RESEARCH PROJECT IN MECHATRONICS ENGINEERING

PART IV PROJECT FINAL REPORT

Enhancing classical control laws with reinforcement learning

Sumukha Viswakarma

Project Report ME038-2023

Co-worker: K'vaan Valabh

Supervisor: Dr Roberto Armellin

Department of Mechanical and Mechatronics Engineering
The University of Auckland

13 October 2023

ME038-2023

ENHANCING CLASSICAL CONTROL LAWS WITH REINFORCEMENT LEARNING

Sumukha Viswakarma

DECLARATION

Student

I …Sumukha Viswakarma……. hereby declare that:

1. This report is the result of the final year project work carried out by my project partner (see cover page) and I under the guidance of our supervisor (see cover page) in the 2023 academic year at the Department of Mechanical and Mechatronics Engineering, Faculty of Engineering, University of Auckland.
2. This report is not the outcome of work done previously.
3. This report is not the outcome of work done in collaboration, except that with a project sponsor as stated in the text.
4. This report is not the same as any report, thesis, conference article or journal paper, or any other publication or unpublished work in any format.

In the case of a continuing project: State clearly what has been developed during the project and what was available from previous year(s):

Signature: 

Date: 11/10/2023

Supervisor

I confirm that the project work undertaken by this student in the 2023 academic year **is / is not** (*strikethrough as appropriate*) part of a continuing project, components of which have been completed previously.

Comments, if any:

Signature: 

Date: 11/10/2023

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank our project supervisor Professor Roberto Armellin, for his continual support, expertise, and guidance throughout this project.

I would also like to thank Harry Holt, for his help with the RL side of the project - he was great at demystifying a lot of tough RL concepts.

Also, a huge thanks to my project partner K'vaan - certainly could not have learnt this much and come this far without his contributions.

1. Introduction

1.1 Project Scope

Since their emergence almost a century ago, classical control laws have been a staple in many industries, offering stability and predictability to several systems and processes. This project concerns itself with the improvement of these control laws, by making use of machine learning concepts. Specifically, we aim to investigate the incorporation of Reinforcement Learning (RL), a subsection of machine learning.

The controllers being explored in this project are the Lyapunov Controller and the Linear Quadratic Regulator (LQR). Both of these controllers have advantages in several applications. The Lyapunov Controller offers an analytically-stable solution, especially for non-linear systems, but often lacks optimality, while LQR controllers can provide highly optimal control, but require rigorous tuning. Each of these control methods have their own associated advantages and drawbacks, and it is these drawbacks that we aim to minimise by reinforcing these methods with RL.

While control and automation are important considerations for many industries, this project will be focused on the aerospace industry. With a reinforced control system, such as the ones described above, spacecraft trajectory design and guidance can be completed on-board, reducing the reliance of these spacecraft on a ground crew. This is just one example of the improvements a more reliable control system can provide. The scope of this project therefore, is to enhance Lyapunov and LQR controllers using RL, and apply these reinforced-controllers to aerospace scenarios, such as spacecraft formation flying.

1.2 Objectives

The objectives of this project can be separated into three sections.

1. Obtaining a deeper understanding of the project and the surrounding material, e.g. choosing the dynamics model and understanding the two control laws.
2. Implementing the two control laws in a chosen spacecraft formation flying context - in this section of the project, the objective is to design and implement both Lyapunov and LQR Controllers in a simple aerospace context, for example, a two dimensional chief-chaser orbit, and investigate the system response.
3. Optimise these controllers using Particle Swarm Optimisation (PSO).
4. Apply Reinforcement Learning - This is the section of the project where we aim to apply reinforcement learning to the controllers implemented earlier, in the hopes of improving upon the PSO.

1.3 Project Background

Classical Control Methods

Lyapunov Controller

The Lyapunov Controller is a powerful controller for many systems, especially non-linear systems, and provides an analytically stable control law. It works on the principle of the Lyapunov Function, $\mathbf{V}(\mathbf{x})$, where $\mathbf{V}(\mathbf{x})$ is a scalar, energy-like function [2], dependent on the state of the system, $\mathbf{x}$. The Lyapunov Function is parabolic in nature, and the goal of Lyapunov Control is to drive $\mathbf{V}(\mathbf{x})$ to zero, and by doing so, drive the system toward a target state.

One drawback of using Lyapunov Control is the difficulty in generating a valid Lyapunov Function for a given system. However, Lyapunov Function prototypes which can be applied to several aerospace systems already exist [2]. The generation of the required function can also be made easier by considering the energy of the system first.

Another disadvantage of Lyapunov Controllers is the lack of optimality that they provide. This is a problem that this project and others [3] aim to mitigate, by introducing RL.

LQR Controller

Linear Quadratic Regulation is a reliable control method, used for linear (or linearised) systems. It is a branch of Optimal Control Theory, and is often used in systems with multiple actuators. It is often times more intuitive to tune a system using LQR over traditional PID controllers, attributed to the more holistic nature of the control method, and LQR control has been evidenced to give better results than a standard PID controller [4].

Though highly optimal, LQR controllers still need to be tuned, by varying certain cost terms within the LQR control law. This project will aim to achieve this using RL methods, training a model to automatically complete this tuning process.

Reinforcement Learning

Reinforcement learning is a branch of machine learning, wherein the relationship between an agent and its surrounding environment is used to produce predictable and repeatable behaviour. There are three main components: the action, which is committed by the agent onto the environment; the state of the environment; and the reward, which is awarded to the agent by the environment [5]. The agent's role is to use this information gathered - the new state of the environment (after the agent's action), the reward obtained from the action, and an inbuilt 'policy' - and determine the next appropriate action that will maximise the reward returned.

There are several factors that can be explored when it comes to the type of reinforcement learning used, including whether the algorithm is model-free or model-based. Model-based algorithms are able to predict the state of the environment, and can therefore prepare actions in advance of happening, while model-free algorithms lack this ability [5]. However, it is not always possible to model the behaviour of the environment, especially for environments with complicated systems. For these cases, model-free algorithms, such as Advantage Actor-Critic (A2C) and Proximal Policy Optimisation (PPO) are generally better.

In order to perform the appropriate action, the agent takes into consideration both its policy and the reward being returned. The policy of an agent is a set of probabilities that

correspond to the state of the environment - it allows the agent to map from the state to the probability of selecting the next action [5]. As for rewards, the goal of the agent is generally to maximise the overall reward obtained. Since this can be never-ending, the reward from environment to agent is typically 'discounted' - each future reward expected is multiplied by the discount rate which falls between the values of 0 and 1.

Relevant Existing Work

On the improvement of classical control methods using RL, extensive work has already been conducted, including the two control methods being investigated in this project.

Howell and Ulrich, on two occasions, have found success using deep reinforcement learning methods to aid in spacecraft pose tracking and docking [6, 7]. Though the control methods applied are not the ones being explored in this project, the system being explored is that of a chaser and target, similar to the systems that will be explored in this project. Altogether, these studies by Howell and Ulrich show that RL has the potential to improve control laws.

When it comes to RL and Lyapunov Control, extensive work has been completed by Holt [3], exploring the effects of using a reinforced Lyapunov controller on trajectory design. Holt's controller uses an actor-critic method, producing a highly-effective control law that is stable and optimal when there is no perturbations. Holt's work sets a solid foundation for the application of RL to a Lyapunov Controller, and provides our project with a Since Holt's work is largely focused on trajectory design, there is scope for looking specifically into spacecraft formation.

On the LQR side, there is comparatively less existing work. In their paper, Caarls [8] explores the combination of LQR and RL by applying three different LQR methods to two different RL algorithms, being DQN (Deep Q Learning) and DDPG (Deep Deterministic Policy Gradient). Simulations were then run on a pendulum swing-up, a cart-pole swing-up, and a 2D flyer, using these combinations. When the DQN algorithm was used, Caarls reports all LQR methods had a shorter rise time and better end performance, while the combinations with the DDPG algorithm showed less consistent results. While Caarls' work has little to no relation with any aerospace applications, it is still a promising glance into the effect of RL on LQR control. This project aims to provide some expansion on this work, by exploring the effect of using A2C and PPO algorithms instead.

Though this project as a whole investigates the improvement of these two control laws using Reinforcement Learning, the following report details processes used to improve only LQR control.

2. Setup

2.1 Dynamical System

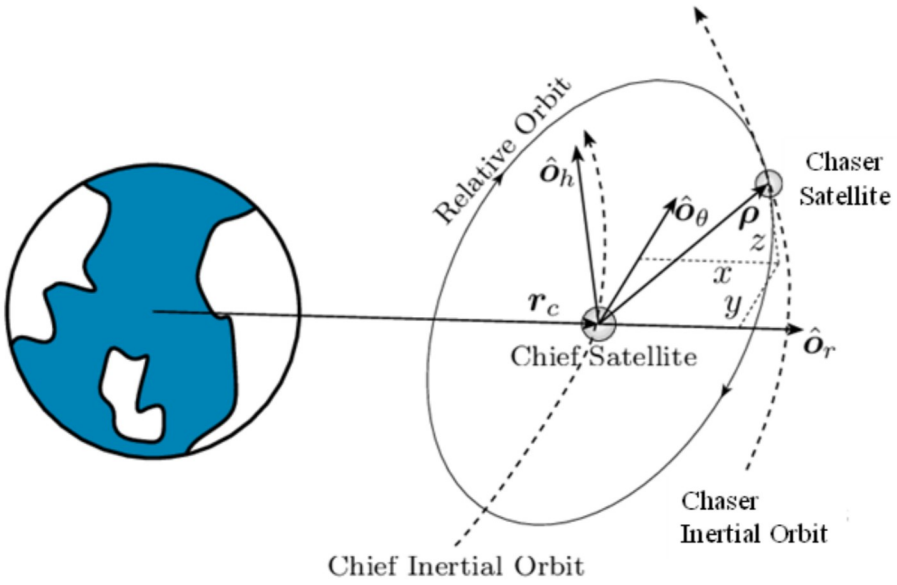

Figure 1 Satellite rendezvous, with chief and chaser satellites. [1]

Spacecraft formation flying is a broad term, and covers several different scenarios, each with their own assumptions and sets of governing equations. The scenario explored in this project is that of spacecraft rendezvous, wherein a spacecraft, known as the chaser, attempts to approach and connect to a second spacecraft, known as the chief. The relative motion between chaser and chief can be described by the Clohessy-Wiltshire equations, shown in Equation 1. These form the basis for the dynamical system investigated in this project. In solving these differential equations, the goal is to drive the relative displacement to zero, resulting in a successful dock between chaser and chief.

$$\begin{aligned} \ddot{x} &= 3n^2x + 2n\dot{y} \\ \ddot{y} &= -2n\dot{x} \\ \ddot{z} &= -n^2z \end{aligned} \tag{1}$$

The CW equations assume that the chief spacecraft moves in its own orbit surrounding a planet or another large mass, with a mean motion equal to n. For this project, a semi-major axis of 7500 km was assumed, which corresponds to an orbit altitude of about 1130 km.

Using the CW equations, the following state-space model was devised (equation 2), where the matrices $\mathbf{A}$ and $\mathbf{B}$ are defined in equation 4. Here, the variable $\mathbf{x}$ represents the state of the spacecraft, containing six elements corresponding to position and velocity, as shown in equation 3. The second term, $\mathbf{Bu}$, represents the evolution of the model in time, taking in an input thrust $\mathbf{u}$, measured in km s^{-2}. The units in use have been km and s.

$$\frac{d\mathbf{x}}{dt} = \mathbf{Ax} + \mathbf{Bu} \tag{2}$$

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \tag{3}$$

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 3n^2 & 0 & 0 & 0 & 2n & 0 \\ 0 & 0 & 0 & -2n & 0 & 0 \\ 0 & 0 & -n^2 & 0 & 0 & 0 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Though investigating the CW equations presents opportunities for improvements to spacecraft formation flying - including, but not limited to, smarter onboard processing decisions, better fuel usage, and quicker flight times - the methods employed in this project can be mirrored and applied to other, similar dynamics problems, and need not be limited to these equations.

2.2 Initial Conditions

For any model represented by a set of differential equations, a valid set of initial conditions is often required in order for solutions to converge and produce repeatable results. Though the aim of this project is to create a controller that is independent of initial conditions, a set of initial conditions had to be selected for comparison's sake. The initial conditions used here were

$$\mathbf{X_0} = \begin{bmatrix} 8.205 \times 10^{-2} \\ 8.16 \times 10^{-1} \\ -3.056 \times 10^{-3} \\ -1.014 \times 10^{-4} \\ -1.912 \times 10^{-4} \\ 9.993 \times 10^{-4} \end{bmatrix}, \tag{5}$$

sourced from a similar formation flying research paper [9]. An initial mass of $m_0 = 750$ kg was assumed [10]. The chaser has converged when the state is brought down from these values to a magnitude of 0.001 km in position and 0.0001 m s^{-1} in velocity (1 m and 0.1 m s^{-1}).

2.3 Control Law

The basis of LQR control are the two matrices $\mathbf{Q}$ and $\mathbf{R}$, which contain covariances for states and inputs respectively. LQR control involves the optimisation of a cost function, $\mathbf{J}$, to which $\mathbf{Q}$ and $\mathbf{R}$ are input matrices. By finding the minimum value of this cost function, one can find the optimal solution to the state-space problem. The minimum value of $\mathbf{J}$ is given by solving the Algebraic Riccati equation, which also outputs the optimal gain matrix, $\mathbf{K}$ - feeding this back into the state-space model in the form of a control law, noted in Equation 6, results in optimal behaviour.

$$\mathbf{u} = -\mathbf{K}\mathbf{x} \tag{6}$$

A requirement of LQR control is that the $\mathbf{Q}$ matrix is positive definite, while the $\mathbf{R}$ matrix must be positive semi-definite. To ensure this, $\mathbf{Q}$ and $\mathbf{R}$ were simplified to diagonal matrices, with each value, or weighting, along the diagonals corresponding to the covariance of a state or input. For example, q_1 in Equation 7 is the weighting for the x state in the state vector $\mathbf{x}$. This gives nine potential optimisation variables to investigate.

$$\mathbf{Q} = \begin{bmatrix} q_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & q_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & q_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & q_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & q_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & q_6 \end{bmatrix}, \mathbf{R} = \begin{bmatrix} r_1 & 0 & 0 \\ 0 & r_2 & 0 \\ 0 & 0 & r_3 \end{bmatrix} \tag{7}$$

The weightings within $\mathbf{Q}$ and $\mathbf{R}$ heavily influence the performance of the controller. Higher values in the $\mathbf{Q}$ matrix generally push the controller to punish large errors in the corresponding states, while higher values within the $\mathbf{R}$ matrix result in the punishment of excessive actuator usage, e.g. thrust in a particular direction. Figure 2 shows the difference in trajectories observed when a single parameter is changed, indicating the importance of selecting the correct values for these weights. Another important characteristic of the LQR method is the focus on the ratios between the weights, rather than the weights themselves - this allows the weights to be varied from 0 to 1 instead of using absolute values.

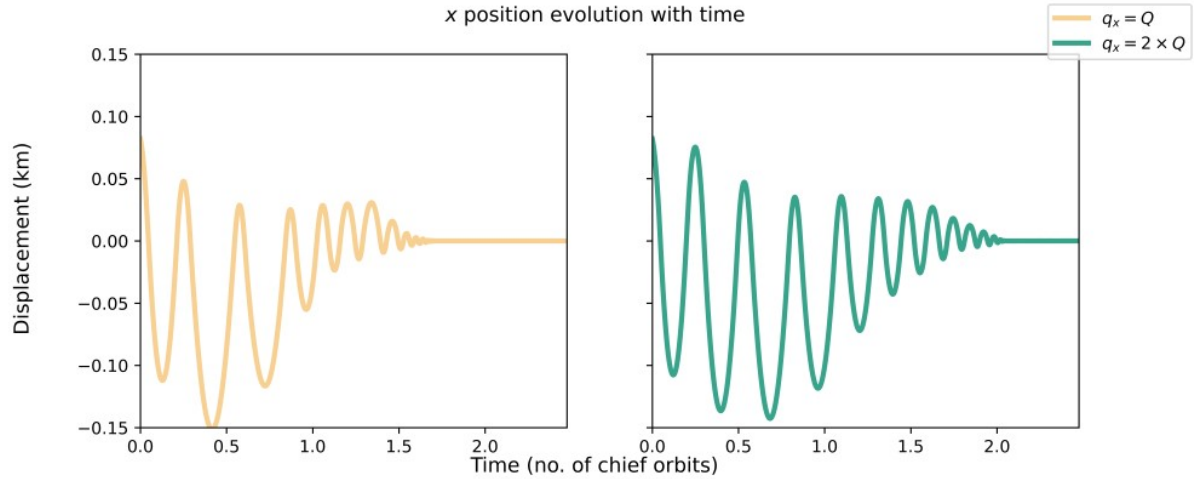

Figure 2 The effect of changing a single $\mathbf{Q}$ parameter. The figure on the left shows the x-trajectory with a nominal weighting, while the figure on the right shows the x-trajectory with double the weighting.

However, to avoid over-complicating the optimisation problem, it was decided to keep the $\mathbf{R}$ matrix constant as a 3×3 identity matrix. This reduced the number of optimisation variables to six, being the six weights recorded in the diagonal of the $\mathbf{Q}$ matrix. It is noted that further work could involve varying the values within $\mathbf{R}$ as well.

2.4 Mass Considerations

In order to apply a more realistic lens to the scenario, and to provide a metric through which the performance of the controller could be measured, the mass of the chaser was included in the state vector. Assuming a constant specific impulse of $I_{\text{sp}} = 1000$ s, and an

initial mass $m_0 = 750$ kg, the following equation was used to quantify the rate of change of the chaser's mass:

$$\dot{m} = -\frac{u}{I_{sp} g_0} m \tag{8}$$

Here, u is the magnitude of the thrust applied, and m is the current mass of the chaser. This change is decoupled from chaser position and velocity, and is calculated independently of the rest of the elements in the state vector. This allows for the calculation of ΔV (as given in equation 9) for a given trajectory and thrust profile. ΔV is a measure of the impulse applied to the chaser in its rendezvous.

$$\Delta V = I_{\text{sp}} g_0 \ln \frac{m_0}{m} \tag{9}$$

2.5 Optimisation and State Dependency

By introducing ΔV as a parameter, steps could then be taken to find the trajectories that minimised this value, saving fuel usage. The optimisation problem was therefore defined as the following:

Find weights such that a diagonal matrix $\mathbf{Q}$ made up of these weights results in an LQR control law with a minimum-ΔV trajectory realisation.

There are many ways to tackle this optimisation problem, two of which will be discussed in this report. The first involves selecting a set of weights to be used throughout the entirety of the chaser's trajectory, integrating from start to finish using the same optimal gain matrix $\mathbf{K}$. This is known as a state-independent controller, as the control weights are independent of the state of the system. This was completed using a Particle Swarm Optimiser (PSO), the methodology behind which is explored in Section 4.

The second method relies on a smarter controller, and changes these weights at selected intervals - in essence, this turns the single-state controller into a state-dependent controller, whereby the weights in matrix $\mathbf{Q}$ change depending on the current state of the system. The main focus of this project, this method was implemented using an RL framework. The methodology and design decisions behind this implementation are detailed in Section 5. By investigating this controller, we aim to investigate whether classical control laws can be improved upon using Reinforcement Learning.

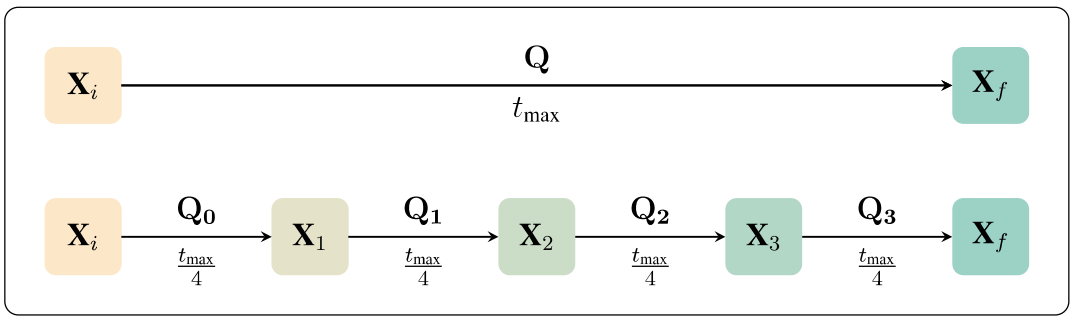

Figure 3 The differences between state-independent (upper flow) and state-dependent (lower flow) control.

3. Software Architecture

This section of the report details some of the base software structure used, including brief definitions of functions and their arguments, libraries used, and design decisions for these.

3.1 Trajectory Integration

To actually solve for the motion of the spacecraft rendezvous, a reliable integrator was required. The integrator used was Python's `solve_ivp`, from the SciPy library. An event function was used in conjunction with this integrator to reduce excessive integration when trajectories became close enough to converge, or when they exceeded a certain outer limit, improving simulation times.

The `lqr()` function from the `control` library was used to generate values for the optimal gain matrix $\mathbf{K}$.

3.2 Gymnasium Environment

The OpenAI Gym library was used as a template to create the environment required for RL. Specifically, the `cartpole.py` environment was chosen due to its similarity in terms of control laws and integration. The exact functions written are detailed in Section 5.1.1.

3.3 Stable Baselines3

Stable Baselines3 was used for the wide range of RL algorithms available. Due to the nature of the dynamics model as well as our own novelty to the subject, a model-free algorithm was chosen. A2C was the first candidates, due to its relative simplicity and wide range of use, as well as the continuous nature of its action space.

3.4 Tensorboard

Tensorboard was used to conveniently display the training logs. This was particularly useful in the case of reward functions. The evolution of the reward during training can be used to evaluate the effectiveness of a reward function before the entirety of the learning period has been completed.

4. State-Independent Controller

This section of the report details the implementation of a state-independent controller. As mentioned in Section 2, the optimisation problem is addressed by selecting the weights of the $\mathbf{Q}$ matrix at the beginning of the trajectories, and keeping these constant through all changes in state. This provides a general benchmark with which the RL-enhanced model can be compared.

4.1 PSO Set-up

In order to find the best possible weights, a Particle Swarm Optimiser (PSO) was used. Initially, MATLAB's built-in `patricleswarm()` function was used, but once a reliable dynamics model had been set-up in Python, a move was made to the `pyswarms` library for the sake of consistency and fair comparison.

PSO works on the basis of a group, or "swarm" of particles within a solution space working to find the minimum realisation of that solution space. By inputting a desired number of dimensions, the PSO algorithm uses these particles and varies along the given dimensions to find the local or global best cost outlined by a cost function. For the case used in this project, the dimensions were simply the six weights in the $\mathbf{Q}$ matrix, varying from 0 to 1.

4.1.1 Cost Function

The following cost function was used for PSO optimisation:

$$C = a \times \Delta V + (1 - a) \times ((1 + x + v) \times 100 + 1000) \tag{10}$$

where C is the total cost, x and v are the magnitudes of the final position and velocity respectively, and a is a Boolean term that is equal to 1 if the chaser converges, and 0 otherwise.

4.2 PSO Results

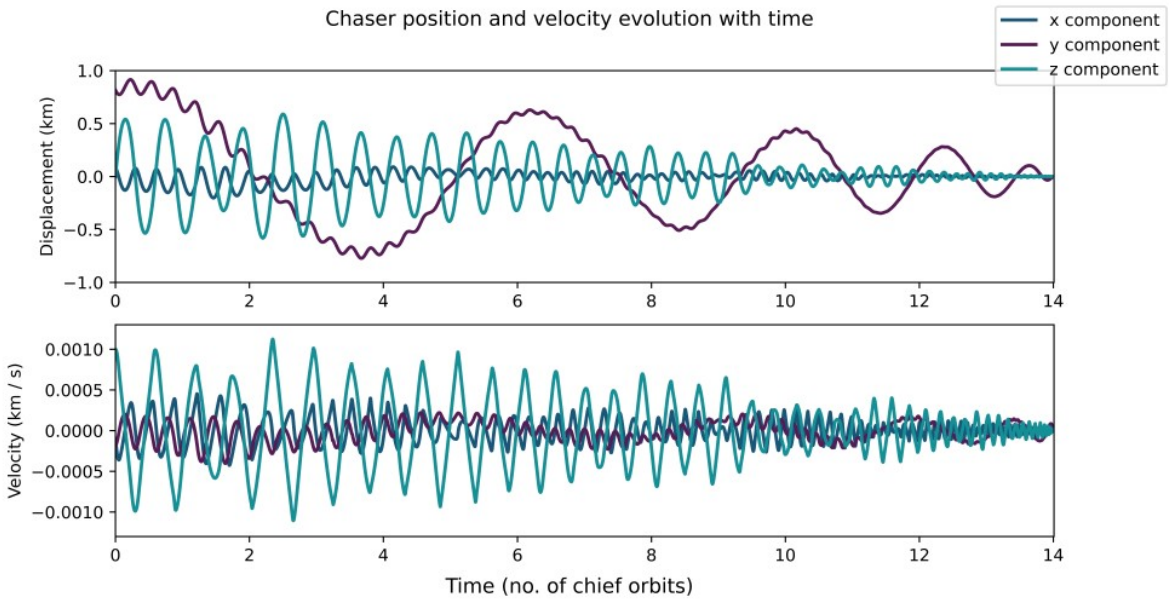

Figure 4 Plots of position and velocity for the PSO optimal solution

Q						ΔV
q_1	q_2	q_3	q_4	q_5	q_6	
0.9538	0.0024	0.2054	0.2359	0.5221	0.6735	90.55

Table 1 Weights and resulting ΔV values for the PSO solution.

The PSO algorithm used produced the following weights and trajectories:

Given the importance of the weights in the Q matrix and their effects on chaser trajectory, it is counter-productive to force these weights to be constant and not vary them given the state of the chaser. This is an area of improvement where we believe Reinforcement Learning can be applied - the following section records the development of a state-dependent controller.

5. State-Dependent Controller

Once the state-independent controller had been set-up, steps could be taken to implement state-dependent architecture. This section details the set up and use of Reinforcement Learning to achieve this.

5.1 RL Set-up

As mentioned previously, Reinforcement Learning makes use of the relationship between an agent and its environment. The agent is given a reward based on the state of the environment. With the aim of maximising this reward, the agent decides the next action, which in turn forms the new state of the system. This feedback loop is depicted simply in Figure 5, while the agent is shown in its wider role in Figure 6.

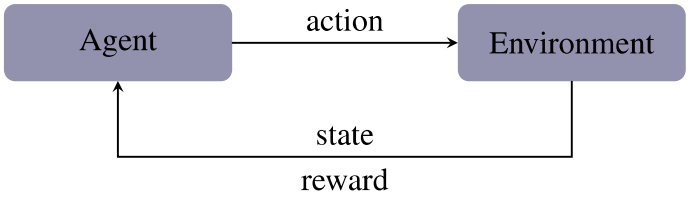

Figure 5 Flowchart depicting the RL process.

5.1.1 Environment

The environment in an RL framework operates using four key functions: `__init__()`, `step()`, `render()`, and `reward()`. As mentioned in Section 3, the Gymnasium library was used as a starting point to create a custom environment with these four functions.

Initialise - `__init__(self, render_mode="human")`

This function initialises the state and sets out the parameters used in the RL framework. The key components of this function are the observation space and the action space.

Initially, the action space was set to have a minimum of 1 and a maximum of 10000, in accordance with values observed during early-stage MATLAB PSO implementation. After reviewing the Stable Baselines3 documentation, this was changed to range between 0 to 1 to reduce harm to learning. However, the conditions of LQR control mean that the Q matrix cannot have any weights equal to zero. In order to circumvent this, the lower limit of the action space was set to a value close to zero, of 0.001, with an upper limit of 1. However,

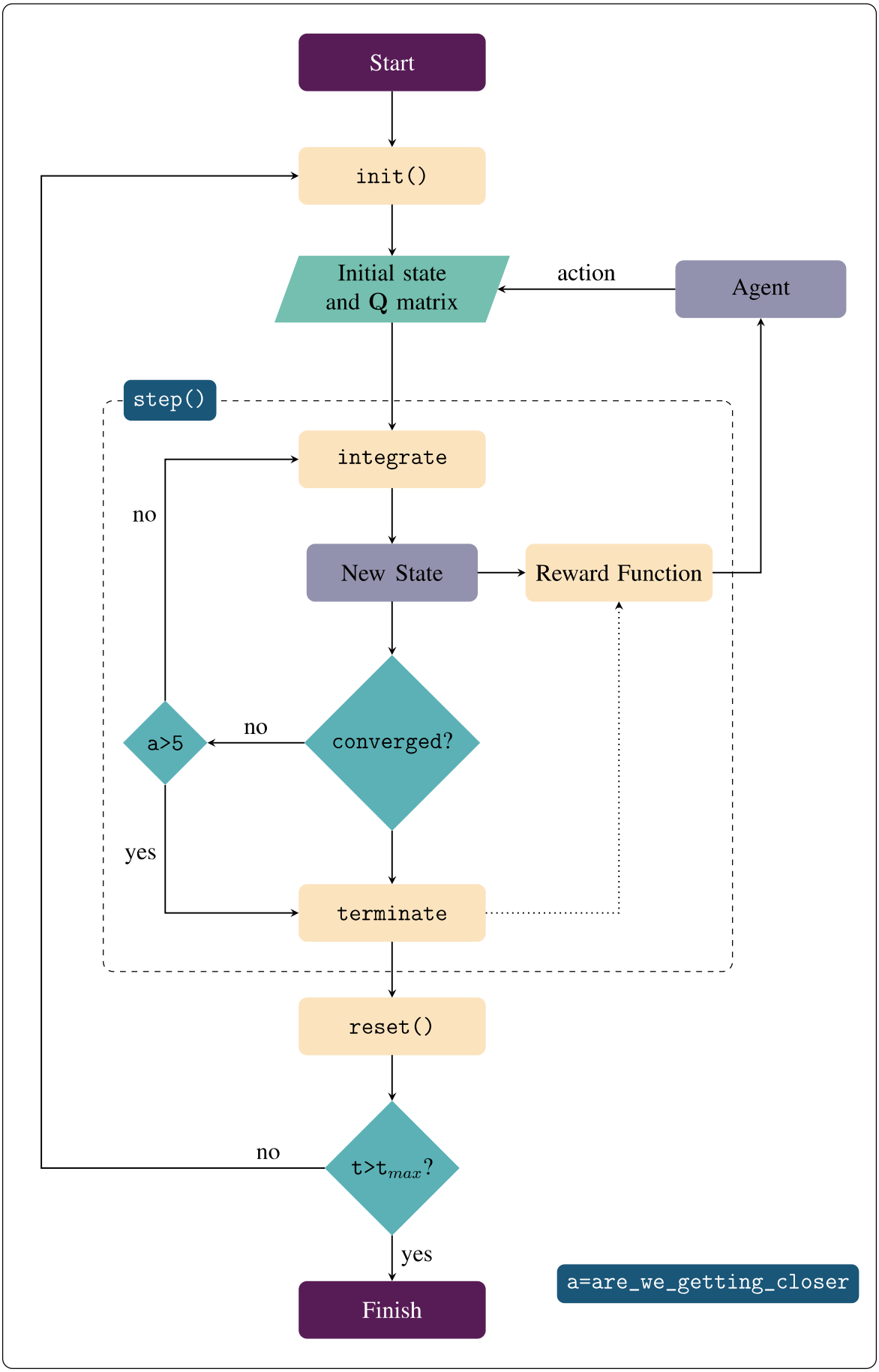

Figure 6 Flowchart depicting the RL process.

during training runs using this action space, the policy would rely heavily on using a weight of 0.001 over others, leading to limited variation in the actions, counterproductive for the aims of this section.

The action space was finally set to alter between a high of -1 and 1 - this was avoided at first because a negative weight does not meet the requirements for the $\mathbf{Q}$ matrix. This was remedied by squaring the weights before using them in the integration method, giving the weights a range of (0,1).

Step - `step(self, action)`

The purpose of the step function in RL is to provide a method by which the agent's actions and current state are used to generate the next state of the system. The action elements are squared before being placed in the diagonal of the $\mathbf{Q}$ matrix, to ensure that the $\mathbf{Q}$ matrix is positive semi-definite. From here, the optimal control matrix is found as described in Section 2 and used to integrate over a given timestep.

The size of this timestep is an important factor. Numerous iterations were made, with step sizes varying from as small as 100 s per integration, to as large as 8000 s. The most successful of these was a timestep of 8000 s, as is described in Section 5.3.

After integration, a Boolean variable `converged` was used to determine whether or not the state had converged, doing so by comparing the magnitude of position and velocity to a pre-determined set of tolerances. The tolerances used are consistent with earlier definitions of the problem - a distance of 1 meter for position, and a velocity of 0.1 m s^{-1}.

If the chaser's orbit evolved beyond five times the magnitude of the initial position, i.e. `are_we_getting_closer` > 5, the `terminated` flag was set to true - this indicates to the model that the episode has terminated, and prevents excessive integration calculations.

Render - `render()`

The render function is an optional function that generates a visual output based on the state of the system. For the purposes of this project, the render function was deemed unnecessary. Actions and rewards were simply printed to the command line in real time, allowing for the user to keep a close eye on the evolution of the state with time.

Reset - `reset(self)`

The reset function is called every time the `terminated` flag is set to true. It sets the state back to the initial conditions and also resets the number of time-steps to zero, while indicating to the model that an episode has concluded.

5.1.2 Hyper-parameters

Along with these four functions, the behaviour of the policy is heavily influenced by the value of certain hyper-parameters. These hyper-parameters are selected before each learning period.

Learning rate, α, is the rate at which the policy updates the actions of the agent while learning. If this value is too high, the action space is not utilised well enough, resulting in poor results when testing the model. If this value is too low, the learning can take excessive amounts of time. A low learning rate was opted for, due to its effectiveness in searching the action-space.

The discount factor, γ, is a parameter that determines the influence of future rewards on the current reward. In the end, the final hyper-parameters decided on were a learning rate of 0.0007 and a discount factor of 0.99.

5.2 Reward Function

The reward function is called within the step function, and is one of the most important components of the RL framework. Thus, it had to be carefully formulated in order to produce a comprehensive RL model. The training for most of the reward functions used in this project was completed in two stages, focusing first on convergence, then on optimising for ΔV.

Convergence

The initial problem that the RL framework had to solve was that of convergence - there is no point in a low ΔV value if the chaser fails to converge in the first place. Therefore, a large component of the reward function awarded the chaser when it got closer to the chief and slowed down (essentially, when the state vector approached a zero vector). This was realised in the reward function by means of three variables:

- `are_we_getting_closer` - an aptly named variable that measures the ratio of the current absolute position to the absolute position of the initial conditions. A value below one indicates that the chaser is closer than initially.
- `position_improvement` - similar to `are_we_getting_closer`, with the only difference being that the current absolute position is compared to the previous absolute position, not the position of the initial conditions.
- `velocity_improvement` - similar to `position_improvement`, with the only difference being that the current absolute velocity is compared to the previous absolute velocity.

In essence, `are_we_getting_closer` measures overall change, while `position_improvement` and `velocity_improvement` measure transient change. Along with these variables, a component of the reward function awarded the model for keeping the chaser in a valid position for longer, in order to prevent the step function from terminating when `are_we_getting_closer` became too high in value. Finally, a terminal reward element was added when the agent was able to make the chaser converge.

Adjusting for ΔV

This was completed by adding a subtractive ΔV element to the reward function. Depending on the values used for other parameters in the reward function, this had to often be multiplied by a factor in order to increase or decrease the impact of ΔV accordingly.

5.3 State-Dependent Results

With all of the previously mentioned architecture set-up, the model was ready to be trained and analysed. Several iterations of the reward function were tried, with various combinations of parameters employed. However, only one set was met with relative success. These results are from using an action space ranging from 0.001 to 1, as described in Section 5.1.1. The effects of this can be seen in Table 2.

Timestep	Q						ΔV
	q_1	q_2	q_3	q_4	q_5	q_6	
1	0.246	0.001	0.001	0.849	0.752	1	8.000
2	0.001	0.001	0.001	0.696	0.406	1	6.388

Table 2 Weights and resulting ΔV values for Set 1.

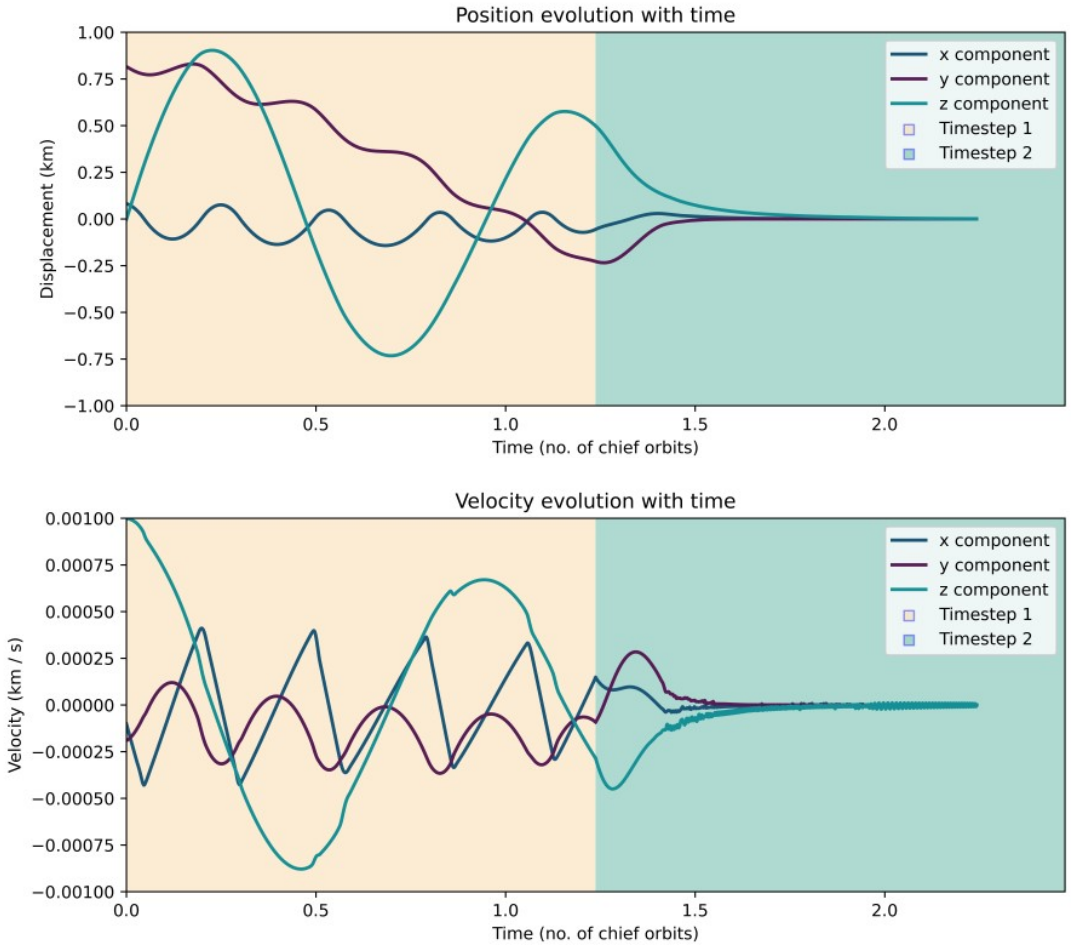

Figure 7 Plots of position and velocity evolution with time for the state-dependent model.

The results summarised in Table 2 are not weights provided by the model during testing. When tested deterministically, the model is successfully able to make the chaser converge, but it does so without changing the weights. In essence, the model is a state-independent controller.

A temporary way around this was found by analysing the training logs of the model, where the weights were routinely changed with state. During testing, the actions of the agent were stored in a separate file. Using a quick script, these actions were tested independently, and the set of actions resulting in the lowest ΔV was extracted, giving the values presented in

Table 2. While this does not directly meet the requirements set out at the beginning of this project, it provides an idea of the level of success to be found with more successful training.

If the model is tested stochastically instead of deterministically, it performs as expected and produces weights that change with the state. However, stochastic models rely too heavily on introduced randomness, and are often not optimised as well as deterministic models.

Reward Function

For this set, the following reward function was used, with the parameters summarised in Table 3.

$$R = 1 - a^{0.4} + p_d \times p_i^{-p_d \times 0.4} + v_d \times v_i^{-v_d \times 0.4} - \Delta V + \frac{t}{n} + 10 \times c \tag{11}$$

This reward function penalises steps that bring the chaser further away from the initial conditions. The terms in the reward function corresponding to `position_improvement` and `velocity_improvement` are additive when either of these variables are less than 1, incentivising moving closer or slower.

Term	Definition
R	reward
a	`are_we_getting_closer`
p_i	`position_improvement`
p_d	1 if $p_i < 0$, -1 otherwise
v_i	`velocity_improvement`
v_d	1 if $v_i < 0$, -1 otherwise
t	no. of timesteps
n	max no. of timesteps in one ep
c	1 if `converged`, 0 otherwise

Table 3 Definitions of terms in Equation 11

Other successful models with smaller step sizes were found, but like this model, they too were unable to produce state-dependent actions. There are probably several reasons for this, but we believe that the largest contributors are

1. the action space used when training this model, and
2. the hyperparameters associated with the training (specifically learning rate)

Difficulty in optimising for ΔV

There is some discussion to be had about the validity of this model as a ΔV-optimal solution. The thrust model for this project works on the basis of saturation. If a calculated value for u, the input thrust to the chaser, is above the permissible u_{max} in magnitude, the thrust applied is simply u_{max}, broken down by direction.

Due to the nature of the calculations, values of u before saturation were found to be much higher than the value of u_{max}. Figure 8 shows the distribution of one such set of u values. A major proportion of the values calculated for u is below the u_{max} line. This implies that the chaser is almost always at maximum saturation.

Per Equation 9, ΔV is heavily dependent on this value of thrust and the ratio of masses before and after thrusting. Given that a low value of thrust results in a low mass loss rate, (Equation 8) this ratio of masses is very close to 1. The outcome of all of this is that ΔV is rendered almost constant for a given length of time, regardless of the value of u. For Set 1, this was around 8 m s^{-1} over a timestep of 8000 s, assuming a u_{max} of $1e-6$ km s^{-2}.

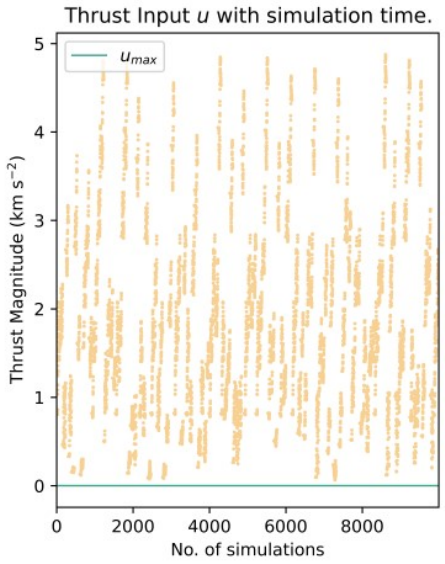

Figure 8 Calculated thrust magnitudes for an example training run.

Constant values for ΔV result in a plateau for the RL algorithm - the agent is unable to decide which actions result in a more optimal ΔV value since there is gradient indicating this direction.

This effectively turns the ΔV optimisation problem into a ΔT problem: if the ΔV for each timestep is a constant number, to minimise it we simply need to minimise the time of flight. This was also used when searching the training data for the best ΔV results. One possible solution to find a more legitimate ΔV-optimal model is to train the model further - exploration of the action space may result in weights that give a u lower than u_{max}. However, employing even lower thrust values can often result in difficulty in getting the chaser to converge.

Another method to address this problem is to add a separate upper limit for u. For example, from Figure 8, the upper value for u may be chosen to be 5. Dividing the value of u by this limit, before saturating with u_{max} results in less saturation and more variation in the values of ΔV. However, this solution was employed very late into the project, and as such, no meaningful results were able to be found in time.

Summary

The results show a total ΔV of 14.39 m s^{-1}. Comparing this to the PSO algorithm's 90.55 m s^{-1}, this is an improvement of 84%.

Though the validity of this improvement is questionable due to the methods employed to get these weights, the low ΔV value indicates that it is a step in the right direction in creating a successful model. The configuration employed in Set 1 is evidence of the potential of RL in producing a ΔV-optimal solution, but requires a lot of work in tuning certain parameters. The recommended steps to improve this framework are detailed in Section 6.

6. Conclusions

This report is the culmination of a year-long investigation into the improvement of classical control laws using Reinforcement Learning. After employing the use of several frameworks and methods, we can conclude that the implementation of a state-dependent controller is a viable outcome, subject to a complete reward function and correctly tuned parameters.

Despite wavering success, the results in Section 5.3 indicate that a highly optimal solution is possible. In terms of the algorithm used, this project lays a foundation for the use of an A2C policy in an LQR scenario. There is evidence to suggest that, with the correct tuning and level of training, this policy can be used to implement a state-dependent controller.

6.1 Recommended Next Steps

While substantial work has been completed toward achieving the goals stated in this report, there are many further steps to be taken to better meet or exceed these goals.

- Improvement of the state-dependent aspect of the policy - while the current policy is able to find weights that result in convergence and optimal values for ΔV, it is not state-dependent by itself. This may be attributed to limited testing and requires further exploration of policy hyper-parameters.
- Better optimisation for ΔV - this can be achieved by implementing the upper limit for u as discussed in Section 5.3. In doing so, this introduces greater variation to ΔV, allowing the algorithm to more easily find weights that optimise it.
- Implementing a time-optimal policy - the reward function can be altered to find a time-optimal policy that minimises the time of flight instead of ΔV.

References

[1] H. Schaub, "Spacecraft relative orbit geometry description through orbit element differences," 01 2002.

[2] H. Schaub and J. L. Junkins, *Analytical Mechanics of Space Systems*. Reston; Virginia: American Institute of Aeronautics and Astronautics, 2018.

[3] H. J. Holt, "Trajectory design using lyapunov control laws and reinforcement learning," Ph.D. dissertation, University of Surrey, 2022.

[4] T. B. Prasad, L. B. and H. O. Gupta, "Optimal control of nonlinear inverted pendulum system using pid controller and lqr: Performance analysis without and with disturbance input," *Wind Energy*, vol. 11, no. 6, pp. 661–670, 2014.

[5] O. Omidvar and D. Elliott, "Neural systems for control," 03 1997.

[6] K. Hovell and S. Ulrich, "Deep reinforcement learning for spacecraft proximity operations guidance," *Journal of Spacecraft and Rockets*, vol. 58, no. 2, pp. 254–264, 2021.

[7] ——, "Laboratory experimentation of spacecraft robotic capture using deep-reinforcement-learningâbased guidance," *Journal of Guidance, Control, and Dynamics*, vol. 45, no. 11, pp. 2138–2146, 2022.

[8] W. Caarls, "Deep reinforcement learning with embedded lqr controllers," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 8063–8069, 2020.

[9] M. Sabatini, D. Izzo, and G. Palmerini, "Analysis and control of convenient orbital configuration for formation flying missions," vol. 124, 01 2006, p. 8.

[10] [Online]. Available: https://science.nasa.gov/mission/dawn/technology/spacecraft/