Engineering Cybernetics
TTK4551 Project Thesis

Machine Learning in Unity

Simon Mork Sætre

Supervisor:
Professor Adil Rasheed

Trondheim, June 9th, 2022

NTNU
Norwegian University of
Science and Technology

Faculty of Information Technology and Electrical Engineering
DEPARTMENT OF ENGINEERING CYBERNETICS

Preface

The work done here is the process, results, and discoveries done during the specialisation project in Autumn 2021 at the Norwegian University of Science and Technology (NTNU). The goal of this specialisation project is to explore the Unity Game Engine and the machine learning capabilities of it. This project is intended to be the pre-phase of a master project built with this project as a foundation, aiming at completion around Summer 2022. Please note the fact that this specialisation project is TTK4551 and not TTK4550. This means the project is 7.5 points and not the usual 15 points.

I would like to thank my supervisor Adil Rasheed for his guidance and logistical assistance. I would also like to thank the PhD student Thomas Nakken Larsen for guidance regarding reinforcement learning. I am grateful for their contribution of knowledge, guidance, and time.

Contents

List of Figures

List of Tables

Abstract

Reinforcement learning is a way of developing an algorithm that is able to control systems. In many cases these algorithms are more capable than what humans are able to program themselves. In some cases the algorithm is able to replace manual control. This can be quite useful in stochastic systems where programming algorithms by hand is near impossible. In this specialisation project, the Unity Game Engine's capabilities and limitations are explored. One approach of importing real life maps is made, a physics model of the Milliampere boat is used here for training the algorithm so it can in theory be used to control a real ship.

The algorithm was able to control the boat with surprising ease, but the model of the vessel had to be simplified because of limitations of the Game Engine. It was tasked to follow a curve, and was tested on a more advanced curve. It showed promising results, but showed signs of not being generalised for the environment. Future work will go into improving the performance of the algorithm, expand it to also support collision and vision, improve the mathematical modeling of the vessel, and real life data for instance wind and waves.

Chapter 1
Introduction

1.1 Motivation and Background

As the world moves onward, more of the world's jobs will be automated and replaced by autonomous machines performing the same job at same or higher efficiency as humans. In the most recent decades, researchers have looked more and more into turning vehicles autonomous, as humans are far from perfect pilots. As a matter of fact, 80% of the accidents at sea happen because of human error Sánchez-Beaskoetxea et al. (2021). Replacing human pilots at sea is thus important if one would like to reduce the economical costs of accidents and to prevent the loss of human lives. Replacing human sea vessel pilots with a regular AI is not a trivial task, as the marine environment is highly stochastic and with near infinite different scenarios. This makes a conventional manually programmed deterministic AI unfeasible. One has instead changed focus towards Machine Learning (ML) instead for creating an AI suitable for sea navigation. This means that one can let the ML program train in a virtual environment and run the simulation until it is sufficient at navigating the virtual environment. This fully trained AI can then be applied in the real world faster than a deterministic AI would ever be. While more conventional machine learning projects create their simulation environment from scratch, this project will try to see how feasible it is to instead use Unity Game Engine. By using an engine with a pre-made assets and libraries, one can possibly speed up the development process significantly. One does not need to program any visualisation as a 3D environment is already made by default. Unity abstracts complex physics and vector calculations and hides them behind functions such that the developer does not need to spend as much time programming advanced mathematics. Unity Technologies, the company behind Unity, has also created a ML library for handling machine learning in the Unity engine. This library handles the creation and training of the Deep Neural Networks, and creates a communication layer between the game engine and the algorithm run in Python. This project is heavily based on the research paper Thomas Nakken Karlsen et al. (2021). While the project relies on this paper, it also wants to further explore autonomous RL navigation of ships and implement the simulation on a more user friendly, robust, and flexible platform.

1.1.1 Literature Review

The paper Thomas Nakken Karlsen et al. (2021) manages to implement an unmanned marine vessel in a Python simulation. It explores different reinforcement learning approaches such as Proximal Policy Optimization (PPO), Deep Deterministic Policy Gradient (DDPG), Twin Delayed DDPG (TD3), Soft Actor-Critic (SAC). These algorithms have different advantages and disadvantages, for instance by being off-policy or robust. The research paper discovers that for RL autonomous sea navigation, PPO is significantly better for the task as it is more stable, easy to implement, and model-free. However, this paper does not apply any disturbances such as wind and sea currents and instead assumes the sea is perfectly calm with no wind. It is yet to be proven if PPO still is the best choice in this scenario.

1.2 Research Objectives and research questions

This project needs some concrete objectives and questions to guide the work onwards.

1.2.1 Objectives

The objectives of this specialisation project can be divided into main tasks and secondary tasks: Main Objective:

1. Develop a reinforcement learning algorithm that can control a simulated boat in Unity.

Secondary Objectives:

1. Import real world map data into the environment.
2. Create an accurate dynamical model of the Milliampere boat.
3. Implement vision and collision for the algorithm.

1.2.2 Research Questions

The Unity Game Engine is primarily a game engine and is not the most well known toolkit for creating simulations, let alone reinforcement learning environments. For this project, one needs to propose a few questions that are essential to this.

1. Is Unity Game Engine a viable toolkit for developing a reinforcement learning environment?
2. Is there really any point in using Unity for reinforcement learning over more conventional reinforcement learning toolkits?

1.3 Outline of Report

This specialisation project contains the following chapters: Chapter 2 contains theory regarding the dynamic model of the boat and thrusters. In addition contains it theory about machine learning before dipping into reinforcement learning. Chapter 3 describes how real world terrain was imported into Unity, here it shows the Trondheim Fjord being imported. It then describes the process of working with Unity together with machine learning. After that it describes the training process of the reinforcement learning algorithm. Chapter 4 shows the performance results of the reinforcement learning algorithm and showcases its performance on advanced paths. At the final chapter, Chapter 5, the project is concluded and suggestions for future work and improvements for a master thesis is given.

Chapter 2
Theory

2.1 Boat dynamics

In marine cybernetics, it is normal to use right-handed 3D cartesian coordinate frames that are either NED (North, East, Down) or ENU (East, North, Up). For this project, NUW (North, Up, West) coordinate frame is used instead such that NUW is x, y, z respectively. This coordinate frame is left-handed and is what Unity Game Engine uses.

For the boat dynamics, one can use a standard differential equation to represent the dynamics. This system is based on the dynamics developed for the Milliampere boat Pedersen (2019).

$$M\ddot{p} + D\dot{p} + R = \tau \tag{2.1.1}$$

$$p = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.1.2}$$

The p vector 2.1.2 represents the position of the boat using the NUW system, which means x is forward, y is up, and z is left. M is the mass of the vessel. The original M is a 3x3 matrix where the inertia affecting the system is dependent on the direction of the vessel. This is mostly because of the vessel also has to push the water along with it. For simulation purposes and simplicity, this matrix has been replaced by a constant mass. D is in the original paper a dampening matrix containing both linear and non-linear elements. Unity does not have native support for matrix calculations, so the dynamics of the system has to be simplified to be more manageable in Unity. The dampening vector is given in equation 2.1.3.

$$\begin{bmatrix} x_d \\ z_d \end{bmatrix} = \begin{bmatrix} X_u + X_{uu}x[t] + X_{uuu}x[t]^2 \\ Z_v + Z_{vv}z[t] + Z_{vvv}Z[t]^2 \end{bmatrix} \tag{2.1.3}$$

For Unity, the dampening matrix has been replaced by a dampening vector 2.1.3 for both x and z direction. A deviation from the real system will happen, as the dynamic system is not quite accurate to a real system. While the dampening vector contains both linear and non-linear elements, the dampening defined in 2.1.3 does not contain the interactions between the velocity directions like they do in the original paper Pedersen (2019). This will cause deviations between the simulated system and the real system.

2.1.1 Thrusters

The thruster dynamics are also derived from the paper about the Milliampere boat Pedersen (2019). The dynamics model is as follows:

$$\tau = T(\alpha)F \tag{2.1.4}$$

$$T(\alpha) = \begin{bmatrix} cos\alpha_1 & cos\alpha_2 \\ sin\alpha_1 & sin\alpha_2 \\ L_x sin\alpha & L_x sin\alpha_2 \end{bmatrix} \tag{2.1.5}$$

F in 2.1.4 is the force each thruster outputs represented in a 2x1 vector. The first column of matrix 2.1.5 shows the thruster force in x direction. Second column is the thruster force in z direction. The final column is the momentum produced by the thrusters, where L_x is the length between the thrusters and center of mass.
The rotation speed of the thruster angle has been simplified. It originally is an S-shaped curve given by:

$$\dot{\alpha} = K_\alpha \frac{(\alpha_d - \alpha)}{\sqrt{(\alpha_d - \alpha)^2 + \epsilon^2}} \tag{2.1.6}$$

where α is the thruster angle, and α_d is the desired angle. K_α and ϵ are tuning parameters for the slope of the angular velocity. This equation has been significantly simplified where it is replaced by a controller with instant response as seen in equation 2.1.7.

$$\dot{\alpha} = K_\alpha N \tag{2.1.7}$$

K_α is the maximum angular velocity and N is the input given by the neural network to adjust the angular velocity of the each individual thruster. This input variable is discrete and outputs the values $-1, 0, 1$. The neural network thus is able to decide if it should rotate the thruster either way, or remain at its current position. One could use a continous input N from the network, but the reinforcement learning network learns faster if a discrete input is used instead.

2.2 Machine Learning

Machine learning, often abbreviated to ML, is a method of developing an artificial intelligence without having a person program the behaviour directly. One simple case for this is to classify information, for example by sending pictures of cats and dogs into an algorithm, where each picture has each "class" marked. The algorithm then from the information given, learns how to classify new unmarked pictures. This method of creating a classifying algorithm is usually faster and easier to develop, but it requires training data, which can be costly depending on the problem.

Reinforcement Learning (RL) has been proven to be quite powerful in many applications. This method of machine learning creates an AI through trial and error, where it trains by interacting with either a real or simulated environment. The AI is being given rewards for performing well, and is punished for performing badly. The goal of the reinforcement algorithm is to maximise this reward.

The AI is then retrained based on the reward score it received during the training episode using gradient descent. One can use this method of machine learning to create powerful AIs which surpass human abilities, like in the case of creating autonomous sea vessels. It is also useful for stochastic systems where programming deterministic behaviour simply becomes infeasible, for instance in noisy environments, systems affected by wind and other external

forces which are hard to predict and take into consideration when programming. This method will unfortunately create a black box where it is hard to review the algorithm and verify if the algorithm will act in a safe and predictable manner.
For this project, the RL algorithm PPO (Proximal Policy Optimisation) is used Schulman et al. (2017). It is a reinforcement algorithm that is high-performing, works for most problems, while remaining simpler than other RL methods to work with. It is the default reinforcement learning method in MLAgents, with SAC (Soft Actor-Critic) as an alternative.

The activation function for the hidden layers used in MLAgents is called Swish Technologies (2021a). The Swish function is defined in equation 2.2.1.

$$f(x) = x \cdot \text{sigmoid}(\beta x) \tag{2.2.1}$$

$$\text{sigmoid}(\beta x) = \frac{1}{1 + e^{-\beta x}} \tag{2.2.2}$$

This activation function has been chosen over the more common Rectified Linear Unit (ReLu) after it has been determined that the function performs better than ReLu in all cases. The Swish function is very similar to ReLu, but it is smoother, contains negative values, and is differentiable everywhere Ramachandran et al. (2017). For reference, the ReLu function is defined in equation 2.2.3.

$$\text{ReLu(x)} = \max(0, x) \tag{2.2.3}$$

When enough steps of the simulation has been recorded, gradient descent is being done on the neural network.

$$x_{(1)} = x_{(0)} - \epsilon \nabla_x f(x) \tag{2.2.4}$$

Chapter 3
Methodology

3.1 Implementation of terrain

For the model to be as realistic as possible, real world data of the terrain has to be used. A point cloud showing height of the Trondheim fjord was ordered from KartverketKartverket (2021). This data is highly accurate with an approximate resolution of one data point per $10m^2$. While highly useful, this data cannot be readily be imported into Unity for terrain generation. It first has to be converted into a heightmap, which is a grayscale image representing the terrain height. The light value of the pixels determine the terrain height on each of the data points.

Kartverket provided the data points in GEOTIFF, which is a common file format for geographic data. This data was then combined into one GEOTIFF image in QGIS. This makes handling the map data drastically easier, but at a cost of map limitations. The map is now limited to one single square. One can attempt to tile the maps, but the height scaling of the maps has to be done correctly to ensure the maps are seamless. To convert this into RAW format, the format Unity uses, the GDAL libraryRouault et al. (2021) was used. This library supports converting the data provided by Kartverket. It was installed by following the instructions provided inRouault et al. (2021). The following command3.1 has to be run to convert the file to one that can be handled by Unity. Navigate to folder containing the .tif and run the following command in terminal:

```
gdal_translate -of ENVI -ot UInt16 -scale -outsize 4097 4097 heightdata.tif
heightmap.raw
```

The data had to be cropped to a resolution of 10 000x10 000 pixels because of Unity engine limitations. Terrain size of 100 000x100 000 units is unfortunately the hard limit of Unity's terrain size. As the data provided by Kartverket has a resolution of one pixel per ten meters, one can simply scale the terrain in Unity to 100 000 x100 000 to make each unit represent one meter. Any higher resolutions would make it impossible to create a map without using map tiles. Creating the terrain with heightmap tiling was attempted, but making the heightmaps align properly was both very time consuming and very prone to misalignment. Simulating on the map as it is right now assumes that the ocean is perfectly flat and with no sea waves or currents.

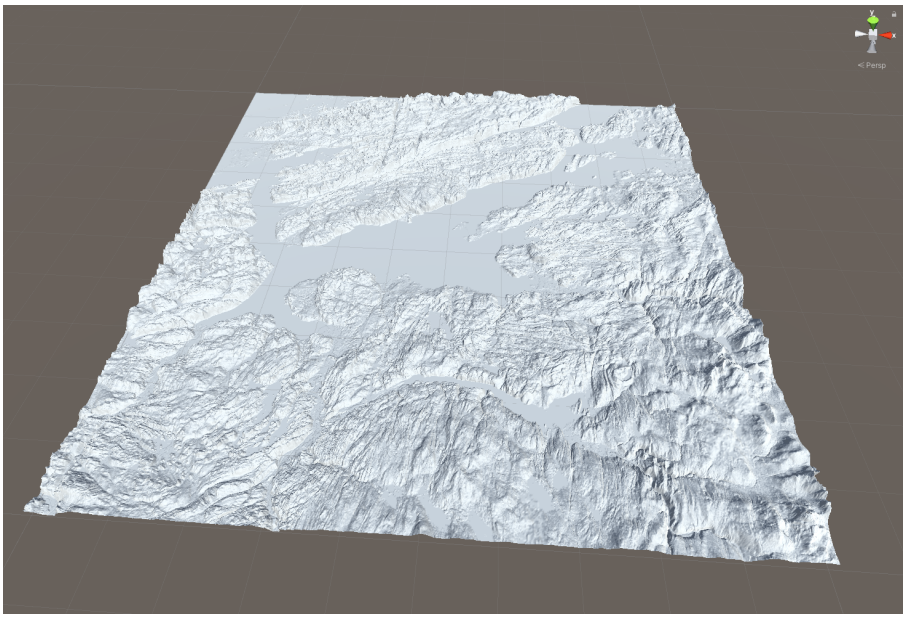

Figure 3.1.1: The entire Trondheim fjord rendered in real time in Unity at a scale of 1 meter per unit in Unity. A total of 10 000 square kilometers

3.2 Unity and Machine Learning

Unity already provides an ML package called ML Agents. This package simplifies the machine learning process by providing ML algorithms from OpenAI. For Reinforcement Learning, one can use two algorithms, Proximal Policy Optimisation (PPO) and Soft-Actor Critic (SAC). PPO is an on-policy algorithm which is useful for this environment, considering the environment can be stochastic. Forces on the vessel provided by wind and sea currents, random starting position and end position, and randomised traveling paths makes this environment stochastic. Because of the stochastic environment, programming an autonomous system by hand would be extremely time consuming and difficult.

3.2.1 Reward Function

The reward function was derived from the works of the RL paperThomas Nakken Karlsen et al. (2021), which again derived its equation from Meyer (2020). The reward function for this RL algorithm is given in the equations below.

$$r_{\text{path}}^{(t)} = \underbrace{\frac{\dot{p}^{(t)}}{U_{\max}}}_{\text{Speed term}} \cdot \underbrace{\frac{1+\cos\left(\psi^{(t)}\right)}{2}}_{\text{Heading term}} \cdot \underbrace{\frac{1}{|\epsilon^{(t)}|+1}}_{\text{Distance term}} \tag{3.2.1}$$

$$r_{\text{exists}} = \lambda\left(2\alpha_r + 1\right) \tag{3.2.2}$$

$$r^{(t)} = \begin{cases} r_{\text{coll}}\,, & \text{if collision} \\ r_{\text{path}}^{(t)} - r_{\text{exists}}\,, & \text{otherwise} \end{cases} \tag{3.2.3}$$

The speed term in equation 3.2.1 makes sure the maximum possible reward given per step is 1. It changes depending on the sea vessel's speed parallel to the path. Just the speed term alone would create issue where the vessel can travel in an arbitrary direction and gain points. The heading term compensates for this by providing a term between 0 and 1, where

Scaling parameter	Interpretation	Value
α_r	Zero-reward relative speed	0.05
r_r	Collision reward	-10000
$\epsilon^{(t)}$	Distance from path	
λ	Objective trade-off coefficient	0.5

Table 3.2.1: The reward parameters used in 3.2.1 and 3.2.3

Parameter name	Explanation	Value
batch size	Amount of steps being performed gradient descent on	512
buffer size	Amount of steps per model training	65536
learning rate	Gradient Descent step size	$1e^{-3}$
β	"Randomness" parameter	$1e^{-2}$
ϵ	Allowed model change	0.2
λ	Regularisation parameter	0.98
epoch	Iterations of gradient descent performed	5
hidden units	The width of each hidden layer in the NN	64
layers	Amount of hidden layers in the NN	2

Table 3.3.1: Parameters used for training

the term is 0 when the vessel faces the opposite direction and 1 when the vessel faces the correct direction. The optimal direction is thus when the vessel heads in the direction parallel to the path. The final term that is missing is the distance term. It scales by the inverse of the absolute distance between the vessel and the path. Minimalising the distance between the vessel and the path will then yield the highest value of 1. This makes sure the vessel stays as close to the path as possible. In addition to the reward for following the path, one also needs to encourage the agent to gain points as fast as possible. The equation 3.2.2 thus grants the agent a negative score for existing. This is to discourage the agent from remaining idle and accumulate random rewards from the reward function.
In short: the reward function gives the maximum reward when the vessel travels parallel to the path at a high speed, and as close to the path as possible.

3.3 Hyperparameters

While Unity abstracts the process of building a NN, one still has to adjust the hyperparameters to make the neural network have the desired functionality and give the correct results. The hyperparameters in Unity is given in table 3.3.1. The explanations provided in 3.3.1 are provided by Unity's documentation Technologies (2021b).

These parameters were the ones used after a lot of trial and error. It is recommended to use a lower buffer size and batch size to allow the algorithm to spend less time training on each iteration. The amount of hidden layers were chosen to be 2, as the mapping between the input and output of the is not too abstract. The width of each hidden layer was chosen to be 64 because it had been proven in earlier works Thomas Nakken Karlsen et al. (2021) that it is a reasonable width for a similar system.

Observation name	variable type
Signed distance between boat and path	float
Angle of node and boat	float
Signed velocity of boat in the path's direction	float
Angular velocity of boat	*3x1 vector*
Thruster angles	*2x1 vector*
Thruster force	*2x1 vector*

Table 3.4.1: Observation space for the agent

3.4 Observation and Action Space

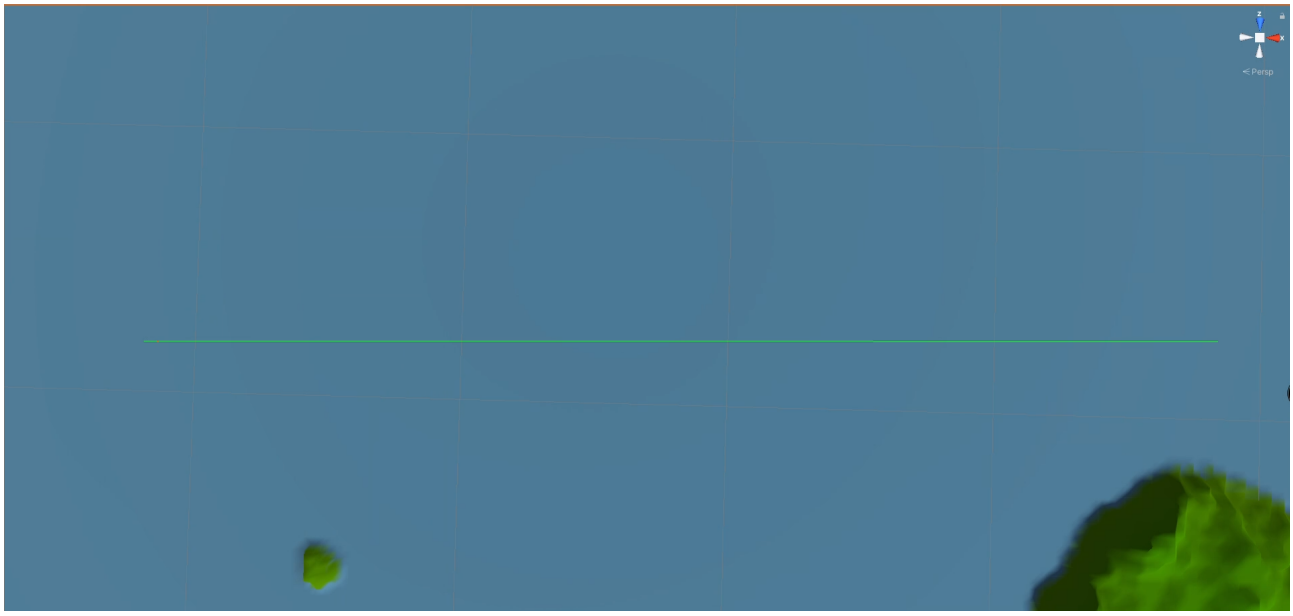

Figure 3.4.1: A straight path for the vessel to follow

The observation space is defined in table 3.4.1.

The size of the observation space given in Table 3.4.1 is 10. While one could provide the ML algorithm more observations in the form of more direct information, this observation space implicitly provides more information per variable than what one normally would have. Providing for example the position of the node and boat, and in addition provide the velocity and rotation of the boat increases the observation of the objects to 18. While it is more readable for humans, a larger observation space will lead to a more complex system which again would require more computation time. It is unsure if this observation space is sufficient when the physics of the boat is expanded, but it is sufficient for this simplified model.

The vessel was given a 4-dimensional action space defined in Table 3.4.2.

The action space used to be a 4x1 continuous action space, but the training time is much faster when a discrete action space is used instead. Each of the 2x1 vectors consists of variables that have three separate "branches" as it is called in Unity's MLAgents. In this case the values can vary between -1,0, and 1. These variables are connected to the thruster angles and thruster force vectors and change them. For example, when the thruster force action is at 1, the thruster force increases, -1 reduces it, and 0 gives no change.

Action name	variable type
Thruster force change	Discrete 2x1 vector
Thruster angle speed change	Discrete 2x1 vector

Table 3.4.2: Action space for the agent

3.5 Training

To start out simple, the agent is first trained on following a straight path as shown in Figure 3.4.1. The optimal path on the ocean between two locations is usually a straight line, which makes this a sufficient starting point. The machine learning problem for now does not include obstructions like land, other ships, or debris. This training is then parallelised by adding additional agents into the environment with their own paths to train independently. This at first seemed fine, but the algorithm would start training a new model before any training was finished. One could increase the buffer size per iteration such that each parallel agent would manage to finish their task before any update of the model, but simply reducing the amount of agents to one and increasing the buffer size to twice the amount was deemed sufficient. There was no noticeable change in training speed compared to having several parallel agents, but the training was substantially more stable than previously.

The training on the straight path showed improvements after a while. The path was then changed to contain a right turn as shown in image 3.5.1. This was done to see if the agent would be able to train on a more advanced and realistic path. A real path most likely contains one or two large turns to avoid large obstacles. This path was thus used for the more realistic training. The agent was trained with the same parameters and with the same observation space and action space.

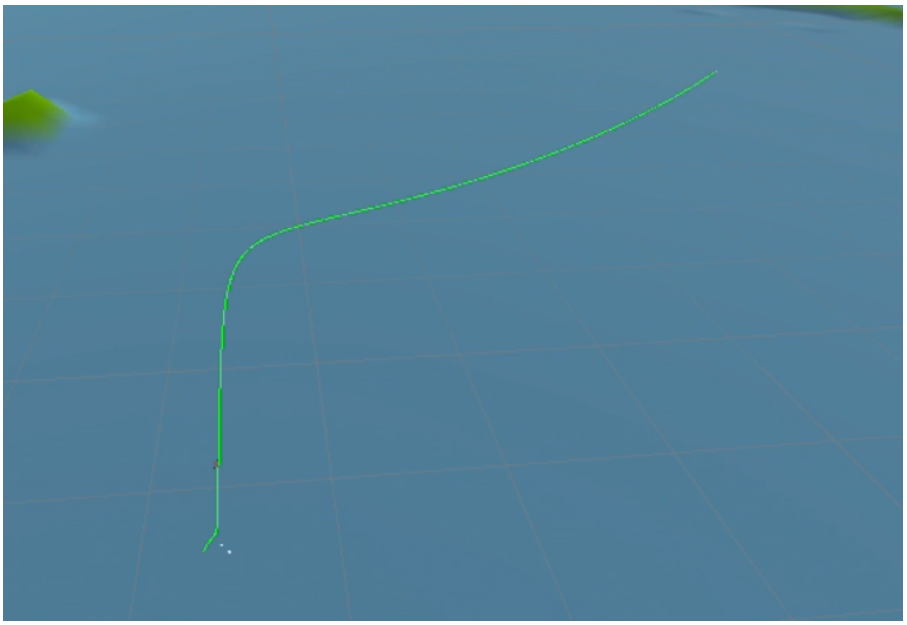

Figure 3.5.1: The curved training path

The agent trained until it was determined to be adequate to navigating the path. The agent trained on the curved path was then given a difficult and unrealistic path to be tested on. While unrealistic, it was given as a test to determine if the agent was generalised enough to handle all types of paths. The path, as shown in image 3.5.2, starts with large curves, and gets increasingly more and more aggressive curves to throw off the agent.

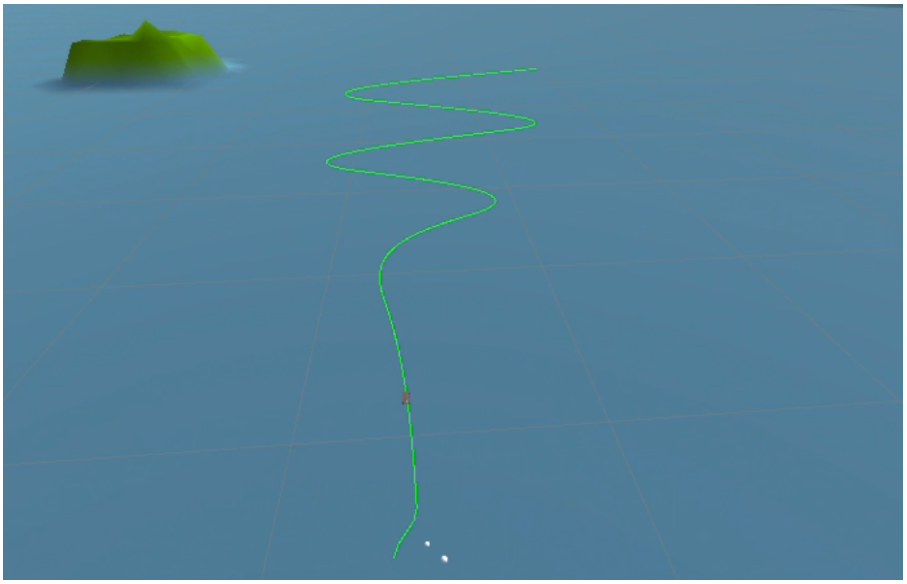

Figure 3.5.2: The test path to test the trained model

Chapter 4
Results and Discussions

4.1 Machine Learning Model

The model that was trained performed well and managed to follow an unseen path relatively easily. This is clearly a sign that Unity works as a machine learning environment for this problem.

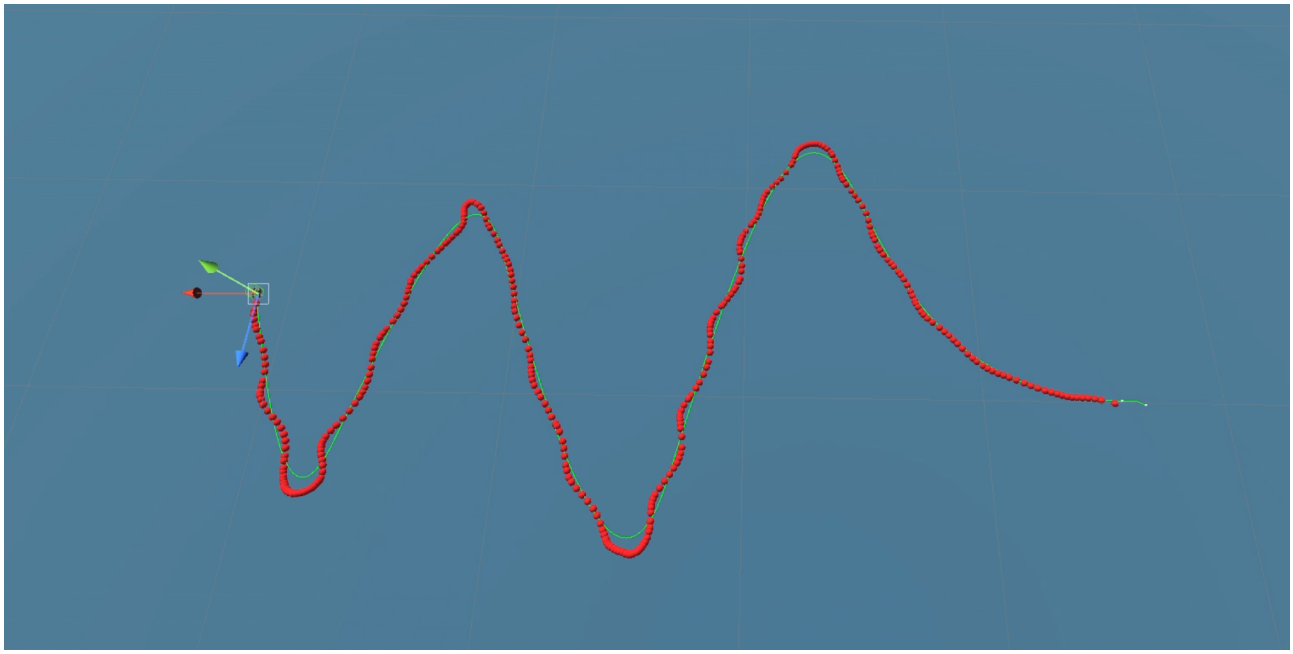

Figure 4.1.1: Testing the boat on an unseen path. Green line representing the desired path, red dots representing the boat's actual path

As seen in image 4.1.1, the model performs quite well on an unseen path. It was observed that the model would struggle on straight parts of the path as seen in 4.1.1. One can see the agent constantly tried to deviate from the path before correcting itself. A possible cause for this might be the fact that the training path 3.5.1 was too simple.

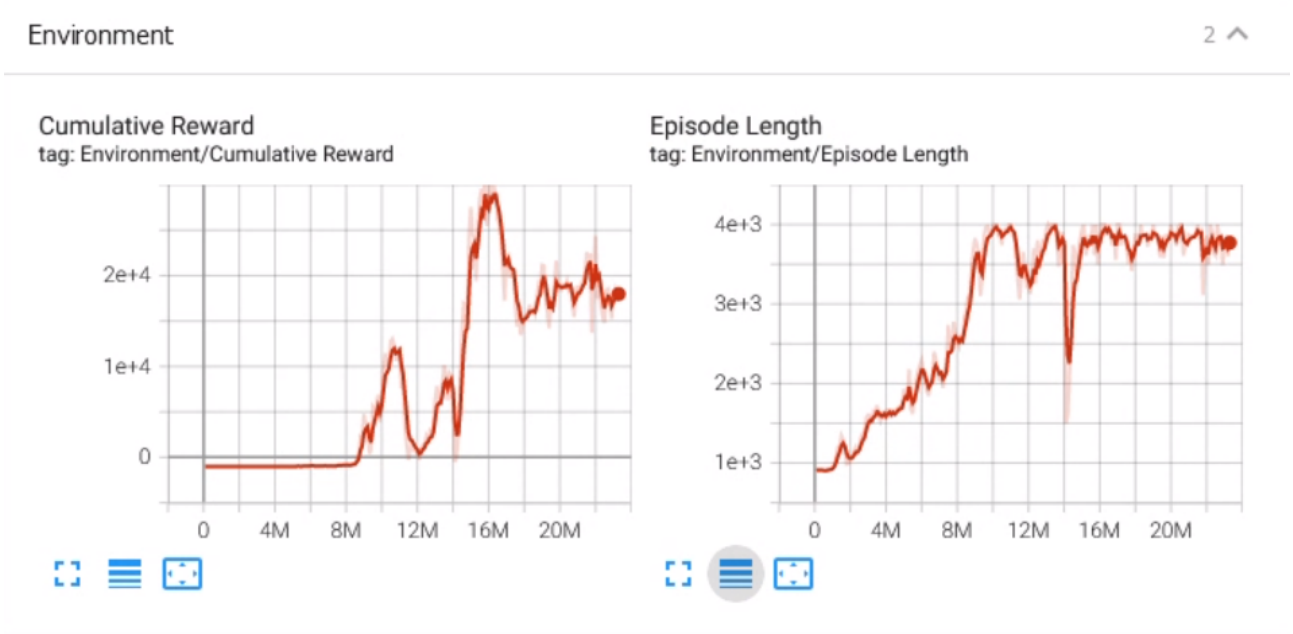

Figure 4.1.2: The cumulative reward and length of the training episodes on the curved path

The cumulative reward for the training of the vessel as shown in 4.1.2 was for some reason quite unstable. In the beginning it was for a long time at the lowest possible reward. This is the case because the agent was not able to finish the full episode lengths before accumulating enough negative score to end the episodes. It is also uncertain why the score suddenly was halved at 16 million steps. This could be a combination of both the curiosity component of MLAgents and the length of the episodes pushing the model into the incorrect direction. One possible theory could also be that the learning rate was constant and not linearly decreasing.

4.2 Limitations of the Unity engine

While the Unity engine is extremely versatile and can be used for everything from scientific simulations and visualisations to games and movies, it still got its limitations. A few of these limitations were encountered during development of this project. The first limitation encountered is the heighmap size. The terrain size is limited to 100 000x100 000 units for each tile. While this limit is large enough for most projects, in this project the limitation was reached and had to be taken into consideration. It was attempted to work around this limitation by stitching several heightmap tiles together to further increase the size of the map. Unfortunately tiling real world data was proven to be quite difficult to execute seamlessly. The environment is therefore limited to an area of 100 000m x 100 000m. This is more than sufficient for the project, but it can prove to be a hindrance if for example the project map should be expanded to represent the entire Norwegian coast. A possible solution for this is to further search for new workarounds for representing a map of such a size. Making the map seamlessly tileable is the issue here.
A second limitation was found while developing the dynamics of the vessel. It was observed that the ship would for some reason not start moving before the rigidbody component of the object reached a speed of 2 units. After performing several experiments, the coordinate values of the vessel was found to be the cause of the problem. Unity handles coordinates in the 32 bit floating point numbers. This means Unity has to represent the higher values using fewer decimals. Since the lower speeds only change the position very slightly every timestep, it gets rounded down to the position from the previous step. This results in no movement at all. The training and simulation of the environment has to take this into consideration such that it does not affect the simulation itself. One could also fix this issue by redefining the origin such that the vessel never moves too far away from the origin and gets affected by this limitation. Increasing the timestep of the physics engine is also a perfectly valid option, but this can affect the accuracy of the simulation.

Chapter 5
Conclusion and future work

5.1 Conclusion

From the results provided, one can conclude that Unity indeed works as a toolkit for reinforcement learning, but more work is needed to make the neural network algorithm work on all cases. The observed behaviour of the model on unseen paths were sufficient, but not optimal as it struggled on paths with straight lines. This at least shows that Unity with MLAgents is viable as a replacement for other reinforcement learning toolkits. A real world map of the Trondheim Fjord was imported in a real world scale, but while the map is highly accurate, it lacks proper colours. More work has to be done to make the environment "prettier".

5.2 Future Work

This is a preproject for a master thesis, and there are several features to expand upon in this project. Here are some of the following proposals for tuture implementations of the project.

- The algorithm does not yet support vision. The Milliampere boat as featured in the Milliampere article Pedersen (2019), contains a LIDAR for detecting objects that the vessel might collide with. Replicating the specifications and functions of the LIDAR. MlAgents already contains a ray casting sensor object to detect objects and distance, so implementing this feature should be possible. This should be implemented in the project such that the algorithm can detect and avoid possible collisions.
- The current model as said, does not perceive any objects. It also cannot collide with any objects right now. Implementing a collision system that handles environment collision and collision with other moving vehicles should be trivial.
- One possibly neat feature to implement could be VR visualisation of the project. The VR capabilities can be used to present works done by other students in a new and different way. Perhaps one can make the user travel onboard the ship being controlled by a RL algorithm? One possible issue with this is performance. VR requires a high framerate to not inflict motion sickness on the user. This means the performance of the algorithm has to be taken into consideration.
- At the moment, the environment looks ugly. The current texturing solution is just a simple colour gradient. This should be improved to make the project more presentable. Perhaps importing satellite imaging of the terrain should be done.
- The dynamics of the current boat is highly simplified, and thus not accurate to real life. More work has to be done to make this model more accurate. Unity's Mathematics package should be able to handle advanced matrix calculations.
- Further work towards making the environment more stochastic should be done. One can perhaps implement real life data, like ocean currents, wind, and sea traffic. The current simulation is not stochastic in particular and adding stochastic forces and events should make the simulation represent the real world better.

Bibliography

Kartverket. Map of norway, 2021. URL https://hoydedata.no/LaserInnsyn/.

E. Meyer. On course towards model-free guidance: A self-learning approach to dynamic collision avoidance for autonomous surface vehicles. Master's thesis, NTNU, 2020. URL https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2780874.

A. A. Pedersen. Optimization based system identification for the milliampere ferry, 2019. URL https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2625699.

P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017. URL https://arxiv.org/abs/1710.05941.

F. W. E. Rouault et al. Gdal - gdal documentation, 2021. URL https://gdal.org/download.html#windows.

J. Sánchez-Beaskoetxea, I. Basterretxea-Iribar, I. Sotés, and M. d. l. M. M. Machado. Human error in marine accidents: Is the crew normally to blame? *Maritime Transport Research*, 2: 100016, 2021.

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

U. Technologies. Unity technologies ml agents activation function, 2021a. URL https://github.com/Unity-Technologies/ml-agents/blob/release_5/ml-agents/mlagents/trainers/models.py#L92-L95.

U. Technologies. Unity technologies ml agents hyperparameters, 2021b. URL https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md.

T. L. Thomas Nakken Karlsen, Halvor Ødegård Teigen et al. Comparing deep reinforcement learning algorithms' ability to safely navigate challenging waters, 2021. URL https://www.frontiersin.org/articles/10.3389/frobt.2021.738113/full.

Chapter 6
Appendix

6.1 Machine Learning Code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Sensors;
using Unity.MLAgents.Actuators;
using PathCreation;
public class MLScript : Agent
{
    public BoatPhysics boatPhysics;
    public GameObject target;
    public GameObject pathNode; //The lookahead point
    public GameObject nearestNode; //The nearest point on the path
    private float nearestPoint;
    private Rigidbody rb;
    public Vector4 controlVector = Vector4.zero; //Controls the thruster
   forces
    public Vector2 controlAngle = Vector2.zero; //Controls thruster angle
   speed
    public PathCreator path;
    public GameObject Path;
    private float nearestNodeTime;
    private float firstNodeTime;
    private Vector3 startPosition;
    private Vector3 startRotation;
    public float rewardview;
    public float angle;
    // Start is called before the first frame update


    //
    // Reward parameters
    //
    private float Ye = 5.0F; //Crosstrack error scaling
    private float Ytheta = 10.0F; //Sensor Angle Scaling
    private float Yx = 0.1F; //Obstacle distance scaling
    private float alphar = 0.05F; //Zero-reward relative speed
    private float rcoll = -10000F; //collision reward
    private float Yr = 0.25F; //Constant reward multiplier
    private float lambda = 0.5F; //Objective trade-off coefficient
    //
    //
    //
    // TODO sensor parameters
    // Not being used yet
    private float maxSpeed = 1F;
    private int amountOfSensors = 180;
    private float sensorAngle;
    private int sensorSectors = 9;
```

```
48      private float sensorRange = 1500F;
49      //private float lookAheadDistance = 3000F;
50      //
51      //
52      //
53      void Start()
54      {
55          nearestPoint = Path.GetComponent<ShortestDistance>().nearestPoint;
56          rb = GetComponent<Rigidbody>();
57          nearestNodeTime = Path.GetComponent<ShortestDistance>().timeOnPath;
58          startPosition = gameObject.transform.position;
59
60          startRotation = transform.localEulerAngles;
61
62      }
63      private void Update()
64      {
65
66
67      }
68      public override void OnEpisodeBegin()
69      {
70          firstNodeTime = nearestNodeTime;
71          //this.transform.localPosition = new Vector3(Random.value * 2, -0.92
    f, Random.value * 2);
72          gameObject.transform.position = startPosition+ new Vector3(Random.
    Range(-5,5),0,Random.Range(-5,5));
73          rb.velocity = new Vector3(0, 0, 0);
74          GetComponent<BoatPhysics>().thrusterAngles = new Vector2(0, 0);
75
76          rb.angularVelocity = new Vector3(0, 0, 0);
77          gameObject.transform.localEulerAngles = startRotation+new Vector3(0,
    Random.Range(-10,10));
78          boatPhysics.thrusterForce = new Vector2(0, 0);
79      }
80
81      public override void CollectObservations(VectorSensor sensor)
82      {
83
84                  //Distance vector between node and vessel. This observation
    gives a signed distance depending on which side the vessel is.
85                  float distanceFromPath = Vector3.Distance(gameObject.
    transform.position, nearestNode.transform.position);
86                  float sideOfPath = Mathf.Sign(Vector3.SignedAngle(path.path.
    GetDirection(nearestNodeTime), gameObject.transform.position -
    nearestNode.transform.position, transform.up));
87
88                  sensor.AddObservation(distanceFromPath*sideOfPath);
89                  //Debug.DrawLine(gameObject.transform.position, pathNode.
    transform.position);
90
91
92                  //Angle between direction of node and boat
93                  sensor.AddObservation(-Mathf.Cos(Mathf.Deg2Rad*Vector3.Angle
    (nearestNode.transform.position - pathNode.transform.position, transform.
    right)));
94                 // Debug.DrawLine(nearestNode.transform.position, nearestNode
    .transform.position + path.path.GetDirectionAtDistance(nearestPoint).
    normalized*10, Color.magenta);
95
```

```
96              //The velocity of the boat in the direction of the node
97               sensor.AddObservation(rb.velocity.sqrMagnitude * -Mathf.Cos(
    Mathf.Deg2Rad * Vector3.Angle(nearestNode.transform.position - pathNode.
    transform.position,rb.velocity)));
98              //Physics of the vessel
99               sensor.AddObservation(Mathf.Rad2Deg * rb.angularVelocity);
100              sensor.AddObservation(boatPhysics.thrusterAngles);
101              sensor.AddObservation(boatPhysics.thrusterForce);
102              //sensor.AddObservation(gameObject.transform.eulerAngles);
103
104
105         /*
106         sensor.AddObservation(transform.localPosition);
107         sensor.AddObservation(transform.localEulerAngles);
108         sensor.AddObservation(transform.InverseTransformVector(rb.velocity))
    ;
109         sensor.AddObservation(nearestNode.transform.localPosition);
110         sensor.AddObservation(rb.velocity.sqrMagnitude * -Mathf.Cos(Mathf.
    Deg2Rad * Vector3.Angle(nearestNode.transform.position - pathNode.
    transform.position, rb.velocity)));
111         sensor.AddObservation(boatPhysics.thrusterForce);
112         sensor.AddObservation(boatPhysics.thrusterAngles);
113         */
114     }
115
116     public override void OnActionReceived(ActionBuffers actionBuffers)
117     {
118         //Changing thruster angle
119         controlAngle.x = actionBuffers.DiscreteActions[0];
120         controlAngle.y = actionBuffers.DiscreteActions[1];
121
122         for (int i = 0; i < 2; i++)//Assigns the speed to the control vector
     depending on the discrete actions of the ML
123         {
124             if (controlAngle[i] == 0) { controlVector[2 + i] = 1; }
125             if (controlAngle[i] == 1) { controlVector[2 + i] = -1; }
126             if (controlAngle[i] == 2) { controlVector[2 + i] = 0; }
127         }
128
129
130         //Force of thrusters
131         //controlVector.x = actionBuffers.ContinuousActions[0];
132         //controlVector.y = actionBuffers.ContinuousActions[1];
133
134         int front = actionBuffers.DiscreteActions[2];
135         int back = actionBuffers.DiscreteActions[3];
136
137
138             if (front == 0) { controlVector[0] = 0; }
139             if (front == 1) { controlVector[0] = 1; }
140             if (front == 2) { controlVector[0] = -1; }
141
142             if (back == 0) { controlVector[1] = 0; }
143             if (back == 1) { controlVector[1] = 1; }
144             if (back == 2) { controlVector[1] = -1; }
145
146
147         //Providing the agent control of  the thruster force and thruster
    angle speed
148         boatPhysics.thrusterAngleSpeed = new Vector2(controlVector.z,
```

```
    controlVector.w);
149
150         boatPhysics.thrusterForce += new Vector2(boatPhysics.forceAccel*
    controlVector.x, boatPhysics.forceAccel * controlVector.y)*Time.
    fixedDeltaTime;
151         boatPhysics.thrusterForce = new Vector2(Mathf.Clamp(controlVector.x*
    boatPhysics.maxForce.y/*boatPhysics.thrusterForce.x*/, boatPhysics.
    maxForce.x, boatPhysics.maxForce.y), Mathf.Clamp(controlVector.y*
    boatPhysics.maxForce.y/*boatPhysics.thrusterForce.y*/, boatPhysics.
    maxForce.x, boatPhysics.maxForce.y));
152         //Reward the vessel for existing and following the path
153         AddReward(CalculateReward());
154         //End episode if
155         if(StepCount >= MaxStep)
156         {
157
158             EndEpisode();
159
160         }
161         else if(GetCumulativeReward() <= -1000)
162         {
163             EndEpisode();
164         }
165         else if(nearestNodeTime >= 0.95)
166         {
167             AddReward(1000F);
168             EndEpisode();
169         }
170         else
171         {
172
173         }
174
175
176     }
177     private void OnCollisionEnter(Collision collision)
178     {
179         if(collision.gameObject.tag == "Terrain" || collision.gameObject.tag
     == "Vehicle")
180         {
181             SetReward(rcoll);
182             EndEpisode();
183         }
184     }
185     float CalculateReward()
186     {
187
188          //Reward function from the paper
189         //Calculates the vector between the nodes to get the angle
    difference between the vessel and the forward direction
190         float absoluteSpeed = rb.velocity.magnitude;
191         Vector3 nodeDirection = path.path.GetDirection(nearestNodeTime);
192
193         //float angleBetweenBoatAndNode = Vector3.Angle(nodeDirection,
    nearestNode.gameObject.transform.position);
194         float angleBetweenBoatAndNode =Vector3.Angle(nearestNode.transform.
    position - pathNode.transform.position, transform.right);
195
196         //Reward for following the path. Punishment for deviating.
197         float pathReward = 1*((Vector3.Project(rb.velocity, nearestNode.
```

```
    transform.position - pathNode.transform.position).magnitude) / maxSpeed)
    * (1-Mathf.Cos(Mathf.Deg2Rad*(angleBetweenBoatAndNode)) / 2 )* (1 / (
    Vector3.Distance(nearestNode.transform.position,transform.position) +1));
198         //
199
200
201         //float pathReward = (nearestNodeTime - firstNodeTime) * 100;
202         //firstNodeTime = nearestNodeTime;
203         //Reward for existing
204        // float existReward = lambda * (2 * alphar +1);
205         float existReward = Yr;
206         //
207
208         float reward = pathReward - existReward;
209         angle = Mathf.Cos(Mathf.Deg2Rad * (angleBetweenBoatAndNode));
210         rewardview = reward;
211         return reward;
212     }
213
214
215
216 }
```

Listing 6.1: Machine learning code

6.2 Boat Physics

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class BoatPhysics : MonoBehaviour
6  {
7      // Start is called before the first frame update
8      public Vector3 inertiaDiagTensors= new Vector3(2389, 5069, 2530);
9      private Rigidbody rb;
10     public Vector2 thrusterAngles = new Vector2(0,0);
11     public Vector2 thrusterAngleSpeed = new Vector2(0, 0);
12     private Vector2 maxThrusterAngles = new Vector2(-90, 90);
13
14     public Vector2 thrusterForce = new Vector2(0, 0);
15     public Vector2 thrusterDistanceFromCO = new Vector2(1, 1);//This number
    is uncertain as the paper at https://ntnuopen.ntnu.no/ntnu-xmlui/handle
    /11250/2625699 does not specify the distances
16     public Vector3 thrusterVectorFront;
17     public Vector3 thrusterVectorBack;
18     public Vector3 thrusterFrontPosition;
19     public Vector3 thrusterBackPosition;
20     public Vector2 maxForce = new Vector2(-300,500); //Minimum and maximum
    thrusterforce on the thrusters
21     public int forceAccel = 50;
22     public float nonlinRotFric;
23     public float nonlinVelFric;
24     public float angleSpeedMax;
25     private MLScript ml;
26
27
28     //Constants for linear and non-linear dampening
```

```
    private float X_u = 0.05F;//-27.632F;
    private float X_uu = 0.001F;//-110.064F;
    private float X_uuu = 0.001F;//-13.965F;
    //N = Y in the original paper
    //Y = N in the original paper
    private float N_r = 52.947F;
    private float N_rr = 116.486F;
    private float N_rrr = 24.313F;
    private float N_vr = 1540.383F;
    private float N_v = -24.732F;
    private float N_rv = -572.141F;
    private float N_vv = 115.457F;
    private float Y_r = -3.524F;
    private float Y_rr = 0.832F;
    private float Y_vr = -336.827F;
    private float Y_v = 0.5F;//122.860F;
    private float Y_vv = 874.428F;
    private float Y_vvv = 0.05F;//0F;
    private float Y_rv = 0.01F;//121.957F;

    //Testing Only!

    //public GameObject front;
    //public GameObject back;
    void Start()
    {
        rb = GetComponent<Rigidbody>();
        rb.inertiaTensor = inertiaDiagTensors;
        ml = GetComponent<MLScript>();

    }

    // Update is called once per frame
    void FixedUpdate()
    {
        //Fetches the controlvector from ML, multiplies it with the maximum positive force, and clamps it between the values -300 and 500 Newtons.

        thrusterForce = new Vector2(Mathf.Clamp(thrusterForce.x, maxForce.x, maxForce.y), Mathf.Clamp(thrusterForce.y, maxForce.x, maxForce.y));
        //

        //Nonlinear dampening rotation
        rb.angularVelocity = rb.angularVelocity - nonlinRotFric * Vector3.Scale(Vector3.Scale(rb.angularVelocity , rb.angularVelocity),new Vector3(Mathf.Sign(rb.angularVelocity.x), Mathf.Sign(rb.angularVelocity.y), Mathf.Sign(rb.angularVelocity.z)));

        //Nonlinear and linear friction
        Vector3 dampeningVector = CalculateDampening();
        dampeningVector = //new Vector3(Mathf.Abs(dampeningVector.x*Mathf.Cos(gameObject.transform.eulerAngles.y)+dampeningVector.z*Mathf.Sin(transform.eulerAngles.y)),dampeningVector.y,Mathf.Abs(dampeningVector.x*Mathf.Sin(transform.eulerAngles.y)+dampeningVector.z*Mathf.Cos(transform.eulerAngles.y)));

        rb.velocity = transform.TransformDirection(transform.InverseTransformDirection(rb.velocity) - Vector3.Scale(dampeningVector, transform.InverseTransformDirection(rb.velocity)));//,new Vector3(Mathf.Sign(rb.velocity.x), Mathf.Sign(rb.velocity.y), Mathf.Sign(rb.velocity.z)
```

```
    ));
77        //Debug.Log(CalculateDampening());
78        //
79
80        RotateThrusters();
81        thrusterFrontPosition = transform.position +new Vector3(Mathf.Cos(-
    Mathf.Deg2Rad*(transform.eulerAngles.y))*thrusterDistanceFromCO.x, 0,
    Mathf.Sin(-Mathf.Deg2Rad*(transform.eulerAngles.y)) *
    thrusterDistanceFromCO.x);
82        thrusterBackPosition = transform.position + new Vector3(-Mathf.Cos(-
    Mathf.Deg2Rad*(transform.eulerAngles.y))*thrusterDistanceFromCO.y,0, -
    Mathf.Sin(-Mathf.Deg2Rad*transform.eulerAngles.y) *
    thrusterDistanceFromCO.y);
83
84
85        thrusterVectorFront = new Vector3(thrusterForce.x * Mathf.Cos(Mathf.
    Deg2Rad*(thrusterAngles.x+transform.eulerAngles.y)), 0, -thrusterForce.x
    * Mathf.Sin(Mathf.Deg2Rad * (thrusterAngles.x + transform.eulerAngles.y))
    );
86
87        thrusterVectorBack = new Vector3(thrusterForce.y * Mathf.Cos(Mathf.
    Deg2Rad * (thrusterAngles.y+transform.eulerAngles.y)), 0, -thrusterForce.
    y * Mathf.Sin(Mathf.Deg2Rad * (thrusterAngles.y+transform.eulerAngles.y))
    );
88        rb.AddForceAtPosition(thrusterVectorFront, thrusterFrontPosition);
89        rb.AddForceAtPosition(thrusterVectorBack, thrusterBackPosition);
90
91        //Testing only! Remove when debugging is finished
92
93        //front.transform.localPosition = (thrusterFrontPosition);
94        //back.transform.localPosition = thrusterBackPosition;
95        //Debug.DrawRay(thrusterFrontPosition, -(thrusterVectorFront)*1,
    Color.red,0.1F);
96        //Debug.DrawRay(thrusterBackPosition, -(thrusterVectorBack)*1,Color.
    blue,0.1F);
97
98    }
99    void RotateThrusters() //Function calculating the new thruster angle
    based on thruster angle speed.
100   {
101       thrusterAngles = thrusterAngles +angleSpeedMax* thrusterAngleSpeed *
     Time.fixedDeltaTime;
102       /*
103       if(thrusterAngles.x >= 360 || thrusterAngles.x <= -360)
104       {
105           thrusterAngles.x = 0;
106       }
107       if (thrusterAngles.y >= 360 || thrusterAngles.y <= -360)
108       {
109           thrusterAngles.y = 0;
110       }
111       */
112       //The line below does the same as the one above but is way more
    flexible and simpler.
113       thrusterAngles = new Vector2(Mathf.Clamp(thrusterAngles.x,
    maxThrusterAngles.x, maxThrusterAngles.y), Mathf.Clamp(thrusterAngles.y,
    maxThrusterAngles.x, maxThrusterAngles.y));
114
115   }
116
```

```
    Vector3 CalculateDampening()
    {
        float x = transform.InverseTransformDirection(rb.velocity).x;
        float z = transform.InverseTransformDirection(rb.velocity).z;

        float d_11 = Mathf.Abs(X_u)+ Mathf.Abs(X_uu * x) + Mathf.Abs(X_uuu * x * x);
        float d_22 = 0;//N_r + N_rr;
                        //Not adding friction in y axis because we assume the speed along y is always zero(for now).
        float d_33 = Mathf.Abs(Y_v) + Mathf.Abs(Y_rv * z) + Mathf.Abs(Y_vvv * z * z);

        return new Vector3(d_11, d_22, d_33);
    }
}
```

Listing 6.2: Boat Physics

6.3 Path Calculations

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using PathCreation;

public class ShortestDistance : MonoBehaviour
{
    // Start is called before the first frame update
    public GameObject vessel;
    public PathCreator Path;
    public GameObject nearestNode;
    public GameObject lookAhead;
    public float nearestPoint;
    public Vector3 closestDistanceVector;
    public float aheadDistance;
    public float timeOnPath; //The path uses a "time" unit where 0 is start of the path and 1 is the end of the path. Useful for tracking progress.
    private float previousTimeOnPath;
    public float differenceTime;

    void Start()
    {
        //vessel = GameObject.FindGameObjectWithTag("Player");

        nearestPoint = Path.path.GetClosestDistanceAlongPath(vessel.transform.position);
        closestDistanceVector = Path.path.GetPointAtDistance(nearestPoint);
        nearestNode.transform.position = closestDistanceVector;
        lookAhead.transform.position = Path.path.GetPointAtDistance(nearestPoint + aheadDistance);
        timeOnPath = Path.path.GetClosestTimeOnPath(closestDistanceVector);
        previousTimeOnPath = timeOnPath;

    }

    // Update is called once per frame
    void Update()
```

```
35      {
36
37          //Finds the shortest vector between the path and the agent(boat).
    Also creates a "lookahead" node.
38          nearestPoint = Path.path.GetClosestDistanceAlongPath(vessel.
    transform.position);
39          closestDistanceVector = Path.path.GetPointAtDistance(nearestPoint);
40          nearestNode.transform.position = closestDistanceVector;
41          lookAhead.transform.position = Path.path.GetPointAtDistance(
    nearestPoint + aheadDistance);
42          //
43          differenceTime = DifferenceTimeOnPath();
44
45
46      }
47
48      float DifferenceTimeOnPath() //Calculates the "time progress" between
    current step and previous step. Used for calculating reward in agent.
49      {
50          timeOnPath = Path.path.GetClosestTimeOnPath(closestDistanceVector);
51          float differenceTimeOnPath = timeOnPath - previousTimeOnPath;
52          previousTimeOnPath = timeOnPath;
53          return differenceTimeOnPath;
54      }
55
56  }
```

Listing 6.3: Path calculations

6.4 Unfinished Boat Physics

```
1       using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class RigidBodyPhysics : MonoBehaviour
6   {
7       private Rigidbody rb;
8       public Vector3 tensors = new Vector3(2389.657F,5068.910F,2533.911F);
9
10      public Matrix4x4 coriolisMatrix;
11      public Matrix4x4 dampingMatrix;
12      private float m_11 = 2389.657F;
13      private float m_22 = 5068.910F;
14      private float m_33 = 2533.911F;
15      private float m_23 = 28.141F;
16      private float m_32 = 62.386F;
17
18
19      // dynamics variables
20      public Vector4 position;
21      public Vector3 lastPosition;
22
23      public Vector4 lastSpeed;
24      public Vector4 acceleration;
25      public Vector4 force;
26      public float xdot;
27      public float zdot;
```

```
    public float ydot;
    public float xddot;
    public float zddot;
    public float yddot;
    public Vector4 speed;
    public Matrix4x4 inertiaMatrix;
    public Matrix4x4 inv;
    private float X_u = -27.632F;
    private float X_uu = -110.064F;
    private float X_uuu = -13.965F;
        //N = Y in the original paper
        //Y = N in the original paper
    private float N_r = 52.947F;
    private float N_rr = 116.486F;
    private float N_rrr = 24.313F;
    private float N_vr = 1540.383F;
    private float N_v = -24.732F;
    private float N_rv = -572.141F;
    private float N_vv = 115.457F;
    private float Y_r = -3.524F;
    private float Y_rr = 0.832F;
    private float Y_vr = -336.827F;
    private float Y_v = 122.860F;
    private float Y_vv = 874.428F;
    private float Y_vvv = 0F;
    private float Y_rv = 121.957F;

    private float c_13;
    private float c_23;
    private float c_31;
    private float c_32;
    private float d_11;
    private float d_22;
    private float d_23;
    private float d_32;
    private float d_33;

    // Start is called before the first frame update
    void Start()
    {
        position = gameObject.transform.localPosition;
        speed = new Vector3(0, 0,0);
        acceleration = new Vector3(0, 0, 0);

        var m1 = new Vector4(m_11, 0, 0, 0);
        var m2 = new Vector4(0, m_22, m_32, 0);
        var m3 = new Vector4(0, m_23, m_33, 0);
        var m4 = new Vector4(0, 0, 0, 0);
        inertiaMatrix = new Matrix4x4(m1, m2, m3, m4);









    }

    // Update is called once per frame
    void Update()
```

```
    {
         c_13 = m_33 * speed.y + m_32 * speed.y;
         c_23 = m_11 * speed.x;
         c_31 = c_13;
         c_32 = -c_23;

        var c1 = new Vector4(0f, 0f, (m_33 * speed.y + m_32 * speed.y), 0f);
        var c2 = new Vector4(0f, 0f, c_23, 0f);
        var c3 = new Vector4(c_13, -c_23, 0f, 0f);
        var c4 = new Vector4(0f, 0f, 0f, 0f);



        coriolisMatrix = new Matrix4x4(c1, c2, c3, c4);


        d_11 = -X_u - X_uu * speed.x - X_uuu * speed.x * speed.x;
        d_22 = N_r + N_rr;
        d_23 = N_v + N_rv * speed.y + N_vv * speed.y * speed.y;
        d_32 = Y_r + Y_rr * speed.y - Y_vr * speed.z;
        d_33 = Y_v + Y_rv * speed.y + Y_vvv * speed.z * speed.z;

        var d1 = new Vector4(-d_11, 0, 0, 0);
        var d2 = new Vector4(0, d_22, d_32, 0);
        var d3 = new Vector4(0, d_23, d_33, 0);
        var d4 = new Vector4(0, 0, 0, 0);

        dampingMatrix = new Matrix4x4(d1, d2, d3, d4);







        lastPosition = position;
        lastSpeed = speed;
        //acceleration = (speed - lastSpeed) * Time.deltaTime;
        //speed = new Vector3( (force.x-m_11*acceleration.x) / (d_11 + c_13)
    , (force.y-(m_22*speed.y+m_23*speed.z))/(d_22  + d_23 +c_23), (force.z-
    m_32*speed.y-m_33*speed.z)/(d_32  + d_33+c_31+c_32)  );

        inv = inertiaMatrix.inverse;
        var accelerationcalc = inertiaMatrix.inverse * (force - (
    coriolisMatrix * dampingMatrix) * speed);
        acceleration =
        speed = speed + acceleration * Time.deltaTime;
        position = position + speed * Time.deltaTime;
       // position = gameObject.transform.localPosition + speed * Time.
    deltaTime;
        gameObject.transform.localPosition = position;

    }
}
```

Listing 6.4: Unfinished advanced boat physics