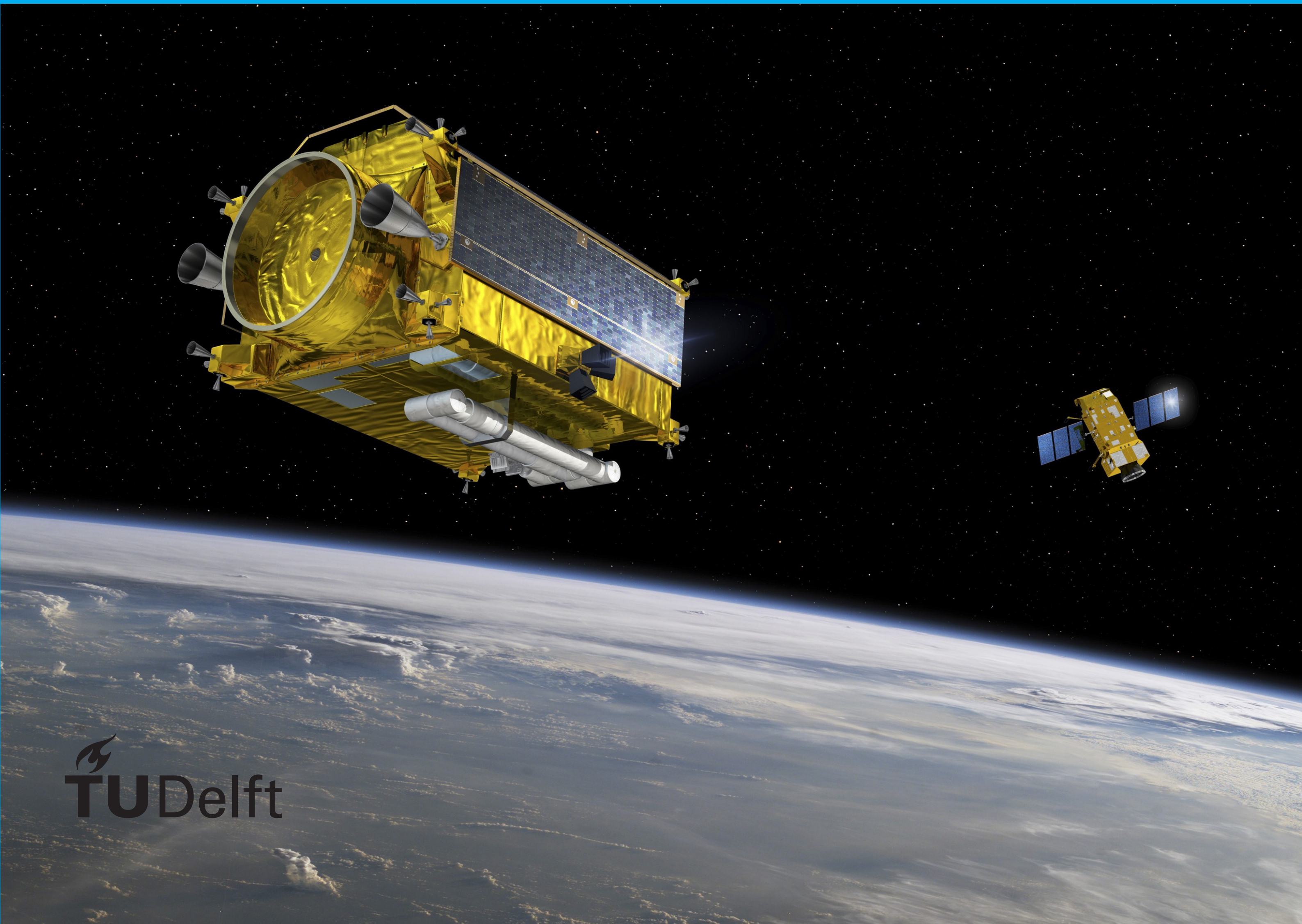
Adaptive Control for Spacecraft Rendezvous

A reinforcement meta-learning approach

Carlos F. De Inza Niemeijer

Delft, June 2023



Adaptive Control for Spacecraft Rendezvous

A reinforcement meta-learning approach

by

Carlos F. De Inza Niemeijer

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on June 27, 2023.

Student number:	5064082	
Project duration:	July 2022 – June, 2023	
Thesis committee:	Dr. J. Guo,	TU Delft, supervisor
	Dr. A. Menicucci,	TU Delft, chair
	Dr. E. van Kampen,	TU Delft, independent examiner

Preface

This thesis marks the conclusion of my journey at TU Delft. It has been a long process involving lots of learning and quite a few struggles, but despite the tough challenges along the way, I am glad I took this path. The lessons I have learned here in Delft have already opened many doors for me to continue my career in Engineering, and they will doubtlessly open many more. I count myself lucky for having had the privilege of being a part of this university. Thus, it is with immense pride and satisfaction that I present my thesis on machine learning applied to rendezvous control. Artificial intelligence has been one of the most exciting subjects of study in the last few years, with amazing new applications seemingly emerging every day. The limits of what can be achieved with artificial intelligence have yet to be found. It has the potential to solve some of the world's most pressing and challenging problems, so I am happy that I had the opportunity to apply it to Aerospace Engineering.

Completing this project would not have been possible without the help of many people. First of all, I would like to thank the members of the aerospace department at TU Delft who helped me with my work. In particular I would like to thank Riupeng Liu and my supervisor Dr. Jian Guo for providing invaluable feedback during our meetings, and for guiding me in the right direction. I would also like to extend my gratitude to all of the friends that I had the pleasure of meeting during my time at TU Delft, as they made this adventure all the more exciting and fun. Last but not least, I am eternally grateful to the three most important people in my life: my parents and my sister. I would not have made it this far without your love and encouragement.

Carlos F. De Inza Niemeijer
Delft, June 2023

Abstract

The continued increase in the number of satellites in low Earth orbit has led to a growing threat of collisions between space objects. On-orbit servicing and active debris removal missions can alleviate this threat by extending the lifetime of active satellites and deorbiting inactive ones, but this requires advanced guidance and control algorithms for the rendezvous phase. Recently, various control policies based on machine learning have been proposed to leverage the advantages of neural networks. One notable technique that has shown much potential in asteroid and planetary landing scenarios is reinforcement meta-learning. This technique consists of training recurrent neural networks in uncertain scenarios in order to develop highly robust control policies that can adapt to unknown conditions in real-time. The goal of this thesis was to apply the meta-learning technique to a rendezvous scenario.

Thus, throughout this project a recurrent neural network was trained via reinforcement meta-learning to generate a control policy that can perform the final approach maneuver of a chaser spacecraft towards a rotating target. A feedforward network was also trained for comparison. The learning algorithm used to train the policy is the Proximal Policy Optimization algorithm, which is a modern actor-critic method that has shown good performance in several continuous control settings. A virtual environment was developed in Python to simulate the rendezvous scenario and collect data to train the policy.

Before beginning with the training process, the hyperparameters of the model were tuned to ensure a smooth and efficient learning process. The three components that required tuning were the learning algorithm, the architecture of the neural networks, and the reward function. Each of these components was tuned in turn, primarily through trial and error. This process required the learning algorithm to be executed multiple times, using a different combination of hyperparameters on each iteration. By repeating this process over a large search space, suitable hyperparameters were found for the learning algorithm and the neural networks. The hyperparameters were chosen to maximize the amount of reward achieved by the policy while maintaining a reasonable training runtime. The reward function was split into several components to guide the policy towards its objective, thereby improving the speed at which the policy learns. Each of the components of the reward function represented some partial goal that the controller had to accomplish. Tuning the relative weight of these components was a challenging process since it often leads to trade-offs between different policy behaviors. Once the tuning process was completed, a sensitivity study was performed to ensure that the model can be used for different kinds of rendezvous trajectories. The sensitivity analysis was performed by training the model on different scenarios, including different orbit altitudes, different distances from the target, different target sizes, and different target rotation speeds. The results of this study showed that the model could be applied to most of these scenarios without the need for any major changes.

After completing the tuning and the sensitivity analysis, the recurrent and the feedforward policies were each trained on a partially observable environment, and their performance was evaluated using a Monte Carlo simulation for a total of one thousand trajectories. The results showed that the recurrent policy was able to learn how to infer hidden information from the environment, which led it to have a much better performance than the feedforward policy. However, the recurrent policy was not without its limitations, since it could not always generate collision-free trajectories, especially when the target rotated at a faster rate. Overall, this thesis showed that reinforcement meta-learning can be a valuable tool for executing complex rendezvous maneuvers, which may be useful now that active debris removal missions are becoming a reality. Furthermore, this thesis also presented a description of how the model was designed and tuned, so that other machine learning practitioners may apply the technique to different scenarios.

Contents

List of Tables

List of Figures

Symbols & Abbreviations

Symbols:

Symbol	Meaning
$\mathbf{A}$	Action
$\hat{A}$	Advantage estimate
c_t	Cell state of a LSTM
$\mathcal{D}$	Machine learning dataset
f	Forget gate of a LSTM
$\mathbf{F}$	Force
G	Return
h	Orbit altitude
h_t	Hidden state of a LSTM
$h(\mathbf{x}, \mathbf{w})$	Supervised learning hypothesis
i_t	Input gate of a LSTM
$\mathbf{I}$	Moment of inertia
k	Reward function coefficient
L	Loss function
$\mathbf{M}$	Torque
m	Mass
n	Mean motion
o_t	Output gate of a LSTM
Q_π	Action-value function
$\mathbf{q}$	Quaternion
R	Reward
R_E	Radius of the Earth
$\mathbf{r}$	Position
$r_t(\boldsymbol{\theta})$	PPO probability ratio
$\mathbf{S}$	State
$\mathcal{U}$	Uniform distribution
$\mathbf{v}$	Velocity
V_π	State-value function
w	Weights of a parametric function
α	Gradient descent step size
Δt	Time step
ΔV	Change in velocity
$\Delta\omega$	Change in rotation rate
μ	Gravitational parameter
π	Reinforcement learning policy
σ	Sigmoid function
$\boldsymbol{\theta}$	Neural network parameters
θ_e	Attitude error
$\boldsymbol{\omega}$	Rotation rate

Abbreviations:

Abbreviation	Meaning
API	Application Programming Interface
C	Chaser body reference frame
CNN	Convolutional neural network
CPU	Central processing unit
CW	Clohessy-Wiltshire
DHPC	Delft High Performance Computing Center
DOF	Degree(s) of freedom
ESA	European Space Agency
LSTM	Long short-term memory network
LVLH	Local-vertical-local-horizontal
MDP	Markov decision process
MLP	Multilayer perceptron
PDF	Probability density function
POMDP	Partially observable Markov decision process
PPO	Proximal policy optimization
ReLU	Rectified linear unit
RK4	Fourth-order Runge-Kutta integration
RNN	Recurrent neural network
SB3	Stable Baselines 3
SGD	Stochastic gradient descent
Tanh	Hyperbolic tangent
T	Target body reference frame
TRPO	Trust Region Policy Optimization
W&B	Weights & Biases

1 Introduction

This thesis explores the feasibility of using machine learning with recurrent neural networks to control the rendezvous of a servicing spacecraft with a tumbling target, which is a common scenario for proposed on-orbit servicing missions or active debris removal missions. In this chapter, section 1.1 provides context and motivation to explain why this research is relevant. Section 1.2 then presents the research questions to be answered and the methodology employed. Lastly, section 1.3 gives a brief overview of the structure of the report.

1.1. Motivation

One of the most significant hazards for current and future spaceflight operations is the growing amount of non-functional artificial objects in orbit around Earth, also known as space debris. Since the beginning of the space age, many kinds of space debris have been created, ranging from items as small as flecks of paint to large structures such as inoperative satellites and used launcher stages. These objects can remain in space for long periods of time, depending on their orbits. Some space debris in low Earth orbit may remain in space for hundreds of years before deorbiting, while objects in higher orbits may remain in space practically forever [1]. Over the years, enough space debris has been generated to cause a significant risk of collisions in orbit. Active satellites must perform collision avoidance maneuvers to prevent impacts that can cause catastrophic failure of their mission, while collisions between two pieces of debris are also concerning, as the high-speed impact can break up the objects into hundreds of pieces, creating even more space debris. It has been theorized that a chain reaction of colliding space debris could form a belt of debris around Earth that would make it impossible to perform space operations in the affected regions [2].

There have been many attempts to mitigate these risks. International guidelines have been published to limit the creation of new space debris via the safe design, operation, and disposal of space vehicles. Meanwhile, programs such as the Space Surveillance Network have made huge efforts to track the existing space debris so that collision events can be predicted and avoided. But despite these efforts, the hazard posed by space debris still persists, and collisions cannot always be prevented. In 2019, the active satellite Iridium-33 unexpectedly collided with the inactive Kosmos-2251, leading to a sharp increase in the amount of space debris from the break-up of both vehicles. The European Space Agency (ESA) has estimated that these collisions will keep occurring, and that the amount of space debris will grow drastically if the current launch trends continue [3].

In light of these facts, more measures are required to limit the growth of space debris. Two promising avenues are on-orbit servicing and active debris removal. The goal of on-orbit servicing is to extend the effective lifetime of space vehicles by approaching them with a servicer spacecraft to perform repairs or maintenance operations while in orbit. Extending the lifetime of existing satellites could reduce the number of new space launches, thereby limiting the creation of new space debris. Alternatively, active debris removal could mitigate the threat of existing space debris by capturing objects in orbit and decommissioning them, either by deorbiting them or placing them in a graveyard orbit. Several demonstration missions have been performed to prove the feasibility of these types of missions. In the past, the *ETS-VII* and *Orbital Express* missions demonstrated some key technologies such as autonomous rendezvous and docking [4]. More recently, the *ELSA-d* mission by Astroscale tested an on-orbit capture in 2021, and in the coming years ESA's *ClearSpace1* mission intends to be the first to deorbit a real piece of space debris.

One feature that these types of missions have in common is that they require one spacecraft to rendezvous with another object in space. A rendezvous trajectory can be highly complex and risky due to the close proximity between the vehicles. The trajectory could be even more difficult to achieve if the target vehicle is not actively controlled, as is the case for space debris. An inactive target would rotate freely, so the service spacecraft would need to match the target's rotational motion in order to capture it. Thus, one of the technical challenges of this type of trajectory is to have a guidance and control system that is precise and responsive enough to autonomously direct the servicing spacecraft towards its objective in a safe manner. In order to achieve this goal, most modern algorithms rely on robust control theory or optimal control theory [5]. Robust control theory is focused on handling disturbances and uncertainties, whereas optimal control theory focuses on optimizing the performance of the system, albeit at the cost of a higher computational load. Recently, machine learning has emerged as an alternative approach to generate guidance and control policies for all sorts of autonomous vehicles, including self-driving cars [6], unmanned aerial vehicles [7], and spacecraft [8]. Much of the success of machine learning has come through the use of artificial neural networks. The main benefit of this approach is that neural networks can approximate highly complex functions with little computational effort, which allows them to output results for a wide range of inputs in real-time. Thus, machine learning can be used to develop advanced control policies that offer the same performance as optimal control, while also maintaining an acceptable computational burden on a spacecraft's onboard computer. Furthermore, different training methods and neural network architectures can be applied to further improve the performance and the adaptability of the control policies.

The primary methods for developing neural network policies are supervised learning and reinforcement learning. With enough training, these neural network policies can generate the optimal guidance signal or the optimal control command to execute a spacecraft's desired trajectory [9]. Training the networks requires a significant computational effort, but once the training is completed the neural networks can generate outputs at a very fast rate. This makes them suitable for either open-loop or closed-loop control applications, as shown in several studies [10, 11]. The supervised learning method trains a neural network using examples of optimal trajectories, so that the network learns to imitate the examples and extrapolate the optimal behavior to new scenarios. Meanwhile, reinforcement learning models use a training environment to gather experience and progressively improve their performance via trial and error. Both of these training methods have shown good performance in simulations of various rendezvous scenarios, including docking [12] and capture [13] missions. A comparative study [14] found that supervised learning models can generate trajectories that are moderately more fuel-efficient than those generated by reinforcement learning models. However, reinforcement learning tends to generate more robust control policies thanks to a wider exploration of the state space, while supervised learning is more prone to overfitting the training data.

The most common type of neural network used to create control policies is the multi-layered feed-forward network. The appeal of these networks is that despite their simplicity they can theoretically approximate any continuous function [15], which makes them suitable for most applications. Other types of networks have also been employed for specific purposes. For instance, convolutional neural networks are often utilized for feature detection on images, and recurrent neural networks are exceptional at processing long sequences of inputs. Recently, recurrent neural networks have been trained via reinforcement learning to develop adaptive policies [16]. This method has been labeled as *reinforcement meta-learning*, and it can produce guidance and control policies that adapt to novel environments even after the training has been completed. Reinforcement meta-learning has been successfully applied to asteroid hovering [17] and planetary landing [18] scenarios. In these works, the policy only received a partial observation of its environment, but the recurrent network allows the policy to remember previous observations, which enables it to detect unobserved information, such as actuator failures, unknown gravitational environments, and external disturbances.

In summary, rendezvous trajectories for active debris removal or on-orbit servicing missions require advanced guidance and control policies to successfully capture their target. The recent advances in machine learning applied to this field show much potential. Through the use of deep learning, many guidance and control policies have been developed using both supervised and reinforcement learning. The reinforcement meta-learning technique seems particularly promising, as it has shown the ability to create highly adaptive policies for a variety of scenarios. The following section of this chapter will present the research plan to employ reinforcement meta-learning for a rendezvous scenario.

1.2. Research framework

The goal of this project is to evaluate how the reinforcement meta-learning technique can be of use in a rendezvous scenario. As described in the previous section, reinforcement meta-learning has recently been used to develop adaptive control policies for various spacecraft landing and hovering scenarios, but not yet for rendezvous scenarios. One of the rendezvous maneuvers that is most critical for an active debris removal mission is the final approach trajectory, where the service spacecraft must reach a position adjacent to its target so that the capture operation may take place. This maneuver is particularly challenging when the target is rotating because the service vehicle must follow the rotational motion of the target before it can proceed with its capture. Thus, a reinforcement meta-learning model will be trained to approach a randomly rotating target, and its performance will be compared to that of a non-meta-learning model. The difference between the performance of these models will show the advantages or disadvantages of using reinforcement meta-learning in a rendezvous scenario.

1.2.1. Research questions

Based on the research objective, the main question that this project will try to answer is the following:

How does reinforcement meta-learning affect the performance of a neural network control policy during a final approach trajectory to a rotating target?

The main rationale behind this main question is to determine if there are any advantages to using reinforcement meta-learning on a rendezvous scenario. If there are no advantages, then it will be necessary to determine why. Three sub-questions stem from the main question. These sub-questions will assess the main performance characteristics of the controller:

1. *Does the reinforcement meta-learning policy achieve collision-free trajectories?*
2. *How is the fuel efficiency of the policy affected by reinforcement meta-learning?*
3. *How well does the policy operate with only partial observations of the scenario?*

The first sub-question focuses on the safety of the trajectory. During a final approach maneuver, the main concern is avoiding a collision between the vehicles, and this can be especially challenging when the target is rotating. Thus, a key performance indicator of the policy will be how often it leads to collisions, if ever. The second sub-question will assess the efficiency of the policy in terms of its fuel consumption. On-board fuel is limited, and its use will determine the effective lifetime of the mission. In the case of a multi-debris removal mission, the fuel efficiency will determine the number of debris objects that can be removed before the fuel is depleted. Hence, it is highly desirable to develop a control policy that can achieve fuel-efficient trajectories. Lastly, the third sub-question will aim to discover how adaptable the policy is when it only has incomplete information of its scenario.

1.2.2. Methodology

A virtual environment will be developed in Python to simulate the rendezvous scenario. The policies will be trained and evaluated in this environment. The two main components of the environment are the dynamics model and the reward function. The dynamics model describes how the system changes based on the actions of the policy. It will be assumed that the target is rotating freely, and the chaser has full control over its own position and attitude relative to the target. The motion of the chaser will be modeled with the Clohessy-Wiltshire equations [19], which are commonly used to represent the relative motion between two vehicles in a circular orbit. The attitude and rotation rate of the vehicles will be modeled assuming rigid body dynamics and using the quaternion kinematics equations [20]. A reward function will be formulated by drawing inspiration from other rendezvous studies that used reinforcement learning controllers for rendezvous [21, 14, 12]. In these studies, the reward function is composed of several terms to encourage different behaviors. For example, in order to encourage collision avoidance, a penalty is given when the chaser enters a keep-out zone, and another penalty is given for using fuel so that the agent learns to be fuel-efficient.

The reinforcement learning algorithm chosen for this project is the Proximal Policy Optimization (PPO) algorithm, which has shown good performance on continuous control tasks [22] and is also more stable than other policy gradient algorithms. Meta-learning will be implemented into the PPO algorithm by applying a recurrent layer to the neural network policy. The policy is then trained in an uncertain environment where the target spacecraft can be rotating in any direction, and the policy does not have full knowledge of this rotation. This technique leverages the ability of recurrent neural networks to detect temporal relationships between sequences of inputs, allowing the policy to deduce the missing information from the observations. The technique was inspired by the work of Wang et al [23], and has been previously proposed for landing scenarios [18] and asteroid intercept scenarios [24].

For this project, the learning algorithm will not be developed from scratch. Instead, the PPO algorithm implemented in the *Stable Baselines 3* [25] repository will be used. This open-source repository contains a library of advanced and reliable reinforcement learning algorithms. It is an ongoing project by the German Aerospace Center's Institute of Robotics and Mechatronics [26], and it is regularly updated by an active community of developers. Furthermore, the PPO algorithm implemented on Stable Baselines 3 is compatible with recurrent neural networks, which will be necessary to implement meta-learning. The neural networks will be generated using the PyTorch [27] deep learning framework, which provides the building blocks to create custom neural networks, and save or load the model.

Once the virtual environment and the learning algorithm have been set up, the tuning process will begin. This is a critical step to optimize the algorithm's learning process and the policies' performance. Three features need to be tuned: the structure of the neural networks, the hyperparameters of the PPO algorithm, and the reward function. Tuning the size of the neural networks is important because a network that is too large can lead to very long training times, and a network that is too small can limit the network's ability to generate a good control policy. The ideal size of a network can vary depending on the dimensionality of the model, hence many researchers [28, 29] follow a trial and error process to find the network size that leads to the smallest loss after a given amount of training steps. Similarly, the hyperparameters of the learning algorithm are also often tuned via trial and error. A simple version of the model will be trained several times with different combinations of these parameters, and the configuration that leads to the highest reward will be chosen to train the final model. As previously mentioned, the reward function will consist of several terms that either penalize unwanted behaviors or reward desired behaviors. Each of these terms will have to be tuned across many simulations to optimize the performance of the policy. Researchers have reported that this can be a lengthy process [30] since it requires many iterations and some amount of intuition. The approach used during this project will be to start from a simple reward function, and then gradually add more complexity to it until the performance is satisfactory.

After the model has been tuned, it will be trained on a range of scenarios with varying initial conditions for the chaser and the target. The performance of the policies will then be evaluated by performing a Monte Carlo simulation with randomized initial conditions. This is the most widely used method for testing neural network controllers, such as the ones developed by Gaudet et al [31] and LaFarge et al [32]. The outcome of the Monte Carlo simulation will be a large set of trajectories. These trajectories will be analyzed to determine how often the controller can achieve a successful rendezvous, and how efficient it is at performing the rendezvous. It is expected that the meta-learning algorithm will make the controller adaptive enough to deal with many different scenarios, but there will most likely still be some instances in which the controller fails. If so, it will be necessary to identify the conditions that lead to failure in order to determine the operational limits of the controller.

The model will be created in Python, a general-purpose programming language that is widely used in the machine learning community. Python has many open-source libraries designed for scientific computing, which will be useful when creating the dynamics model. In particular, NumPy [33] and SciPy [34] will be used to perform matrix algebra and numerical integration. Python will also be used to plot the trajectories generated by the policy. Simple 2D plots will be generated with the Matplotlib [35] library, which is the default library for making plots in Python. However, Matplotlib does not fully support 3D graphics, so the Plotly [36] and Vpython [37] libraries will be used to make 3D plots and animations of the trajectories.

Training neural networks can be a slow process, therefore the *DelftBlue* supercomputer will be used to run the experiments throughout this project. *DelftBlue* is the most recent addition to the Delft High Performance Computing Center (DHPC) [38], which provides hardware and software solutions for researchers and students to run complex analyses and simulations.

1.3. Report structure

The remainder of the report is structured into 7 chapters. Chapter 2 delves deeper into the topics of rendezvous trajectories and machine learning. The purpose of this chapter is to provide background information that is relevant to understand the rest of the thesis. Chapter 3 describes the model scenario that was developed to gather the data that is used to train and evaluate the controller. This includes a detailed description of the mission scenario, the dynamics model, and an explanation of how the environment was implemented in Python. Chapter 4 examines the learning algorithm that was employed to train the controller. Specifically, it covers the practical details of the PPO algorithm, as well as the architecture of the recurrent and feedforward policies, and the design of the reward function that the learning algorithm uses to learn the desired behaviors. Before training the policies, the hyperparameters of the model were tuned to improve the training process. Chapter 5 explains how this tuning was performed, and chapter 6 shows the results of a sensitivity analysis that was performed to assess the applicability of the learning algorithm to different scenarios. Chapter 7 describes how the controller was trained and evaluated with a Monte Carlo simulation, and it presents the results of the trained policies. Finally, chapter 8 concludes the report with a summary of the results and recommendations for future work.

2 Background

Two important topics of this thesis are rendezvous and machine learning. Rendezvous is a key element of on-orbit servicing missions, and machine learning is a broad subject with several different sub-fields. This chapter provides theoretical background information that will assist the reader in understanding the following chapters. Section 2.1 covers the main features of rendezvous trajectories, the reference frames used for relative navigation, and the dynamics and kinematics models that describe the motion of the vehicles in orbit. Section 2.2 presents several machine learning concepts significant to this project. It outlines the two branches of machine learning that have been most relevant for the development of control policies: supervised learning and reinforcement learning. This section of the chapter also explains the benefits of using deep learning and different network architectures such as recurrent neural networks. Another prominent branch of machine learning is *unsupervised learning*, but it is not discussed in this work because it is mainly used for finding patterns in datasets, not for training control policies.

2.1. Rendezvous

In the context of space missions, rendezvous is the process of bringing two spacecraft or other objects close to each other in space. The close proximity between the two vehicles can be used to transfer personnel and cargo, or to perform inspections or repairs. Hence, rendezvous is an essential part of on-orbit servicing missions and active debris removal missions. In order to perform a rendezvous, at least one of the two spacecraft needs to be actively controlled to change its position and velocity relative to the other spacecraft. The vehicle that is being controlled is referred to as the *chaser*, while the other vehicle is referred to as the *target*.

2.1.1. Phases

A rendezvous trajectory is a sequential process involving several steps. The specific maneuvers performed during a rendezvous can vary from mission to mission, but the whole operation can be generalized into four different phases. Fehse [39] described the phases as follows:

- Phasing: The goal of this phase is to adjust the orbit of the chaser in order to reduce the phase angle between the two vehicles. Phasing ends with the acquisition of an *initial aim point* or *entry gate*, which is usually a few tens of kilometers from the target.
- Far-range rendezvous: This phase is also referred to as *homing* in other texts. Its purpose is to reduce the approach velocity while also reducing the distance to the target to a few kilometers. Relative navigation is often initiated during this phase.
- Close-range rendezvous: This phase is usually divided into two sub-phases, although in some cases it is not possible to make a clear distinction between the two. First, during the *closing* subphase the distance to the target is further reduced, and the chaser aligns itself with the approach corridor. Then, during the *final approach* the chaser advances towards the target along the approach corridor. This phase ends when the chaser achieves the conditions required to initiate the mating phase. The *final approach* is the maneuver that will be simulated in this project.

- Mating: In this phase, the mating interfaces of the chaser interact with the mating interfaces of the target, and a structural connection is established between the two vehicles, either by docking or capture. The mating operation itself is outside the scope of this study, but it is worth noting that the mating conditions will depend on the type of mating mechanism used. For instance, to achieve docking, the chaser must move toward the target along the docking axis at a low speed. And to achieve a capture with robotic manipulators, the chaser must cancel its motion relative to the target to allow the robotic manipulators to grasp the capture point(s).

Additional maneuvers may be necessary depending on the mission. For example, if the properties of the target are not known ahead of time, then an extra phase may need to be added to characterize the target. During this *characterization* phase, the chaser would observe the target to identify properties such as suitable capture points. The rotational rate of the target can also introduce further complications to the final approach. A target capable of maintaining a stable attitude can be approached relatively easily in a straight-line motion. But if the target is rotating then the orientation of the approach corridor will change over time, and a straight-line approach may not be possible. Pieces of space debris seldom have a functioning attitude control system, so it is necessary to prepare for this scenario if an active debris removal mission is ever to be achieved.

2.1.2. Dynamics

During the early stages of rendezvous - when the distance between the vehicles is still large - navigation is generally expressed in absolute measurements, based on an Earth-centered reference frame, such as the Earth-centered inertial frame. However, as the rendezvous progresses and the vehicles come closer together, it becomes more convenient to express the motion of one vehicle relative to the other, using relative navigation. To this end, a Local-vertical-local-horizontal (LVLH) reference frame is defined with its origin located on the center of mass of the target. The x-axis of this reference frame is pointed in the outward radial direction of the orbit, the y-axis is pointed tangential to the orbit in the direction of motion, and the z-axis is pointed in the direction perpendicular to the plane of the orbit, completing the right-hand reference frame. A diagram of the LVLH reference frame can be seen in Figure 2.1a, where the z_{LVLH} axis is pointing out of the page. Using the LVLH frame, the motion of the chaser can be described relative to the target. A dynamics model that employs the LVLH frame is the Clohessy-Wiltshire (CW) model [19]. This model is derived from Kepler's and Newton's laws, and it uses linearized dynamics to represent the 3D position and velocity of the chaser over time. The CW equations of motion are shown in equation 2.1, where $x,\ y\ \&\ z$ are the LVLH coordinates of the chaser, and n is the mean motion of the target's orbit. It can be seen that the x and y variables are coupled, whereas the variable z is independent of the other two, indicating that the out-of-plane motion is decoupled from the in-plane motion.

$$\begin{aligned} \ddot{x} &= 3n^2x + 2n\dot{y} \\ \ddot{y} &= -2n\dot{x} \\ \ddot{z} &= -n^2z \end{aligned} \tag{2.1}$$

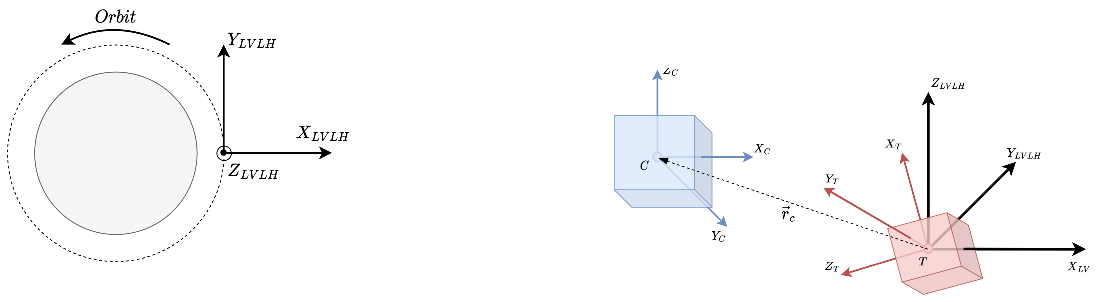
For these equations to be valid, it is assumed that the target is in a circular orbit and that the distance between the chaser and the target is small relative to the radius of the orbit. The accuracy of the CW model degrades for lengthy trajectories, but for a final approach maneuver the duration of the trajectory is expected to be short, so the CW model is suitable for this scenario. Overall, the CW model is fairly simple but still accurate for rendezvous trajectories, and thus it is more widely used than more complex models such as the Tschauner-Hempel [40] equations. The analytical solution of the CW model is shown below:

$$\begin{bmatrix} \mathbf{r} \\ \mathbf{v} \end{bmatrix}_{t_0+\Delta t} = \Phi_{cw} \begin{bmatrix} \mathbf{r} \\ \mathbf{v} \end{bmatrix}_{t_0} \tag{2.2}$$

Where $\mathbf{r}$ is the 3D position of the chaser, $\mathbf{v}$ is the 3D velocity of the chaser, and Φ_{cw} is the state-transition matrix, defined in 2.3. Given an initial position and velocity at t_0, the analytical solution of the CW equations can be used to compute the position and velocity of the chaser at any forward point in time.

$$\Phi_{cw} = \begin{bmatrix} 4-3\cos(n\Delta t) & 0 & 0 & \frac{1}{n}\sin(n\Delta t) & \frac{2}{n}[1-\cos(n\Delta t)] & 0 \\ 6[\sin(n\Delta t)-n\Delta t] & 1 & 0 & \frac{2}{n}[\cos(n\Delta t)-1] & \frac{1}{n}[4\sin(n\Delta t)-3n\Delta t] & 0 \\ 0 & 0 & \cos(n\Delta t) & 0 & 0 & \frac{1}{n}\sin(n\Delta t) \\ 3n\sin(n\Delta t) & 0 & 0 & \cos(n\Delta t) & 2\sin(n\Delta t) & 0 \\ 6n[\cos(n\Delta t)-1] & 0 & 0 & -2\sin(n\Delta t) & 4\cos(n\Delta t)-3 & 0 \\ 0 & 0 & -n\sin(n\Delta t) & 0 & 0 & \cos(n\Delta t) \end{bmatrix} \tag{2.3}$$

To express the attitude and rotation rate of the vehicles, two additional reference frames must be defined: the chaser body (C) and the target body (T) reference frames. These are located on the center of mass of the chaser and the target, and they remain fixed to the vehicles, hence the orientation of these reference frames describes the the attitude of the vehicles themselves. The body frames *C* and *T* are shown in Figure 2.1b, with the target spacecraft depicted as a red cube at the origin of the LVLH frame, and the chaser spacecraft depicted as a blue cube. With these reference frames, the attitude and rotation rate of each vehicle can be expressed with respect to the LVLH frame.



(a) LVLH reference frame.

(b) Chaser and target body reference frames.

Figure 2.1: Reference frames.

The dynamics of the body frames are defined by Euler's rotation equation for rigid bodies, which can be seen in equation 2.4. This equation can be used to compute the angular acceleration of a body as a function of the body's moment of inertia ($\mathbf{I}$) and the torque acting on the body ($\mathbf{M}$), as shown in equation 2.5. The rotation rate of the body can then be computed via numerical integration of the equation.

$$\mathbf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \mathbf{I}\boldsymbol{\omega} = \mathbf{M} \tag{2.4}$$

$$\dot{\boldsymbol{\omega}} = \mathbf{I}^{-1}\left[\mathbf{M} - \boldsymbol{\omega} \times \mathbf{I}\boldsymbol{\omega}\right] \tag{2.5}$$

The orientation of the reference frames can be expressed with quaternions. Unit quaternions can represent any rotation in 3D space using only four numbers. This makes them much more compact than rotation matrices, and thus they are widely used [12, 13, 41]. A quaternion is composed of two parts: a scalar part consisting of one value (q_w), and a vector part consisting of three values (q_x, q_y q_z), as shown in equation 2.6. It should be noted that some works use a different order, placing the vector part ahead of the scalar part, but in this work only the scalar-first formulation will be used. The quaternion represents a rotation of size θ around a unit vector $\hat{e}$. The rotation defined by a quaternion $\mathbf{q}$ can easily be applied to any three-dimensional vector $\mathbf{p}$ as shown in equation 2.7, where $\mathbf{q}^*$ is the conjugate of the quaternion $\mathbf{q}$, the operator $\otimes$ is the Hamilton product, and $\mathbf{p}'$ is the rotated vector. During this operation, the three-dimensional vector $\mathbf{p}$ is treated as a quaternion with a scalar value of zero and a vector value of $\mathbf{p}$.

$$\mathbf{q} = \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} \cos(\theta/2) \\ \hat{e}_x\sin(\theta/2) \\ \hat{e}_y\sin(\theta/2) \\ \hat{e}_z\sin(\theta/2) \end{bmatrix} \tag{2.6}$$

$$\mathbf{p}' = \mathbf{q} \otimes \mathbf{p} \otimes \mathbf{q}^* \tag{2.7}$$

In a dynamic system the value of a quaternion may change over time, therefore it is often necessary to know the time-derivative of the quaternion. This time-derivative can be computed via the quaternion kinematics relation shown in equation 2.8, where ω is the rotation rate of the body frame represented by the quaternion. This time-derivative can then be integrated to compute the value of the unit quaternion over time. The derivation for this equation can be found in Appendix A.

$$\dot{\mathbf{q}} = \begin{bmatrix} \dot{q}_w \\ \dot{q}_x \\ \dot{q}_y \\ \dot{q}_z \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & -\omega_z & \omega_y \\ \omega_y & \omega_z & 0 & -\omega_x \\ \omega_z & -\omega_y & \omega_x & 0 \end{bmatrix} \begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} \tag{2.8}$$

2.2. Machine learning

Machine learning is a branch of artificial intelligence that focuses on the development of algorithms and models that can learn from data. These models are not explicitly programmed to complete specific tasks, but they progressively learn to generate outputs as they process more data. Machine learning first originated in the mid-twentieth century, and it has seen a rise in popularity in the last couple of decades thanks to the improvement of computers' processing capabilities and the emergence of Big Data [42]. There are several approaches to machine learning. The rest of this chapter will cover the working principles of supervised learning, reinforcement learning, and deep learning.

2.2.1. Supervised learning

Supervised learning is a type of machine learning in which a model is trained on a labeled dataset of input-output pairs. As shown in equation 2.9, each sample in a dataset $\mathcal{D}$ must contain an input and the corresponding output to that input. The goal of a supervised learning model is to learn a function that can map an output to the given input. Once the function has been learned, it can be applied to new, unseen input data to make predictions.

$$\mathcal{D} = \left\{ \left(x^{(i)}, y^{(i)}\right),\ \left(x^{(ii)}, y^{(ii)}\right),\ \dots,\ \left(x^{(n)}, y^{(n)}\right) \right\} \tag{2.9}$$

The learned function is sometimes referred to as a *hypothesis*, since it makes predictions based on inputs. Such a hypothesis can take many forms depending on the nature of the problem, but is is generally expressed as a parametrized function. For example, equation 2.10 shows a hypothesis that outputs either a +1 or a -1, depending on the sign of the dot product of the parameters ($\mathbf{\Theta}$) and the inputs ($\mathbf{x}$) added to the bias (θ_0). This hypothesis is a simple example for a linear binary classifier, but other types of hypotheses can be used to implement more complex mappings such as regression or multi-class classification.

$$h(\mathbf{x}, \mathbf{\Theta}) = sign\left(\mathbf{\Theta}^T \mathbf{x} + \theta_0\right) \tag{2.10}$$

The purpose of the supervised learning algorithm is to adjust the parameters to find a hypothesis that matches the labeled inputs to the labeled outputs as closely as possible. To assess the accuracy of a hypothesis, learning algorithms use loss functions (L) which measure the difference between the output predicted by the hypothesis and the real output (y). Loss functions can take many forms, such as mean squared squared error as shown in equation 2.11, cross-entropy loss, and others. The choice of loss function is an important one, as it can have a significant impact on the outcome of the learning process.

$$L = \frac{1}{n} \sum_{i=1}^{n} \left[h(\mathbf{x}_i, \mathbf{\Theta}) - y_i\right]^2 \tag{2.11}$$

To find the optimal parameters for the hypothesis, the learning algorithm tries to minimize the total losses. This optimization is usually performed by means of gradient descent, where the learning algorithm computes the gradient of the loss function with respect to the parameters of the hypothesis, and then updates the parameters in the direction of steepest descent, as shown in equation 2.12. Stochastic gradient descent (SGD) is often used as an alternative. SGD operates similarly to gradient descent, but it computes the gradient based on a small batch of the samples in the dataset instead of the entire dataset. This approach can improve the convergence time of the algorithm, and can prevent the algorithm from getting stuck in local optima [43].

$$\mathbf{\Theta}_{new} = \mathbf{\Theta}_{old} - \alpha \nabla_{\Theta} L \tag{2.12}$$

2.2.2. Reinforcement learning

Reinforcement learning is another type of machine learning that does not require a labeled dataset of input-output pairs. Instead, the data is gathered gradually during training, by having an agent interact with its environment. As the agent explores more of the environment, it develops a decision-making policy that dictates which actions to take in each state of the environment to achieve the most positive outcomes. The reinforcement learning paradigm is very useful for applications such as control systems, robotics, and game-like scenarios where a goal has to be achieved through a sequential decision-making process [44].

One of the key elements in reinforcement learning is the Markov Decision Process (MDP). The MDP is a mathematical framework used to model sequential decision-making problems where two entities - the agent and the environment - interact with each other. Figure 2.2 depicts how the agent and the environment interact together. On each time step, the agent selects an action that affects the state of the environment. The new state of the environment is then passed to the agent along with a scalar reward, and the agent uses this information to pick the next action. This cycle can continue indefinitely or until some termination condition is met. A similar framework to the MDP is the Partially Observable Markov Decision Process (POMDP). A POMDP is a variant of the MDP where the agent does not have complete knowledge of the state of the environment. Instead of receiving the state on each time step, the agent receives an observation with limited information, such as an incomplete state, or a noisy version of the state. This framework is suitable to simulate systems where the exact state is not well known due to reasons such as lack of sensor data, or biased and noisy measurements.

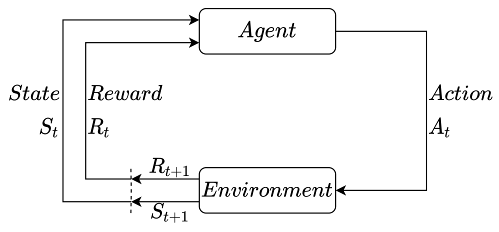


Figure 2.2: Interaction between agent and environment, from Sutton & Barto [45]

The role of a reinforcement learning algorithm is to gather data from the MDP and use it to learn a decision-making policy (π) that will maximize the cumulative reward obtained over time. This cumulative reward is also referred to as *return* (G), and it is computed as the sum of the rewards achieved by the agent at each time step, discounted over time. Equation 2.13 shows how the return is computed, where R is the reward at each time step, and γ is the discount factor. The discount factor is used to define the relative importance of future rewards and immediate rewards. A low discount factor will give more importance to immediate rewards, while a discount factor that is close to one will give more importance to future rewards.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.13}$$

The return is used to compute the value functions. These functions evaluate the quality of a given policy by keeping track of the expected returns. There are two types of value functions: the state value function V_π and the action value function Q_π. The formulae for these two functions are shown in equations 2.14 and 2.15. Intuitively, the state value function is the expected return when starting from some given state s and following some given policy π. Similarly, the action value function is the expected return when starting from state s, taking some action a and following the policy π.

$$V_\pi(s) = \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \tag{2.14}$$

$$Q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid S_t = s,\, A_t = a\right] \tag{2.15}$$

Some reinforcement learning algorithms focus entirely on learning the optimal value functions. For instance, the off-policy algorithm Q-learning progressively updates the Q function until it converges to the optimal one [46]. Off-policy algorithms learn the optimal policy without using said policy to explore the environment. This sets them apart from on-policy algorithms, where the agent explores the environment using the current estimate of the optimal policy. Other relevant algorithms are *policy gradient methods*, which calculate the gradient of the expected reward with respect to a parametrized policy to perform gradient-based optimization, and also *actor-critic methods*, which learn both the policy and the value function.

One of the most modern and widely-used actor-critic methods is the *Proximal Policy Optimization* (PPO) algorithm. It was developed by Schulman et al [22] in 2017, and it follows the same basic behavior as any on-policy algorithm, repeatedly performing the following steps: collect data from the environment using the current policy $\pi(\boldsymbol{\theta})$, compute the losses and then update the policy and the value function via gradient descent. The difference introduced by PPO is in how it computes the loss. The formula for PPO's loss function is based on the Trust Region Policy Optimization (TRPO) algorithm, which tries to limit the size of the policy update in order to prevent large decreases in performance.

The TRPO algorithm optimizes the loss function shown in equation 2.16, subject to a Kullback-Leibler divergence constraint to avoid excessively large policy updates. PPO on the other hand removes the divergence constraint and instead applies a clipping function to the loss function. The modified loss function used by the PPO algorithm is shown in equation 2.17, where the term $r_t(\theta)$ is the probability ratio between the new policy and the previous policy, as depicted in equation 2.18, and the term $\hat{A}_t$ is the estimate of the advantage function. Conceptually, the advantage represents the benefit of choosing a particular action over the policy's average performance. Mathematically, the advantage of taking action a_t at state s_t is defined as the difference between the discounted return measured before and after executing the action, as shown in equation 2.19.

$$L^{TRPO}(\boldsymbol{\theta}) = \hat{\mathbb{E}}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] \tag{2.16}$$

$$L^{CLIP}(\boldsymbol{\theta}) = \hat{\mathbb{E}}\left[\min\left(r_t(\boldsymbol{\theta})\hat{A}_t,\quad clip(r_t(\boldsymbol{\theta}),\ 1-\epsilon,\ 1+\epsilon)\hat{A}_t\right)\right] \tag{2.17}$$

$$r_t(\boldsymbol{\theta}) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2.18}$$

$$\hat{A}_t = R_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.19}$$

In the clipped loss function, the parameter ϵ is the clip range, which essentially determines how different the new policy is allowed to be from the old policy. The first term inside the $\min$ function is equivalent to the loss of the TRPO algorithm. The second term inside the $\min$ function is a modified version of the TRPO loss, where the probability ratio $r_t(\boldsymbol{\theta})$ is clipped within the interval $[1-\epsilon,\ 1+\epsilon]$, as shown in equation 2.20. Thus, L^{CLIP} takes the minimum of the unclipped and the clipped TRPO loss, effectively setting an upper bound on the TRPO loss function. This effect can be seen in Figure 2.3, which depicts the TRPO loss, the clipped TRPO loss, and the PPO loss (L^{CLIP}) as a function of the policy ratio $r_t(\boldsymbol{\theta})$, for both positive advantages (on the left) and negative advantages (on the right). By limiting the loss function, the size of the policy update is also limited. Overall, PPO is conceptually simpler than TRPO, and it has shown a better sample efficiency than TRPO in various continuous control environments, meaning that it requires fewer samples to learn a suitable policy.

$$clip\left(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon\right) = \begin{cases} 1+\epsilon, & \text{if} \quad r_t(\boldsymbol{\theta}) > 1+\epsilon \\ r_t(\boldsymbol{\theta}), & \text{if} \quad 1-\epsilon < r_t(\boldsymbol{\theta}) < 1+\epsilon \\ 1-\epsilon, & \text{if} \quad r_t(\boldsymbol{\theta}) < 1-\epsilon \end{cases} \tag{2.20}$$

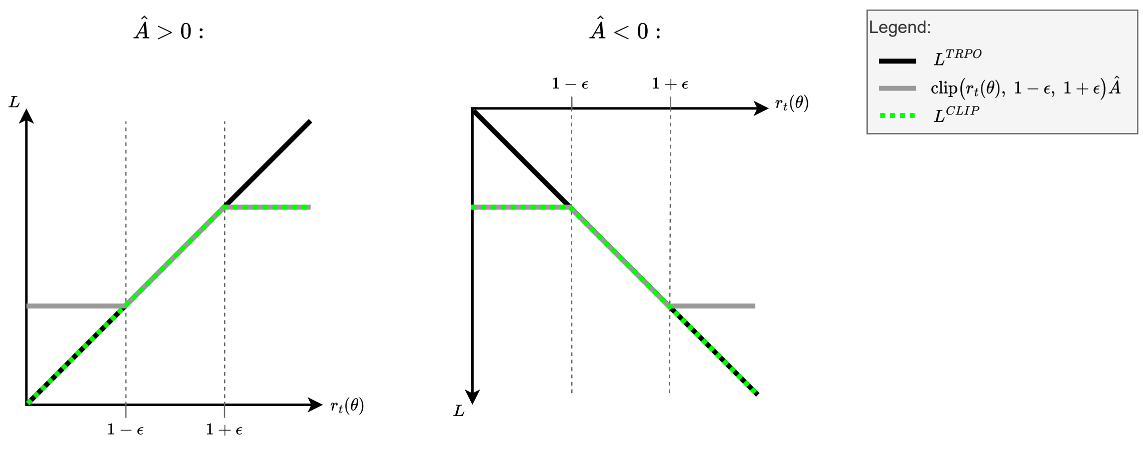


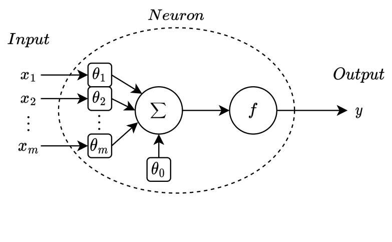
Figure 2.3: Loss as a function of $r_t(\theta)$

2.2.3. Deep learning

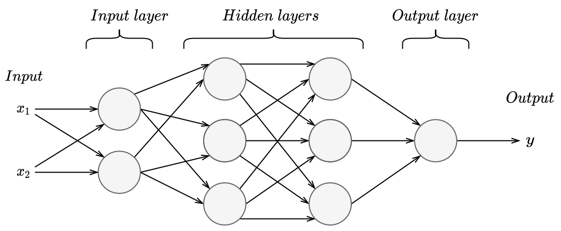
Deep learning refers to the use of multi-layer artificial neural networks in machine learning. Artificial neural networks are composed of multiple interconnected neurons, also called nodes. In this context, a neuron is simply a function that takes in inputs and produces an output, as shown in figure 2.4a. Each input is first multiplied by a real-valued parameter θ and then added to the rest of the inputs along with a bias term θ_0. The result of the weighted sum is then passed through a nonlinear *activation function* that produces the output of the neuron. The purpose of the activation function is to introduce nonlinearities into the model, otherwise the output could only be a linear combination of the input values. Some of the most commonly used activation functions are the rectified linear unit (ReLU), the sigmoid (σ), and the hyperbolic tangent (Tanh). Equation 2.21 shows the formula for the output of a neuron with parameters $\mathbf{\Theta}$ and a sigmoid activation function.

$$y = \sigma\left(\mathbf{\Theta} \cdot \mathbf{x} + \theta_0\right) \tag{2.21}$$

In a neural network the neurons are organized in layers. The first layer is the input layer, the last layer is the output layer, and any layers in between are called hidden layers. An example of a fully connected feedforward neural network is shown in figure 2.4b. This network has an input layer with two neurons, followed by two hidden layers with three neurons each, and finally an output layer with a single neuron. It is a *fully connected* neural network because all of the neurons in each layer are connected to all of the neurons in the adjoining layers. The network is also *feedforward* because the outputs of each layer only flow forward into the next layer, without looping back to previous layers.



(a) Structure of a neuron.



(b) Fully-connected neural network.

Figure 2.4: ANN architecture.

The combined effect of the successive layers of neurons with nonlinear activation functions makes artificial neural networks excellent function approximators, and this makes them extremely useful in machine learning. Recall that the goal of a supervised learning algorithm is to approximate a function that maps the given inputs to the corresponding outputs. Neural networks are perfectly suited for this task, as they can approximate virtually any function [15]. All that is required is calculating the gradient of the loss function with respect to the parameters of the network $(\nabla_{\Theta} L)$. This can be easily achieved by means of the *backpropagation* algorithm. Backpropagation is a gradient-based algorithm that can efficiently optimize the parameters of a neural network to minimize the losses. The algorithm first performs a forward pass, where it computes the output of the network to calculate the loss. Then it executes a backward pass starting from the last layer, making its way back to the first layer. In this backward pass, the algorithm computes the partial derivatives of the loss function with respect to the parameters in the network. Once the partial derivatives are known, the parameters can be quickly updated via gradient descent. The backpropagation algorithm is widely used, as it can be applied to all kinds of neural networks, regardless of their architecture. This is highly advantageous, since some architectures can be more suitable for certain applications.

One of the simplest neural network architectures is the multi-layer perceptron (MLP). The MLP is a fully connected feedforward neural network that is composed of an input layer, one or more hidden layers, and an output layer. It is frequently used since its structure is fairly simple, and it can be easily implemented using machine learning frameworks. Despite its simplicity, a multi-layer perceptron with a single hidden layer can in theory approximate any continuous function [47], although in practice it is common to use more than one hidden layer. The number of layers and neurons required to achieve efficient learning depends on the dimensionality of the problem. Many authors tune their neural networks through a trial and error process to determine the best structure for the network [28, 29, 48]. Preliminary experiments are performed with varying network sizes, and then the network with the lowers losses or highest reward is chosen. Other parameters such as the learning rate or activation functions are also often tuned through a similar process. Li et al [49] and Federici et al [14] successfully used multi-layer perceptrons to develop control policies for rendezvous scenarios.

Another type of artificial neural network is the convolutional neural network (CNN), which was introduced by Fukushima [50] as a method for pattern recognition in images. CNNs are a type of feedforward network that use three basic building blocks: convolutional layers, pooling layers, and fully-connected layers. The convolutional layers apply a series of filters for the purpose of detecting features in the input image. Pooling layers are added in between the convolutional layers to reduce the dimensionality of the CNN's outputs, and fully-connected layers are then used at the end of the network to compute the output based on the detected features. CNNs are very efficient at image processing, so there has been much research to use them for navigation purposes [51, 52]. However, there is not much research into using these networks for guidance and control purposes. This can be partly attributed to reliability concerns, since many factors make it challenging to use vision-based systems in space [53], such as extreme ranges of illumination conditions, and lack of real images to use for training.

One limitation of feedforward architectures such as the MLP and the CNN is that they can only generate outputs based on the current input they receive. They cannot use previous information to modify their output. Rumelhart et al [54] developed recurrent neural networks (RNN) to address this issue. RNNs are neural networks that are especially effective at processing sequences of inputs. They achieve this by introducing a feedback loop into the network, so that a neuron can receive its own output as an input during the next time step. The feedback loop allows the network to ”remember” past data, as the outputs from previous samples can influence current ones. A notable issue with recurrent neural networks is the vanishing (or exploding) gradient problem, which occurs when the influence of an input on the network either decays or increases exponentially as it repeatedly cycles over the recurrent connections.

Long short-term memory (LSTM) networks are a type of recurrent neural network that manages to solve the vanishing gradient issue [55], making it very useful for remembering long-term dependencies in a sequence of inputs. Its architecture can be seen in Figure 2.5, where x_t is the input and h_t is the output. In a reinforcement learning scenario, x_t would be the current observation of the state of the environment. The output h_t of the LSTM is referred to as the hidden state because it is fed backward to become part of the input to the network. Therefore the full input of the LSTM is a combination of the current observation (x_t) and the previous output (h_{t-1}). One of the key components within the LSTM is the cell state (c_t). It is the element that stores information from previous time steps, allowing the LSTM to detect the temporal relationships between the inputs.

The rest of the components within the LSTM are the forget gate (f_t), the input gate (i_t), and the output gate (o_t). Each of these is composed of a fully connected feedforward layer (or two in the case of the input gate) with its own parameters $\mathbf{\Theta}$ and activation function, as shown in equations 2.22-2.24. The purpose of the forget gate and the input gate is to modify the cell state at each time step. The forget cell determines what information is removed from the cell state, while the input cell adds information from the current observation to the cell state, as expressed in equation 2.25. Finally, the output gate determines what part of the cell state is passed to the output (h_t) of the LSTM. This structure is much more complex than conventional feed forward networks, but it is essentially four neural layers acting together to update the cell state and output the hidden state. Together, these four layers allow the LSTM to memorize data from an indefinite amount of previous inputs, making it suitable for tasks that involve temporal sequences of data, such as trajectory control [56].

Various researchers have leveraged the LSTM to implement reinforcement meta-learning [24]. In this context, the term *meta-learning* is used to emphasize that the neural network policy not only learns to perform well on a specific task, but also learns to adapt to a wide range of tasks. These tasks are represented as POMDPs where the policy cannot directly observe the complete state of the system. For instance, the policy may have to face unknown situations such as different environmental dynamics, actuator failures, or measurement errors [57]. The LSTM policy can learn to adapt to these uncertain conditions by detecting the temporal relationships between sequences of inputs. This allows the LSTM to generate outputs based on all the previous inputs, as opposed to a feedforward network which can only use the current input to generate an output.

$$f_t = \sigma\left(\mathbf{\Theta}_f \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix}\right) \tag{2.22}$$

$$i_t = \sigma\left(\mathbf{\Theta}_i \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix}\right) \odot \tanh\left(\mathbf{\Theta}_g \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix}\right) \tag{2.23}$$

$$o_t = \sigma\left(\mathbf{\Theta}_o \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix}\right) \tag{2.24}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{2.25}$$

$$h_t = o_t \odot \tanh(c_t) \tag{2.26}$$

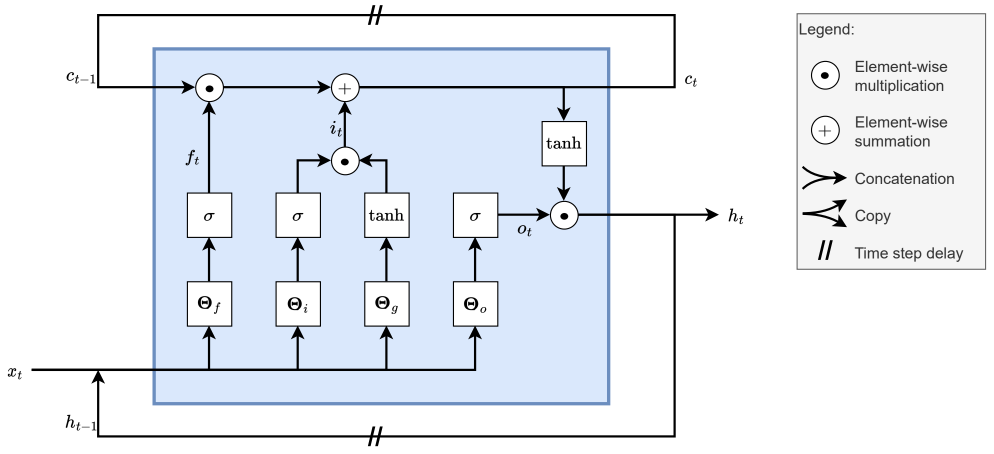


Figure 2.5: Structure of a LSTM

3 Modeling

In order to learn via reinforcement learning, the controller needs to gather experience from somewhere. Ideally, data can be taken from the real world, but that data can be scarce, expensive, or otherwise difficult to acquire. Such is the case for rendezvous missions. Hence, the training data had to be gathered from a simulated environment that was developed during this project. This chapter describes how this environment was modeled to simulate a rendezvous scenario. Section 3.1 describes the chosen mission scenario, section 3.2 covers the dynamics model for the translational and rotational motion, section 3.3 explains how the environment must be implemented to interact with the learning algorithm, and lastly section 3.4 shows the experiments that were performed to verify the dynamics model.

3.1. Mission scenario

As previously mentioned, the mission scenario is the final approach of a chaser towards a rotating target. The goal during this scenario is to make a chaser vehicle approach the target without colliding with it. The chaser is a small spacecraft represented as a uniform-density cube with a length of 1 meter and a mass of 100 kilograms. It is assumed that the chaser is equipped with an attitude and orbit control system consisting of thrusters and reaction wheels. The thrusters can generate a net force of $\pm 10\ N$ along each of the chaser's three body axes, while the reaction wheels can generate a net torque of $\pm 0.2\ N.m$ along the three body axes. The torque and thrust constraints were defined based on the current state of the art of commercial off-the-shelf components for small spacecraft [58]. Together, these actuators give the chaser full control over its position, velocity, attitude, and rotation rate. The chaser is also equipped with navigation instruments that measure the state of the chaser and the target. During the approach trajectory, these navigation instruments must remain pointed at the target to enable relative navigation.

The target is also represented as a uniform cube, but it also has two long appendages along one of its axes. These represent solar arrays. In order to prevent collisions between the chaser and the target, the latter is surrounded by a keep-out zone that the chaser must avoid. The keep-out zone has a spherical shape with a radius of 5 meters and a conical cut-out that represents an entry corridor. A visual representation of the scenario can be seen in Figure 3.1, where the chaser is depicted as the green cube, and the target is depicted as the yellow cube surrounded by the red keep-out zone. In this figure, the entry corridor is pointed directly at the target, but the target is inactive, so it can rotate freely and has no means to control its own position or attitude. As the target rotates, the entry corridor rotates with it. This adds another layer of difficulty to the chaser's objective, as it can only approach the target through the entry corridor. Once the chaser is within the entry corridor, it must complete the trajectory by arriving at a given position near the target where it must achieve the required terminal conditions to finish the final approach phase. The terminal conditions require the chaser to cancel its motion relative to the target so that the capture phase can begin. In other words, the chaser must reduce its position, velocity, attitude, and rotation rate relative to the target. These terminal conditions can be seen in Table 3.1. For a trajectory to be considered successful, the chaser needs to achieve the terminal conditions without ever colliding with the keep-out zone.

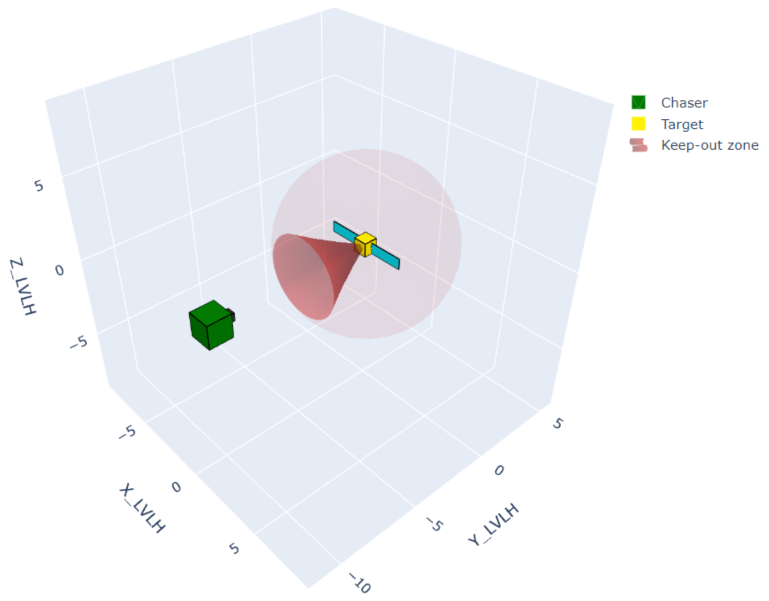


Figure 3.1: Mission scenario.

Parameter	Condition
Position	$r_e < 0.5$ m
Velocity	$v_e < 0.1$ m/s
Attitude	$\theta_e < 5$ deg
Rotation rate	$\omega_e < 1$ deg/s

Table 3.1: Scenario terminal conditions.

3.2. Dynamics

The dynamics determine how the state $\mathbf{S}$ of the environment evolves over time. In this scenario, the state of the environment is defined by the position, velocity, attitude, and rotation rate of the chaser and the target. Position and velocity are expressed in the LVLH reference frame, which was defined back in section 2.1 and is suitable for rendezvous scenarios. In this reference frame, the target is located at the origin, and thus the position of the chaser $(\mathbf{r})$ and the velocity of the chaser $(\mathbf{v})$ are expressed relative to the target:

$$\mathbf{r} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix}_{LVLH} , \quad \mathbf{v} = \begin{bmatrix} \dot{r}_x \\ \dot{r}_y \\ \dot{r}_z \end{bmatrix}_{LVLH} \tag{3.1}$$

Meanwhile, the attitude of the vehicles is expressed using unit quaternions. As described in section 2.1, a unit quaternion is a four-element array that can represent any rotation in three-dimensional space, and thus can be used to represent the orientation of a reference frame with respect to another reference frame. In this rendezvous scenario, the quaternion $\mathbf{q}_C$ represents the attitude of the chaser body frame with respect to the LVLH frame, and similarly $\mathbf{q}_T$ represents the attitude of the target body frame with respect to the LVLH frame. Lastly, the rotation rates of the chaser and the target body frames relative to the LVLH frame are expressed with the two vectors $\boldsymbol{\omega}_C$ and $\boldsymbol{\omega}_T$, respectively. Hence, the full state vector $(\mathbf{S})$ of the system is as shown in equation 3.2. The six terms in the state vector describe the following six state variables:

- $\mathbf{r}$: Position of the chaser expressed in the LVLH frame
- $\mathbf{v}$: Velocity of the chaser expressed in the LVLH frame
- $\mathbf{q}_C$: Attitude of the chaser body frame relative to the LVLH frame

- $\boldsymbol{\omega}_C$: Rotation rate of the chaser body frame relative to the LVLH frame
- $\mathbf{q}_T$: Attitude of the target body frame relative to the LVLH frame
- $\boldsymbol{\omega}_T$: Rotation rate of the target body frame relative to the LVLH frame

$$\mathbf{S} = \begin{bmatrix} \mathbf{r} \\ \mathbf{v} \\ \mathbf{q}_C \\ \boldsymbol{\omega}_C \\ \mathbf{q}_T \\ \boldsymbol{\omega}_T \end{bmatrix} \tag{3.2}$$

The control policy can affect the state of the system by directing the actuators to exert a net force ($\mathbf{F}_C$) and torque ($\mathbf{M}_C$) on the chaser. The forces and torques act along the three axes of the chaser body frame, since the actuators are fixed to the body frame. It is assumed that the net force executed by the control policy causes an instantaneous change in velocity ΔV on the chaser at the beginning of the time step. The magnitude of this change in velocity is proportional to the length of the time step Δt, and inversely proportional to the mass (m) of the chaser, as shown in equation 3.3. In reality, the ΔV would occur gradually throughout the time step, but making the ΔV instantaneous greatly reduces the computational effort of running the dynamics model, and it only introduces a small error into the model. This will be verified in a later section of this chapter. Similarly, it is assumed that the net torque causes an instantaneous change in rotation rate $(\Delta\omega)$, which can be estimated from Euler's rotation equation, as shown in equation 3.5.

$$\mathbf{F}_C = \begin{bmatrix} F_{x,C} \\ F_{y,C} \\ F_{z,C} \end{bmatrix}, \qquad \mathbf{M}_C = \begin{bmatrix} M_{x,C} \\ M_{y,C} \\ M_{z,C} \end{bmatrix} \tag{3.3}$$

$$\Delta\mathbf{V} = \frac{\mathbf{F}_C}{m}\Delta t \tag{3.4}$$

$$\Delta\boldsymbol{\omega} = \mathbf{I}^{-1}\left[\mathbf{M} - \boldsymbol{\omega} \times \mathbf{I}\boldsymbol{\omega}\right]\Delta t \tag{3.5}$$

The position and velocity of the chaser will change over time according to the CW model. The closed form solution described back in equation 2.2 is used to compute the new position and velocity on each time step of the trajectory, adding the exerted ΔV at the start of each time step, as shown in equation 3.6. It should be noted that the ΔV vector must be rotated from the chaser body frame to the LVLH reference frame before being applied to the CW model.

The attitude and rotation rate of the chaser and the target are determined via numerical integration with a fourth-order Runge-Kutta method (RK4), as shown in equation 3.7. The inputs of the RK4 function are the length of the time step and the current values of attitude and rotation rate, as well as their derivatives. With the given inputs, the RK4 function computes the new attitude and rotation rate at the end of the time step. The derivative of the attitude ($\dot{\mathbf{q}}$) is obtained from the quaternion kinematics model shown in equation 2.8, and the derivative of the rotation rate ($\dot{\boldsymbol{\omega}}$) is obtained from equation 2.5. It should be noted that the $\Delta\boldsymbol{\omega}$ is applied only to the chaser and not to the target, since the target does not experience any torque.

$$\begin{bmatrix} \mathbf{r} \\ \mathbf{v} \end{bmatrix}_{t+\Delta t} = \Phi_{CW} \begin{bmatrix} \mathbf{r} \\ \mathbf{v} + \Delta\mathbf{V} \end{bmatrix}_t \tag{3.6}$$

$$\begin{bmatrix} \mathbf{q} \\ \boldsymbol{\omega} \end{bmatrix}_{t+\Delta t} = RK4\left(\begin{bmatrix} \mathbf{q} \\ \boldsymbol{\omega} + \Delta\boldsymbol{\omega} \end{bmatrix}_t, \begin{bmatrix} \dot{\mathbf{q}} \\ \dot{\boldsymbol{\omega}} \end{bmatrix}_t, \Delta t\right) \tag{3.7}$$

3.3. Implementation

In order to be compatible with the reinforcement meta-learning framework, the model of the rendezvous scenario needs to be formulated as a POMDP. The POMDP framework in this case is comparable to a discrete dynamical system, where the state of the system evolves in discrete time steps in response to the output of the control policy. The current state of the system is relayed to the control policy in the form of an observation that contains limited information. The observation is represented as a one-dimensional array of the state variables, as shown in equation 3.8. This observation contains all the state variables described in equation 3.2, with the exception of the target's rotation rate (ω_T). Such a partial observation can be representative of scenarios where the onboard capabilities of the chaser are limited due to factors such as a lack of sensors or instrument failure. Since the control policy cannot directly observe the target's rotation rate, it must learn to infer it indirectly via meta-learning.

After receiving an observation, the control policy relays its output to the POMDP in the form of an action, expressed as a one-dimensional array containing the net forces and torques exerted by the actuators on the chaser. The values of the observations and the actions are normalized to a range of $[-1,\ 1]$ as recommended by the developer guide on SB3 [59]. Normalizing the observations can improve the convergence rate of the learning algorithm by ensuring that the gradients in the network have similar magnitudes.

$$Observation = \begin{bmatrix} \mathbf{r} \\ \mathbf{v} \\ \mathbf{q}_C \\ \boldsymbol{\omega}_C \\ \mathbf{q}_T \end{bmatrix}_{Norm.} \tag{3.8}$$

$$Action = \begin{bmatrix} \mathbf{F}_C \\ \mathbf{M}_C \end{bmatrix} = \begin{bmatrix} F_{x,C} \\ F_{y,C} \\ F_{z,C} \\ M_{x,C} \\ M_{y,C} \\ M_{z,C} \end{bmatrix}_{Norm.} \tag{3.9}$$

After receiving the action from the control policy, the POMDP updates the state of the system using the dynamics equations described in section 3.2. The POMDP then outputs a new observation representing the new state of the environment. In addition, the POMDP also has to generate a reward value based on the new state of the environment. Further information on how the reward is generated can be found in section 4.3. The new observation is once again fed into the control policy, and the cycle continues. Training is divided into episodes, and each episode represents a single rendezvous attempt. An episode is ended automatically if a given time limit is reached, or if the chaser strays too far from its objective. Once an episode ends, the MDP is reset to its initial conditions so that a new episode may begin.

The functionality described above was implemented in Python, using the OpenAI Gym package [60]. This open-source package is specifically designed to be used with reinforcement learning algorithms. It provides a framework that allows users to create custom environments in order to train and evaluate a control policy. A template for such a custom environment can be seen in Appendix B.1.

3.4. Dynamics verification

Several trajectories were tested to verify the correct implementation of the CW model and the rigid body rotation model. First, the implementation of the CW model was tested on a free drift trajectory. In this type of trajectory there is no thrust applied to the chaser, so its motion is determined entirely by the initial conditions. Free drift maneuvers are often executed during the far-range rendezvous phase in order to save fuel. For this test case, the chaser was initialized at a position that was Δx meters lower than the target's orbit, and with an initial velocity of $3n\Delta x/2$ m/s along the y-direction. According to the CW model, these initial conditions should cause the chaser to move linearly along the y-axis at a constant velocity [61], and the expected distance traveled after one orbit should be equal to $3\pi\Delta x_0$. In addition to these initial conditions, the initial z-coordinate of the chaser is set to a non-zero value Δz. The non-zero z-coordinate should lead to a periodic motion in the out-of-plane direction, while not affecting the motion in the in-plane direction. The values for each of the initial conditions can be seen in equation 3.10. The attitude and rotation rate of the chaser are ignored in this scenario because this test is only concerned with the position and velocity of the chaser.

$$n = 0.00104\text{rad/s}, \qquad \mathbf{r}_0 = \begin{bmatrix} -\Delta x_0 \\ \Delta y_0 \\ \Delta z_0 \end{bmatrix} = \begin{bmatrix} -10\text{ m} \\ 0\text{ m} \\ 10\text{ m} \end{bmatrix}, \qquad \mathbf{v}_0 = \begin{bmatrix} 0 \\ \frac{3}{2}n\Delta x_0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0\text{ m/s} \\ 0.0156\text{ m/s} \\ 0\text{ m/s} \end{bmatrix} \tag{3.10}$$

Figure 3.2 shows the results after propagating the initial conditions over one full orbit. The plot on the left displays the x, y, and z components of the chaser's position as a function of time, and the plot on the right shows the components of the chaser's velocity. As expected, the position along the y-direction increases linearly while the position along the x-direction remains constant. After one orbit, the distance covered in the y-direction is 94.25 meters, which is equivalent to the expected $3\pi\Delta x_0$. Meanwhile, the z-coordinate displays the expected periodic motion, and it does not affect the motion in the x and y directions, proving that the out-of-plane motion is decoupled from the in-plane motion. The results of this experiment show that the dynamics model of the learning environment behaves as expected during free drift trajectories.

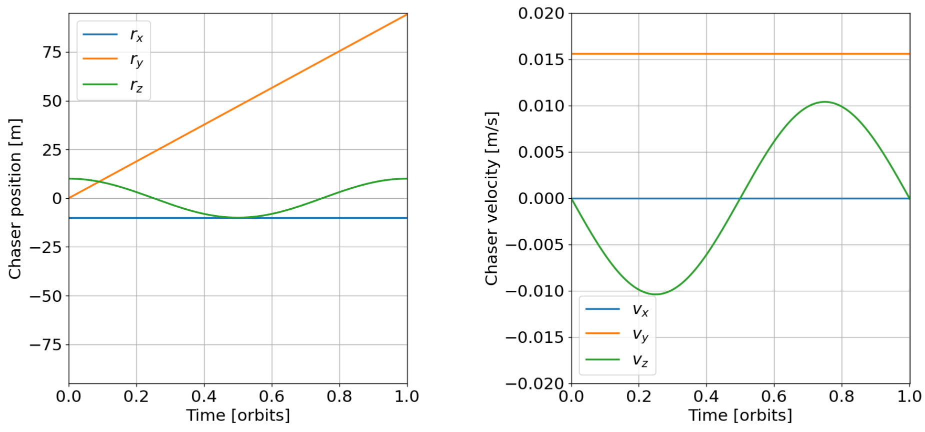


Figure 3.2: Verification of a free drift trajectory.

Next, the dynamics model was tested on a continuous-thrust trajectory. In this type of trajectory, a thrust force is constantly acting on the chaser. Trajectories of this kind are commonly used for the close-range rendezvous phase, where the chaser needs more maneuverability to perform the final approach to the target. A common example of a continuous thrust maneuver is a straight-line approach along the $-y$ direction. During this maneuver, the chaser must constantly apply thrust in the $-x$ direction in order to maintain a constant velocity and avoid drifting sideways. The magnitude of the required thrust can be computed as a function of the chaser's velocity, as shown in equation 3.11, where m_c is the mass of the chaser and v_y is the desired velocity of the chaser along the y axis. The initial conditions chosen for this test can be seen in equation 3.12. Just like in the previous test, the attitude dynamics of the chaser are ignored. It is simply assumed that the chaser body frame is oriented in the same direction as the LVLH frame, so that the force F_x acting on the chaser's x body axis is also acting along the x axis of the LVLH frame.

$$F_x = -2v_y m_c n \tag{3.11}$$

$$n = 0.00104\text{ rad/s}, \qquad \mathbf{r}_0 = \begin{bmatrix} 0\text{ m} \\ -120\text{ m} \\ 0\text{ m} \end{bmatrix}, \qquad \mathbf{v}_0 = \begin{bmatrix} 0\text{ m/s} \\ 2\text{ m/s} \\ 0\text{ m/s} \end{bmatrix}, \qquad \mathbf{F} = \begin{bmatrix} -0.42\text{ N} \\ 0\text{ N} \\ 0\text{ N} \end{bmatrix} \tag{3.12}$$

Since the chaser starts at a distance of 120 meters from the origin and moves at a constant rate of 2 m/s, it is expected that it will reach the origin after 60 seconds. Thus, the simulated trajectory was run for 60 seconds to see if the chaser reached the origin of the LVLH reference frame as expected. The results of this experiment can be seen in Figure 3.3, which displays the x and y coordinates of three trajectories. Without applying any thrust, the path of the chaser would quickly drift sideways, as shown by the blue curve. In theory, if a constant thrust was continuously applied, then the trajectory of the chaser would be a straight-line arriving precisely at the origin, as shown by the dotted line in the plot. The result of this experiment did not achieve a straight-line trajectory, since it drifted towards the $-x$ direction by 6 centimeters. The reason for this drift is that the dynamics model assumes that the thrust is executed instantly at the beginning of each time step, instead of continuously during the full length of the time step. For the purposes of this model, a drift of 6 centimeters for a trajectory of 120 meters is not significant, so the dynamics model is considered to be sufficiently accurate.

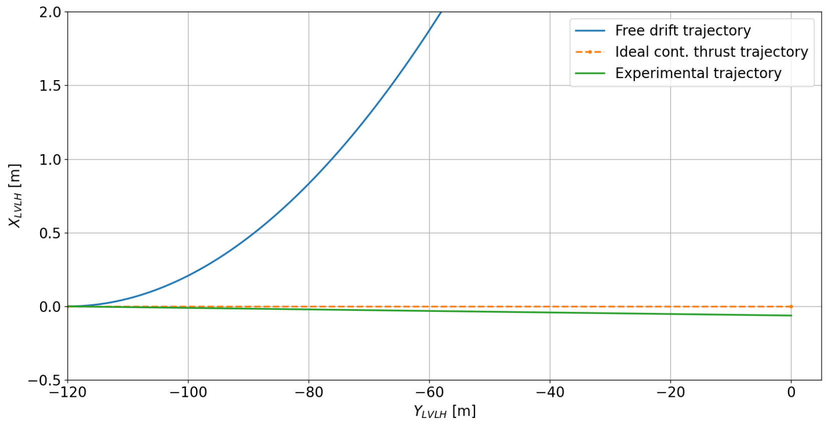


Figure 3.3: Verification of a constant thrust trajectory.

Having proven that the implementation of the CW model works as expected, the following step was to verify the attitude dynamics model. Just like in the previous case, two types of trajectories were tested: a free drift trajectory and a continuous control trajectory. For the free drift example, a special case was chosen to demonstrate that the dynamics model follows the intermediate axis theorem. This theorem dictates that for a body with distinct principal axes of inertia, the rotation around the major and minor axes of inertia is stable, whereas the rotation around the intermediate axis of inertia is unstable. Although the effects of the intermediate axis theorem are difficult to observe on Earth, they can be clearly seen in space, as an object can rotate undisturbed for a longer period of time. If the object is rotating about its intermediate axis of inertia, it will start wobbling, and its intermediate axis will then switch direction by 180 degrees, even though there is no torque acting on the object. To simulate this kind of motion, the dynamics model of the learning environment was initialized with the initial conditions shown in equation 3.13. The value of the quaternion $\mathbf{q}_{c0}$ corresponds to a rotation of zero degrees, indicating that the chaser body frame is initially oriented in the same direction as the LVLH frame. The moment of inertia $\mathbf{I}$ was defined so that the chaser's y axis is the intermediate axis of inertia, and therefore the initial rotation rate of the chaser $\boldsymbol{\omega}_{c0}$ is also around the y axis. The position and velocity of the chaser are irrelevant in this case, so the chaser is assumed to be stationary at the origin of the LVLH frame.

$$\mathbf{q}_{c0} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \qquad \boldsymbol{\omega}_{c0} = \begin{bmatrix}0 \text{ deg/s}\\5 \text{ deg/s}\\0 \text{ deg/s}\end{bmatrix}, \qquad \mathbf{I} = \begin{bmatrix}0.8 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1.2\end{bmatrix}\frac{1}{6}d^2m = \begin{bmatrix}\frac{80}{6} & 0 & 0\\0 & \frac{100}{3} & 0\\0 & 0 & \frac{120}{6}\end{bmatrix} \text{kg.m}^2 \tag{3.13}$$

The results of running the dynamics model with these initial conditions can be seen in Figure 3.4. The orthogonal red, green, and blue arrows represent the x, y & z axes of the chaser body frame, respectively. The translucent arrows indicate the initial attitude of the chaser, while the opaque arrows indicate its final attitude, and the dotted line displays the motion of the y axis. Initially, the chaser body frame is aligned with the LVLH frame, rotating around the body y axis at a rate of 5 deg/s. But a wobble appears and grows larger over time, causing the y axis to change direction. At the end of the simulation, the chaser is still rotating at a rate of 5 deg/s, but its y axis has flipped around 180 degrees, as predicted by the intermediate axis theorem. If the simulation is run for longer, the y axis begins wobbling again and returns to its original position, and then continues repeating the same motion back and forth. The result of this experiment proves that the attitude dynamics model of the learning environment is accurate enough to replicate complex dynamics effects.

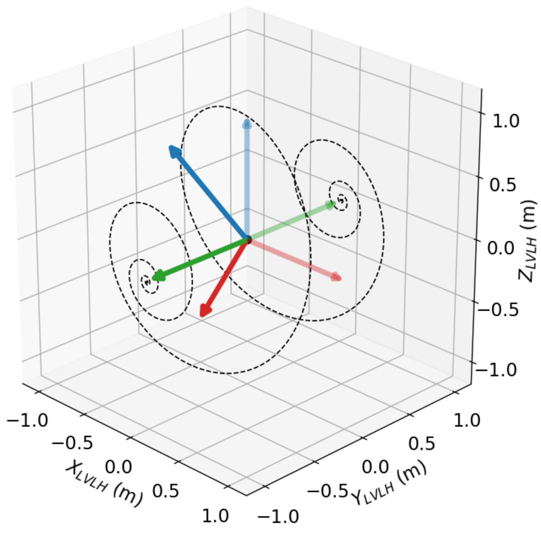


Figure 3.4: Verification of the intermediate axis theorem.

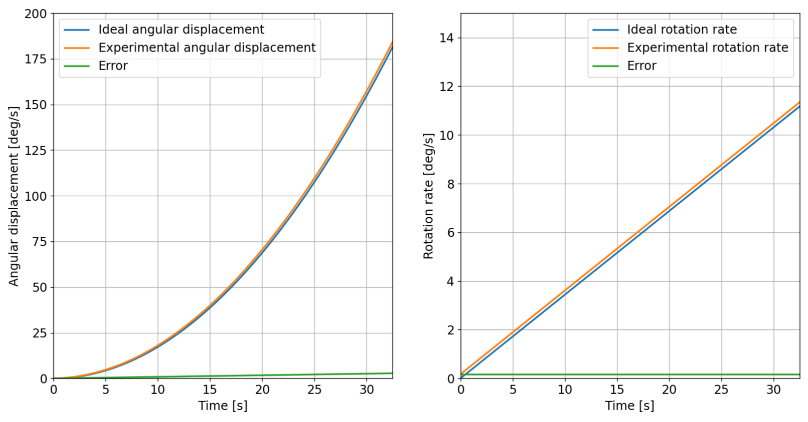


Figure 3.5: Verification of the rotation induced by torque.

Lastly, one final experiment was performed to verify that the attitude of the chaser behaves as expected when a control torque is applied. The initial conditions for this test can be seen in equation 3.14. The chaser was initialized in a stationary state, and a constant control torque of 0.1 N.m was applied on the z body axis. The expected angular acceleration can be derived from Euler's rotation equation as shown in equation 3.15. Given that the principal moments of inertia are equal and the control torque is constant, the angular acceleration must also be constant. Knowing that the angular acceleration is constant, the total change in rotational velocity ($\Delta\omega_z$) and the total angular displacement ($\Delta\theta_z$) can be computed as shown in equations 3.16 and 3.17, respectively. Using these formulas, it can be determined that after applying the control torque for 32.4 seconds the chaser should have rotated approximately 180 degrees around the body z axis, and its rotation rate should have linearly increased from zero to 11.2 deg/s. A trajectory was simulated with the dynamics model for this amount of time to observe if the results matched the expected behavior. The results can be seen in Figure 3.5, where the plot on the left shows the angular displacement as a function of time, and the plot on the right shows the rotation rate as a function of time. The angular displacement obtained from the simulation is very similar to the expected value, although there is a small difference that grows over time. At the end of the trajectory this difference has grown to 2.8 deg. The reason for this small error is that the dynamics model assumes that the control torque causes an instantaneous increase in rotation rate at the beginning of each time step, instead of a gradual increase throughout the duration of the time step. This choice was made to reduce the computational effort of running the model. Since the errors caused by this assumption are small, no changes will be made to the model.

$$\mathbf{q}_{c0} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{\omega}_{c0} = \begin{bmatrix} 0 \text{ deg/s} \\ 0 \text{ deg/s} \\ 0 \text{ deg/s} \end{bmatrix}, \quad \mathbf{I} = \begin{bmatrix} \frac{50}{3} & 0 & 0 \\ 0 & \frac{50}{3} & 0 \\ 0 & 0 & \frac{50}{3} \end{bmatrix} \text{kg.m}^2, \quad \mathbf{M} = \begin{bmatrix} 0 \text{ N.m} \\ 0 \text{ N.m} \\ 0.1 \text{ N.m} \end{bmatrix} \tag{3.14}$$

$$\dot{\omega}_z = \frac{1}{I_{zz}} \left[M_z - (I_{yy} - I_{xx}) \omega_x \omega_y \right] = \frac{M_z}{I_{zz}} \tag{3.15}$$

$$\Delta\omega_z = \omega_{z0} + \dot{\omega}_z \Delta t \tag{3.16}$$

$$\Delta\theta_z = \theta_{z0} + \omega_{z0} \Delta t + \dot{\omega}_z (\Delta t)^2 \tag{3.17}$$

4 Learning Algorithm

The learning algorithm is the element that is responsible for training the controller. During training, the algorithm collects data from the virtual environment and then uses the data to update the parameters of the controller, thereby modifying its performance to maximize the cumulative reward. For this project, the PPO algorithm with recurrent neural networks was chosen. A theoretical overview of the PPO algorithm was introduced in section 2.2.2. The current chapter presents the practical details of how the learning algorithm was implemented, and how it interacts with the neural network policy. Section 4.1 explains the processes that are executed when training the algorithm, and how to use them in a Python script. Section 4.2 then describes the structure of the neural network policies that are responsible for controlling the spacecraft. Lastly, section 4.3 presents the reward function that the learning algorithm uses to improve the performance of the controller.

4.1. Algorithm

For this project, the PPO algorithm was chosen to train the control policy. PPO is a modern learning algorithm that uses a clipped loss function to limit the size of each policy update, making it more stable than other policy gradient algorithms. It has been shown to perform well on various continuous control scenarios, and it is also conceptually simpler and more sample efficient than similar algorithms such as TRPO. The version of PPO used throughout this project is from the Stable Baselines 3 (SB3) library. This version was chosen because SB3 is a reliable library with ample documentation, making it safe and easy to use. It also provides an easy interface for modifying hyperparameters, which is useful for tuning the algorithm.

To train the control policy, the PPO algorithm uses an iterative process. On each iteration, it first collects data from the environment using its current version of the control policy. In reinforcement learning, the process of data collection is often referred to as *collecting rollouts*. Once the data has been collected, it is separated into mini-batches, and the data from each mini-batch is then used to compute the clipped loss function L^{CLIP}. Next, it computes the gradient of the loss function with respect to the parameters of the policy, and it updates those parameters via gradient descent for a given number of epochs.

The learning process can last several hours, which is why it is often performed on remote servers or high-performance systems such as DelftBlue. While the learning algorithm is running, it is useful to periodically evaluate the performance of the control policy in order to assess how well the training is progressing. For this purpose, a callback function was introduced into the learning algorithm. The callback function is executed on every iteration of the training loop, just before the rollouts are collected. When executed, the callback function runs the current control policy on several episodes, and then computes the average reward obtained throughout those episodes. The average reward is then logged onto W&B's online platform, where the progress of the learning algorithm can be visualized in real-time. In addition, the current policy is saved if the average reward is higher than on previous evaluations. At the end of the training process, the callback function will have recorded the progress of the learning algorithm, and it will also have saved the policy that achieves the best reward on average. The Python function used by the callback to evaluate the current policy can be seen in Appendix B.2.

A flowchart of the operations performed by the learning algorithm can be seen in Figure 4.1. The process begins by setting the step count to zero, and then it repeats the main loop until the step count reaches the desired number of training steps. At the start of each iteration, the callback function is used to log the performance of the current policy, and save it if necessary. After the callback has been executed, the learning algorithm starts collecting rollouts, where the current policy repeatedly interacts with the environment for a given number of steps. Once the data is collected, the learning algorithm optimizes the policy to minimize the loss. Then the step count is updated, and then the process repeats itself. After enough iterations are performed, the policy learns to output the actions that maximize the cumulative reward over an episode.

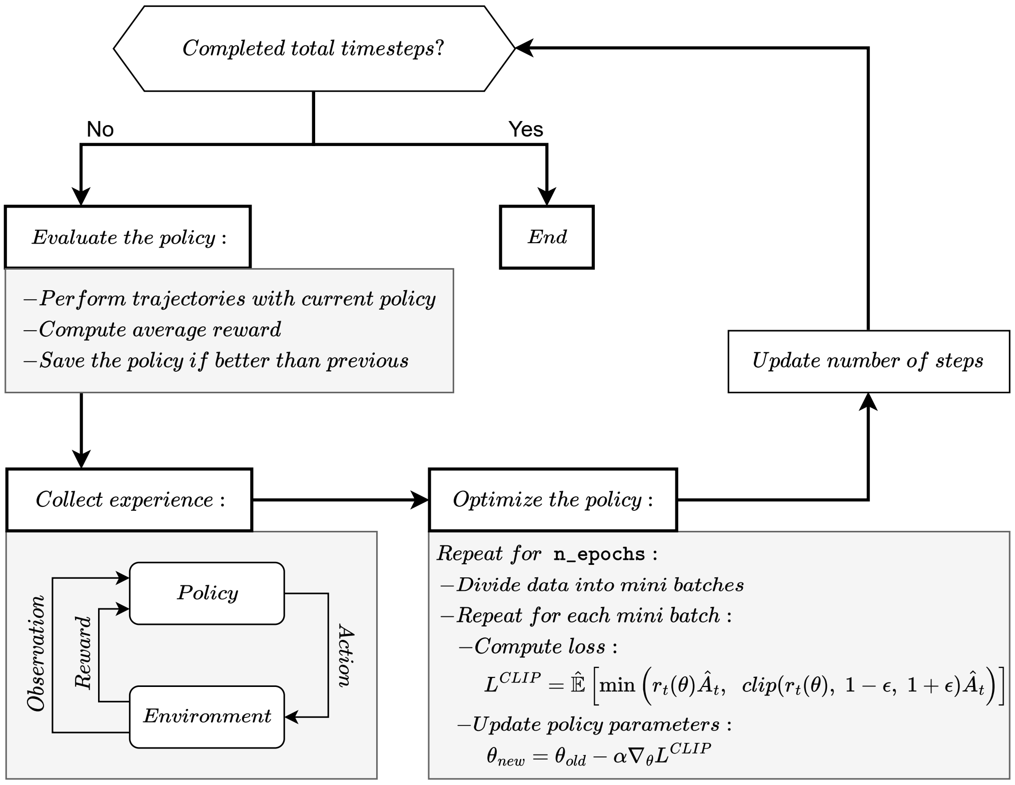

Completed total timesteps?
No
Yes
Evaluate the policy :
End
−Perform trajectories with current policy
−Compute average reward
−Save the policy if better than previous
Update number of steps
Collect experience :
Optimize the policy :
Policy
Environment
Observation
Reward
Action
Repeat for n_epochs :
−Divide data into mini batches
−Repeat for each mini batch :
−Compute loss :
$L^{CLIP} = \hat{\mathbb{E}}\left[\min\left(r_t(\theta)\hat{A}_t,\ clip(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon)\hat{A}_t\right)\right]$
−Update policy parameters :
$\theta_{new} = \theta_{old} - \alpha\nabla_\theta L^{CLIP}$

Figure 4.1: Overview of the learning algorithm.

4.2. Policies

In deep reinforcement learning, the policy consists of one or more neural networks whose parameters (θ) are repeatedly updated during training. Actor-critic methods such as PPO have two networks in their policy: the actor and the critic. Both the actor and the critic receive observations as inputs, but their outputs are different. The actor generates the control action that will be applied to the environment, and the critic outputs the value (V) of the given observation. Hence, the output of the critic is always a scalar, whereas the output of the actor is an array of six values representing the control forces and torques. For a feedforward (MLP) policy, the architecture of the neural networks can be defined by three attributes:

- Number of layers
- Number of neurons per layer
- Activation function

There is no ideal architecture that works for every problem, so a policy can be customized by modifying these three parameters to better fit the use case. Chapter 5 will explain how a custom policy was created by tuning the network architecture.

To implement reinforcement meta-learning, a recurrent (RNN) policy is necessary. The specific RNN chosen for this project is the Long Short-Term Memory (LSTM) network. This type of network was selected because LSTMs are known for being efficient at detecting long-term relationships in sequences of data. The recurrent policy contains one neural network for the actor and another network for the critic, much like the non-recurrent policy does. However, in the recurrent policy the networks are more complex, since they contain a LSTM unit followed by a multilayer perceptron. The structure of the recurrent networks can be seen in Figure 4.2.

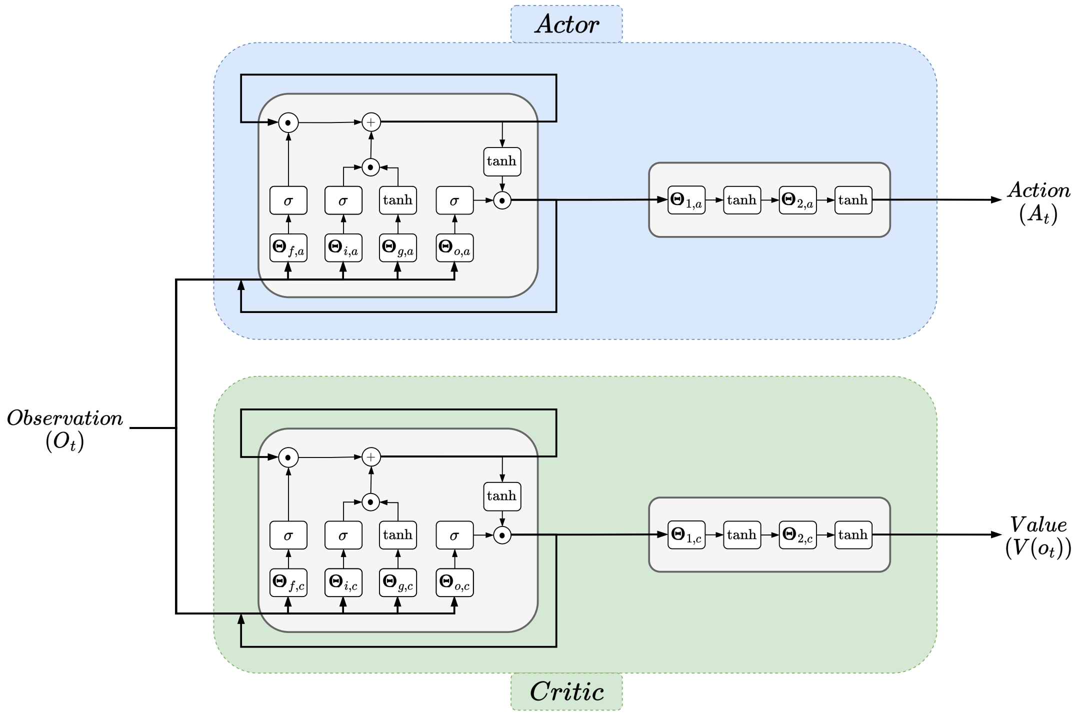

Actor
Action (A_t)
Observation (O_t)
Value $(V(o_t))$
Critic
σ
tanh
$\Theta_{f,a}$ $\Theta_{i,a}$ $\Theta_{g,a}$ $\Theta_{o,a}$
$\Theta_{1,a}$ $\Theta_{2,a}$
$\Theta_{f,c}$ $\Theta_{i,c}$ $\Theta_{g,c}$ $\Theta_{o,c}$
$\Theta_{1,c}$ $\Theta_{2,c}$

Figure 4.2: Structure of the recurrent policy.

The actor and the critic have the same architecture, but the parameters $\mathbf{\Theta}$ of the networks evolve differently as the training progresses. Both networks take the current observation of the environment as their input. The observation is passed to the LSTM, where it is combined with the previous output of the LSTM and then passed through the forget gate, the input gate, and the output gate, as described in section 2.2.3. Next, the output of the LSTM is passed to a multilayer perceptron which produces the output of the network. In the case of the actor, the multilayer perceptron outputs the next action that will be applied to the environment, i.e. the thrust and torque exerted on the chaser. And in the case of the critic, the multilayer perceptron outputs the estimated value of the current observation of the environment's state, which is used by the learning algorithm to compute the loss function.

Throughout this project, the recurrent policy was used to implement reinforcement meta-learning, while the non-recurrent policy was used for comparison. Appendix B.3 shows how to train the recurrent and non-recurrent policies using SB3.

4.3. Reward function

The reward function determines the behavior of the trained controller. Since the learning algorithm will try to maximize the cumulative reward, the reward function should encourage the policy to accomplish the objective of the scenario. In the rendezvous final approach scenario, the ultimate objective is to make the chaser achieve the terminal conditions required to initiate the capture phase. A logical idea would then be to give the policy a positive reward when it achieves the terminal conditions, and give it zero rewards otherwise, as shown in equation 4.1, where k_* is a positive constant. The issue with this approach is that it leads to a sparse reward function. In other words, only a very small section of the environment's state space provides rewards to the environment, while the rest of the state space provides no feedback at all. With a sparse reward function, the policy may take a long time to improve because it has very few rewards to learn from. It would essentially have to explore the entire state space without any rewards to guide it until it eventually found the terminal conditions.

$$R_* = \begin{cases} 0 & \text{if terminal conditions are NOT achieved} \\ k_* & \text{if terminal conditions are achieved} \end{cases} \tag{4.1}$$

The solution to the sparse reward problem is to shape the reward function by using domain knowledge to give smaller intermediate rewards that guide the policy towards the ultimate objective. The shaping reward can be used to encourage behaviors that lead to a successful trajectory, while also penalizing behaviors that don't. In order to shape the reward function, it is useful to decompose the overall goal of the scenario into multiple smaller objectives. For instance, one objective can be to keep the chaser pointed towards the target, another objective can be to avoid collisions, and yet another objective can be to preserve fuel. The overall reward function can then be composed of several different terms, as shown in equation 4.2, where each of the first three terms addresses one of the minor objectives, and the last term is the sparse reward from equation 4.1.

$$R = R_\theta + R_f + R_c + R_* \tag{4.2}$$

The purpose of the reward term R_θ is to keep the chaser pointed at the target throughout the trajectory. In equation 4.3, the term θ_e is the chaser's pointing error, which is measured as the angle between the direction in which the chaser is currently pointing and the direction in which it should be pointing, as depicted in Figure 4.3. The term $\theta_{e,max}$ is the maximum allowed pointing error. If the chaser's pointing error exceeds $\theta_{e,max}$ then the episode is terminated early, because the target is no longer within the chaser's field of view.

$$R_\theta = k_\theta \left(1 - \frac{\theta_e}{\theta_{e,max}}\right) \tag{4.3}$$

The reward term R_f encourages the policy to generate fuel-efficient trajectories. to use less fuel during the trajectory. The amount of fuel used on each time step is estimated by the ΔV applied to the chaser. The value of the reward varies linearly, so that the agent receives no reward on the time steps where the maximum ΔV is applied, but receives a reward of k_f when no ΔV is applied. The purpose of this reward is to prevent the policy from using fuel unnecessarily.

$$R_f = k_f \left(1 - \frac{|\mathbf{\Delta V}|}{\mathbf{\Delta V}_{max}}\right) \tag{4.4}$$

The reward term R_c penalizes the agent for causing collisions. An essential requirement for the trajectories is that they do not lead to a collision between the chaser and the target. Therefore, a constant penalty k_c is applied on every time step that the chaser spends within the keep-out zone. This penalty discourages the policy from directing the chaser into the keep-out zone. In addition to this penalty, the reward term R_* is also used to encourage the policy to avoid collisions. The bonus reward k_* is only given if the terminal conditions are met *and* if the chaser has not been within the keep-out zone during the current episode. The goal of this constraint is to teach the policy that the bonus reward can only be achieved when the trajectory is collision-free. Otherwise, the policy could possibly disregard the collision penalty k_c, and enter the keep-out zone for a few time steps only to reach the bonus reward faster.

$$R_c = \begin{cases} 0 & \text{if chaser NOT in keep-out zone} \\ -k_c & \text{if chaser in keep-out zone} \end{cases} \tag{4.5}$$

As an example of how to compute the reward, consider the environment state shown in Figure 4.3. In this instance, the chaser is positioned within the keep-out zone and it has a large pointing error θ_e. The policy is applying a velocity change of ΔV_x along the chaser's x axis and ΔV_y along the y axis.

The overall reward obtained on this time step can then be computed as shown in equation 4.6. A positive reward is given for the chaser's pointing error and fuel usage, and a negative penalty is given for entering the keep-out zone. The success bonus R_* is not achieved on this time step because the

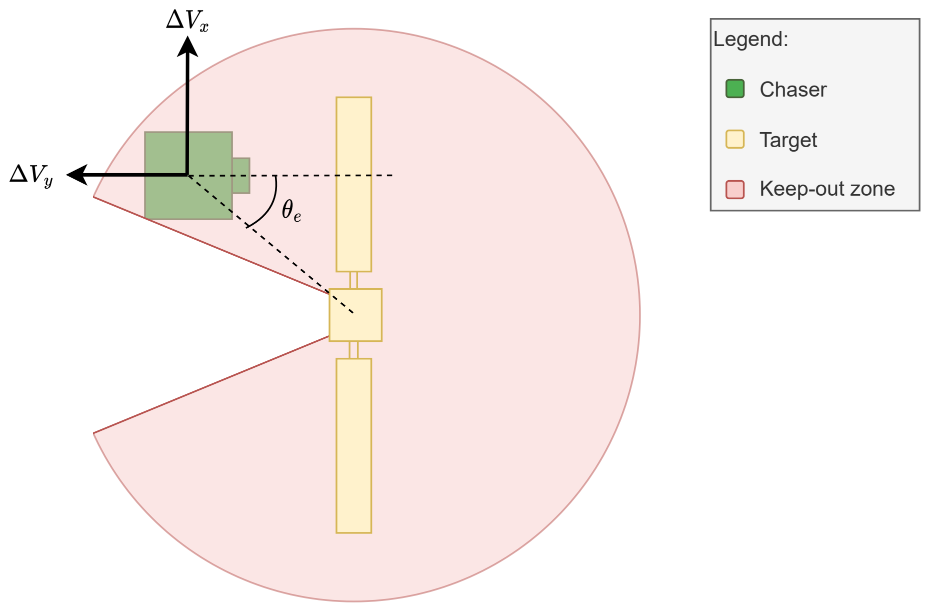


Figure 4.3: Example of chaser pointing error.

terminal conditions have not been met. The bonus also cannot be achieved for the remainder of this trajectory because the chaser has entered the keep-out zone.

$$R = k_\theta \left(1 - \frac{\theta}{\theta_{max}}\right) + k_f \left(1 - \frac{\Delta V_x + \Delta V_y}{\Delta V_{max}}\right) - k_c + 0 \tag{4.6}$$

Another detail related to the reward function is that the available state space is reduced over time. At the beginning of each episode, the chaser is allowed to move away from the target up to a distance d_{max}. But as the episode progresses, d_{max} is gradually lowered, shrinking the available space where the chaser can be. If the chaser goes beyond d_{max}, the current episode is terminated, therefore the chaser must stay within d_{max} in order to continue the current episode and earn more rewards. The value of d_{max} continues shrinking until it reaches the edge of the keep-out zone, so that the chaser then has more time to explore the state space closer to the target. This dynamic motivates the chaser to move towards the target, without needing to add an additional position term to the reward function.

The shaping reward terms can help to guide the policy towards the intended goal, but they may also lead to unexpected behaviors if some of the reward terms are more dominant than others. The relative importance of some of the reward terms with respect to others may lead the agent to accomplish some of the objectives while ignoring the rest of the objectives. For example, the authors of a similar rendezvous study [21] reported that the collision-avoidance capabilities of their policy degraded when their reward function emphasized fuel efficiency, and vice-versa. In order to avoid this, careful tuning of the reward function is required, which can be a time-consuming process [30]. Despite this difficulty, it is possible to successfully train a reinforcement learning model to achieve all of the desired objectives throughout the trajectory. Several researchers have seen good results with this type of shaped reward function [12, 62].

5 Model tuning

Before training the policy, the hyperparameters of the model must be tuned to improve the efficiency of the learning process. In the context of machine learning, the term *hyperparameters* refers to the parameters that have to be defined by the user prior to running the learning algorithm. This nomenclature is used to distinguish the user-defined parameters from the parameters of the neural network ($\boldsymbol{\theta}$) which are progressively updated by the learning algorithm. Three components of the model had to be tuned: the algorithm, the neural networks, and the reward function. Sections 5.1, 5.2, and 5.3 describe respectively how the hyperparameters of each of these components were tuned. Lastly, section 5.4 provides a summary of the tuning process.

Throughout the tuning process, *Weights & Biases* (W&B) was used to execute experiments and track results. W&B is a machine learning platform that provides many tools for developers to build and test their machine learning models. It has an easy-to-use Python API that allows developers to set up experiments and log the results, which can then be visualized and analyzed on an intuitive online dashboard. Besides of its usefulness for logging and visualizing data, W&B also offers a convenient tool called *sweeps*. With a sweep, the hyperparameter search can be automated using only a few lines of code. An example of how to perform a sweep is shown in Appendix B.4.

5.1. Algorithm tuning

The first component of the model to be tuned was the learning algorithm. The goal of this tuning is to find suitable set of hyperparameters that improves the algorithm's convergence time. SB3's implementation of the PPO algorithm has been thoroughly tested by its developers on several different control environments, so it has some fairly reliable default values for its hyperparameters. Regardless of this, a sweep was performed to check if the algorithm's performance can be improved. The four parameters to be tuned are: the learning rate, the clip range, the batch size, and the number of epochs. The learning rate determines the size of the gradient descent step that is used to optimize the networks. A large learning rate will perform larger policy updates, which may lead the policy to learn faster but may also make the learning less stable. On the other hand, a smaller learning rate will perform small policy updates and may slow down the learning process. Next, the clip range is the parameter (ε) described in Section 2.2.2. This hyperparameter is unique to the PPO algorithm, and it determines how different the new iteration of the policy can be from the previous one. Similarly to the learning rate, a large clipping parameter will allow the policy to change more quickly, which could reduce the algorithm's stability. Lastly, batch size and the number of epochs affect how the policy is updated on each iteration. These two hyperparameters define the size of the mini-batches, and the amount of gradient descent steps that are performed each time the policy is updated. Mini-batches allow the algorithm to perform more policy updates using smaller subsets of data. Separating the data into mini-batches can increase the computation time of the algorithm, but it can improve its sample efficiency and it may also prevent it from getting stuck on local optima. Similarly, a higher number of epochs also increases the computation time, but it can help the algorithm fit the data better, although too many epochs can also lead to overfitting.

In summary, for each of these hyperparameters there is a trade-off between different performance metrics. For this reason, a sweep was performed to find a suitable combination of hyperparameters that leads to efficient learning and reasonable computational time. The search space chosen for each of these hyperparameters is shown in Table 5.1. In this table, the expression $\mathcal{U}[a,\ b]$ indicates a continuous uniform distribution between the limits a & b, and the expression $\mathcal{U}\{a_1,\ a_2, \dots\ a_n\}$ represents a discrete uniform distribution wherein each of the values $a_1, \dots\ a_n$ is equally likely to be sampled. This search space was selected to cover at least one order of magnitude around the default values defined by SB3. Originally, batch sizes as low as 8 were also sampled, but this led to prohibitively long computation times, which severely limited the number of runs that could be performed. Therefore the lower limit of the batch size was set at 32.

Hyperparameter	Default value	Search space
Learning rate	3×10^{-4}	$\exp\left(\mathcal{U}\left[\ln(10^{-6}),\ \ln(10^{-2})\right]\right)$
Clip range	0.2	$\mathcal{U}\{0.02,\ 0.1,\ 0.15,\ 0.2,\ 0.25,\ 0.3,\ 0.5,\ 1,\ 2\}$
Batch size	64	$\mathcal{U}\{32,\ 64,\ 128,\ 512\}$
Number of epochs	10	$\mathcal{U}\{1,\ 2,\ 3,\ \dots,\ 100\}$

Table 5.1: Search space of the algorithm sweep.

Initially, a uniform distribution was chosen for the learning rate, but this distribution did not properly explore the full range of values, especially on the lower end, as shown in Figure 5.1. Note that the scale of the y-axis on this figure is logarithmic. Despite having set the search space between 10^{-6} and 10^{-2}, the uniform distribution rarely samples a value lower than 10^{-4} in 1000 samples. To avoid this narrowed sampling, a log-uniform distribution was chosen (also known as a reciprocal distribution). The log-uniform distribution is characterized by the probability density function (PDF) shown in equation 5.1, which leads to a much more even sampling across all the orders of magnitude of the search space, as depicted on the rightmost plot in Figure 5.1. This kind of distribution is better for sampling search spaces that cover multiple orders of magnitude.

$$PDF(x) = \frac{1}{x\left[\ln(b) - \ln(a)\right]} \quad \text{for} \quad a <= x <= b \quad \text{and} \quad a > 0 \tag{5.1}$$

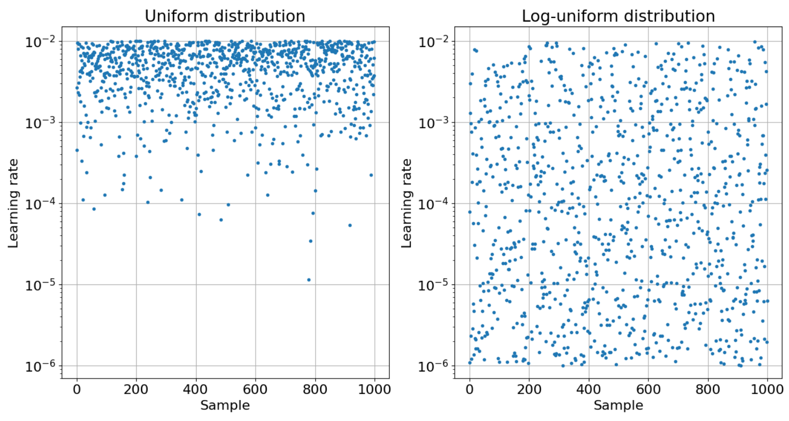


Figure 5.1: Result of sampling the learning rate with a uniform distribution (left), and a log-uniform distribution (right).

On each run of the sweep, the learning algorithm trained the policy for 100,000 time steps using randomly sampled hyperparameters. The results of the sweep can be seen in the two parallel coordinates plots in Figure 5.2. These plots depict the hyperparameters and the reward achieved on every run. The four vertical axes on the left side of the plot show the values of the hyperparameters that were randomly sampled, and the axis on the right shows the maximum reward achieved during each run. Each line in the plot represents one of the runs executed during the sweep. Lines are color-coded according to the reward of the run, where blue indicates a low reward, and yellow indicates a high reward.

The plot on the top of Figure 5.2 shows the results for all of the 110 runs that were executed during the sweep. The majority of the runs achieved low rewards, as can be seen by the predominantly dark blue color of the lines. The low reward indicates that the trained policies were not able to steer the chaser into the entry corridor due to the poor performance of the learning algorithm.

Some of the runs did achieve higher rewards. These runs can be identified by the pink and yellow lines, which have been highlighted in the lower plot of Figure 5.2. The distribution of these lines shows the range of values that are suitable for each hyperparameter. For the batch size, all of the sampled values were capable of achieving at least one run with high rewards, suggesting that it has no significant effect on the learning process. On the other hand, the clipping parameter has a very limited range of values that lead to high rewards. Specifically, the only runs that achieved high rewards had a clipping parameter between 0.2 and 0.3. This result agrees with the findings of the authors of the PPO algorithm [22], who concluded that PPO achieves the best results with a clipping parameter of 0.2, while larger clipping values lead to excessively large policy updates.

A similar pattern appears for the learning rate. The highest rewards are achieved by runs that have a learning rate between 7×10^{-4} and 2×10^{-3}, while the runs with a learning rate closer to the default value of 3×10^{-4} generated lower rewards. This suggests that for this scenario the algorithm benefits from a larger learning rate to make the policy learn faster. Lastly, the number of epochs also appears to have an effect on the reward, since the highest rewards correspond to runs with a number of epochs between 33 and 85. Notably, the runs with the default number of epochs never achieved high rewards, indicating that 10 epochs may not be sufficient for the algorithm to properly fit the data, so a larger amount of epochs should be used to achieve better results.

Another performance metric to consider is the runtime of the training process. Although the batch size does not have a significant effect on the reward, it does have a noticeable effect on the runtime of the sweep. This is evidenced by the fact that the average runtime of the runs increases as the batch size decreases. For instance, the runs with a batch size of 512 have an average runtime of 12 minutes, while the runs with a batch size of 32 have an average runtime of 23 minutes. This was to be expected, because the batch size affects how many policy updates are performed. As explained in section 4.1, one policy update has to be executed for each mini-batch. Thus, a smaller batch size leads to more mini-batches and more policy iterations, and therefore takes more time to execute. Similarly, the number of epochs also affects the runtime because a higher number of epochs increases the number of policy updates. Therefore, it is best to avoid using a low batch size and a high number of epochs, as it has no significant effect on the reward but it does slow down the learning algorithm considerably. Based on the results of this sweep, the hyperparameters shown in Table 5.2 were chosen for the learning algorithm. The learning rate and the clip range were chosen to achieve a high reward, while the batch size and the number of epochs were chosen to avoid unnecessarily long training times.

Hyperparameter	Default value	Tuned value
Learning rate	3×10^{-4}	3×10^{-3}
Clip range	0.2	0.25
Batch size	64	128
Number of epochs	10	40

Table 5.2: Hyperparameter values chosen as a result of the algorithm sweep.

5.2. Neural network tuning

The next component to be tuned was the architecture of the neural network policies, which can have a significant effect on the learning process and on the performance of the trained policy. Just as with the learning algorithm, the default policies on SB3 have been extensively tested, and thus have a fairly reliable architecture. However, the optimal architecture for a neural network is not the same for every scenario, as it often depends on the dimensionality of the problem. Therefore, the policies were tuned to ensure that their architecture is suitable for this rendezvous scenario. The following sections describe the process and the results of tuning the feedforward and the recurrent policies.

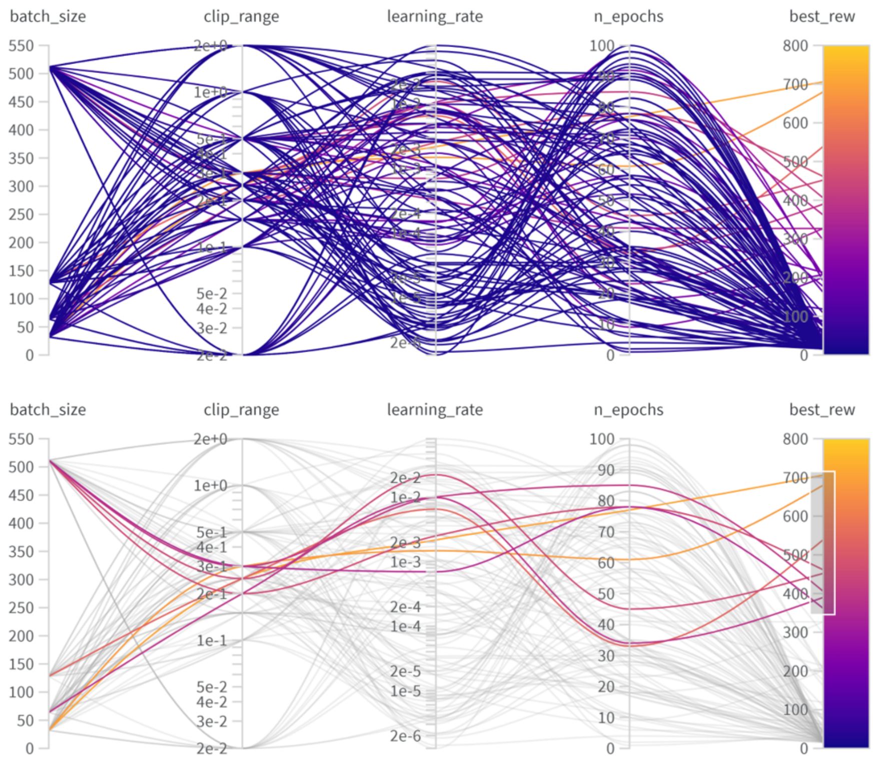


Figure 5.2: Results of algorithm sweep (top), with the best results highlighted (bottom).

5.2.1. MLP tuning

For the non-recurrent policy, three hyperparameters were tuned: the number of hidden layers, the number of neurons per layer, and the activation functions. The three activation functions that were tested during this sweep are the rectified linear unit (ReLU), the sigmoid (σ), and the hyperbolic tangent (Tanh) functions. The formulas for these nonlinear functions are given in equations 5.2-5.4, and their plots can be seen in Figure 5.3. ReLU is a piece-wise linear function with an output equal to zero when the input is negative, and equal to the input when the input is positive. Thus, the ReLU function has no upper limit. On the other hand, the output of the sigmoid function is bounded between 0 and 1. For large negative inputs, its output asymptotically approaches 0, and for large positive values it asymptotically approaches 1. The Tanh function is similar to the sigmoid, but its lower bound is -1 instead of 0. Some works claim that the ReLU function is more successful than other activation functions for general purposes [63], while the sigmoid is preferred for binary classification problems. However, there is no general agreement on which activation function is universally better than the others. For this reason, all three of these activation functions were tested during the tuning process.

$$ReLU(x) = \begin{cases} 0 & \text{if} \quad x < 0 \\ x & \text{if} \quad x \leq 0 \end{cases} \tag{5.2}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5.3}$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{5.4}$$

The optimal amount of hidden layers and neurons in the neural network depends on the number of dimensions of the problem. Several sources [64, 65] claim that the optimal number of neurons per layer is somewhere between the size of the input and the size of the output of the neural network. In this project, the input observation contains 17 values, and the output action has 6 values. Hence, 16 neurons per layer should be sufficient for this scenario, but higher values were also tested for comparison.

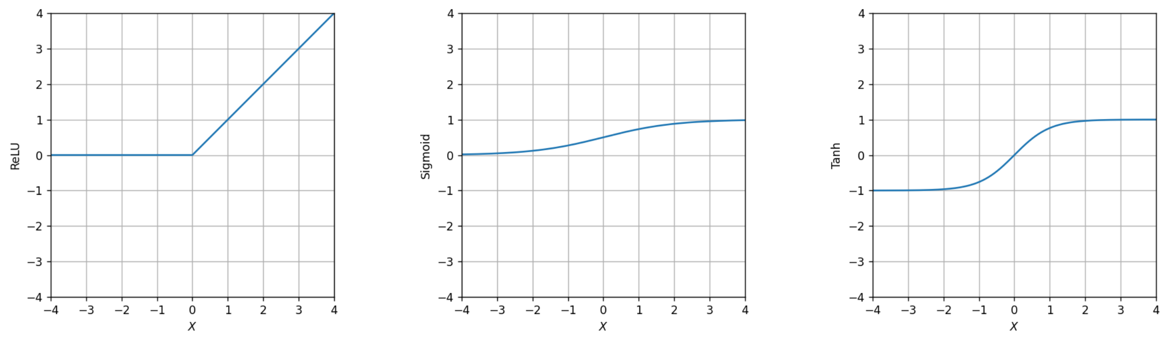


Figure 5.3: Activation functions tested. ReLU (left), Sigmoid (center) and Tanh (right).

In theory, a neural network requires only one hidden layer to approximate any continuous function. In practice, however, many researchers use deeper networks [28, 29, 48]. Therefore, this study also included deeper networks. Table 5.3 lists the values that were sampled for each of the three hyperparameters mentioned. A sweep was executed using a grid search strategy so that each of the possible combinations of hyperparameters was tested. On each run of the sweep, a policy was created using the hyperparameter values sampled from the search space. The policy was then trained for 100,000 time steps, and the maximum reward was recorded.

Hyperparameter	Default value	Search space
Number of layers	2	{2, 3, 4}
Neurons per layer	32	{16, 32, 64}
Activation function	Tanh	{ReLU, σ, Tanh}

Table 5.3: Search space of the non-recurrent network sweep.

In total, 27 runs were executed. The maximum reward obtained on each of these runs can be seen in Figure 5.4. The plot on the left shows the reward achieved by the policies that used a ReLU activation function, and the plots in the center and on the right show the results for the Sigmoid and Tanh functions, respectively. Of the three activation functions, Tanh clearly displayed the best performance, as it achieved much higher rewards than both of the other activation functions. On the other hand, the sigmoid function showed very poor results, as seen by the low rewards in all of its nine runs. One possible explanation for these poor results is the phenomenon known as saturation [66]. This issue arises due to the asymptotic nature of the sigmoid function, which can reduce its output to a binary state when the input is large (in either the positive or negative directions). Updating the input parameters of a saturated neuron will have a very small effect on its output, and therefore the gradient of the loss function will be small as well. As a result, the parameters of the network will be updated more slowly, because the size of the update is proportional to the size of the gradient. Hence, the learning process will be slower, and it will take more time for the algorithm to converge. Another possible explanation for the poor results of the sigmoid function is the vanishing gradient problem. This problem refers to how the gradients of the loss function become increasingly smaller in the early layers of the network as a result of the repeated multiplication of small gradients that is performed by the backpropagation algorithm. Because of the shrinking gradients, the early layers of the network cannot be properly updated, thus preventing the network from learning a suitable control policy. The sigmoid function is especially susceptible to this problem due to its asymptotic nature, which tends to result in smaller gradients when the input is not near zero.

The Tanh function has a similar shape to the sigmoid function, so in theory it should suffer from similar problems. However, the Tanh function consistently achieved higher rewards than the sigmoid function. One possible explanation for this outcome is that the Tanh function has a larger derivative, which results in larger gradients and may have helped to prevent saturation and vanishing gradients. Another possible reason for the difference in performance is that the output of the Tanh function matches the output range of the environment's action space ($[-1, 1]$), whereas the output of the ReLU and

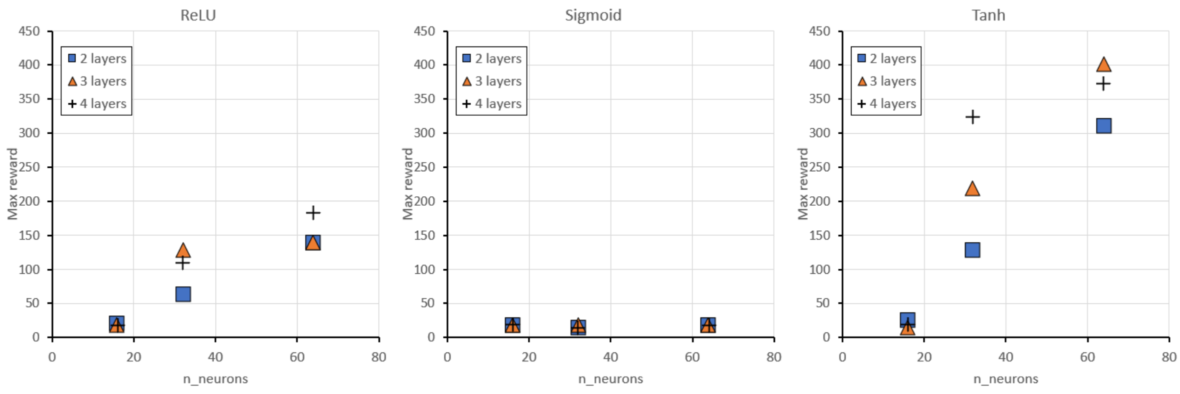


Figure 5.4: Results of the network sweep for ReLU (left), Sigmoid (center) and Tanh (right).

Sigmoid functions needs to be normalized to match the action space. It is also likely that the Tanh function is capable of learning more complex relationships between inputs and outputs thanks to its ability to output both positive and negative values. The other two activation functions can only output positive values, which may limit the network's ability to learn features from the given data.

The plots in Figure 5.4 also show that the number of neurons per layer has an effect on the policy's performance. Rewards were always low during the runs that used only 16 neurons per layer, suggesting that 16 neurons are not enough for the network to learn a suitable control policy. On the other hand, the runs that used 32 and 64 neurons per layer achieved much better rewards, especially when using the ReLU and Tanh functions. The number of layers does not have such a significant effect. This agrees with the universal approximation theorem, which states that one hidden layer should be sufficient to enable a neural network to approximate any continuous function.

5.2.2. LSTM tuning

Another sweep was performed to tune the recurrent policy, which has three hyperparameters: the size of the hidden state, the number of LSTM layers, and the sharing of the LSTM. The size of the hidden state refers to the size of the output of the LSTM. A larger hidden state can capture more information from the sequence of inputs fed to the LSTM, but it can also lead to overfitting. Ultimately, the optimal size of the hidden state depends on the dimensions of the problem, therefore several different sizes were tested during the sweep. Next, the number of LSTM layers indicates how many consecutive LSTM units are included in the policy. For example, if the number of LSTM layers is set to two, then the network will contain two LSTM units stacked sequentially one after the other, such that the output of the first LSTM is the input to the second LSTM. In theory, one LSTM layer should be sufficient for the policy to remember information from previous inputs, but more layers were also tested to see if it led to any changes in performance. Lastly, sharing the LSTM means that both the actor and the critic networks use the same LSTM. Sharing the LSTM reduces the total amount of parameters in the network, which can help to lower the computational time and may prevent overfitting. It may also be beneficial for the actor and the critic to share information. However, when the LSTM is shared, its parameters are only updated with the actor losses. This may lead the actor to dominate the LSTM, reducing the effectiveness of the critic. Both shared and separate LSTM configurations were tested during the sweep. The full search space of the sweep can be seen in Table 5.4.

Hyperparameter	Default value	Search space
Size of hidden layer	128	{32, 64, 128, 256}
LSTM layers	1	{1, 2, 3}
Shared LSTM	No	{Yes, No}

Table 5.4: Search space of the recurrent policy sweep.

Just as with the non-recurrent policy, a sweep was performed with a grid search method to check every combination of hyperparameters. Figure 5.5 shows the results of the sweep in terms of the reward achieved by each run and the time it took to complete each run. The plot on the left shows the runtime of the experiments expressed as a percentage of the average time it took to run the same experiment with a non-recurrent policy. It can be seen that all of the recurrent policies are slower to train than the non-recurrent policy. This difference was expected, since the non-recurrent policy only contains a MLP, whereas the recurrent policies contain at least one LSTM in addition to a MLP. The LSTMs drastically increase the number of parameters in the network, and they also have to perform more sequential operations to compute an output, as explained in section 2.2.3. This explanation is supported by the fact that using more LSTM layers increased the training time even further, as seen in the plot. There is also a clear difference between the policies that shared the LSTM between the actor and the critic and those that did not. The policies that used a separate LSTM were on average 32% slower to train than the MLP. Meanwhile, the policies that shared the LSTM between the actor and the critic were 14% faster than their non-sharing counterparts, but still slower than the MLP policy. Despite being faster to train, the policies with a shared LSTM achieved a lower reward on average than the policies with the separate LSTM, as shown in the rightmost plot of Figure 5.5.

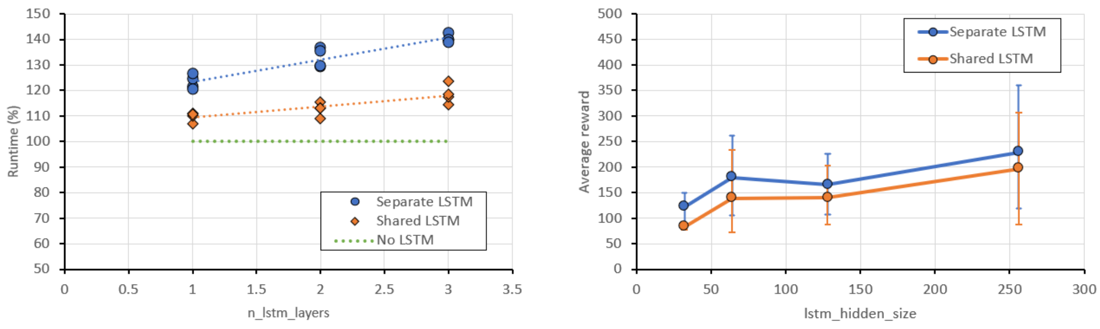


Figure 5.5: Results of the RNN network sweep in terms of runtime (left) and average reward (right).

Having more than one LSTM layer on the policy did not improve the reward. This was expected, since one LSTM layer should be sufficient for the network to remember previous inputs, and adding more LSTM layers cannot provide any added benefit. Contrary to the effect of increasing the number of LSTM layers, increasing the size of the hidden state does not increase the training time significantly, but it does appear to improve the reward by a small amount. These results indicate that in this case it is best to use only one LSTM layer to reduce the computational time, and also use a large hidden state without sharing the LSTM to achieve a higher reward.

Based on the outcome of the recurrent and the non-recurrent sweeps, the tuned values for the hyperparameters of the policies were selected to maximize the reward while keeping a low runtime. The tuned value for each hyperparameter can be seen in Table 5.5. Overall, the tuned values are similar to the default values, but there were still some unexpected results while tuning the networks. The Tanh activation function performed significantly better than the other two alternatives, and the size of the feedforward layers and the hidden state had to be increased to achieve higher rewards. If the hyperparameters had been left as is, the resulting policy would have most likely yielded lower rewards. This shows the importance of tuning the model before committing to training the policy.

Hyperparameter	Default value	Tuned value
Number of (feedforward) layers	2	3
Neurons per (feedforward) layer	32	64
Activation function	Tanh	Tanh
Size of hidden state	128	256
Number of LSTM layers	1	1
Shared LSTM	No	No

Table 5.5: Hyperparameter values chosen as a result of the network sweeps.

5.3. Reward function tuning

The last component of the model that required tuning is the reward function. During the training process, the learning algorithm will update the policy to try to maximize the cumulative reward. Therefore, the reward function essentially defines what the goal of the scenario is, and what the behavior of the trained policy will be like. As described in Section 4.3, the reward function is composed of multiple terms that reward the desired behaviors and penalize unwanted behaviors. The reward function was tuned by varying the coefficients (k) of each component of the function until a suitable behavior was observed.

The first parameter to be tuned was the collision coefficient (k_c), which penalizes the policy for causing collisions. A sweep was executed to assess how different collision coefficients affect the performance of the policy. The search space of the sweep can be seen in Table 5.6. On each run, a policy was trained using a different value for the collision coefficient, ranging between zero and one. In the case where the collision coefficient is equal to zero, the policy is not penalized for collisions, and in the other four cases the policy receives a penalty equal to k_c on every time step that the chaser is inside the keep-out zone. To keep the reward function simple during this sweep, the rest of the coefficients were set to zero, except for the success coefficient, so that the policy still has a goal to accomplish.

Hyperparameter	Search space
Collision coefficient (k_c)	$\{0,\ 0.1,\ 0.25,\ 0.5,\ 1\}$
Fuel coefficient (k_f)	0
Attitude coefficient (k_θ)	0
Success coefficient (k_*)	1

Table 5.6: Search space of the collision coefficient sweep.

It was expected that many collisions would occur when the collision coefficient was equal to zero, because the policy does not receive any penalties for entering the keep-out zone. It was also expected that a high collision coefficient would prevent the chaser from reaching the entry corridor, because the policy would be too intent on avoiding collisions. However, the results of the sweep showed a different outcome. In each of the runs, the trained policy was able to successfully enter the corridor without causing any collisions, even when there was no collision penalty. The most likely explanation for this behavior is the fact that the success bonus (R_*) can only be achieved when the chaser has not entered the keep-out zone during the current episode. As mentioned in section 4.3, this constraint was added to the success bonus to provide additional motivation for the policy to avoid collisions. Based on the results, the constraint appears to have the desired effect, because the policies learned to avoid collisions in order to receive the success bonus, even when the collisions are not explicitly penalized.

Next, the fuel efficiency component (R_f) was added to the reward function. As described in section 4.3, the purpose of the R_f component is to encourage the policy to use less ΔV throughout the trajectory, thereby conserving fuel. To this end, the R_f component gives the policy a reward that is inversely proportional to the ΔV used at every time step. In other words, the policy receives higher rewards for using less ΔV, and vice-versa. The size of this reward is determined by the fuel coefficient k_f, as shown in equation 4.4. If the coefficient is set to a low value then the reward R_f will also be low, so the policy will have little motivation to generate fuel efficient trajectories. On the other hand, when the fuel coefficient is high, the policy will receive a large reward for using less ΔV. This may lead the policy to be more fuel-efficient, but it may also distract it from its ultimate objective of reaching the desired terminal position. Other authors that used similar reward functions reported that varying the fuel coefficient caused a trade-off between fuel efficiency and other performance parameters [21], so a similar behavior is expected in this test.

During this experiment, several policies were trained using different values for the fuel coefficient. In total, six training runs were executed with fuel coefficients ranging between zero and one, and the performance of the policy was evaluated at the end of each run. The results for these runs can be seen in Figure 5.6. The bar chart on the left shows the total ΔV of the trajectory generated by each of the six trained policies, and the bar chart on the right shows the terminal position error of those same trajectories (prior to any collisions). The blue bars indicate the trajectories that did not collide with the keep-out zone, while the red bars indicate the trajectories that did.

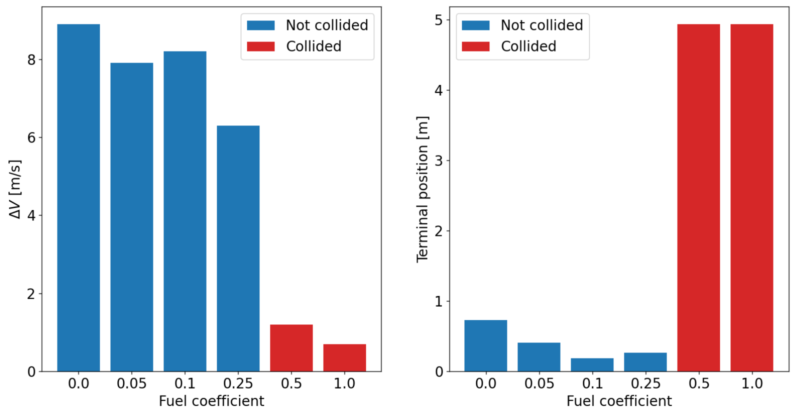


Figure 5.6: Results of using different fuel coefficients.

As expected, the highest ΔV corresponds to the policy that had a fuel coefficient of zero. This policy did not receive any reward from the R_f component, and therefore it did not have any motivation to learn fuel-efficient trajectories. The rest of the policies were trained with non-zero fuel coefficients, and as a result they generated trajectories with a lower total ΔV. As the fuel coefficient increases, the ΔV of the trajectories decreases, suggesting that the policies learned to use less ΔV to increase the amount of reward obtained throughout the episode. However, making the fuel coefficient too large can be detrimental to the policy's performance. This is evidenced by the results of the two policies that had the highest fuel coefficients during training. These two policies learned to execute trajectories with a very low ΔV, but they did not learn to approach the target without colliding with the keep-out zone, as depicted by their large terminal position error. The terminal position error of these two policies was equal to the radius of the keep-out zone, indicating that the policies were not even able to enter the corridor without colliding with the keep-out zone, while the other four policies were able to reduce the position error to less than one meter. This result proves that there is indeed a trade-off between the different components of the reward function, and one of the terms may dominate over the rest if the coefficients are not tuned carefully. Using a low fuel coefficient makes the policy prioritize collision avoidance over saving fuel, whereas using a high fuel coefficient does the opposite. For any given on-orbit servicing mission it is desirable to have fuel-efficient trajectories, but not at the cost of higher collision risks. Based on this requirement, a fuel coefficient of 0.25 was chosen for the final reward function. This value strikes a good balance between conserving fuel while still avoiding collisions when approaching the target.

The last components of the reward function that need tuning are the attitude term (R_θ), which rewards the policy for pointing at the target, and the success term (R_*) which rewards the policy when it successfully achieves the terminal constraints without colliding. The attitude term R_θ is significant because the chaser must remain pointed towards the target throughout the duration of the trajectory in order to have accurate knowledge of its state relative to the target. However, the R_θ reward term must not outweigh the R_* term, as this would distract the policy from its ultimate objective of achieving the terminal conditions of the trajectory. To find a good combination, the coefficients of the two reward terms (k_θ and k_*) were varied over the search space shown in Table 5.7. A grid search was performed over this search space, training one policy with each combination of parameters.

Hyperparameter	Search space
Collision coefficient (k_c)	0.5
Fuel coefficient (k_f)	0.25
Attitude coefficient (k_θ)	$\{0,\ 1,\ 2,\ 4\}$
Success coefficient (k_*)	$\{0,\ 2,\ 4,\ 8\}$

Table 5.7: Search space of the attitude and success coefficients sweep.

In total, 16 policies were trained. Their performance was evaluated in terms of the final position error at the end of the trajectory, and the average attitude error throughout the trajectory. The results for the 16 policies can be seen in Figure 5.7, where the x-axis displays the position error, the y-axis displays the attitude error, and the dashed lines indicate the terminal constraints required for a successful trajectory. From this graph it can be seen that the performance of the policies varied greatly. Some of the policies achieved large position errors and small attitude errors, while others did the opposite. The policies with large position errors correspond to the reward functions that had a large attitude coefficient (k_θ). As expected, in these cases the weight of the attitude reward term R_θ prevented the policy from approaching the target, as it was more focused on maintaining a small attitude error. Many of the policies fulfilled the required position constraint, but only two of them fulfilled both the position and the attitude constraints. The best performance was achieved by the reward function with an attitude coefficient of 1 and a success coefficient of 8, so these values were chosen for the tuned reward function. The tuned values for all of the coefficients can be seen in Table 5.8. With this, the model tuning process was concluded.

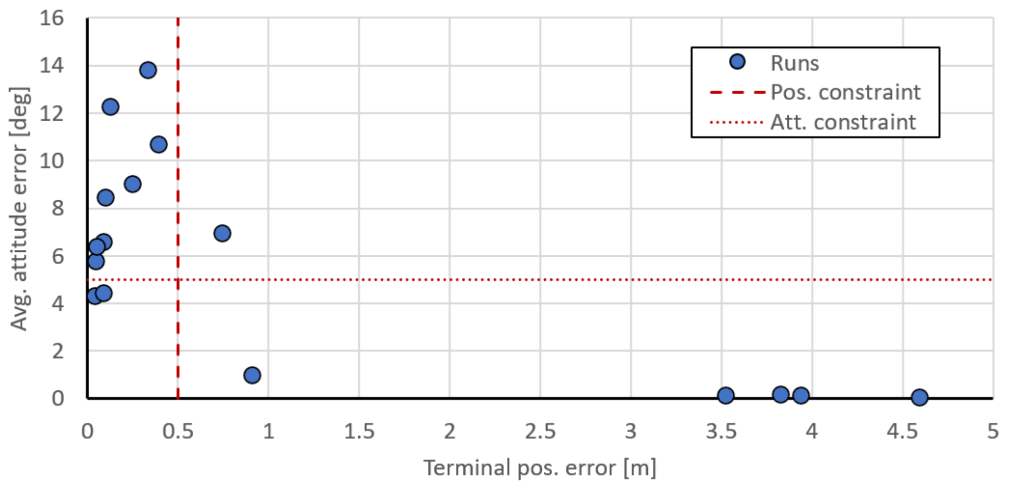


Figure 5.7: Results of using different fuel coefficients.

Hyperparameter	Tuned value
Collision coefficient (k_c)	0.5
Fuel coefficient (k_f)	0.25
Attitude coefficient (k_θ)	1
Success coefficient (k_*)	8

Table 5.8: Parameter values chosen as a result of the reward sweeps.

5.4. Tuning summary

Three components of the model were tuned: the hyperparameters of the learning algorithm, the architecture of the neural network policies, and the coefficients of the reward function. Although W&B made it easy to automate the experiments, tuning the model's components was still a lengthy process, since it largely relies on a trial-and-error methodology where the learning algorithm is executed multiple times with different combinations of hyperparameters. Given enough trials, a suitable set of hyperparameters can be identified based on the performance of the learning algorithm.

The first component to be tuned was the learning algorithm. The purpose of tuning the algorithm is to find a suitable combination of hyperparameters that will lead to a more efficient and stable training process. First, the relevant hyperparameters were identified using prior knowledge of the algorithm as well as SB3's documentation pages. Then, a wide search space was designated for each hyperparameter, centered around the default values defined by SB3. The search space was centered around the default values because these values are often fairly well suited for a wide range of problems, so they provide a good starting point for the search. A wide search space was chosen so that the search covers a broad range of possible hyperparameter combinations. If necessary, the search space can be later narrowed to ignore certain ranges that yield poor results. Once the search space was defined, a sweep was performed, randomly selecting different combinations of hyperparameters. A random search was chosen instead of a grid search because a grid search would have taken too long to cover the entire search space. The results of the sweep showed the effects that the different hyperparameters have on the training process. The clip range and the learning rate had a significant effect on the algorithm's ability to achieve high rewards, while the batch size and the number of epochs mostly affected the total runtime of the training. Based on the results, suitable hyperparameter values were then selected to maximize the amount of reward achieved, while also avoiding long training times. The tuning procedure for the neural networks was performed in a similar manner, sweeping over a search space to find a network architecture that is adequate for the scenario. The results showed that the choice of activation function can have a large effect on the network's ability to learn, and the Tanh function performs significantly better than the other alternatives. Furthermore, the size of the network can also affect the network's performance, so it is best to check several different sizes to see which architecture is best for the dimensionality of the given scenario.

Lastly, the reward function was tuned. This process is very dependent on the format chosen for the reward function. In this case, the reward function was designed as a composite function. This was done by first identifying the behaviors or goals that the policy has to achieve throughout the trajectory, and then representing each of those goals as an element in the reward function. The composite reward format yields a shaped reward that gradually guides the policy towards its objective, which makes it learn faster than it would with a sparse reward format. Some domain knowledge was also incorporated to terminate episodes early if the policy deviates too far from its objective. Each of the elements in the composite reward function was given a coefficient. The purpose of tuning the reward function was to find suitable values for these coefficients so that the policy learns to follow the objectives represented by the reward function. Initially, the coefficients were set to zero, and then they were gradually introduced to modify the behavior of the policy. This process involved some trial and error because the policy may ignore some reward terms if the coefficients are not chosen adequately. As a general rule, the coefficient of each reward component should be commensurate with the relative importance of the corresponding goal. For example, during an on-orbit servicing mission it is more important to prevent collisions than to perform a fuel-optimal trajectory, therefore the reward coefficient that is responsible for avoiding collisions should be larger. It is also beneficial to keep the overall reward positive rather than negative to prevent the policy from purposefully trying to terminate the episodes early. The tuned reward function was able to accomplish the goals of the scenario reasonably well.

6 Sensitivity analysis

After tuning the hyperparameters of the model, a sensitivity analysis was performed. The goal of this sensitivity analysis was to evaluate the learning algorithm's ability to train a policy in different rendezvous scenarios. Ideally, the tuned algorithm should be suitable for a wide variety of scenarios, but it is possible that the tuning process may have optimized the algorithm's performance for only a small set of initial conditions. If so, the applicability of the learning algorithm would be quite limited, and it would have to be tuned anew for different scenarios. Hence the need for this sensitivity analysis.

The analysis was performed by executing the learning algorithm several times using different initial conditions during each training. Three different parameters were varied: the orbit altitude, the initial distance to the target, and the rotation rate of the target. Sections 6.1, 6.2, and 6.3 present the results for each of the three sensitivity studies.

6.1. Sensitivity to orbit altitude

The first parameter to be varied was the altitude of the orbit in which the chaser and the target are located. This parameter is relevant because it affects the dynamics of the scenario. As described in section 2.1.2, the CW model dictates that the motion of the chaser relative to the target depends on the mean motion of the orbit (n), which itself is a function of the orbit altitude. Assuming that the chaser and the target are orbiting Earth in a circular orbit, the relationship between the mean motion and the orbit altitude is defined as shown in equation 6.1 where μ is the gravitational parameter of Earth, R_E is the radius of Earth, and h is the altitude of the orbit.

$$n = \sqrt{\frac{\mu_E}{(R_E + h)^3}} \tag{6.1}$$

Five tests were performed to check the performance of the learning algorithm at different altitudes. On each of these five tests a different altitude was used, and the algorithm was executed for $400,000$ time steps. The five altitudes that were tested are 400 km, 600 km, 800 km, 1000 km, and 2000 km. This range of altitudes was chosen to assess if the learning algorithm can produce good results at any altitude in low Earth orbit, where most space debris is located. The results of the five tests can be seen in Figure 6.1, where the graph shows the learning process throughout each training in terms of the terminal position error of the trajectories. It can be seen that in each case the algorithm is capable of teaching a policy to approach the desired terminal position, reducing the position error from 12 meters at the start of training to less than 1 meter at the end of training. Furthermore, the similarity of the learning curves shows that the learning process is not affected by the different orbit altitudes. After approximately 70,000 time steps all of the runs have managed to reduce the terminal error to less than 1 meter, and by the end of the training process all of the five policies can achieve a terminal position error of 20 cm or less. Other performance parameters such as cumulative reward and total runtime were also unaffected by the change in orbit altitude.

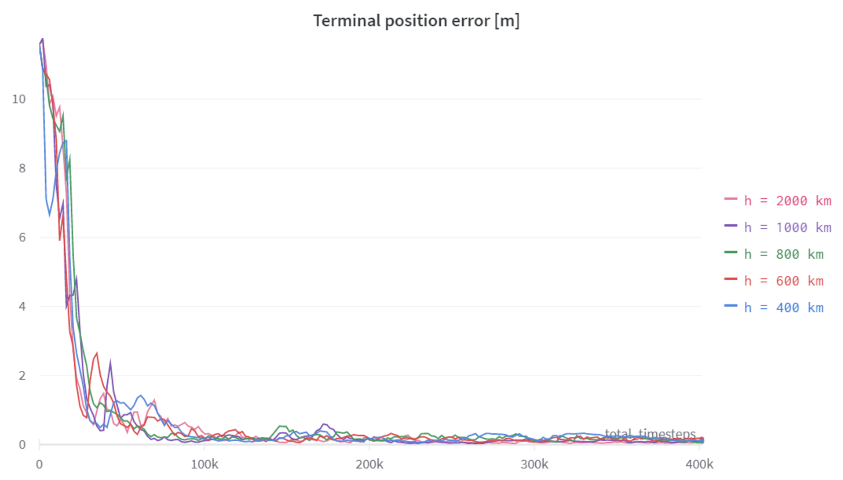


Figure 6.1: Terminal position error improvement during training.

This result proves that the learning algorithm can be used for final approach trajectories at any altitude in low Earth orbit. However, it should be remembered that the final approach trajectories tested here only have a duration of 60 seconds. During this short period of time, the different orbit altitudes will not have a large effect on the relative motion of the vehicles. If the trajectories were longer, then the orbit altitude would play a larger role in the dynamics of the system, and therefore could have an impact on the algorithm's performance.

6.2. Sensitivity to target distance

The next parameter to be tested was the initial distance between the chaser and the target. It is relevant to test different values for this parameter because different rendezvous missions have different definitions for where the final approach phase begins. Fehse [39] describes the Soyuz beginning its final approach to the international space station at a distance of approximately 150 meters, while Li et al [13] proposed a mission to capture Envisat with the final approach phase commencing at a distance of 15 meters. Thus, another sensitivity study was performed to evaluate how the initial distance between the chaser and the target affects the learning process of the algorithm. In this study, the initial position of the chaser $(\mathbf{r}_0)$ was varied between 10 and 50 meters along the negative direction of the y-axis of the LVLH frame. In addition to the initial position, the size of the target was also varied in this study. This was included because the target of a debris removal mission may have a wide range of sizes. The intended target could be anything as small as a CubeSat, or a larger body such as the aforementioned Envisat. Hence, different target sizes were represented by using three different radii for the keep-out zone (r_{KOZ}). The search spaces for the distance and size of the target are shown in Table 6.1. A grid search was performed, checking each of the 15 combinations of parameter values. In each of the 15 runs, a policy was trained for $400,000$ time steps, and the performance of each policy was evaluated at the end of the training.

Parameter	Search space
$\lvert\mathbf{r}_0\rvert$ [m]	{10, 20, 30, 40, 50}
r_{KOZ} [m]	{2, 5, 10}

Table 6.1: Parameters of the target size sensitivity analysis.

Of the 15 policies trained in this experiment, all of them learned to avoid collisions. It was expected that perhaps the larger keep-out zones would have been harder to avoid, but this was not the case, as all of the trained policies achieved collision free-trajectories regardless of the size of the target and the initial distance. Other performance parameters did get affected by the distance to the target, such as the total ΔV and the cumulative reward. For instance, the ΔV was larger when the initial distance to the target was larger. This was to be expected, because the chaser must spend more fuel to approach a target that is further away. Likewise, the cumulative reward will also be smaller when the initial distance to the target is larger, because the chaser will require more time to reach the desired terminal conditions of the trajectory, and hence it will accumulate less reward throughout the episode. Despite these differences in reward, almost all of the trained control policies were capable of successfully performing the final approach. As shown in the bar chart in Figure 6.2, the terminal position error of most of the policies was below the required 0.5 meters, but three of the policies were not able to achieve this and still had a large position error at the end of the trajectory. These three policies were all trained and evaluated in the scenario with the largest keep-out zone, suggesting that the learning algorithm is less effective when the target is larger.

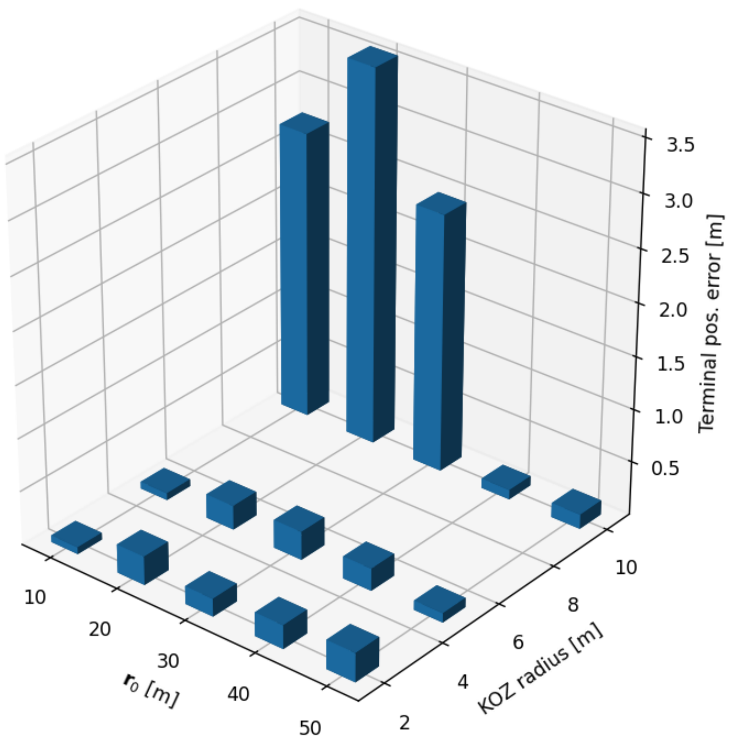


Figure 6.2: Terminal position error for each run.

Notably, the terminal position error of the three unsuccessful policies is smaller than the radius of the keep-out zone, meaning that the policies did manage to enter the corridor, but they were not able to find the terminal position required for a successful trajectory. The most likely reason for this partial success is the reduction of the available state space that was explained in section 4.3. As previously mentioned, there is a maximum distance d_{max} that the chaser is allowed to be from the target, and the value of d_{max} shrinks during the episode, encouraging the chaser to move towards the target. However, d_{max} only shrinks until reaching the edge of the keep-out zone. When an episode reaches that point, the chaser no longer has any explicit motivation to continue moving towards the desired terminal position. This was not a problem when the target is small, because during training the chaser will continue exploring the remaining space and will eventually find the desired terminal position. But for larger targets, it is possible that the remaining space within the keep-out zone is too large, and therefore the policy struggles to find the terminal position during training. This is what might have happened to the three unsuccessful policies that had to deal with a keep-out zone of 10 meters.

Interestingly, two other policies also had to deal with large keep-out zones, yet they still managed to achieve the terminal conditions despite starting from a distance of 40 and 50 meters away from the target. Perhaps starting from further away was in fact beneficial for these two policies. Since they had to start their trajectories from further away, they could have built up more speed as they approached the target, and their momentum may have driven them deeper into the corridor, helping them find the desired terminal conditions. It is also possible that the success of these two policies is purely due to random exploration, and that training the three unsuccessful policies for longer would have led them to achieve successful trajectories as well. Regardless, this study showed that the learning algorithm can deal fairly well with different distances and target sizes. One change that could be helpful would be to shrink the value of d_{max} even further to continue encouraging the chaser to move towards the target.

6.3. Sensitivity to target rotation rate

The last parameter to be tested was the rotation rate of the target. This parameter is significant because it can add a lot of complexity to the maneuver, since the chaser must rotate along with the target in order to achieve the terminal conditions. Therefore it is much harder to execute the final approach with a rotating target than with a static target. Initially, three policies were trained using different rotation rates for the target: 0 deg/s, 1.25 deg/s, and 2.5 deg/s. Some of the learning curves showed sudden decreases in reward throughout the training process. So the test was repeated five times to see if the same effect occurred again. The resulting learning curves can be seen in Figure 6.3, where the colored lines show the average reward for each rotation rate during training, and the shaded regions show the standard deviation of the samples.

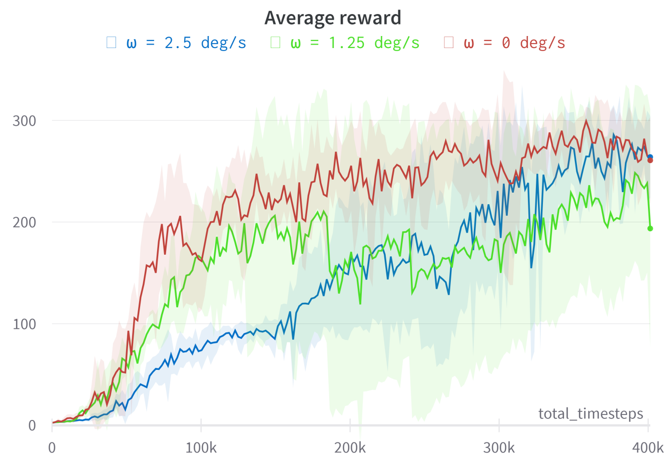


Figure 6.3: Reward achieved during training.

ω_T [deg/s]	Maximum reward
0.00	313 ± 5
1.25	301 ± 46
2.50	300 ± 26

Table 6.2: Maximum reward for different target rotation rates.

In the cases where the target was static, the reward increased rapidly during training, and the results for the five samples were fairly similar, as depicted by the relatively small standard deviation. But when the target was rotating, the results were less consistent. Just as in the initial tests, there were sudden decreases in reward occurring at random times during training. This issue can often happen in deep reinforcement learning methods because small changes in the policy can occasionally lead to a policy that is worse than the previous one, especially if the size of the update is too large. The issue can be aggravated by the format of the reward function. In this rendezvous scenario, any collision with the keep-out zone will prevent the policy from earning any success rewards for the remainder of the episode. Hence, a tiny change in the policy could lead to a collision, which would dramatically decrease the total reward obtained.

Authors of similar studies [14, 67] also encountered dips in their reward curves, particularly around the beginning of the training process when the algorithm is still discovering new areas of the state space, and the dips tend to disappear as the training progresses. Hence, the dips in reward are tolerable as long as they are not permanent and the general trend in the reward is still upward. This was the case in this sensitivity study, where each of the policies still had a fairly decent final performance despite the reward drops. All of the fifteen policies achieved a similar reward at the end of the training, as shown in Table 6.2, and the resulting trajectories were collision-free. The only significant difference between the performance of the three policies was the attitude error, which was smaller when the target was static. A smaller attitude error is normal for that scenario, because the lack of rotation makes it easier for the control policy to point the chaser in the desired direction.

If the dips in reward had persisted for longer periods of time then it could have been helpful to adjust some of the model's hyperparameters such as the learning rate or the clip range in order to limit the size of the policy updates, but this could also lead to a slower learning process. Overall, these sensitivity studies found that the performance of the learning algorithm is mostly unaffected when varying the parameters of the scenario. However, some minor changes to the reward function could be beneficial when dealing with large targets, and some irregular learning curves may occur when the target is rotating.

7 Performance evaluation

This chapter describes how the control policies were trained and evaluated. Section 7.1 shows the set up and the results of the training process for both policies. Then, section 7.2 evaluates the performance of the policies on the nominal scenarios in terms of the terminal state errors, collisions, and fuel usage. Section 7.3 evaluates the same performance metrics for 1000 randomly-generated scenarios. Lastly, section 7.4 verifies that the response time of the policies is quick enough for closed-loop control applications.

7.1. Training

Two neural networks were trained: one using the recurrent policy, and the other using the feedforward policy. Both of the policies were trained using the tuned values for the hyperparameters described in previous chapters. During training, the performance of the policies is periodically evaluated using the callback function described in section 4.1. The callback function is executed before each policy update. It computes the average of the cumulative reward obtained by the policy throughout fifty episodes, where each episode is initialized with random initial conditions. In order to simulate uncertain scenarios, the policies are only given a partial observation of the environment. This partial observation includes all of the state variables mentioned in section 3.1, with the exception of the target rotation rate. Thus, the challenge for the control policies is to infer the target's rotation rate without being able to observe it directly.

To prevent overfitting the policy, every training episode is initialized with randomly sampled initial conditions. The range of the initial conditions for each state variable can be seen in Table 7.1 along with the nominal values. The nominal initial state of the chaser is at a hold-point, located 10 meters away from the target along the y-axis. In the nominal initial state the chaser is pointing at the target, and likewise the target's entry corridor is pointed at the chaser. At the beginning of each episode, the initial conditions are sampled from a random uniform distribution around the nominal value. Varying the initial conditions is helpful to prevent overfitting the networks to a single set of initial conditions, improving the robustness of the policies. Moreover, varying the target rotation rate will show how well the policies deal with uncertain scenarios.

Parameter	Nominal value	Range
r	$10 \quad m$	$[9-11] \quad m$
v	$0 \quad m/s$	$[0-0.1] \quad m/s$
q_C	$0 \quad deg$	$[0-1] \quad deg$
ω_C	$0 \quad deg/s$	$[0-0.1] \quad deg/s$
q_T	$0 \quad deg$	$[0-45] \quad deg$
ω_T	$0 \quad deg/s$	$[0-3] \quad deg/s$

Table 7.1: Scenario initial conditions.

To visualize what the initial state distribution looks like, Figure 7.1 shows 100 samples of the initial position and velocity of the chaser. The location of each arrow indicates the initial position of the chaser, and the direction and color of each arrow indicate the direction and the magnitude of the initial velocity vector, respectively. As shown in the picture, the initial position of the chaser can be anywhere within one meter of the nominal initial position of $-10\ \hat{y}_{LVLH}$, and its initial velocity can be pointed in any direction, but is bounded within the range $[0, 0.1]\ m/s$. Similarly, the initial attitude of the chaser is bounded within one degree of nominal attitude, and so forth, as given by the ranges shown in Table 7.1. It is worth noting that the rotation rate vector of the target $(\boldsymbol{\omega}_T)$ can be pointed in any direction, which makes the learning more challenging since the policies cannot observe the target's rotation rate.

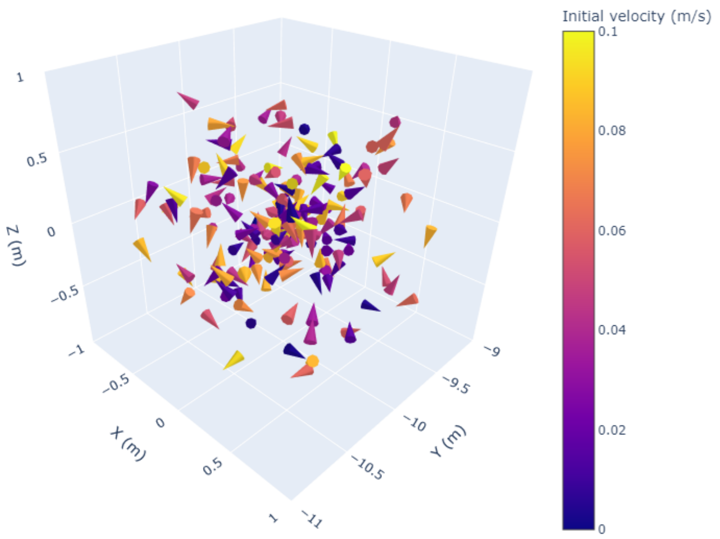


Figure 7.1: Chaser initial position and velocity distribution.

The training process was stopped when the algorithm converged and the policies stopped improving. Figure 7.2 shows the learning curves for the two policies. These curves show the average reward computed by the callback function every time the policy is updated. It can be seen that at the beginning of training, the policies learn very quickly, as indicated by the steep increase in reward. As the training progresses, the learning slows down until the reward eventually stops increasing. Notably, the MLP policy learns quicker at the start of the training, reaching an average reward of 100 after only 200,000 training steps, whereas the RNN policy takes approximately 600,000 steps to reach the same level of reward. However, the RNN policy soon surpasses the MLP policy, eventually reaching an average reward of approximately 250, while the reward of the MLP policy peaks at around 150. This indicates that the RNN policy performed better during the evaluations, at least in terms of reward. Interestingly, both policies reach their peak reward at approximately nine million times steps. After this point, the reward remains fairly constant, indicating that the learning algorithm has converged. The fact that both policies peak after the same amount of time steps shows that the learning algorithm is equally sample efficient for recurrent and non-recurrent policies. After the learning algorithm has converged, the learning curves still display some small variations in reward. This noise in the learning curves is attributed to the random initial conditions sampled during the policy evaluation. When the sampled initial conditions result in ”easier” scenarios, the policy achieves a slightly higher reward, and vice-versa. The average reward is a useful metric for visualizing the learning process, but it is not a good indicator of the overall performance of the policies. In the following sections of this chapter, the performance of the policies will be assessed using more intuitive metrics.

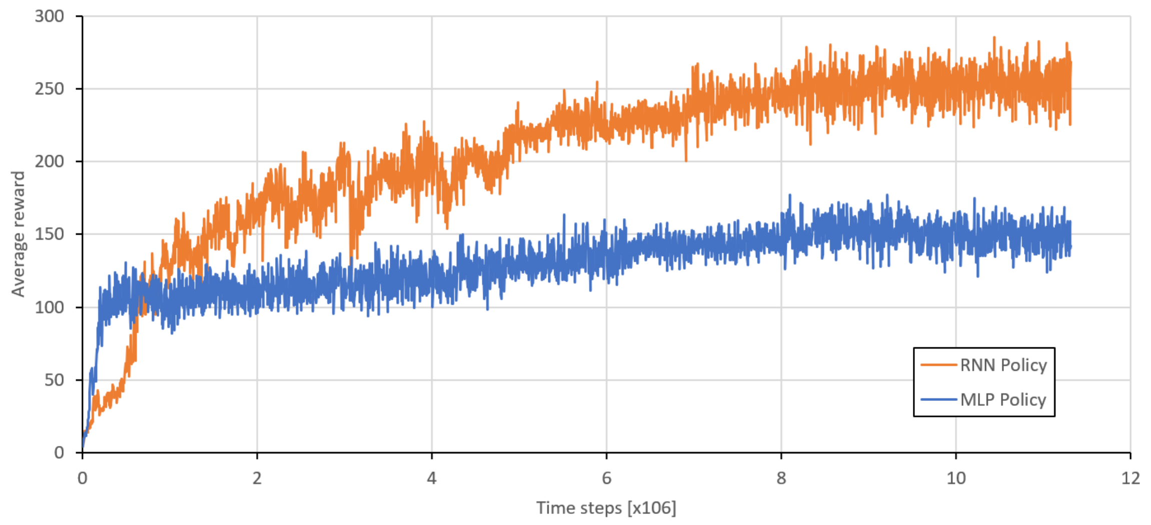


Figure 7.2: Learning curves of the feedforward policy and the recurrent policy.

7.2. Nominal scenario performance

The nominal initial conditions represent the simplest final approach scenario that the control policy may encounter. The chaser is initially static at a hold-point 10 meters away from the target, pointing directly at it. The target is not rotating, so its entry corridor is oriented directly towards the chaser, and it remains in that orientation for the remainder of the episode. In this configuration, the chaser should perform a straight-line approach towards the target, maintaining a constant attitude throughout the trajectory. The following performance metrics are used to assess the resulting trajectories:

- Terminal position error: difference between the desired capture position and the chaser terminal position.
- Terminal velocity error: difference between the desired capture velocity and the chaser terminal velocity.
- Terminal attitude error: difference between the desired capture attitude and the chaser terminal attitude.
- Terminal rotation rate error: difference between the desired capture rotation rate and the chaser terminal rotation rate.
- Success: Whether or not the chaser achieved all of the terminal conditions at once.
- Collisions: Number of time steps during which the chaser was within the keep-out zone.
- Total ΔV: total amount of ΔV used throughout the episode. Acts as a measure of fuel efficiency.

Figure 7.3 shows the closed-loop response of the recurrent and feedforward policies when the scenario is initialized with the nominal conditions. In clockwise order starting from the top left, the plots display the position error, velocity error, rotation rate error, and attitude error of the chaser as a function of time. The terminal constraint required to accomplish a successful trajectory is shown by the dotted line in each plot. It can be seen that the policies have a similar behavior in terms of position and velocity. After approximately 20 seconds both policies achieve the terminal position constraint, and after 25 seconds they achieve the velocity constraint, successfully reaching the desired capture point and remaining there for the rest of the trajectory. However, the response in terms of attitude and rotation rate shows is less stable. Despite having no need to alter their attitude during this trajectory, both policies made the chaser rotate, reaching a maximum rotation rate of approximately 3 deg/s. The recurrent policy was capable of correcting its attitude error and eventually achieved the four required terminal conditions for a successful trajectory, whereas the feedforward policy was not able to achieve the rotation rate constraint. This result shows that the feedforward policy will struggle to perform a final approach without explicitly knowing the rotation rate of the target, even when the rotation rate is zero. Meanwhile, the recurrent policy can infer the target's rotation rate from the sequence of inputs it receives throughout the episode.

One explanation for the initially large rotation rate error of the recurrent policy is that it does not yet know the target's rotation rate at the beginning of the episode. The LSTM layer needs data from several time steps to deduce the target's rotation rate, and only then can it correct the chaser's rotation state. Another possible explanation for the large error in rotation rate is that the reward function only penalized the policy for the applied net force, and not for the applied net torque on the chaser. The rationale behind this design choice was that it is more important to limit the amount of net force applied because it is assumed to be produced by chemical thrusters which require fuel to operate. Meanwhile, the net torque is produced by reaction wheels. Hence, reducing the net torque does not reduce the amount of fuel used. But not penalizing the net torque may have indirectly led the policies to apply torque needlessly, causing larger attitude and rotation rate errors during the trajectory.

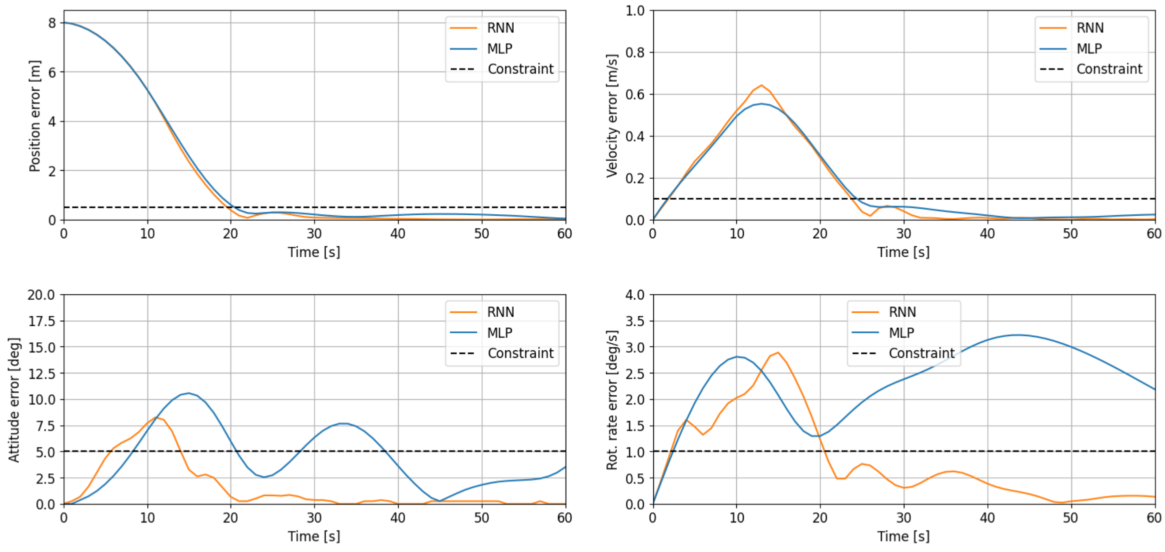


Figure 7.3: Closed-loop response of the policies.

Table 7.2 shows the performance metrics of the two trajectories. The average errors (r_e, v_e, θ_e & ω_e) are computed starting from the time step when the position and velocity constraints are achieved until the end of the trajectory. It is clear to see from the data in the table that the recurrent policy performed better than the feedforward policy across almost all metrics. One notable exception is the total amount of ΔV used during the trajectory. In this scenario, the feedforward policy used 30% less ΔV than the recurrent policy. This was unexpected, since in theory the recurrent network has better knowledge about the state of the environment than the feedforward policy does, and this should enable it to approach the target more efficiently. A likely cause for this difference in fuel efficiency is that the recurrent policy has experienced more success during training, and has therefore learned that it can get a higher reward from the success term (R_*) of the reward function than from the fuel efficiency term (R_f). This causes the recurrent policy to ignore the fuel penalty in order to arrive at the terminal conditions faster and get higher rewards. The feedforward policy does not suffer as much from this effect because - as shown by this example - it is harder for this policy to achieve success, therefore the reward from the fuel term (R_f) is still relevant.

7.3. Monte Carlo simulation

Monte Carlo simulations are often used as test-based verification methods to evaluate the robustness and reliability of neural network policies [14, 32, 30]. The objective of this kind of simulation is to evaluate the performance of a policy over a wide range of randomly-sampled initial conditions. After enough samples, one can get an overview of the range of possible outcomes. A Monte Carlo simulation by itself cannot prove that a neural policy will always behave as expected, but it can be a useful tool to

Parameter	Units	RNN Policy	MLP Policy
Average r_e	m	0.07	0.18
Average v_e	m/s	0.01	0.03
Average θ_e	deg	0.25	3.56
Average ω_e	deg/s	0.32	2.73
Total ΔV	m/s	2.14	1.49
Total $\Delta\omega$	deg/s	17.80	21.04
Collisions	-	No	No
Success	-	Yes	No

Table 7.2: Nominal scenario results.

determine the policy's limitations and to identify any particular instances where it performs poorly. For this Monte Carlo simulation, 1000 initial conditions were sampled from the same random distribution that was employed during the training process. Both the recurrent and feedforward policies were used to generate trajectories starting from these initial conditions. The same metrics were recorded as during the nominal scenario experiments.

Table 7.3 shows the average results and standard deviation of each policy. It can be seen that the performance of the RNN policy is better than that of the MLP policy, since it achieves lower errors, particularly in terms of position and attitude. The only metric in which the RNN policy is notably outperformed by the MLP policy is the fuel efficiency, since the MLP policy used 16% less ΔV on average. However, it should be noted that the MLP policy also tended to terminate some episodes prematurely due to large deviations from the objective, so this may have reduced the total amount of fuel used.

The most striking difference between the policies is in the number of collisions and successful trajectories. For the MLP policy, almost 17% of the tested trajectories resulted in the chaser entering the keep-out zone, and less than 55% of the trajectories were successful. On the other hand, the RNN policy achieved a successful final approach in over 85% of the trajectories, and the chaser only entered the keep-out zone in less than 9% of the trajectories. Once again this demonstrates the benefits of using a recurrent layer in the neural network, as it enables the policy to draw more information from uncertain environments, allowing it to perform better than a feedforward network even though both policies receive the same observations as inputs.

Parameter	Units	RNN Policy	MLP Policy
Average r_e	m	0.16	0.49
Average v_e	m/s	0.03	0.06
Average θ_e	deg	2.37	8.69
Average ω_e	deg/s	1.08	1.02
Total ΔV	m/s	2.52	2.11
Collision %	-	8.8	16.6
Success %	-	85.6	54.5

Table 7.3: Monte Carlo simulation results.

A correlation table was generated to better understand the reasons why some of the trajectories were unsuccessful. The table shows the correlation coefficients between the input parameters and performance metrics, as seen in Figures 7.4a and 7.4b for the RNN and MLP policies, respectively. The rows represent the initial conditions of the Monte Carlo simulation, while the columns represent the performance metrics, including the average errors, the number of collisions, and successes. The first nine rows of the charts refer to the initial conditions related to the chaser, namely its initial position, its initial velocity, and its initial attitude relative to the target. As can be seen in the figures, the correlations on these rows are close to zero, indicating that the initial conditions of the chaser do not have a significant impact on the performance of the policy. This shows that the policies can still perform well even when the initial conditions of the chaser are not nominal. The variable in the tenth row (θ_T) is the target's initial deviation from its nominal orientation. As seen by the color of this row, there is a weak positive correlation between this variable and the average errors, indicating that the errors during the trajectory tend to be larger when the initial deviation of the chaser is larger. This was to be expected, since the nominal orientation for the target has the entry corridor pointed directly towards the chaser, therefore any deviation from the nominal state will rotate the entry corridor further away from the chaser, making it harder for the chaser to perform the approach trajectory. However, the performance of the policies is even more susceptible to the rotation rate of the target (ω_T). As seen by the last row of the charts, there is a strong correlation between the errors and the target's rotation rate, indicating that the errors are larger when the target rotates faster. Furthermore, there is a strong positive correlation between the rotation rate and the number of collisions, and also a strong negative correlation between the rotation rate and the success rate. This suggests that the main reason behind the occurrence of collisions and lack of successful trajectories is the target's rotation.

	Pos. error	Vel. error	Att. error	Rot. error	Collisions	Success
r_x	-0.07	-0.06	-0.06	-0.03	-0.07	0.06
r_y	-0.05	-0.04	-0.05	-0.04	-0.02	0.05
r_z	0.02	0.01	0.01	-0.01	-0.03	0.03
$\lvert r\rvert$	-0.03	-0.01	0.0	-0.05	0.0	0.02
v_x	-0.05	-0.07	-0.07	-0.04	-0.06	0.06
v_y	-0.03	-0.04	-0.02	0.03	-0.04	0.03
v_z	0.03	0.02	0.04	-0.01	0.04	-0.07
$\lvert v\rvert$	0.0	-0.02	-0.02	0.01	-0.04	0.01
θ_c	0.05	0.06	0.06	0.0	0.04	-0.03
θ_T	0.14	0.18	0.16	0.05	0.19	-0.14
$\omega_{c,x}$	-0.05	-0.06	-0.04	-0.01	-0.05	0.01
$\omega_{c,y}$	-0.03	-0.03	-0.03	-0.03	-0.03	0.06
$\omega_{c,z}$	0.01	0.04	0.04	0.0	0.05	-0.03
$\lvert\omega_c\rvert$	0.0	-0.01	-0.01	-0.02	0.01	0.02
$\omega_{T,x}$	-0.01	0.08	-0.03	0.18	0.14	-0.18
$\omega_{T,y}$	0.09	0.04	0.07	-0.14	0.03	-0.25
$\omega_{T,z}$	0.01	-0.02	-0.05	0.0	0.06	0.03
$\lvert\omega_T\rvert$	0.72	0.63	0.62	0.68	0.34	-0.46

(a) Correlation coefficients of the RNN policy.

	Pos. error	Vel. error	Att. error	Rot. error	Collisions	Success
r_x	-0.06	-0.05	-0.06	-0.02	-0.03	0.06
r_y	-0.04	-0.07	-0.07	-0.04	-0.06	0.04
r_z	-0.04	-0.01	-0.01	-0.02	-0.05	0.06
$\lvert r\rvert$	-0.03	-0.01	-0.04	-0.02	-0.03	0.02
v_x	-0.03	-0.05	-0.03	-0.05	-0.02	0.04
v_y	-0.1	-0.07	-0.1	-0.05	-0.05	0.04
v_z	0.04	0.0	0.03	0.04	0.06	0.0
$\lvert v\rvert$	0.05	-0.01	0.01	-0.05	0.04	0.02
θ_c	0.04	0.01	0.02	0.05	-0.01	-0.05
θ_T	0.18	0.3	0.22	0.2	0.14	-0.23
$\omega_{c,x}$	-0.02	-0.04	-0.05	0.01	0.02	-0.02
$\omega_{c,y}$	-0.04	-0.02	-0.04	-0.03	-0.07	0.05
$\omega_{c,z}$	-0.01	0.01	0.0	0.03	-0.03	0.05
$\lvert\omega_c\rvert$	-0.02	-0.01	0.0	-0.02	-0.01	0.01
$\omega_{T,x}$	-0.03	0.11	0.02	0.4	0.13	-0.15
$\omega_{T,y}$	0.02	-0.02	0.06	0.16	0.11	-0.01
$\omega_{T,z}$	0.13	0.07	0.1	-0.03	0.03	0.01
$\lvert\omega_T\rvert$	0.49	0.62	0.55	0.57	0.48	-0.63

(b) Correlation coefficients of the MLP policy.

Figure 7.4: Correlation between initial conditions and performance metrics.

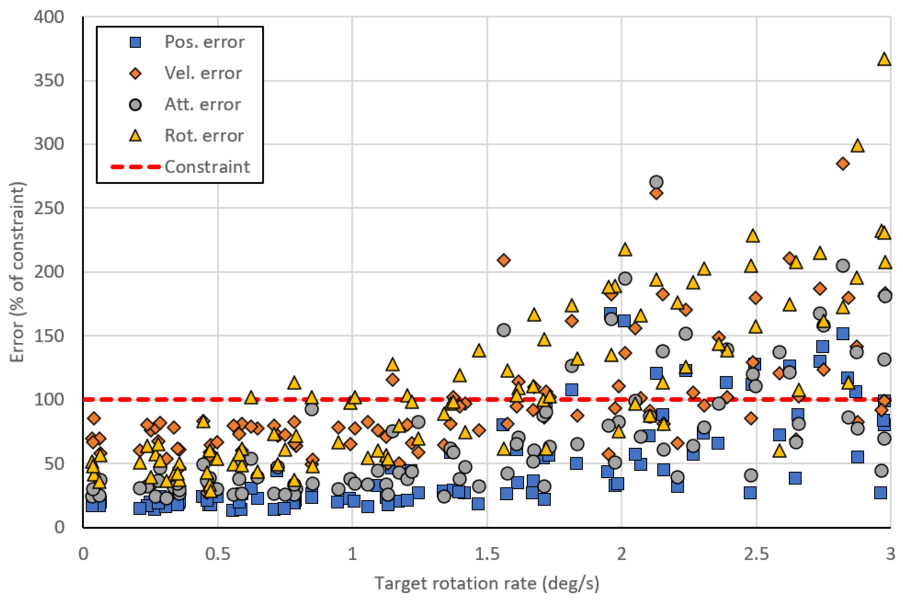


Figure 7.5: Errors of the RNN policy as a function of the target's rotation rate.

The effects caused by the target's rotation rate on the policy's performance can be seen more clearly in Figure 7.5, where the state errors from some of the RNN trajectories are plotted as a function of the target's rotation rate. Note that in this figure the errors are expressed as a percentage of the terminal constraint. It can be seen that when the rotation rate is less than 1 deg/s practically all of the state errors are within the constraint, but as the rotation rate increases the errors begin to grow. This is not unexpected, since a target that rotates faster is more difficult for the chaser to approach. Overall, the performance of the RNN policy is satisfactory, especially when considering that it does not receive full observations of the environment.

7.4. Response time

One final experiment was performed to ensure that the neural network policies can generate an output in a short amount of time, which is required for real-time applications. One thousand random observations were fed to the trained actor network of both the MLP and the RNN models, and the time was measured to evaluate how long it took each network to output an action. The experiment was performed on a personal notebook computer equipped with an Intel Core i7-8550U CPU.

Figure 7.6 shows the result of the experiment. As expected, both networks are capable of generating outputs in a matter of milliseconds, proving that they are suitable for real-time control applications. The MLP network showed an average response time of 1.02 ms, which is similar to the times obtained by other authors [12]. Meanwhile, the RNN network showed a somewhat slower response time of 2.63 milliseconds, making it 160% slower than the MLP on average. It is to be expected that the RNN network would be slower, since it contains both a MLP and a LSTM, and therefore it requires more operations to execute a forward pass through the network. The RNN network also shows a larger difference between its fastest and slowest response times. The MLP network had a range of 0.64 ms between its slowest and fastest responses, whereas the RNN network had a difference of 1.90 ms. It is unclear where this difference stems from. No correlation was found between the response time and any particular region of the state space.

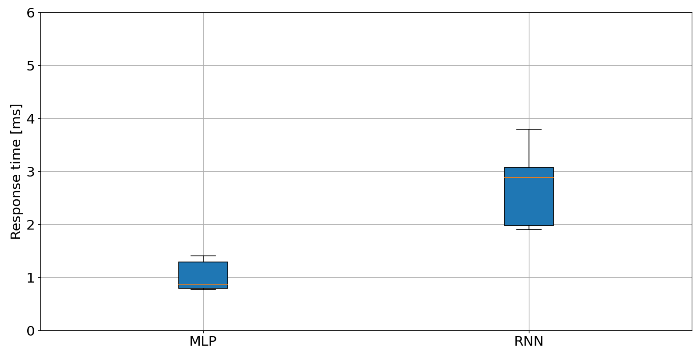


Figure 7.6: Response time of the neural networks.

8 Conclusions and Recommendations

This chapter concludes the main content of the report. Section 8.1 presents a summary of the project, including its goals, results, and answers to the research questions. Lastly, section 8.2 presents several recommendations and ideas for possible future work.

8.1. Summary

The goal of this thesis was to use a technique known as reinforcement meta-learning to develop an adaptive control policy for the final approach phase of a rendezvous trajectory. Although this technique has been previously proposed for landing missions and asteroid intercept missions, this thesis is one of the first to utilize it for a rendezvous scenario with a rotating target, which can have useful applications for on-orbit servicing and active debris removal missions. Furthermore, this thesis tackled a more complex problem than previous studies by randomizing the rotation rate of the target, highlighting the ability of the control policy to adapt to different scenarios. In addition, a custom reward function was designed using reward shaping so that the policy could learn more efficiently. The tuning process for this reward function and for the rest of the model was described in detail so that this technique may be repeated for other applications.

The reinforcement meta-learning technique was implemented by training a recurrent neural network policy in a partially-observable environment, so that the policy may learn to detect long-term dependencies over time. The policy was trained using a reliable implementation of the PPO algorithm. A virtual environment was created in Python to simulate the rendezvous scenario from which the learning algorithm collects experience during training. The environment included a 6DOF dynamics model and a custom-made reward function which had to be tuned before commencing the training process. Tuning the reward function was a critical step, since it determines what the final behavior of the policy will be. There are several terms in the reward function to encourage or penalize different behaviors, so the tuning process becomes a matter of trade-offs between these different terms, ideally arriving at a solution that balances all the requirements without ignoring others. Besides of tuning the reward function, the hyperparameters of the algorithm and the policy were also tuned to find a suitable balance between the stability and the computational time of the training process.

Once the tuning process was completed, a sensitivity analysis was performed to assess how well the model generalized different types of trajectories. The results of this study showed that the tuned model can be successfully used in a variety of scenarios, although some minor modifications to the reward function may be beneficial for the training process in certain cases. Finally, the recurrent policy was trained in a wide range of scenarios without having explicit knowledge of the target's rotation rate. A feedforward policy was also trained under the same conditions for comparison. Both of the policies were trained until their learning curves peaked, and then their performance was evaluated by means of a Monte Carlo simulation, using 1000 randomly generated initial conditions to execute 1000 trajectories with each policy. The results of the Monte Carlo simulation showed that the recurrent policy had a better performance than the feedforward policy, achieving successful trajectories more often. However, both policies occasionally caused collisions, especially when the target rotated faster. Based on these results, the research questions can be answered. Starting with the first sub-question:

1. *Does the reinforcement meta-learning policy achieve collision-free trajectories?*

Avoiding collisions is a major priority during rendezvous missions, so several measures were taken to prioritize safety. A keep-out zone was defined around the target to discourage the chaser from coming too close to it, and the reward function was specifically designed with collision avoidance in mind, since the policy would not receive any success bonuses if there had been any collisions throughout the trajectory. With these measures in place, it was believed that both the meta policy and the non-meta policy would perhaps be equally adept at collision avoidance, but this was not the case. Compared to the feedforward policy, the meta-learning policy was vastly superior in terms of avoiding collisions. However, there were still a few instances in which it did lead the chaser to enter the keep-out zone, as seen in approximately 9% of the sampled trajectories. The collisions occurred mainly when the rotation rate of the target was high, thus making it more difficult for the chaser to enter the corridor without touching the keep-out zone. Still, this result illustrates one of the difficulties of trying to use neural networks as control policies. Neural networks are black boxes, so it is very challenging to guarantee that some given behavior will always be enforced, even when it is specified in the reward function. Hence, the answer to the first sub-question is that the meta-learning policy appears to be better at avoiding collisions than a regular feedforward policy, but it cannot guarantee collision-free trajectories. The second sub-question was related to the efficiency of the policy:

2. *How is the fuel efficiency of the policy affected by reinforcement meta-learning?*

The fuel efficiency of the policies was evaluated in terms of how much total ΔV they used during their trajectories. On average, the feedforward policy was more fuel efficient than the meta-learning policy, as it used 16% less ΔV throughout the 1000 sampled trajectories. However, this difference is also partly due to the early termination of episodes, which occurred more frequently for the feedforward policy than for the meta-learning policy. Furthermore, based on the results it appears that the meta-learning policy was more concerned with achieving the terminal conditions than with being fuel efficient, because it learned that it can obtain a higher cumulative reward from the former than from the latter. This is not a bad behavior from the policy per se, but it does suggest that the tuning process did not yield such a balanced reward function after all. So, the answer to the second sub-question is that the meta-learning policy was more efficient on average, but it could have been better with further tuning. The last of the sub-questions was:

3. *How well does the policy operate with only partial observations of the scenario?*

To assess how the policies performed in highly uncertain environments, the target's rotation rate (ω_T) was hidden from the policies. In other words, at every time step the policies could see the instantaneous orientation of the target, but not its rate of change. This partially-observable environment made the final approach trajectory more challenging, because the policy needs to make the chaser follow the target's motion without precisely knowing that motion. To make the scenario even more difficult, the target's rotation rate was randomized at the beginning of every training episode, meaning that the target could be rotating in any direction. The basic feedforward policy struggled significantly in this environment, as evidenced by its low success rate, and large state errors during the Monte Carlo simulation. The appeal of a meta-learning policy is that in theory it should be able to learn how to deduce the missing information simply by interacting with the environment, using its recurrent layer to learn the relationships between sequences of observations. This technique has previously been seen to work successfully on asteroid and planetary landing simulations, and the results of this project show that it can also be applied to rendezvous scenarios. The meta-learning policy used in this project was able to outperform the feedforward policy in almost every aspect, repeatedly learning to deduce the hidden state variables within the short time frame of a single episode. With this information, the main research question can be answered.

How does reinforcement meta-learning affect the performance of a neural network control policy during a final approach trajectory to a rotating target?

Reinforcement meta-learning can be highly beneficial for the performance of a control policy during rendezvous, as it can generate highly adaptive policies that can react to unknown situations in real-time. In this project, the meta-learning policy learned to detect the unknown rotation rate of the target, allowing it to approach the target in a variety of different rotation states. Such a controller can be useful for rendezvous missions, especially if its adaptability is improved further. Future studies could investigate the adaptability of the controller to features such as actuator failures or sensor bias. However, the meta-learning policy developed in this project did have some limitations. It did not always yield successful trajectories, especially when the target rotated faster. In a few instances the policy also caused collisions between the chaser and the target, despite having designed the reward function to prevent collisions. A deeper study into the tuning of the reward function may help to yield better results. Lastly, one of the main drawbacks of deep learning models is that they are hard to validate due to the black-box nature of neural networks. So perhaps it may take some time before meta-learning policies start being applied to safety-critical operations such as active debris removal.

8.2. Recommendations

Deep learning is a rapidly changing field, and new techniques and tools are constantly emerging or being used in new ways, so there are many modifications that one can apply to this project to improve upon it. For instance, imitation learning could help to improve the performance of the trained policy. In imitation learning, the reward function is modified to reward the policy for imitating an example behavior. Peng et al [68] used this type of ”example-guided” reinforcement learning to create realistic character animations, but the same approach can be applied to trajectory control. If the given examples are optimal trajectories, then the policy could learn to imitate the optimal trajectories more closely, and perhaps achieve the same levels of efficiency as supervised learning while still retaining the robustness of a reinforcement learning model. However, the example trajectories would have to be computed with an optimal control solver for every scenario encountered during training, and this could add a significant amount of time to the training process.

Different types of RNNs could also be used to see if they improve the policy’s performance. For instance, Gated Recurrent Units are similar to LSTMs but have fewer parameters, making them more lightweight and efficient. Another option would be to use *transformers*, which are a more recent kind of neural network that have been widely used for natural language processing since they are effective at modeling long-term dependencies between inputs [69], so they could also prove useful for a control policy. And a different approach to meta-learning altogether would be the use of frame-stacking instead of RNNs. In frame-stacking, the last several observations are fed to the policy instead of only the current observation. With this method, an MLP could discern the temporal relationship between the observations, much like a RNN does. However, the benefit of RNNs is that they can remember an indefinite number of observations, whereas with frame-stacking only a limited number of observations can be stacked without increasing the dimensionality of the problem too much.

In this project, a Monte Carlo simulation was used to assess the performance of the policies, but there are other test-based verification tools that use more systematic approaches to detect errors in the model. Koren et al [70] developed Adaptive Stress Testing, which uses reinforcement learning to find the most likely path to a failure event. Similarly, DeepXplore [71] can reveal erroneous behaviors in neural network controllers. Both of these methods have been successfully tested in simulations of self-driving cars. They are more efficient at finding errors than the Monte Carlo method, but they still cannot provide any guarantees about the safety of the model. Like every other test-based method, they can discover the presence of errors, but they cannot prove their absence.

An alternative to test-based methods are formal verification techniques, which thoroughly analyze a system to provide assurances about whether or not it will meet the necessary requirements. The development of formal verification techniques for deep learning models is an active area of research, but there have been promising advances such as Reach-LP [72], which seeks to estimate the forward reachable set of a closed-loop system controlled by a neural network. Formal verification methods such as Reach-LP can guarantee that the state of the system will remain within certain bounds, whereas test-based methods cannot provide these guarantees. However, formal verification methods often have many limitations which restrict their use. For instance, Reach-LP is only applicable to feed-forward neural networks. Hence, test-based methods are still more widely used.

One practical recommendation is to have appropriate hardware to train neural network policies. During this project, training a policy could take hours, and running some of the tuning scripts took several days to complete. It is highly recommended to use a cloud computing service to run computationally intensive programs on remote machines, saving time and resources. But one caveat to be aware of is that the cloud-based services offered online often have limitations such as reduced storage space. TU Delft has many services such as the DHPC that are often freely available to students and researchers.

A

Derivation of quaternion time-derivative

Recall that a quaternion represents a 3D rotation of θ radians around a unit vector $\hat{e}$, which is expressed as shown below:

$$\mathbf{q} = \left[\cos\frac{\theta}{2},\quad \hat{e}\sin\frac{\theta}{2}\right] \tag{A.1}$$

If a body is rotating at a constant rate $\boldsymbol{\omega}$, then after a time Δt the body will have rotated by an amount $|\omega(t)|\Delta t$. This rotation can be expressed as the following quaternion:

$$\mathbf{q}(\Delta t) = \left[\cos\frac{|\omega(t)|\Delta t}{2},\quad \sin\frac{|\omega(t)|\Delta t}{2}\frac{\boldsymbol{\omega}(t)}{|\omega(t)|}\right] \tag{A.2}$$

At a time $t_0 + \Delta t$, the orientation of the body would be given by the result of first rotating it by $\mathbf{q}(t_0)$, and then rotating it by $\mathbf{q}(\Delta t)$. The result of these two rotations would be as shown below, where the $\otimes$ operator is the Hamilton product:

$$\mathbf{q}(t_0 + \Delta t) = \left[\cos\frac{|\omega(t_0)|\Delta t}{2},\quad \sin\frac{|\omega(t_0)|\Delta t}{2}\frac{\boldsymbol{\omega}(t_0)}{|\omega(t_0)|}\right] \otimes \mathbf{q}(t_0) \tag{A.3}$$

Making the substitution $t = t_0 + \Delta t$, the previous equation can be expressed as:

$$\mathbf{q}(t) = \left[\cos\frac{|\omega(t_0)|(t-t_0)}{2},\quad \sin\frac{|\omega(t_0)|(t-t_0)}{2}\frac{\boldsymbol{\omega}(t_0)}{|\omega(t_0)|}\right] \otimes \mathbf{q}(t_0) \tag{A.4}$$

Differentiating with respect to t yields:

$$\frac{d}{dt}\mathbf{q}(t) = \left[-\frac{|\omega(t_0)|}{2}\sin\frac{|\omega(t_0)|(t-t_0)}{2},\quad \frac{|\omega(t_0)|}{2}\cos\frac{|\omega(t_0)|(t-t_0)}{2}\frac{\boldsymbol{\omega}(t_0)}{|\omega(t_0)|}\right] \otimes \mathbf{q}(t_0) \tag{A.5}$$

At time $t = t_0$ the derivative evaluates to:

$$\left.\frac{d}{dt}\mathbf{q}(t)\right|_{t_0} = \left[-\frac{|\omega(t_0)|}{2}\sin 0,\quad \frac{|\omega(t_0)|}{2}\cos 0\frac{\boldsymbol{\omega}(t_0)}{|\omega(t_0)|}\right] \otimes \mathbf{q}(t_0) = \left[0,\quad \frac{|\omega(t_0)|}{2}\frac{\boldsymbol{\omega}(t_0)}{|\omega(t_0)|}\right] \otimes \mathbf{q}(t_0) \tag{A.6}$$

Thus, the time-derivative of the quaternion is:

$$\frac{d}{dt}\mathbf{q}(t) = \left[0,\quad \frac{1}{2}\boldsymbol{\omega}(t)\right] \otimes \mathbf{q}(t) \tag{A.7}$$

Expressed as a matrix multiplication:

$$\frac{d}{dt}\mathbf{q}(t) = \frac{1}{2}\begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & -\omega_z & \omega_y \\ \omega_y & \omega_z & 0 & -\omega_x \\ \omega_z & -\omega_y & \omega_x & 0 \end{bmatrix}\begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} \tag{A.8}$$

B
Code listings

B.1. Custom environment

The rendezvous scenario was implemented in Python using the OpenAI Gym library. This library offers a variety of training environments, and it also allows users to create custom environments. Code listing 1 shows a template for how to create such an environment. The environment must be defined as a Python class, using the `Env()` base class as a parent to inherit its core functionality. A template for this custom environment can be seen below in code listing 1. As shown in the code, there are five functions that need to be defined to complete the environment class: `__init__()`, `step()`, `reset()`, `render()`, and `close()`.

```python
from gym import Env

class CustomEnv(Env):

    def __init__(self):
        # Define the attributes of the environment.
        return

    def step(self, action):
        # Run one timestep of the environment's dynamics.
        return obs, rew, done, info

    def reset(self):
        # Reset the state of the environment.
        return obs

    def render(self):
        # Generate the current frame of the environment visualization.
        return

    def close(self):
        # Close any resources used by the environment (e.g. figures, etc)
        return
```

Listing 1: Custom environment template

B.2. Callback function

Code listing 2 shows how a callback function can be used to evaluate and save a neural network policy created in SB3. The `evaluate_policy` function requires four arguments: `policy`, `env`, `n_evals`, and `previous_best`. The `policy` argument is the current policy, and `env` is the environment where the policy will be tested. The `n_evals` argument determines how many episodes will be evaluated, and `previous_best` keeps track of the best result achieved on previous calls to the function. The callback function runs several episodes of the environment, using the current policy to select the action on each time step. Then it computes the average reward across all of the episodes and logs the result using *Weights & Biases* (W&B). The logged values can be viewed on W&B's online platform to get a real-time view of the algorithm's progress. In addition, the current policy is saved as a zip file if the average reward is higher than on previous evaluations. At the end of the training process, the callback function will have recorded the progress of the learning algorithm, and it will also have saved the policy that achieves the best average reward.

```python
def evaluate_policy(policy, env, n_evals, previous_best):

    total_reward = 0

    # Evaluate episodes:
    for i in range(n_evals):
        obs = env.reset()
        done = False
        while not done:
            action = policy.predict(observation=obs,
                                    deterministic=True
                                    )
            obs, reward, done, info = env.step(action)
            total_reward += reward

    # Log avg. reward and save policy:
    average_reward = total_reward / n_evals
    wandb.log(average_reward)
    if average_reward > previous_best:
        policy.save()
        previous_best = average_reward

    return previous_best
```

Listing 2: Callback function

B.3. Training scripts

A basic Python script to run the learning algorithm would look as shown in code listing 3. On line 5, the variable named `model` is created as an instance of the `PPO()` class, which uses the `RendezvousEnv` environment described in section 3.3 and the policy named `MlpPolicy`, which will be described in the following section. Optional keyword arguments can also be passed to the algorithm to define properties such as the learning rate, mini-batch size, or the number of epochs. Then the `learn()` function is called on line 8 to train the model for $100,000$ time steps, using the callback to track and save the results.

The commands for training a recurrent policy in SB3 are different to that of a feedforward policy. This is because the default `PPO` class is not compatible with recurrent neural network architectures.Instead, the `RecurrentPPO` class needs to be used. The behavior of this class is identical to that of the `PPO` class. The only difference is that `RecurrentPPO` supports recurrent policies such as `MlpLstmPolicy()`. As a result, the training shown in code listing 3 needs to be slightly modified to

```python
from stable_baselines3 import PPO
from local import RendezvousEnv, EvaluationCallback

# Create the PPO model:
model = PPO(env=RendezvousEnv(), policy='MlpPolicy', **kwargs)

# Train the model:
model.learn(total_timesteps=100_000, callback=EvaluationCallback())
```

Listing 3: Training script

```python
from sb3_contrib import RecurrentPPO
from local import RendezvousEnv, EvaluationCallback

# Create the PPO model:
rnn_model = RecurrentPPO(env=RendezvousEnv(), policy='MlpLstmPolicy')

# Train the model:
rnn_model.learn(total_timesteps=100_000, callback=EvaluationCallback())
```

Listing 4: Training script for a recurrent policy

train a recurrent policy. The modified script can be seen in code listing 4. Throughout this project, `MlpLstmPolicy` was used to create recurrent policies, while `MlpPolicy` was used to create the feedforward policies. The callback described in section B.2 is compatible for both the recurrent and non-recurrent policies.

B.4. Tuning sweep

An example of how to perform a sweep is shown in code listing 5. This example explains how to set up and execute a sweep that tests how the learning algorithm performs with different learning rates. In lines 6-12, the variable `config` contains the configuration details of the sweep. It defines the sweep's search method and the hyperparameters that will be varied during the sweep. In this example, the only hyperparameter that is varied throughout the sweep is the learning rate, but more hyperparameters can also be added to the sweep if desired. The search method is set to random, so the learning rate is randomly sampled from a uniform distribution between 0.01 and 1. On lines 13 and 14, the command `wand.sweep()` creates the sweep using the given configuration data, and then the command `wandb.agent()` starts executing runs. On each run, the learning rate is randomly sampled from the given uniform distribution, and the `single_run()` function is executed. This function creates an instance of the learning algorithm with the given learning rate, then trains a policy for 100,000 time steps and logs the results to the W&B dashboard. In this example, the sweep has no limit on the number of runs, so it will continue executing runs indefinitely until it is stopped by the user. Sweeps such as this one were used throughout the tuning process to find suitable hyperparameters for the model.

```python
import wandb
from sb3_contrib import RecurrentPPO
from stable_baselines3.common.evaluate import evaluate_policy
from rendezvous_env import RendezvousEnv

config = {"method": "random",                # Define the search method
          "parameters": {                     # Define the hyperparameters
              "learning_rate": {
                  "distribution": "uniform",
                  "min": 0.1,
                  "max": 1}
              }
          }

sweep_id = wandb.sweep(sweep=config)          # Create the sweep
wandb.agent(sweep_id, function=single_run)   # Execute the sweep

def single_run():
    with wandb.init() as run:
       model = RecurrentPPO(
             policy="MlpLstmPolicy",
             env=RendezvousEnv(),
             learning_rate=wandb.config["learning_rate"])
        model.learn(total_timesteps=100_000)               # Train
        wandb.log(evaluate_policy(model, RendezvousEnv())) # Log data
```

Listing 5: Example of a sweep script.

Bibliography

[1] Wigbert Fehse. “Rendezvous with and Capture / Removal of Non-Cooperative Bodies in Orbit: The Technical Challenges”. In: *Journal of Space Safety Engineering* 1.1 (2014), pp. 17–27. ISSN: 24688967. DOI: 10.1016/S2468-8967(16)30068-4. URL: http://dx.doi.org/10.1016/S2468-8967(16)30068-4.

[2] Donald J. Kessler and Burton G. Cour-Palais. “Collision frequency of artificial satellites: The creation of a debris belt”. In: *Journal of Geophysical Research* 83.A6 (June 1978), pp. 2637–2646. DOI: 10.1029/JA083iA06p02637.

[3] ESA. *Annual Space Environment Report*. ESA, 2021. URL: https://www.sdo.esoc.esa.int/environment_report/Space_Environment_Report_latest.pdf.

[4] Alex Ellery. “Tutorial review on space manipulators for space debris mitigation”. In: *Robotics* 8.2 (2019), p. 34. ISSN: 22186581. DOI: 10.3390/ROBOTICS8020034.

[5] “Review of advanced guidance and control algorithms for space/aerospace vehicles”. In: *Progress in Aerospace Sciences* 122.December 2020 (2021). ISSN: 03760421. URL: https://doi.org/10.1016/j.paerosci.2021.100696.

[6] Jianjun Ni et al. “A Survey on Theories and Applications for Self-Driving Cars Based on Deep Learning Methods”. In: *Applied Sciences* 10.8 (2020). ISSN: 2076-3417. DOI: 10.3390/app10082749. URL: https://www.mdpi.com/2076-3417/10/8/2749.

[7] Su Yeon Choi and Dowan Cha. “Unmanned aerial vehicles using machine learning for autonomous flight; state-of-the-art”. In: *Advanced Robotics* 33.6 (2019), pp. 265–277. DOI: 10.1080/01691864.2019.1586760. eprint: https://doi.org/10.1080/01691864.2019.1586760. URL: https://doi.org/10.1080/01691864.2019.1586760.

[8] Dario Izzo, Marcus Märtens, and Binfeng Pan. “A survey on artificial intelligence trends in spacecraft guidance dynamics and control”. In: *Astrodynamics* 3.4 (2019), pp. 287–299. DOI: 10.1007/s42064-018-0053-6. URL: https://doi.org/10.1007/s42064-018-0053-6.

[9] Dario Izzo, Dharmesh Tailor, and Thomas Vasileiou. “On the Stability Analysis of Deep Neural Network Representations of an Optimal State Feedback”. In: *IEEE Transactions on Aerospace and Electronic Systems* 57.1 (2021), pp. 145–154. DOI: 10.1109/TAES.2020.3010670.

[10] Cole George, Joshuah Hess, and Richard Cobb. “Open-and closed-loop neural network control for proximal spacecraft maneuvers”. In: *Advances in the Astronautical Sciences* 168 (2019), pp. 3453–3469. ISSN: 00653438.

[11] Elisabeth A. Youmans and Frederick H. Lutze. “Neural network control of space vehicle intercept and rendezvous maneuvers”. In: *Journal of Guidance, Control, and Dynamics* 21.1 (1998), pp. 116–121. ISSN: 15333884. DOI: 10.2514/2.4206.

[12] Charles E. Oestreich, Richard Linares, and Ravi Gondhalekar. “Autonomous Six-Degree-of-Freedom Spacecraft Docking with Rotating Targets via Reinforcement Learning”. In: *Journal of Aerospace Information Systems* 18.7 (2021), pp. 1–12. DOI: 10.2514/1.i010914.

[13] Hongjue Li, Yunfeng Dong, and Peiyun Li. “Real-Time Optimal Approach and Capture of ENVISAT Based on Neural Networks”. In: *International Journal of Aerospace Engineering* 2020 (2020). ISSN: 16875974. DOI: 10.1155/2020/8165147.

[14] Lorenzo Federici, Boris Benedikter, and Alessandro Zavoli. “Machine Learning Techniques for Autonomous Spacecraft Guidance during Proximity Operations”. In: *AIAA Scitech 2019 Forum* January (2021). DOI: 10.2514/6.2021-0668.

[15] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8.

[16] Jane X Wang et al. *Learning to reinforcement learn*. 2016. DOI: 10.48550/ARXIV.1611.05763. URL: https://arxiv.org/abs/1611.05763.

[17] Brian Gaudet, Richard Linares, and Roberto Furfaro. "Six Degree-of-Freedom Hovering using LIDAR Altimetry via Reinforcement Meta-Learning". In: *AIAA Scitech 2020 Forum* January (2020). DOI: 10.2514/6.2020-0953.

[18] Brian Gaudet, Richard Linares, and Roberto Furfaro. "Adaptive guidance and integrated navigation with reinforcement meta-learning". In: *Acta Astronautica* 169.April 2020 (2020), pp. 180–190. ISSN: 00945765. DOI: 10.1016/j.actaastro.2020.01.007. arXiv: 1904.09865. URL: https://doi.org/10.1016/j.actaastro.2020.01.007.

[19] W. H. Clohessy and R. S. Wiltshire. "Terminal Guidance System for Satellite Rendezvous". In: *Journal of the Aerospace Sciences* 27.9 (1960), pp. 653–658. ISSN: 1936-9999. DOI: 10.2514/8.8704. URL: https://arc.aiaa.org/doi/10.2514/8.8704.

[20] Y. Yang. "Spacecraft attitude determination and control: Quaternion based method". In: *Annual Reviews in Control* 36.2 (2012), pp. 198–219. ISSN: 1367-5788. DOI: https://doi.org/10.1016/j.arcontrol.2012.09.003. URL: https://www.sciencedirect.com/science/article/pii/S1367578812000387.

[21] Andrea Scorsoglio et al. "Actor-critic reinforcement learning approach to relative motion guidance in near-rectilinear orbit". In: *29th AAS/AIAA Space Flight Mechanics Meeting*. 2019, pp. 1737–1756. ISBN: 9780877036593.

[22] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347.

[23] Jane X. Wang et al. "Learning to reinforcement learn". In: *CogSci*. 2017. URL: https://mindmodeling.org/cogsci2017/papers/0252/index.html.

[24] Lorenzo Federici et al. "Image-based Meta-Reinforcement Learning for Autonomous Terminal Guidance of an Impactor in a Binary Asteroid System". In: *AIAA Science and Technology Forum and Exposition, AIAA SciTech Forum 2022* (2022), pp. 1–22. DOI: 10.2514/6.2022-2270.

[25] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[26] DLR. *Institute of Robotics and Mechatronics*. https://www.dlr.de/rm/en/desktopdefault.aspx/tabid-8017. Accessed: 2022-07-14.

[27] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[28] Dario Izzo, Christopher Iliffe Sprague, and Dharmesh Vijay Tailor. "Machine Learning and Evolutionary Techniques in Interplanetary Trajectory Design". In: *Springer Optimization and its Applications*. Vol. 144. 2019, pp. 191–210. DOI: 10.1007/978-3-030-10501-3_8. arXiv: 1802.00180.

[29] Shanshan Yin, Jian Li, and Lin Cheng. "Low-thrust spacecraft trajectory optimization via a DNN-based method". In: *Advances in Space Research* 66.7 (2020), pp. 1635–1646. ISSN: 18791948. DOI: 10.1016/j.asr.2020.05.046. URL: https://doi.org/10.1016/j.asr.2020.05.046.

[30] Brian Gaudet, Richard Linares, and Roberto Furfaro. "Deep reinforcement learning for six degree-of-freedom planetary landing". In: *Advances in Space Research* 65.7 (2020), pp. 1723–1741. ISSN: 18791948. DOI: 10.1016/j.asr.2019.12.030. URL: https://doi.org/10.1016/j.asr.2019.12.030.

[31] Brian Gaudet and Roberto Furfaro. "Robust spacecraft hovering near small bodies in environments with unknown dynamics using reinforcement learning". In: *AIAA/AAS Astrodynamics Specialist Conference 2012* August (2012), pp. 1–20. DOI: 10.2514/6.2012-5072.

[32] Nicholas B Lafarge et al. “Guidance for Closed-Loop Transfers using Reinforcement Learning with Application to Libration Point Orbits”. In: *AIAA Scitech 2020 Forum* January (2020). DOI: 10.2514/6.2020-0458.

[33] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[34] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[35] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[36] Plotly Technologies Inc. *Collaborative data science*. 2015. URL: https://plot.ly.

[37] Dieter Scherer, Pierre Dubois, and Beth Sherwood. “VPython: 3D interactive scientific graphics for students”. In: *Comput. Sci. Eng.* 2 (2000), pp. 56–62.

[38] Delft High Performance Computing Centre (DHPC). *DelftBlue Supercomputer (Phase 1)*. https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1. 2022.

[39] Wigbert Fehse. *Automated Rendezvous and Docking of Spacecraft*. Cambridge Aerospace Series. Cambridge University Press, 2003. DOI: 10.1017/CBO9780511543388.

[40] J. Tschauner and P. Hempel. “Rendezvous zu einem in elliptischer Bahn umlaufenden Ziel”. In: *Astronautica Acta* 11.2 (1965), pp. 104–109. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-50549203494&partnerID=40&md5=7e39b762954471288892564261eb9eed.

[41] Jacob G. Elkins, Rohan Sood, and Clemens Rumpf. “Autonomous spacecraft attitude control using deep reinforcement learning”. In: *Proceedings of the International Astronautical Congress, IAC*. Vol. October. 2020.

[42] Alexander L. Fradkov. “Early History of Machine Learning”. In: *IFAC-PapersOnLine* 53.2 (2020). 21st IFAC World Congress, pp. 1385–1390. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.12.1888. URL: https://www.sciencedirect.com/science/article/pii/S2405896320325027.

[43] E. Alpaydin. *Introduction to Machine Learning, fourth edition*. Adaptive Computation and Machine Learning series. MIT Press, 2020. ISBN: 9780262358064.

[44] “Reinforcement learning in robotics: A survey”. In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274. DOI: 10.1177/0278364913495721.

[45] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. MIT Press, 2018.

[46] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards”. In: (1989).

[47] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals and Systems* 2 (4 1989), pp. 303–314. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[48] Kyongsik Yun et al. “Multi-agent motion planning using deep learning for space applications”. In: *Accelerating Space Commerce, Exploration, and New Discovery Conference, ASCEND 2020*. 2020. ISBN: 9781624106088. DOI: 10.2514/6.2020-4233.

[49] Hongjue Li et al. “Optimal Real-Time Approach and Capture of Uncontrolled Spacecraft”. In: *Journal of Spacecraft and Rockets* (2021), pp. 1–12. ISSN: 0022-4650. DOI: 10.2514/1.a34687.

[50] Kunihiko Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological Cybernetics* 36.4 (1980), pp. 193–202. ISSN: 1432-0770. DOI: 10.1007/BF00344251. URL: https://doi.org/10.1007/BF00344251.

[51] L. Pasqualetto Cassinis et al. “CNN-based pose estimation system for close-proximity operations around uncooperative spacecraft”. In: vol. 1. 2020. DOI: 10.2514/6.2020-1457. URL: https://doi.org/10.2514/6.2020-1457.

[52] Haoran Huang et al. “Non-Model-Based Monocular Pose Estimation Network for Uncooperative Spacecraft Using Convolutional Neural Network”. In: *IEEE Sensors Journal* 21.21 (2021), pp. 24579–24590. DOI: `10.1109/JSEN.2021.3115844`.

[53] Jianing Song, Duarte Rondão, and Nabil Aouf. “Deep Learning-based Spacecraft Relative Navigation Methods: A Survey”. In: *ArXiv* (2021). URL: `https://arxiv.org/abs/2108.08876`.

[54] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536. ISSN: 1476-4687. DOI: `10.1038/323533a0`. URL: `https://doi.org/10.1038/323533a0`.

[55] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: `10.1162/neco.1997.9.8.1735`. eprint: `https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf`. URL: `https://doi.org/10.1162/neco.1997.9.8.1735`.

[56] L. Medsker and L.C. Jain. *Recurrent Neural Networks: Design and Applications*. International Series on Computational Intelligence. CRC Press, 1999. ISBN: 9781420049176. URL: `https://books.google.nl/books?id=ME1SAkN0PyMC`.

[57] Brian Gaudet, Richard Linares, and Roberto Furfaro. “Terminal adaptive guidance via reinforcement meta-learning: Applications to autonomous asteroid close-proximity operations”. In: *Acta Astronautica* 171.June 2020 (2020), pp. 1–13. ISSN: 00945765. DOI: `10.1016/j.actaastro.2020.02.036`.

[58] *State-of-the-art of Small Spacecraft Technology*. Accessed: 2022-09-01. Oct. 2021. URL: `https://www.nasa.gov/smallsat-institute/sst-soa`.

[59] Stable Baselines3 Documentation. *Reinforcement learning tips and tricks*. `https://stable-baselines3.readthedocs.io/en/master/guide/rl_tips.html`. Accessed: 2022-08-01.

[60] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[61] Wigbert Fehse. “Orbit dynamics and trajectory elements”. In: *Automated Rendezvous and Docking of Spacecraft*. Cambridge Aerospace Series. Cambridge University Press, 2003, pp. 29–75. DOI: `10.1017/CBO9780511543388.004`.

[62] Andrea Scorsoglio et al. “Image-based deep reinforcement learning for autonomous lunar landing”. In: *AIAA Scitech 2020 Forum*. 2020. ISBN: 9781624105951. DOI: `10.2514/6.2020-1910`.

[63] Cenk Bircanoğlu and Nafiz Arıca. “A comparison of activation functions in artificial neural networks”. In: *2018 26th Signal Processing and Communications Applications Conference (SIU)*. 2018, pp. 1–4. DOI: `10.1109/SIU.2018.8404724`.

[64] Timothy Masters. *Practical Neural Network Recipes in C++*. Morgan Kaufmann, 1993. ISBN: 9780124790407. DOI: `10.1016/C2009-0-22399-3`.

[65] Jeff Heaton. *Introduction to Neural Networks for Java, 2nd Edition*. 2nd. Heaton Research, Inc., 2008. ISBN: 1604390085.

[66] Anna Rakitianskaia and Andries Engelbrecht. “Measuring Saturation in Neural Networks”. In: *2015 IEEE Symposium Series on Computational Intelligence*. 2015, pp. 1423–1430. DOI: `10.1109/SSCI.2015.202`.

[67] Brian Gaudet, Richard Linares, and Roberto Furfaro. “Six Degree-of-Freedom Hovering using LIDAR Altimetry via Reinforcement Meta-Learning”. In: *AIAA Scitech 2020 Forum* January (2020). DOI: `10.2514/6.2020-0953`.

[68] Xue Bin Peng et al. “DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills”. In: *ACM Transactions on Graphics* 37.4 (Aug. 2018), pp. 1–14. ISSN: 0730-0301. DOI: `10.1145/3197517.3201311`. arXiv: `1804.02717`. URL: `https://dl.acm.org/doi/10.1145/3197517.3201311`.

[69] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[70] Mark Koren et al. “Adaptive Stress Testing for Autonomous Vehicles”. In: *2018 IEEE Intelligent Vehicles Symposium (IV)* (2018), pp. 1–7.

[71] Kexin Pei et al. “DeepXplore: Automated Whitebox Testing of Deep Learning Systems”. In: *Communications of the ACM* 62.11 (Oct. 2019), pp. 137–145. ISSN: 0001-0782. DOI: `10.1145/3361566`. URL: `https://doi.org/10.1145/3361566`.

[72] Michael Everett, Golnaz Habibi, and Jonathan P. How. “Efficient Reachability Analysis of Closed-Loop Systems with Neural Network Controllers”. In: *Proceedings - IEEE International Conference on Robotics and Automation*. 2021, pp. 4384–4390. ISBN: 9781728190778. DOI: `10.1109/ICRA48506.2021.9561348`. arXiv: `2101.01815`.