/12825	28561/56430	-9/50	2/55

Table 4.2: Butcher tableau for embedded Runge-Kutta-Fahlberg method (15).

Remark. The default distance unit used in this package is 10 m (decameters). Return values, such as those returned by `length` and `position` attributes, must be multiplied by 10 to obtain the values in meters. This choice was made out of convenience, given that the terrain data has a 10x10 metric resolution.

Chapter 5

Results and Conclusion

In this chapter, we present and discuss the results obtained during the training and evaluating phase. Furthermore, we conclude the study and outline the most notable directions for future work based on this project.

5.1 Training process

The agent's training process consisted of a total number of 1000 simulated episodes, which corresponds to more than 4 million simulated time-steps. Using a Intel Core i7-8550U CPU with 8 parallel simulation environments, this amounted to a total training time of more than 50 hours. Training was stopped when the agent's policy had converged to a desirable performance level. This was backed up by the observed training metrics used for monitoring the progress, whose plots can be seen in the Appendix. At the end of the training process, we observed that the agent converged to an almost 100% success rate.

5.2 Evaluation results

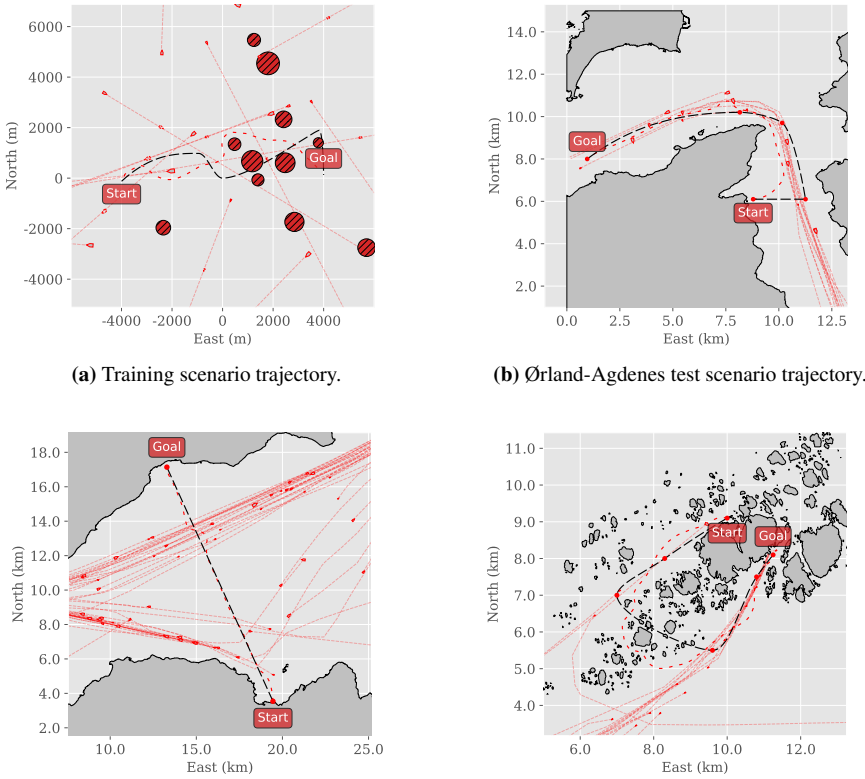
As outlined in Section 3.5, our strategy for evaluating the agent's guidance performance was based on repeated trials in multiple test scenarios. Specifically, we simulated the agent's trajectory in new, random iterations of the training environment, as well as three real-world based scenarios from the Trondheim Fjord: Ørland-Agdenes, Trondheim and Froan, where the scenario traffic was randomly sampled as a subset of the total recorded AIS data in the area. For each of the four test domains, 100 episodes were simulated. A larger sample size for evaluation would, of course, yield stronger statistical significance, but due to the lack of time and computational resources, 100 episodes was a reasonable compromise. The results obtained from quantitative evaluation are presented in in Table 5.1. A 100 % success rate, i.e. the agent reaching the goal position without colliding in every episode, is observed in all test environments. This excellent set of results testifies to the agent's capability to navigate unknown waters in a safe manner. As can also be

seen from the test results, the average cross-track error is highly scenario-dependant. This is expected, as the vessel traffic and obstacle density dictates how closely the agent can follow the desired path in a safe manner.

Scenario	Success rate	Avg. cross-track error
Training scenario	100%	450 m
Ørland-Agdenes	100%	300 m
Trondheim	100%	90 m
Froan	100%	300 m

Table 5.1: Quantitative test results obtained from 100 episode simulations in each scenarios.

In Figure 5.1, the vessel trajectory from selected test episodes is plotted for each of the four test environments.



(a) Training scenario trajectory.

(b) Ørland-Agdenes test scenario trajectory.

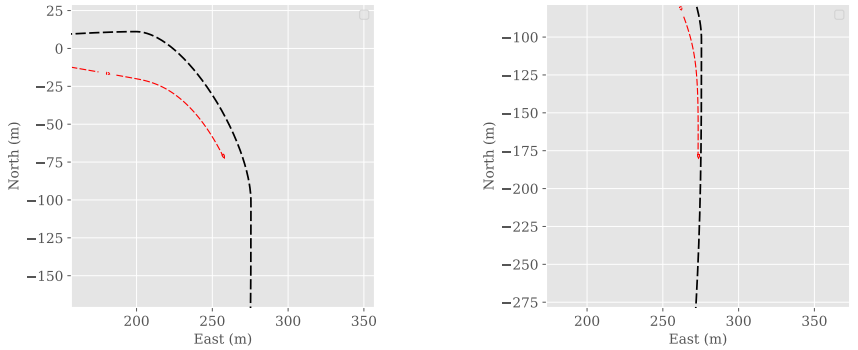
(c) Trondheim test scenario trajectory.

(d) Froan test scenario trajectory.

Figure 5.1: Selected example trajectories from the test scenarios. The vessel trajectories are drawn as red dashed lines.

In addition to the statistical evaluation, snapshots of some selected situations which are

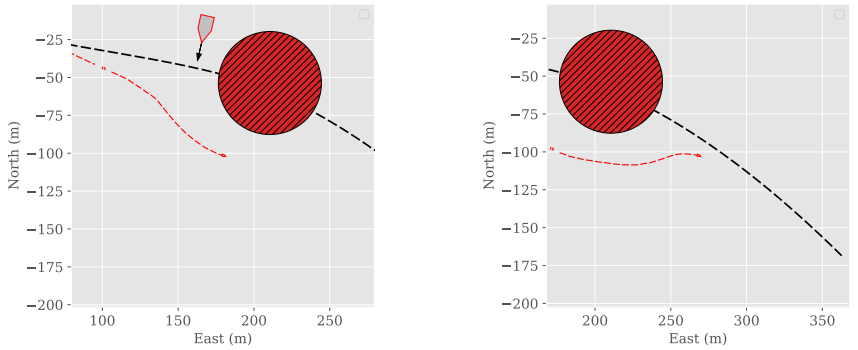
deemed representative for the agent are presented. This should provide some insight into the vessel's guidance behavior in commonly occurring situations. First, the pure path-following ability of the agent is showcased in Figure 5.2, where it is evident that the agent converges to the desired trajectory in a satisfactory manner after having deviated from it. Then, the agent's ability to go around static obstacles to avoid stranding is showcased in Figure 5.3. Here, the agent performs an evasive maneuver to avoid stranding (i.e. colliding with the circular obstacle). Finally, selected examples of the agent's behavior when faced with vessel encounters are shown in Figure 5.4.



(a) Snapshot 1

(b) Snapshop 2 (+5 seconds)

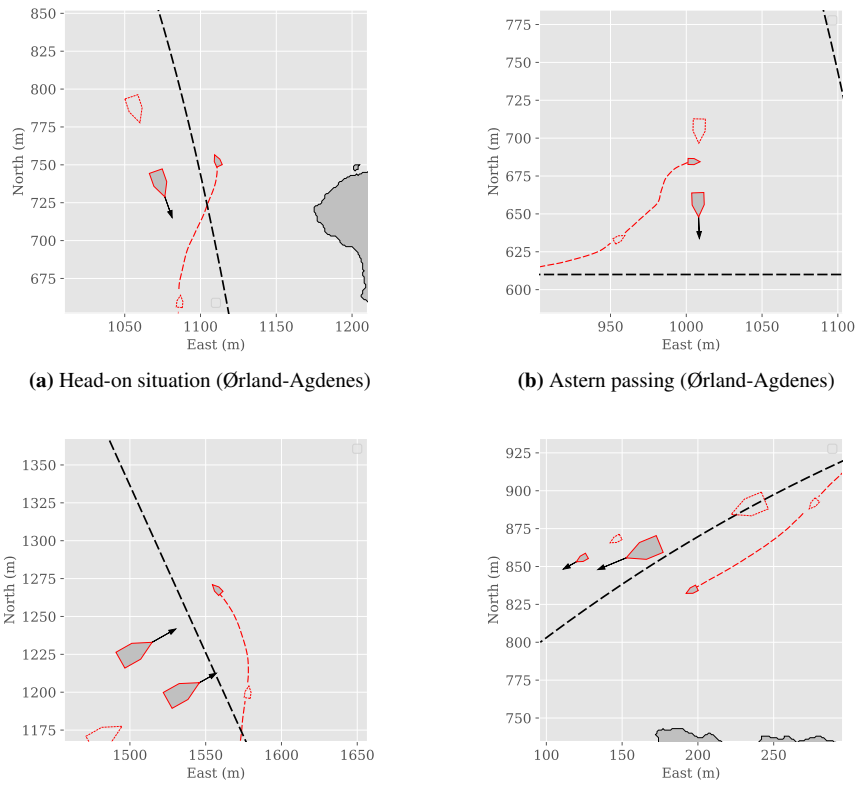
Figure 5.2: Showcase of the path-following ability demonstrated by the agent in one of the training scenarios. After the turn, the agent converges perfectly to the desired trajectory.



(a) Snapshot 1

(b) Snapshot 2 (+5 seconds)

Figure 5.3: Showcase of the agent's ability to perform an evasive maneuver to avoid stranding. Here, the agent makes a correct decision to sacrifice path adherence in order to avoid colliding with the static obstacle.



(a) Head-on situation (Ørland-Agdenes)

(b) Astern passing (Ørland-Agdenes)

(c) Crossing (Trondheim)

(d) Overtaking (Ørland-Agdenes)

Figure 5.4: Agent performing common naval collision avoidance maneuvers upon encountering other vessels.

5.3 Conclusion

Our results suggest that DRL agents, if trained in a stochastic, generic obstacle environment, are capable of performing complex guidance tasks. Specifically, we have shown that our trained agent to a very satisfactory degree avoids collisions with other vessels while, at the same time, adheres to a desired trajectory without getting stranded.

Furthermore, the successful experiments in simulated real-world-based environments show great promise for the viability of implementing it on a real vessel. As the approach requires no knowledge of the internal dynamics, and allows us to easily adapt the agent behavior by customizing the performance measure, our paper lays the groundwork for further research which may, given equally positive results, bring significant value to the field of autonomous guidance.

Still, it should be noted that DRL algorithms heavily rely on deep neural networks, which entail an enormous amount trained parameters. Interpreting and explaining such networks are very challenging, if not humanly impossible at the current moment. Thus, this is generally considered a drawback for using these algorithms in safety critical applications. However, the results obtained in this study, despite the lack of formal stability and robustness guarantees associated with the approach, do demonstrate the feasibility of achieving reliable and high-performing guidance intelligence - even in a safety critical domain.

Finally, it is important to stress that the disturbance-free environment in which the agent was trained and evaluated does not accurately reflect a real-world marine environment, even if it perhaps could be reproduced for a pool operation. This encompasses not only disturbances to the vessel dynamics, but also measurement noise. The lack of such disturbances in our software simulation have undoubtedly simplified the guidance task considered in this work.

5.4 Suggestions for future work

5.4.1 Implicit handling of dynamic COLAV

In this work, we have, in a feed-forward manner, engineered the agent's observation vector using velocity attributes of the simulated obstacles. Of course, this approach is not fully realistic, as velocity measurements may not in general be obtainable given a standard rangefinder sensor suite. Luckily, methods for estimating such velocities based on recent sensor reading can be found in literature; object tracking is a well-researched computer vision discipline (22), and should be considered as an approach to make the DRL guidance system more easily transferable to a real-world application. Also, further research should be directed at investigating whether recurrent approaches, i.e. using a recurrent neural network as policy network, can be applied successfully for this purpose. Here, the idea would be that, based on sequential data processing, where historic (i.e. recent) sensor values are fed into the network, the agent could make deliberate guidance decisions where the dynamic properties of the obstacle environments are taken into account in an implicit manner.

5.4.2 COLREGs compliance

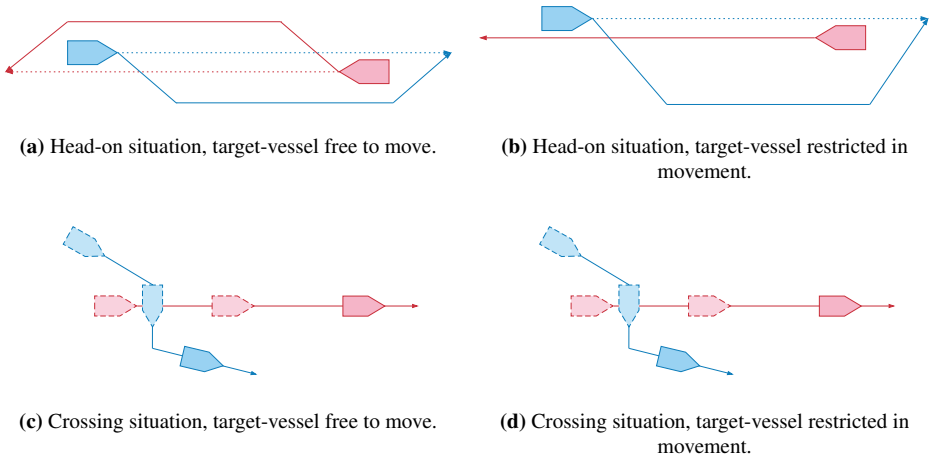
Adherence to the International Regulations for Preventing Collisions at Sea (COLREGs) (28) is a central part of maritime navigation. These rules mandate how vessels should behave upon ship encounters. In this work, we have not considered adherence to these as a part of the DRL agent's mission statement; instead, the agent was rewarded simply for obstacle avoidance, without concern for whether its evasive maneuvers are in line with COLREG regulation. However, by incorporating the central COLREG rules in the reward function, it is likely that the agent's behavior policy could be altered in such a manner that it would become COLREG-compliant. In particular, implementing the two rules listed below in the reward function design would be a huge step towards COLREG-compliance:

Rule 14: Head-on situation

(a) When two power-driven vessels are meeting on reciprocal or nearly reciprocal courses so as to involve risk of collision each shall alter her course to starboard so that each shall pass on the port side of the other.

Rule 15: Crossing situation

When two power-driven vessels are crossing so as to involve risk of collision, the vessel which has the other on her own starboard side shall keep out of the way and shall, if the circumstances of the case admit, avoid crossing ahead of the other vessel.



(a) Head-on situation, target-vessel free to move.

(b) Head-on situation, target-vessel restricted in movement.

(c) Crossing situation, target-vessel free to move.

(d) Crossing situation, target-vessel restricted in movement.

Figure 5.5: Expected behavior from the own-ship (colored in blue) in head-on and crossing encounters according to COLREGs.

5.4.3 Multi-agent environments

In this study, we have considered the collision avoidance problem from the perspective of a single agent. Even during testing, the other vessels were interpreted simply as moving obstacle with pre-determined trajectories. This does not allow for any form of collaboration between the vessels, something which, if facilitated, could present opportunities for co-operative guidance missions which could have useful applications in the real world.

5.4.4 Increased realism

In this work, we have neglected environmental forces such as wind, waves and current, which can pose a serious challenge to a marine vessel. As can be read in (17), such effects

can be accurately represented in the mathematical vessel dynamics, and thus incorporated in the simulation. If a static disturbance, such as an ocean current, was applied to the vessel, there is reason to believe that it would be within the agent's reach to learn a control policy that counters the effect of the disturbance. This is backed by the results in (37), where a path following DRL agent was successfully trained even under the influence of ocean currents. The more challenging case of dynamic disturbances, such as winds with varying direction and speed, would likely require us to extend the vessel's sensor suite to access or estimate the current disturbance characteristics.

5.4.5 Other application domains

Even though autonomous marine vessels has been the the focus of this project, the technology explored is likely to be applicable to other domains as well, including especially (but not exhaustively) unmanned aerial vehicles (UAVs) and autonomous underwater vehicles (AUVs). Here, a major challenge will be the introduction of a third movement direction. Given the increased computational challenges associated with sensor scanning in 3D, it is clear that intelligent sensor data pre-processing and dimensionalty reduction would be of even higher importance. Unfortunately, extending the feasibility pooling algorithm 2 to three dimensions is a highly non-trivial endeavor. Thus, research should be done in this direction to investigate how this or similar approaches can be made feasible.

Bibliography

[1] BELLETER, D. J. W., MAGHENEM, M., PALIOTTA, C., AND PETTERSEN, K. Y. Observer based path following for underactuated marine vessels in the presence of ocean currents: A global approach - with proofs, 2018.

[2] BITAR, G., BREIVIK, M., AND LEKKAS, A. M. Energy-optimized path planning for autonomous ferries. *IFAC-PapersOnLine 51*, 29 (2018), 389 – 394. 11th IFAC Conference on Control Applications in Marine Systems, Robotics, and Vehicles CAMS 2018.

[3] BORENSTEIN, J., AND KOREN, Y. The vector field histogram - fast obstacle avoidance for mobile robots. *Robotics and Automation, IEEE Transactions on 7* (07 1991), 278 – 288.

[4] BREIVIK, M., AND FOSSEN, T. I. Path following for marine surface vessels. In *Oceans '04 MTS/IEEE Techno-Ocean '04 (IEEE Cat. No.04CH37600)* (2004), vol. 4, pp. 2282–2289 Vol.4.

[5] BROCK, O., AND KHATIB, O. High-speed navigation using the global dynamic window approach. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)* (May 1999), vol. 1, pp. 341–346 vol.1.

[6] BROCKMAN, G., CHEUNG, V., PETTERSSON, L., SCHNEIDER, J., SCHULMAN, J., TANG, J., AND ZAREMBA, W. Openai gym, 2016.

[7] CAHARIJA, W., PETTERSEN, K. Y., BIBULI, M., CALADO, P., ZEREIK, E., BRAGA, J., GRAVDAHL, J. T., SØRENSEN, A. J., MILOVANOVIĆ, M., AND BRUZZONE, G. Integral line-of-sight guidance and control of underactuated marine vehicles: Theory, simulations, and experiments. *IEEE Transactions on Control Systems Technology 24*, 5 (2016), 1623–1642.

[8] CHEN, Y., PENG, H., AND GRIZZLE, J. Obstacle avoidance for low-speed autonomous vehicles with barrier function. *IEEE Transactions on Control Systems Technology 26*, 1 (Jan 2018), 194–206.

[9] CODEVILLA, F., SANTANA, E., LÓPEZ, A. M., AND GAIDON, A. Exploring the limitations of behavior cloning for autonomous driving, 2019.

[10] DHARIWAL, P., HESSE, C., KLIMOV, O., NICHOL, A., PLAPPERT, M., RADFORD, A., SCHULMAN, J., SIDOR, S., WU, Y., AND ZHOKHOV, P. Openai baselines. `https://github.com/openai/baselines`, 2017.

[11] ERIKSEN, B. H., BREIVIK, M., PETTERSEN, K. Y., AND WIIG, M. S. A modified dynamic window algorithm for horizontal collision avoidance for auvs. In *2016 IEEE Conference on Control Applications (CCA)* (Sep. 2016), pp. 499–506.

[12] ERIKSEN, B.-O., BREIVIK, M., WILTHIL, E., FLÅTEN, A., AND BREKKE, E. The branching-course mpc algorithm for maritime collision avoidance. *Journal of Field Robotics 36* (06 2019), 1222–1249.

[13] EUROPEAN MARITIME SAFETY AGENCY. Annual overview of marine casualties and incidents 2019, 2019. Available at `http://www.emsa.europa.eu/emsa-documents/latest/download/5854/3734/23.html`.

[14] EUROPEAN MARITIME SAFETY AGENCY. Marine casualties and incidents - preliminary annual overview of marine casualties and incidents 2014-2019, 2020. Available at `http://www.emsa.europa.eu/accident-investigation-publications/annual-overview/download/6132/2713/23.html`.

[15] FEHLBERG, E. Klassische runge-kutta-formeln vierter und niedrigerer ordnung mit schrittweiten-kontrolle und ihre anwendung auf wärmeleitungsprobleme. *Computing 6*, 1 (Mar 1970), 61–71.

[16] FIORINI, P., AND SHILLER, Z. Motion planning in dynamic environments using velocity obstacles. *I. J. Robotics Res. 17*, 7 (1998), 760–772.

[17] FOSSEN, T. *Handbook of Marine Craft Hydrodynamics and Motion Control*, 1 ed. 05 2011.

[18] FOSSEN, T. I. *Handbook of Marine Craft Hydrodynamics and Motion Control*. 05 2011.

[19] FOSSEN, T. I., BREIVIK, M., AND SKJETNE, R. Line-of-sight path following of underactuated marine craft. *IFAC Proceedings Volumes 36*, 21 (2003), 211–216.

[20] FOX, D., BURGARD, W., AND THRUN, S. The dynamic window approach to collision avoidance. *IEEE Robotics Automation Magazine 4*, 1 (March 1997), 23–33.

[21] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. The MIT Press, 2016.

[22] GRANSTROM, K., BAUM, M., AND REUTER, S. Extended object tracking: Introduction, overview and applications, 2016.

[23] GUERRERO, J., TORRES, J., CREUZE, V., AND CHEMORI, A. Observation-based nonlinear proportional–derivative control for robust trajectory tracking for autonomous underwater vehicles. *IEEE Journal of Oceanic Engineering* (2019), 1–13.

[24] GUO, S., ZHANG, X., ZHENG, Y., AND DU, Y. An autonomous path planning model for unmanned ships based on deep reinforcement learning. *Sensors 20*, 2 (Jan 2020), 426.

[25] HAGEN, I. B., KUFOALOR, D. K. M., BREKKE, E. F., AND JOHANSEN, T. A. Mpc-based collision avoidance strategy for existing marine vessel guidance systems. In *2018 IEEE International Conference on Robotics and Automation (ICRA)* (May 2018), pp. 7618–7623.

[26] HENDERSON, P., ISLAM, R., BACHMAN, P., PINEAU, J., PRECUP, D., AND MEGER, D. Deep reinforcement learning that matters, 2017.

[27] HILL, A., RAFFIN, A., ERNESTUS, M., GLEAVE, A., KANERVISTO, A., TRAORE, R., DHARIWAL, P., HESSE, C., KLIMOV, O., NICHOL, A., PLAPPERT, M., RADFORD, A., SCHULMAN, J., SIDOR, S., AND WU, Y. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[28] INTERNATIONAL MARITIME ORGANIZATION. Colregs - international regulations for preventing collisions at sea, 1972.

[29] KHATIB, O. Real-time obstacle avoidance for manipulators and mobile robots. *Int. J. Rob. Res. 5*, 1 (Apr. 1986), 90–98.

[30] KOREN, Y., AND BORENSTEIN, J. Potential field methods and their inherent limitations for mobile robot navigation. In *Proceedings. 1991 IEEE International Conference on Robotics and Automation* (April 1991), pp. 1398–1404 vol.2.

[31] KUFOALOR, D., BREKKE, E., AND JOHANSEN, T. Proactive collision avoidance for asvs using a dynamic reciprocal velocity obstacles method. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems* (10 2018), pp. 2402–2409.

[32] LIANG, E., LIAW, R., MORITZ, P., NISHIHARA, R., FOX, R., GOLDBERG, K., GONZALEZ, J. E., JORDAN, M. I., AND STOICA, I. Rllib: Abstractions for distributed reinforcement learning. *arXiv preprint arXiv:1712.09381* (2017).

[33] LILLICRAP, T. P., HUNT, J. J., PRITZEL, A., HEESS, N., EREZ, T., TASSA, Y., SILVER, D., AND WIERSTRA, D. Continuous control with deep reinforcement learning, 2015.

[34] LIN, C., WANG, H., YUAN, J., YU, D., AND LI, C. An improved recurrent neural network for unmanned underwater vehicle online obstacle avoidance. *Ocean Engineering 189* (2019), 106327.

[35] LIN, C., WANG, H., YUAN, J., YU, D., AND LI, C. Research on uuv obstacle avoiding method based on recurrent neural networks. *Complexity 2019* (2019), 6320186:1–6320186:16.

[36] LIPTON, Z. C., BERKOWITZ, J., AND ELKAN, C. A critical review of recurrent neural networks for sequence learning, 2015.

[37] MARTINSEN, A. B. End-to-end training for path following and control of marine vehicles, 2018.

[38] MARTINSEN, A. B., AND LEKKAS, A. M. Curved path following with deep reinforcement learning: Results from three vessel models. In *OCEANS 2018 MTS/IEEE Charleston* (2018), pp. 1–8.

[39] MARTINSEN, A. B., AND LEKKAS, A. M. Straight-path following for underactuated marine vessels using deep reinforcement learning. *IFAC-PapersOnLine 51*, 29 (2018), 329–334.

[40] MARTINSEN, A. B., LEKKAS, A. M., GROS, S., GLOMSRUD, J. A., AND PEDERSEN, T. A. Reinforcement learning-based tracking control of usvs in varying operational conditions. *Frontiers in Robotics and AI 7* (2020), 32.

[41] MEYER, E. Python simulation framework based on openai gym for drl-based collision avoidance and path following for unmanned surface vehicle, 2019-. Available at `https://github.com/EivMeyer/gym-auv`.

[42] MEYER, E., ROBINSON, H., RASHEED, A., AND SAN, O. Taming an autonomous surface vehicle for path following and collision avoidance using deep reinforcement learning. *IEEE Access 8* (2020), 41466–41481.

[43] MITCHELL, I. M., BAYEN, A. M., AND TOMLIN, C. J. A time-dependent hamilton-jacobi formulation of reachable sets for continuous dynamic games. *IEEE Transactions on Automatic Control 50*, 7 (July 2005), 947–957.

[44] MNIH, V., BADIA, A. P., MIRZA, M., GRAVES, A., LILLICRAP, T. P., HARLEY, T., SILVER, D., AND KAVUKCUOGLU, K. Asynchronous methods for deep reinforcement learning, 2016.

[45] MNIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A., VENESS, J., BELLEMARE, M., GRAVES, A., RIEDMILLER, M., FIDJELAND, A., OSTROVSKI, G., PETERSEN, S., BEATTIE, C., SADIK, A., ANTONOGLOU, I., KING, H., KUMARAN, D., WIERSTRA, D., LEGG, S., AND HASSABIS, D. Human-level control through deep reinforcement learning. *Nature 518* (02 2015), 529–33.

[46] MOE, S., CAHARIJA, W., PETTERSEN, K. Y., AND SCHJØLBERG, I. Path following of underactuated marine underwater vehicles in the presence of unknown ocean currents. In *ASME 2014 33rd International Conference on Ocean, Offshore and Arctic Engineering* (2014), American Society of Mechanical Engineers, pp. V007T05A014–V007T05A014.

[47] MOE, S., AND PETTERSEN, K. Y. Set-based line-of-sight (los) path following with collision avoidance for underactuated unmanned surface vessel. In *2016 24th Mediterranean Conference on Control and Automation (MED)* (2016), IEEE, pp. 402–409.

[48] MÍNGUEZ, J., AND MONTANO, L. Robot navigation in very complex, dense, and cluttered indoor/outdoor environments. *IFAC Proceedings Volumes 35*, 1 (2002), 397 – 402. 15th IFAC World Congress.

[49] NORWEGIAN MAPPING AUTHORITY. Høydedata og terrengmodeller for landområdene, Mar 2019. Available at `https://www.kartverket.no/data/hoydedata-og-terrengmodeller/`.

[50] PALIOTTA, C., LEFEBER, E., PETTERSEN, K. Y., PINTO, J., COSTA, M., AND DE FIGUEIREDO BORGES DE SOUSA, J. T. Trajectory tracking and path following for underactuated marine vehicles. *IEEE Transactions on Control Systems Technology 27*, 4 (2019), 1423–1437.

[51] PANAGOU, D. Motion planning and collision avoidance using navigation vector fields. *Robotics and Automation, IEEE Transactions on* (10 2014).

[52] REIS, M. F., JAIN, R. P., AGUIAR, A. P., AND DE SOUSA, J. B. Robust moving path following control for robotic vehicles: Theory and experiments. *IEEE Robotics and Automation Letters 4*, 4 (2019), 3192–3199.

[53] SCHULMAN, J., LEVINE, S., MORITZ, P., JORDAN, M. I., AND ABBEEL, P. Trust region policy optimization. *CoRR abs/1502.05477* (2015).

[54] SCHULMAN, J., MORITZ, P., LEVINE, S., JORDAN, M., AND ABBEEL, P. High-dimensional continuous control using generalized advantage estimation.

[55] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., RADFORD, A., AND KLIMOV, O. Proximal policy optimization algorithms, 2017.

[56] SEAN GILLIES AND OTHERS. Shapely: Manipulation and analysis of geometric objects, 2007–. Available at `https://github.com/Toblerity/Shapely`.

[57] SKJETNE, R., SMOGELI, Ø. N., AND FOSSEN, T. I. A nonlinear ship manoeuvering model: Identification and adaptive control with experiments for a model ship.

[58] SKJETNE, R., ØYVIND SMOGELI, AND FOSSEN, T. I. Modeling, identification, and adaptive maneuvering of cybership ii: A complete design with experiments. *IFAC Proceedings Volumes 37*, 10 (2004), 203 – 208. IFAC Conference on Computer Applications in Marine Systems - CAMS 2004, Ancona, Italy, 7-9 July 2004.

[59] SOCIETY OF NAVAL ARCHITECTS AND MARINE ENGINEERS (U.S.). TECHNICAL AND RESEARCH COMMITTEE. HYDRODYNAMICS SUBCOMMITTEE. *Nomenclature for Treating the Motion of a Submerged Body Through a Fluid: Report of the American Towing Tank Conference*. Technical and research bulletin. Society of Naval Architects and Marine Engineers, 1950.

[60] SUTTON, R., MCALLESTER, D., SINGH, S., AND MANSOUR, Y. Policy gradient methods for reinforcement learning with function approximation. *Adv. Neural Inf. Process. Syst 12* (02 2000).

[61] SUTTON, R. S., AND BARTO, A. G. *Reinforcement Learning: An Introduction*, second ed. The MIT Press, 2018.

[62] SUTTON, R. S., ET AL. *Introduction to reinforcement learning*, vol. 135.

[63] S⊘RENSEN, M. E. N., BREIVIK, M., AND ERIKSEN, B. H. A ship heading and speed control concept inherently satisfying actuator constraints. In *2017 IEEE Conference on Control Technology and Applications (CCTA)* (2017), pp. 323–330.

[64] TAI, L., ZHANG, J., LIU, M., BOEDECKER, J., AND BURGARD, W. A survey of deep network solutions for learning control in robotics: From reinforcement to imitation, 2016.

[65] VAN ANDEL, B., BIENIEK, T., AND BØ, T. I. Bidirectional utm-wgs84 converter for python, 2012–. Available at `https://github.com/Turbo87/utm`.

[66] VINYALS, O., BABUSCHKIN, I., CZARNECKI, W., MATHIEU, M., DUDZIK, A., CHUNG, J., CHOI, D., POWELL, R., EWALDS, T., GEORGIEV, P., OH, J., HORGAN, D., KROISS, M., DANIHELKA, I., HUANG, A., SIFRE, L., CAI, T., AGAPIOU, J., JADERBERG, M., AND SILVER, D. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature 575* (11 2019).

[67] VIRTANEN, P., GOMMERS, R., OLIPHANT, T. E., HABERLAND, M., REDDY, T., COURNAPEAU, D., BUROVSKI, E., PETERSON, P., WECKESSER, W., BRIGHT, J., VAN DER WALT, S. J., BRETT, M., WILSON, J., JARROD MILLMAN, K., MAYOROV, N., NELSON, A. R. J., JONES, E., KERN, R., LARSON, E., CAREY, C., POLAT, İ., FENG, Y., MOORE, E. W., VAND ERPLAS, J., LAXALDE, D., PERKTOLD, J., CIMRMAN, R., HENRIKSEN, I., QUINTERO, E. A., HARRIS, C. R., ARCHIBALD, A. M., RIBEIRO, A. H., PEDREGOSA, F., VAN MULBREGT, P., AND CONTRIBUTORS, S. . . SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints* (Jul 2019), arXiv:1907.10121.

[68] WOO, J., YU, C., AND KIM, N. Deep reinforcement learning-based controller for path following of an unmanned surface vehicle. *Ocean Engineering 183* (2019), 155 – 166.

[69] YAN, Z., ZHAO, Y., HOU, S., ZHANG, H., AND ZHENG, Y. Obstacle avoidance for unmanned undersea vehicle in unknown unstructured environment. *Mathematical Problems in Engineering 2013* (11 2013), 1–12.

[70] ZHANG, Q., LIN, J., SHA, Q., HE, B., AND LI, G. Deep interactive reinforcement learning for path following of autonomous underwater vehicle, 2020.

[71] ZHAO, L., AND ROH, M.-I. Colregs-compliant multiship collision avoidance based on deep reinforcement learning. *Ocean Engineering 191* (2019), 106436.

Appendix

Training plots

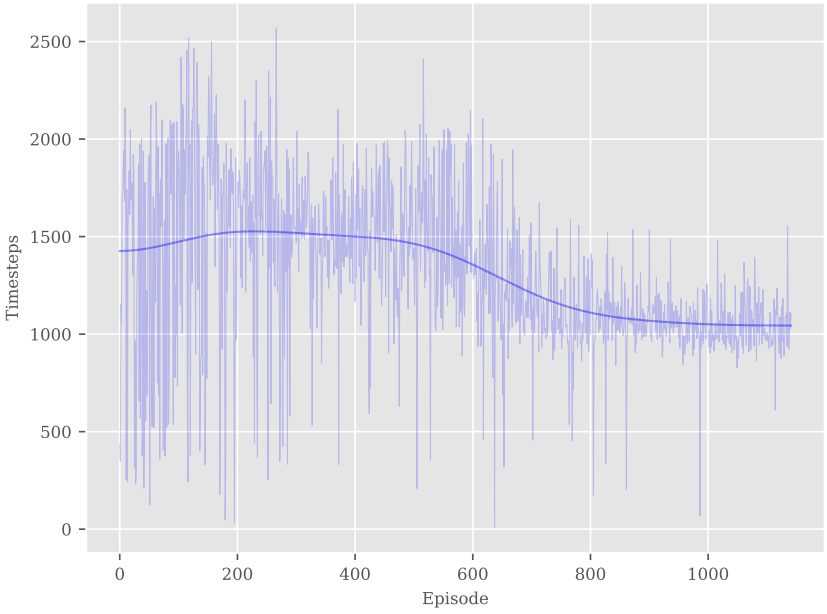


Figure A.1: Episode timesteps (i.e. duration of episode) during training process.

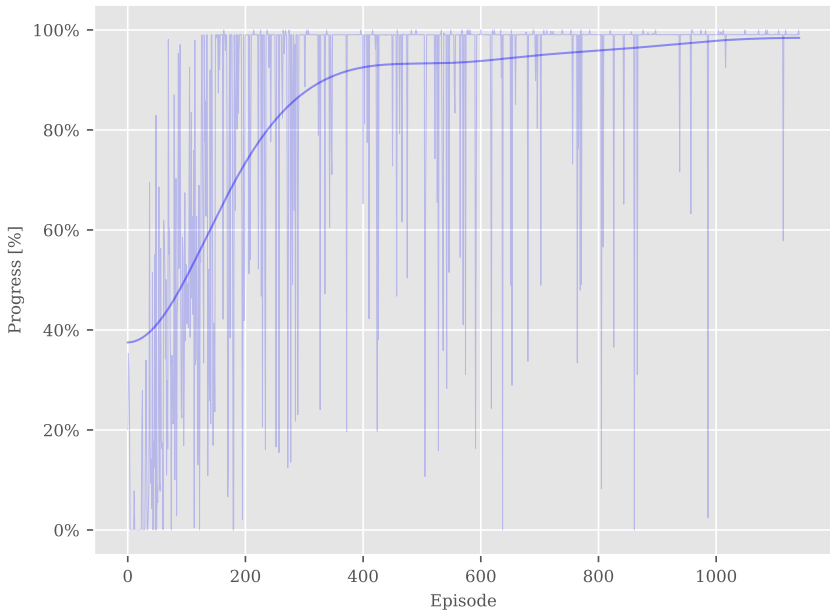


Figure A.2: Achieved progress (i.e. percentage of full path distance travelled at the end of the episode).

Training scenario algorithm

Algorithm 3 Generate and simulate random training scenario with moving obstacles.

Require:

Number of static obstacles $N_{o,stat} \in \mathbb{N}_0$
Number of dynamic obstacles $N_{o,dyn} \in \mathbb{N}_0$
Number of path waypoints $N_w \in \mathbb{N}_0$
Path length $L_p \in \mathbb{N}_0$
Mean static obstacle radius $\mu_{r,stat} \in \mathbb{R}^+$
Mean dynamic obstacle radius $\mu_{r,dyn} \in \mathbb{R}^+$
Obstacle displacement distance standard deviation $\sigma_d \in \mathbb{R}^+$
procedure SIMULATETRAININGENVIRONMENT($N_{o,stat}$, $N_{o,dyn}$, N_w, L_p, $\mu_{r,stat}$, $\mu_{r,dyn}$m σ_d)
Draw θ_{start} from *Uniform*$(0, 2\pi)$
Set of static obstacles $\mathcal{O}_{stat} \leftarrow \{\}$
Set of moving obstacles $\mathcal{O}_{dyn} \leftarrow \{\}$
Path origin $\boldsymbol{p}_{start} \leftarrow 0.5L_p \left[\cos\left(\theta_{start}\right), \sin\left(\theta_{start}\right)\right]^T$
Goal position $\boldsymbol{p}_{end} \leftarrow -\boldsymbol{p}_{start}$
Generate N_w random waypoints between $\boldsymbol{p}_{start}$ and $\boldsymbol{p}_{end}$.
Create smooth arc length parameterized path $\boldsymbol{p}_p(\bar{\omega}) = [x_p(\bar{\omega}), y_p(\bar{\omega})]^T$ from 1D Piecewise Cubic Hermite Interpolation (PCHIP).
repeat
Draw arclength $\bar{\omega}_{obst}$ from *Uniform*$(0.1L_p, 0.9L_p)$.
Draw obstacle displacement distance d_{obst} from $\mathcal{N}(0, \sigma_d^2)$
Path angle $\gamma_{obst} \leftarrow \operatorname{atan2}\left(\boldsymbol{p}_p{}'(\bar{\omega}_{obst})_2, \boldsymbol{p}_p{}'(\bar{\omega}_{obst})_1\right)$
$\boldsymbol{p}_{obst} \leftarrow \boldsymbol{p}_p(\bar{\omega}_{obst}) + d_{obst}[\cos\left(\gamma_{obst} - \frac{\pi}{2}\right), \sin\left(\gamma_{obst} - \frac{\pi}{2}\right)]^T$
Draw obstacle radius r_{obst} from *Poisson*$(\mu_{r,stat})$.
Add static obstacle $(\boldsymbol{p}_{obst}, r_{obst})$ to $\mathcal{O}_{stat}$.
until $N_{o,stat}$ static obstacles are created
repeat
Draw arclength $\bar{\omega}_{obst}$ from *Uniform*$(0.1L_p, 0.9L_p)$.
Draw obstacle displacement distance d_{obst} from $\mathcal{N}(0, \sigma_d^2)$
Path angle $\gamma_{obst} \leftarrow \operatorname{atan2}\left(\boldsymbol{p}_p{}'(\bar{\omega}_{obst})_2, \boldsymbol{p}_p{}'(\bar{\omega}_{obst})_1\right)$
$\boldsymbol{p}_{obst} \leftarrow \boldsymbol{p}_p(\bar{\omega}_{obst}) + d_{obst}[\cos\left(\gamma_{obst} - \frac{\pi}{2}\right), \sin\left(\gamma_{obst} - \frac{\pi}{2}\right)]^T$
Draw obstacle radius r_{obst} from *Poisson*$(\mu_{r,dyn})$.
Draw movement direction ψ_{obst} from *Uniform*$(0, 2\pi)$.
Draw phase shift ϕ_{obst} from *Uniform*$(0, 2\pi)$.
Add moving obstacle $(\boldsymbol{p}_{obst}, r_{obst}, \psi_{obst}, \phi_{obst})$ to $\mathcal{O}_{dyn}$.
until $N_{o,dyn}$ moving obstacles are created
Time-step $t \leftarrow 0$
repeat
for $\boldsymbol{p}_{obst}, r_{obst}, \psi_{obst}, \phi_{obst} \in \mathcal{O}_{dyn}$ **do**
$dx \leftarrow \frac{1}{r} \sin\left(S_0 t + \phi_{obst}\right) \cos\psi_{obst}$
$dy \leftarrow \frac{1}{r} \sin\left(S_0 t + \phi_{obst}\right) \sin\psi_{obst}$
$\boldsymbol{p}_{obst} \leftarrow \boldsymbol{p}_{obst} + [dx, dy]^T$
$t \leftarrow t + 1$
until End of episode