Waypoint Control of a Quadrotor Carrying a Slung Payload Using Reinforcement Learning

Nourah Al Saud

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the
Master of Applied Science in Mechanical Engineering

Faculty of Engineering
Department of Mechanical Engineering

© Nourah Al Saud, Ottawa, Canada, 2024

Abstract

In recent years, global e-commerce has experienced a significant rise. This surge has spurred a demand for efficient delivery services, specifically in the last-mile segment, which faces considerable challenges. Implementing Uninhabited Aerial Vehicles (UAVs), especially with the slung payload system, offers effective maneuverability and flexibility to address this issue. That being said, controlling a drone with a slung load poses unique challenges such as real-time adaptability and unpredictable dynamics. The objective of this work was to develop an on-board model free Reinforcement Learning (RL) controller capable of achieving precise waypoint tracking for a drone with a slung load while ensuring stability and accuracy in the presence of payload-induced perturbations. A Temporal Difference (TD) RL control algorithm was developed to enable the drone to achieve waypoint tracking capabilities through commanding changes in attitude. This algorithm was trained in simulation using a developed planar model of the drone-payload system. The trained controller was implemented on a prototype quadrotor in an indoor testing environment with an integrated motion capture system in charge of relaying position and velocity data in real time to the aircraft. Waypoint tracking was applied to the quadrotor using two separate RL agents to achieve full control in the horizontal plane. The proposed fully on-board controller was able to lead the aircraft to the desired goal successfully, closely matching simulation results. In both sets of results there was a small steady state error that could be mitigated in the future through step size optimisation along with additional training.

Acknowledgements

I would like to express my deepest gratitude to the many individuals who have contributed to the realization of this project. To my advisor, Dr, Eric Lanteigne, your guidance and unwavering support have been the cornerstone of this journey. Your insights, encouragement, and dedication to my growth have been invaluable.

A heartfelt thank you to my family and friends for their boundless encouragement and understanding. Your belief in me sustained me through the highs and lows, and I am grateful for the love and strength you've provided.

I extend my appreciation to the University of Ottawa for fostering an environment of academic excellence and providing the resources necessary for this endeavor. The collaborative spirit and intellectual stimulation have been instrumental in shaping my perspective.

To my dear friends Ahmed and Eleni and Osama, thank you for making the long days in the lab full of laughs and amazing memories that I will cherish for life. Thank you for the camaraderie, shared insights and collaborative efforts. Your diverse perspectives have enriched this project and made the journey truly memorable.

A special thank you to Fatimah. Your unwavering support and encouragement have been a constant throughout not only the duration of this project but in every aspect of my life. Your steadfast presence and uplifting spirit have been a source of strength.

Lastly, to all those whose names may not appear here but who have played a role, no matter how small, in this endeavor − I am profoundly thankful. This achievement is a collective effort, and I am humbled by the support and contributions of each person who has crossed my path.

This work stands as a testament to the power of collaboration, mentorship, and the support of loved ones. Thank you, from the depths of my heart.

This thesis is dedicated to my parents for their endless love and encouragement, and my little brother for always believing in me and motivating me to be better everyday.

Learning is the only thing the mind never exhausts, never fears, and never regrets.

- Leonardo da Vinci

Contents

Abstract **ii**

Acknowledgements **iii**

1 Introduction **1**

2 Literature Review **4**

2.1 Nonlinear Control Methods 5

2.1.1 PID and PD Control 5

2.1.2 Feedback Linearisation (FBL) 8

2.1.3 Optimal Control 8

2.2 Modelling Methods 14

2.2.1 Euler-Lagrange Modelling Method 14

2.2.2 Newton-Euler Modelling Method 15

2.2.3 Modelling Assumptions for a UAV Carrying a Slung Load 16

2.3 Implementation on the Physical System 18

3 Quadcopter-Payload Dynamic Model **19**

3.1 Frames of Reference & The Rotation Matrix 20

3.1.1 Quadcopter Position and Rotation Matrix 20

3.1.2 Payload Position and Rotation Matrix 21

3.2 Quadcopter Operation Principles 24

3.3 Euler-Lagrange Model Formulation 28

3.3.1 Modelling Assumptions 28

3.3.2 Euler's Kinematic Equations 28

3.3.3 Equations of Motion 30

3.3.4 Simplified Equations of Motion For Training 33

4 Control Algorithm Development **35**

4.1 Reinforcement Learning Review [55] 36

4.1.1 Elements of Reinforcement Learning 37

4.1.2 Finite Markov Decision Processes [56] [57] 40

4.1.3 Temporal Difference Learning 46

4.1.4 Approximate Solution Methods (Continuous State and Action Space) . . 49

4.2 Waypoint Tracking Algorithm 63

4.2.1 Actor - Critic $\boldsymbol{TD(0)}$ Algorithm for Waypoint Tracking 63

5 Experimental Setup **66**

5.1 Experimental Platform 67

5.2 Testing Area Setup - Motion Capture 70

5.3 Data Transfer 72

5.3.1 Receiving and Processing Motive's Data 72

5.3.2 Sending Attitude Commands to the AutoPilot 75

6 Results and Discussion **77**

6.1 Simulated Results 78

6.1.1 Control Input Signal Modelling 78

6.1.2 Aircraft Response Modelling 79

6.1.3 Reward Function 83

6.1.4 Simulated Quadcopter Training & Waypoint Tracking 87

6.2 Real Quadcopter Waypoint Tracking Results 100

7 Future Improvements and Conclusions **109**

7.1 Conclusions 110

7.2 Future Improvements 110

A Actor Critic Neural Networks **118**

B Constructing The Rotation Matrix **120**

C Fourth Order Runge Kutta ODE Solver **122**

D Environment Classification for Reinforcement Learning 125

D.1 Fully Observable vs. Partially Observable: 125

D.2 Deterministic vs. Stochastic: 126

D.3 Static vs. Dynamic 127

D.4 Discrete vs Continuous 127

D.5 Episodic vs. Sequential 127

E Markov Decision Process - The State Transition Matrix 129

F Dynamic Programming (DP) 131

F.1 Policy Evaluation 131

F.2 Policy Improvement 131

F.3 Policy Iteration Algorithm 133

F.4 Value Iteration Algorithm 133

F.5 Asynchronous Dynamic Programming 133

G Monte Carlo Methods 136

G.1 Monte Carlo Prediction 136

G.2 Monte Carlo Control 137

H Auto-Tuning ArduPilot's PID Controllers 140

I PID Input Signal Modelling 141

List of Tables

6.1 Table of the step sizes, standard deviation and discount factor used for the reinforcement learning controller 83

List of Figures

1.1 Possible configurations for payload delivery [6]. 2

2.1 Diagram of the control architecture of the RCAC and PD trajectory controller proposed by [15] for a UAV-payload system 7

3.1 The quadcopter body frame $\{B\}$ shown relative to the inertial body frame $\{I\}$. The position of the quadcopter is measured by the position vector $\boldsymbol{p}_Q$ from the inertial to the body frame. 20

3.2 The Euler sequence of rotations to convert from the body frame $\{B\}$ to the inertial frame {I}. Start with rotating about the z-axis, then the new y-axis and then the final x-axis to achieve the resultant orientation. 21

3.3 The payload fixed frame $\{S\}$ 22

3.4 The angles used to measure the position of the cable relative to the inertial frame. α denotes the swing angle of the cable measured from the z_I axis and β denotes the rotational position of the cable measured about the z_I axis. Both angles are measured using the right hand notation. 22

3.5 Possible quadcopter configurations a) +-configuration b) $\times$-configuration 24

3.6 A quadcopter rolling by an angle ϕ about the inertial x-axis x_I to move in the $-y$ direction using the vector of the total thrust force T_{xy} 25

3.7 Figure showing changes in forces required to achieve the three types of moments a quadcopter can experience. a) A quadcopter rolling by an angle ϕ by increasing forces f_1 and f_4 and decreasing forces f_2 and f_3. b) A quadcopter pitching by an angle θ by increasing forces f_3 and f_4 and decreasing forces f_1 and f_2. c) A quadcopter yawing by an angle ψ by increasing forces f_1 and f_3 and decreasing forces f_2 and f_4 26

3.8 The modified angles used to measure the position of the cable relative to the inertial frame to prevent the occurrence of a singularity at the equilibrium. α denotes the swing angle of the cable measured from the x_I axis and β denotes the rotational position of the cable measured about the x_I axis from the $-z_I$ axis. Both angles are measured using the right hand notation. 31

3.9 Figure showing the alignment of frames $\{S\}$ and $\{I\}$ at $\alpha = 90^{\circ}$ and $\alpha > 90^{\circ}$ . . 32

3.10 Figure of the planar model of the quadcopter in the $X - Z$ plane. 33

4.1 Figure of the path the quadcopter must take to reduce the x position error. Here, the x position goal is represented using a yellow star. 36

4.2 Reward values around the vicinity of a goal state for a self driving car. 38

4.3 Two one-way paths that a self driving car can take to reach its goal. The red path has a higher value than the blue path. 39

4.4 Figure showing the action distributions dictated by policy π when starting from state s. 44

4.5 Figure showing the state distribution dictated by system dynamics $\mathcal{P}$ when starting from state s and taking action a. 44

4.6 Comparison of the effect of the training duration and step-size choice α on a Fourier basis function approximation ($n = 15$). 52

4.7 Fourier series approximation of the function shown in Figure 4.6 with an excessively step size $\alpha = 1.5 \times 10^{-2}$ showing over and under shoot of the actual function. 53

4.8 Fourier series approximation of a step function with varying order n. 54

4.9 Comparison of the affect of the Fourier order n on the function approximation after 6×10^4 updates. 57

4.10 Figure of normal distributions with varying mean μ and standard deviation σ values. 61

5.1 Picture of the quadcopter's components without the propellers or protective case. 67

5.2 3D rendition of (a) vibration dampening base and (b) the full autopilot stack enclosed within the protective case 68

5.3 Picture of the fully assembled quadcopter 68

5.4 Figure of the wiring of Raspberry Pi, Navio2, the power module, the ESC, the motors, and the battery. 69

5.5 Pictures of (a) The confined testing area in the robotics laboratory (b) The OptiTrack cameras. 70

5.6 (a) Photo of the markers positioned on the quadcopter's case (b) Figure of how OptiTrack interpreted the quadcopter. 70

5.7 Schematic of the socket programming data transfer from the Motive software to the RL controller and autopilot on the UAV. 72

5.8 Table of the bytes of data sent from motive and their corresponding information for bytes 0 to 95 to illustrate what data is being extracted to be sent to the controller. 74

5.9 Figure showing the alignment of the UAV's frame once rotated $\{Q\}$, the MOCAP frame$\{M\}$ and the ArduPilot frame $\{A\}$ in the testing area. 75

6.1 Figure comparing the ArduPilot input signal with a time constant of zero to a step signal for a 20 degree roll command. 79

6.2 Figure showing the simulated command signals compared to the actual ArduPilot command signals. 80

6.3 Figure showing a comparison between transfer functions created using the actual and simulated input. 80

6.4 Figure showing the simulated roll response of the aircraft achieved through rearranging the validated transfer function. 82

6.5 Comparison of the performance of the simulated quadcopter following $1,000$ episodes of training with reward functions with a c constant of (a) 1. (b) 5. (c) 15. 84

6.6 Comparison of the performance of the simulated quadcopter following $5,000$ episodes of training with reward functions with a c constant of (a) 1. (b) 15. . . 84

6.7 Comparison of the value function approximation and performance of a quadcopter with a reward function with $c = 200$ 85

6.8 Figure comparing the waypoint tracking performance of a quadcopter after $2,000$ episodes of training (a) without any positive reward. (b) with the introduction of positive reward when the error is < 0.2m. 86

6.9 Figure showing the (a) Performance (b) Commanded roll of the simulated quadcopter (c) The value function estimate and (d) The policy's mean following $1,000$ episodes of training. 88

6.10 Figure showing the (a) Performance (b) Commanded roll of the simulated quadcopter (c) The value function estimate and (d) The policy's mean following $10,000$ episodes of training. 89

6.11 Figure indicating the updated portion of the (a) $\hat{V}$ (b) μ after $10,000$ episodes of training 90

6.12 Figure presenting the rise times of the controller after (a)$10,000$ (b)$25,000$ episodes of training. 90

6.13 Figure illustrating the (a) TD error and (b) Return for $30,000$ episodes of training. 91

6.14 Figure illustrating the (a) Performance and (b) Commanded roll of the simulated quadcopter following the decrease in the standard deviation. 91

6.15 Figure showing the (a) Performance (b) Commanded roll of the simulated quadcopter (c) The value function estimate and (d) The policy's mean following $25,000$ episodes of training. 92

6.16 Figuring illustrating the split of the state space with labeled numbers used in the text. 93

6.17 Figure illustrating the chosen value function split into the quadrants and subquadrants defined in Figure 6.16 from (a) an isometric view. (b) a top view. . . 94

6.18 (a) Figure illustrating the areas of unexpected behaviour within the value function. The areas encircled in white show the complex learning capabilities of the RL agent and how it reflects basic intuitive reasoning. The enclosed areas in black indicate unexpected behaviour. (b) Figure showing 'ringing' behaviour occurring at the edges of low value updated states. 95

6.19 Figure showing the trained policy's mean μ split into the quadrants and sub-quadrants defined in Figure 6.16 from (a) an isometric view. (b) a top view. . . 96

6.20 (a) Figure illustrating the areas of unexpected behaviour within the action selection. (b) Figure showing a clearer view of the unupdated portion of the action selection space. 97

6.21 (a) Figure showing the error and action selection when starting at (a,b) positive e_x and positive $\dot{x}$. (c,d) negative e_x and negative $\dot{x}$. 98

6.22 Figure illustrating the oscillatory behavior present in the action selection due to strong jumps in action choices paired with the Fourier basis approximation method. 99

6.23 Figure illustrating the x_M goals chosen for Tests 1 and 2. 101

6.24 Figure comparing the actual and simulated performance of the quadcopter in Test 1 (a) Error in the $x_M(=-y_Q)$ axis. (b) ϕ command produced by the controller. 102

6.25 Figure showing the drift of the quadcopter in the y_M axis in Test 1. 102

6.26 Figure comparing the a PID controller to the RL controller. 103

6.27 Figure comparing the actual and simulated performance of the quadcopter in Test 2 (a) Error in the $x_M(=-y_Q)$ axis. (b) ϕ command produced by the controller. 104

6.28 Figure comparing the actual and simulated performance of the quadcopter in Test 2 (a) Error in the $y_M(=x_Q)$ axis. (b) θ command produced by the controller. 104

6.29 Figure illustrating the steady state error in the x_M axis of Test 2. 105

6.30 Figure illustrating the x_M goals chosen for Test 3. 105

6.31 Figure comparing the actual and simulated performance of the quadcopter in Test 3 (a) Error in the $x_M(=y_Q)$ axis. (b) ϕ command produced by the controller. 106

6.32 Figure comparing the actual and simulated performance of the quadcopter in Test 3 (a) Error in the $y_M(=x_Q)$ axis. (b) θ command produced by the controller. 106

6.33 Figure illustrating the (x_M, y_M) goals chosen for Test 4. 106

6.34 Figure comparing the actual and simulated performance of the quadcopter in Test 4 (a) Error in the $x_M(=-y_Q)$ axis. (b) ϕ command produced by the controller. 107

6.35 Figure comparing the actual and simulated performance of the quadcopter in Test 4 (a) Error in the $y_M(=x_Q)$ axis. (b) $-\theta$ command produced by the controller. 107

A.1 The architecture of a feedforward neural network where each node of a layer is connected to every node in the succeeding layer. 119

B.1 Illustration of the rotations of the quadcopter frame 120

D.1 An illustration of card game mid-play as a representation of a partially observable state space for a card game playing agent. 126

D.2 Figure showing the possible states that could result from a car attempting to turn into the right-most lane in icy conditions. 127

E.1 Figure of the Markov chain and state transition probability matrix of a simple mars rover that can go left or right [57]. 130

H.1 Figure of the twitching behaviour of the roll of the UAV during the autotune process. 140

I.1 Figure showing the d variable value of the sigmoid function 142

Nomenclature

$\mathcal{A}$	Action space	
$\{B\}$	Quadcopter body fixed frame	
C_D	Coefficient of drag	
C_L	Coefficient of lift	
G	Return of an episode	
H_0	Angular momentum	kgm^2/s
$\{I\}$	Earth fixed frame	
j	Scalar performance measure	
L	Lagrangian of a system	J
$\mathcal{P}$	System dynamics	
$\boldsymbol{p}_Q$	Position vector of the quadcopter measured in the inertial frame {I}	m
$\boldsymbol{p}_L$	Position vector of the payload measured in the inertial frame {I}	m
Q	State-action value function	
R	Rotation matrix	
$\mathcal{R}$	Reward function	
S	Surface area of the blade	m^2
$\mathcal{S}$	State space	
$\{S\}$	Payload body fixed frame	
T	Kinetic energy of the system (In Sections 2.2.1 and 3.2)	J
T	Total thrust generated by all 4 motors	N

V	Potential energy of the system (In Sections 2.2.1 and 3.2)	J
V	State value function	
a	Tangential acceleration	ms^{-2}
d_i	Drag force generated by the ith motor	N
f_i	Thrust force generated by the ith motor	N
l	Horizontal distance between the centers of the motors	m
n	Fourier basis order	
r_t	Reward at time t	
s_t	State of the system at time t	
v	Tangential velocity	$\mathrm{m\,s}^{-1}$
$\boldsymbol{w}$	Value function weight vector	
x	Distance of the quadcopter along the x-axis of the inertial frame t	m
x_L	Distance of the payload along the x-axis of the inertial frame t	m
y	Distance of the quadcopter along the y-axis of the inertial frame t	m
y_L	Distance of the payload along the y-axis of the inertial frame t	m
z	Distance of the quadcopter along the z-axis of the inertial frame t	m
z_L	Distance of the payload along the z-axis of the inertial frame t	m

Greek Symbols

Ω_i	Angular velocity of the blades on the ith motor	$\mathrm{rad\,s}^{-1}$
α	Swing angle of the payload about the inertial y-axis	rad
α_μ	SGA step size for the weight updates of the policy	
α_V	SGD step size for the weight updates of the value function	
β	Swing angle of the payload about the inertial x-axis	rad
δ	Temporal difference error	

γ	Discount factor	
ϵ	ϵ-greedy probability of selecting a random action	
θ	Pitch of the quadcopter	rad
θ_μ	Weight vector of the policy	
λ	Length of the payload suspension cable	m
μ	Normal distribution mean value	
$\boldsymbol{\zeta}$	Attitude of the quadcopter relative to the inertial frame $\{I\}$	rad
π	Reinforcement learning policy	
ρ	Fluid density	$\mathrm{kgm^{-3}}$
σ	Normal distribution standard deviation	
τ_i	Reaction torque generated by the ith motor	N m
ϕ	Roll of the quadcopter	rad
$\boldsymbol{\chi}$	Features vector	
ψ	Yaw of the quadcopter	rad
$\boldsymbol{\omega_B}$	Angular velocity vector of the quadcopter about the x, y and z axes of the body-fixed frame $\{B\}$	$\mathrm{rad\,s^{-1}}$

List of Abbreviations

CCW	Counter ClockWise
CW	ClockWise
DP	Dynamic Programming
DPG	Deterministic Policy Gradient
ESC	Electronic Speed Controller
FBL	FeedBack Linearisation
GCS	Ground Control Station
IMU	Inertial Measurement Unit

LQR	Linear Quadratic Regulators
LTI	Linear Time Invariant
MC	Monte Carlo
MDP	Markov Decision Process
ML	Machine Learning
MPC	Model Predictive Control
MP	Markov Process
MRP	Markov Reward Process
NED	North-East-Down
NN	Neural Network
NWU	North-West-Up
ODE	Oridinary Differrential Equation
OS	Operating System
PBC	Passivity Based Controller
PD	Proportional-Derivative
PID	Proportional-Integral-Derivative
PPN	Pure Proportional Navigation
PPO	Proximal Policy Optimization
PVTOL	Planar Vertical Take off and Landing
PWM	Pulse Width Modulation
RCAC	Retrospective Cost Adaptive Controller
RK4	Fourth Order Runge-Kutta
RL	Reinforcement Learning
SARSA	State-Action-Reward-State-Action
SGA	Semi-Gradient Ascent
SGD	Semi-Gradient Descent

TD	Temporal Difference
UAV	Uninhabited Aerial Vehicle
UDP	User Datagram Protocol
VFA	Value Function Approximation

Chapter 1

Introduction

Recent years have experienced a notable surge in e-commerce world-wide. In 2020, countries like the United States, the United Kingdom, China, India, Germany, Japan, Spain and France saw a two to five fold increase in the growth rate of their e-commerce shares when compared to the rate prior to the COVID-19 pandemic [1]. Following the worldwide trend, the Canadian e-commerce shares percentage increased from 3.9% in 2019 to 6.2% in 2022 [2]. Upon lifting the health restrictions, the e-commerce shares initially declined from their peak but have since stabilised at a higher value than pre-pandemic shares [2]. With this permanent boost in e-commerce shares in the retail market, there is an apparent increase in demand for small package home delivery services [3]. This presents some logistical challenges for the current delivery service infrastructure. Most notably, the last mile section of long-haul deliveries faced significant challenges. Although the last-mile is the shortest part of a delivery service, it is associated with the most costs. With the rise in consumer demand, additional expenses of accommodating and restructuring adds to this already costly process [4]. The issue of last-mile delivery delays can rack up to be a debilitating cost on the business. Delays in global shipping, although more costly, are fairly predictable and can be accounted for [4]. Unfortunately, the same cannot be said for the shorter haul deliveries as they can be affected by insufficient routes, driver error, and traffic, among other factors. A possible solution to this problem is the implementation of autonomous delivery robots.

Robotics have seen a shift from being implemented almost exclusively for research and development purposes to being more heavily used to serve the public. Robots have now seamlessly integrated in day to day life with applications in the fields of manufacturing, healthcare, and agriculture [5]. Specifically, Uninhabited Aerial Vehicles (UAVs) have spread more rapidly due to their versatility, cost, and flexible operation characteristics [6]. UAV operations can range from land surveying, to wild life monitoring, and forest fire tracking. However, an operation of particular interest is parcel delivery. Recently, parcel delivery has been an extensively researched UAV application driven by large online corporations' desire to find faster and more cost effective package delivery methods, specifically to solve the last-mile delivery issue [7]. An example of a corporation actively working on implementing autonomous drones in delivery services is FedEx Corp who partnered with Elroy Air to test autonomous air cargo delivery [8].

There are several configurations in which a UAV can carry a given payload. These include rigid body connection, slung payload, and a manipulator arm as shown in Figure 1.1 [6].

The most intuitive payload carrying method is the rigid body connection as this will limit any

Figure 1.1: Possible configurations for payload delivery [6].

payload swings in transport. That being said, having the payload attached to or close to the body can alter the moment of inertia and adversely affect UAV maneuverability, especially when transporting heavier payloads. Furthermore, a shift in the contents of the package can lead to aircraft instability and possible catastrophe. Additionally, the aircraft will have to fully land to deliver the parcel, making the aircraft unable to deliver a package in places where the UAV cannot land, greatly limiting delivery possibilities.

A manipulator arm equipped UAV faces the same issues as the rigid body connection with the added complexity of maneuvering and controlling the arm. It does have the advantage of being able to pick up and let go of the parcel, but still needs a clearing in which to land and faces the possibility of instability due to shifting inertia.

A slung payload delivery system solves both the inertia and space issues. By hanging the payload on a cable, the masses can be modeled as separate bodies, and that retains the inertia of the quadcopter. This also allows for the payload to be delivered without landing.

Traditional control methods, while effective in many scenarios, often struggle to adapt seamlessly to the intricate dynamics of slung payloads. The inherent variability arising from payload movement necessitates real-time adjustments in flight control, a demand that is difficult for traditional algorithms. This challenge has paved the way for the exploration of innovative and adaptive control strategies, and among them, Reinforcement Learning (RL) stands out.

Reinforcement learning offers an interesting paradigm shift. By enabling UAVs to learn from experience, RL algorithms empower these systems to refine their control policies over time. Through continuous interaction with the environment, quadcopters equipped with RL systems not only adapt to diverse payload dynamics but also optimize their actions in response to real-time feedback. This adaptability, grounded in learning from trial and error, shows significant potential for enhancing the stability, maneuverability, and overall performance of slung payload quadcopters.

In particular, model free RL emerges as a powerful methodology for slung payload applications due to its adaptability enabling it to handle the intricate and unpredictable dynamics inherent in slung payload systems. Slung payloads introduce complexities such as varying weights, sizes, and shapes, making it challenging to formulate accurate models of the system's behavior. In traditional control methods, precise models are essential, but their creation in these unpredictable contexts is often impractical. Model-free RL eliminates this hurdle by learning directly from interactions with the environment, making it inherently suited for situations where creating accurate models is difficult.

Additionally, slung payload applications demand real-time adaptability. Quadcopters must swiftly respond to changing conditions, ensuring stability and precision while handling dynamic payloads. Model-free RL algorithms excel in real-time decision-making. They continuously learn and update their strategies based on immediate feedback, allowing the UAV to adjust its

actions promptly, especially in environments where conditions change rapidly.

The goal of this thesis is to produce a model-free RL controller which can be implemented fully on board an actual UAV with a slung payload to achieve full waypoint tracking capabilities. It must be able to overcome the perturbations caused by the slung load such that it is able to move towards the goal and be able to be stationary at that point.

Chapter 2

Literature Review

This chapter of the thesis investigates existing research on how quadcopters control their position both with and without slung loads. It starts by discussing the practical control techniques used in previous works. Proportional Integral Derivative (PID) controllers, which are a standard in control systems, are first examined in terms of effectiveness in achieving control when coupled with nonlinear methods. The exploration then moves on to more advanced methods such as FeedBack Linearisation (FBL), which is a method used to simplify the complex dynamics of these UAVs. Optimal control strategies, and how they optimize the quadcopter's movements while managing slung loads, are also discussed.

After discussing the control methods, the focus shifts to methods of deriving the mathematical model of the system. This involves looking into two common methods: the Euler-Lagrange and Newton-Euler techniques. For each method, the study explores their advantages, disadvantages and assumptions made during the modeling process, as well as considerations of their practical use.

Finally, implementation of these methods on real life systems is examined. This includes the types of quadcopters, autopilots and motion capture systems used. Additionally, the ways in which the controllers were integrated onto the actual quadcopter and whether it was a fully on-board process is examined.

2.1 Nonlinear Control Methods

Nonlinear systems have parameters that vary in time or just naturally exhibit nonlinear dynamics which makes solving them more difficult in comparison to linear systems. However, when the nonlinearities of the system are not too extreme, local linearisation can be applied to derive approximate linear models of a nonlinear system. This enables the implementation of linear controllers [9].

An important consideration is that no data stream from a mechanical system is continuous in time. The sensors are unable to send a continuous stream of data and therefore, the controller is dependent on the speed at which the sensors can obtain and relay back the required information. This introduces discrete-time control theory. However, it is generally assumed that computations can be performed fast and often enough that they can be assumed to be continuous time systems [9].

2.1.1 PID and PD Control

Simple Proportional-Integral-Derivative (PID) and Proportional-Derivative (PD) controllers are sufficient to control a simple quadcopter system as demonstrated by [10] and [11]. However, the controller's performance can be greatly improved when coupled with a feedback linearisation technique like the input-output nonlinear dynamic inversion technique applied in [10].

For a slung payload-quadrotor system, simple PD control is no longer adequate due to the additional complexities and considerations introduced by incorporating the load. Hence, to use a PD controller on the UAV-load system, it is paired with a secondary control technique to accommodate for the additional dynamics and requirements of the system [12].

Pure Proportional Navigation Guidance Law

In [12], a PD controller is grouped with a Pure Proportional Navigation (PPN) inspired guidance law to aid in landing with a payload and avoid crashing the load to the ground. The work presented in [6] also implements a PPN approach, but proposes a higher level controller which relies heavily on the PIDs present in the autopilot. Traditional PPN is based on the *Constant Bearing Principle*. In this method, the UAV aligns its velocity vector with its line of sight to the target. This alignment results in the rotation rate of the line of sight to be zero resulting in a direct collision course between the UAV and target [13].

Instead of using an acceleration command to track a constant velocity, the dimension of the problem is lowered by one such that a velocity command is used to track a constant position [12]:

$$\dot{\boldsymbol{x}}_{cmd} = \begin{bmatrix} \dot{x}_{cmd} \\ \dot{y}_{cmd} \\ \dot{z}_{cmd} \end{bmatrix} = \begin{bmatrix} k_{p,x}(x_t - x_q) \\ k_{p,y}(y_t - y_q) \\ k_{p,z}(z_t - z_q) \end{bmatrix} = \boldsymbol{k}_p(\boldsymbol{x}_t - \boldsymbol{x}_q) \tag{2.1}$$

where the subscripts t and q indicate target and quadcopter.

This control law is generated with the assumption that there is an autopilot position controller that can deal with the generated position commands. The simple inspiration of this guidance law is that $\dot{\boldsymbol{x}}_{cmd} \to 0$ as $\boldsymbol{x}_q \to \boldsymbol{x}_t$. To obtain the commanded positions (2.1) is integrated [6]:

$$\boldsymbol{x}_{cmd,t} = \int \dot{\boldsymbol{x}}_{cmd} dt = \boldsymbol{x}_{cmd,t-1} + \dot{\boldsymbol{x}}_{cmd} \Delta t \tag{2.2}$$

The work presented in [12] uses a PD-like position controller coupled with the PPN guidance law to produce a thrust and quaternion command; the latter is then fed into the attitude PID controller. Firstly, the model is linearised using a Jacobian linearisation method about the hover equilibrium: $\boldsymbol{F}_{eq} = -(m_p + m_q)\boldsymbol{g}$ (for a detailed derivation of the Jacobian linearisation method refer to [14]). A PD-like position controller coupled with the PPN guidance law is then utilised to produce a command signal $\boldsymbol{F} = -\boldsymbol{k}_x\boldsymbol{x}_{cmd} - \boldsymbol{F}_{PD} - \boldsymbol{F}_{\boldsymbol{eq}}$ where $\boldsymbol{F}_{PD}$ is the command signal produced by the designed PD controller: $\boldsymbol{F}_{PD} = \boldsymbol{k}_v\boldsymbol{v}_q^T + \boldsymbol{k}_x\boldsymbol{x}_q^T$. The PD gains $\boldsymbol{k}_x$ and $\boldsymbol{k}_v$ along with the PPN gains $\boldsymbol{k}_p$ were chosen based on the linearised model.

In [6], the PPN guidance law produces $\boldsymbol{x}_{cmd}$ and the rest is left to the autopilot's translational and rotational PIDs which are assumed to be adequate. The gains of the PPN guidance law $\boldsymbol{k_p}$ were tuned using an approximated model of the system through generating a transfer function using input-output data from a step input on the experimental quadcopter.

This method showed an improvement in performance when landing with a payload using a PPN guidance law as opposed to algorithms that solely rely on PD and PID controllers. Specifically, the results showed reduced overshoot along a slower velocity gradient which are essential components for slung payload transportation.

The shortcoming of both proposed PPN guidance law controllers is their heavy reliance on the model of the system. In the presence of disturbances or changes in load mass or tether length, the gains of the controller would no longer be valid and would require user interference and retuning. Furthermore, the controller presented in [6], although viable, depends on the soundness of the three onboard PID controllers. This can be improved by designing a controller that produces lower level commands similar to the one presented in [12]. This increases the transparency of how the command is generated which in turn provides the user more flexibility and allows more room for improvement in the control algorithm design.

Adaptive Control

To address the issue of changing dynamics, controllers can be integrated with an adaptive control method as presented in [15]. As a general term, adaptive control pertains to a collection of techniques that provide a method for automatic adjustment of controllers in real-time. The primary goal is to attain or sustain a desired level of performance in a control system, especially when the parameters of the plant's dynamic model are either unknown or subject to changes over time [16]. Retrospective Cost Adaptive Controllers (RCACs) are a discrete time adaptive control method applicable to stabilization, command following and disturbance rejection. The discrete nature of RCACs allows for theit implementation at the sensor sample rate, eliminating the need for controller discretization. This also implies that the required model information can be estimated through input-output data of system and an accurate model is not required [17].

The work presented in [15] proposes the use of an RCAC in tandem with a fixed gain PD controller on a quadcopter-payload system to deliver a cable-suspended load without knowledge of the payload mass. The nonlinear PD trajectory controller is designed to achieve adequate performance for some known payload mass and produce an three-dimensional thrust force $\boldsymbol{u}_p$:

$$\boldsymbol{u}_p = -\boldsymbol{k}_x\boldsymbol{e}_x - \boldsymbol{k}_{\dot{x}}\boldsymbol{e}_{\dot{x}} - \sum_{i=1}^{n} k_{\omega_i}\boldsymbol{e}_{\omega_i} - \sum_{i=1}^{n} k_{q_i}\boldsymbol{e}_{q_i} - (m - \tilde{m}_n)g\begin{bmatrix}0 & 0 & 1\end{bmatrix} \tag{2.3}$$

where $\boldsymbol{e}_x = \boldsymbol{x} - \boldsymbol{x}_d + \sum_{i+1}^{n} l_i\boldsymbol{e}_3$ and $\boldsymbol{e}_{\dot{x}} = \dot{\boldsymbol{x}}$ are the errors for the quadcopter position and velocity, $\boldsymbol{e}_{\omega_i} = \begin{bmatrix}\boldsymbol{e}_1 & \boldsymbol{e}_2\end{bmatrix}^T \boldsymbol{\omega}_i$ and $\boldsymbol{e}_{q_i} = \begin{bmatrix}\boldsymbol{e}_1 & \boldsymbol{e}_2\end{bmatrix}^T (\boldsymbol{e}_3 \times \boldsymbol{q}_i)$ are the cable segments' errors in

angular velocity and unit direction vectors, and $\tilde{m}_n$ is the mass uncertainty making $(m - \tilde{m}_n)$ the total mass of the system with a nominal payload mass. Note that here $\boldsymbol{e}_1$, $\boldsymbol{e}_2$ and $\boldsymbol{e}_3$ are unit vectors in x, y and z consecutively.

The purpose of the controller is to compensate for the payload mass uncertainty, meaning that the RCAC portion of the controller is in charge of producing an additional z-component of the thrust $\boldsymbol{u}_R(k) = \begin{bmatrix} 0 & 0 & u_R(k) \end{bmatrix}^T$ to accommodate for differences from the nominal mass that the PD is trained for. The performance vector associated with the RCAC algorithm is defined as the following:

$$z(k) = \begin{bmatrix} \tilde{x}_{p,3}(k) \\ \dot{x}_3(k) \end{bmatrix} \tag{2.4}$$

where the first entry is the error of third coordinate of the payload position, and the second entry is the quadcopter velocity error in the z-direction measured in the inertial frame.

Furthermore the parameters for the RCAC algorithm are obtained from an impulse response generated from the PD controller along with the simplified dynamics of the model. Finally both inputs generated by the PD controller and RCAC are combined to generate the final three-dimensional thrust force $\boldsymbol{F}_c$ and sent to an attitude PD controller that serves as the final control level of the quadcopter as shown in Figure 2.1. The shortcoming of this control

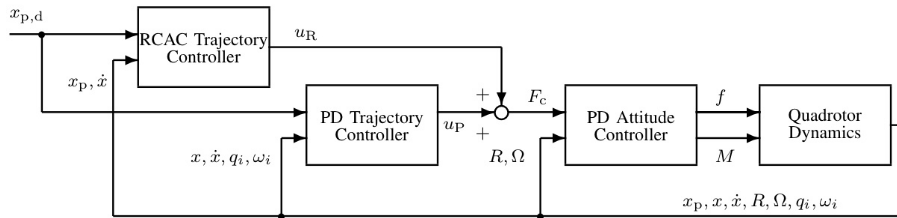


Figure 2.1: Diagram of the control architecture of the RCAC and PD trajectory controller proposed by [15] for a UAV-payload system

algorithm is that while it can account for changes in mass of the payload, it will have trouble in accommodating other changes that might affect the dynamics of the system.

Additional Nonlinear Coupling Terms

The performance of a traditional PD controller on a nonlinear system can be greatly improved through the integration of a nonlinear coupling term. This term is obtained by determining what states in the system are coupled through the analysis of the mathematical model. After that determination has been made, a nonlinear term relating those coupled states can be added to the PD controller to mitigate the adverse dynamics resulting from that relationship. In [18], a UAV-slung load system was analysed in the $X - Z$ plane and the model was computed. From there it was concluded that the swing angle α of the payload affects both the x and z dynamics of the quadcopter through the following equation:

$$\ddot{\alpha} = -\frac{1}{l}\ddot{x}\cos\alpha - \frac{\ddot{z} + g}{l}\sin\alpha \tag{2.5}$$

The output of the designed controller is $\boldsymbol{u}_o = \begin{bmatrix} \theta_{d_o} & f_o \end{bmatrix}^T$ where the first output is the desired pitch to stabilise the x position and the second output is the total thrust in the z-direction to control the aircraft's altitude. For theoretical clarity, the output can be described mathematically

as being the sum of the output of a traditional PD control $\boldsymbol{u}_{PD}$ and the nonlinear coupling term $\boldsymbol{u}_C$.

$$\boldsymbol{u}_o = \boldsymbol{u}_{PD} + \boldsymbol{u}_C = \begin{bmatrix} -k_{p,x}(x - x_d) - k_{v,x}\dot{x} \\ \frac{-k_{p,z}(z-z_d)-k_{v,z}\dot{z}-(m_p+m_l)g}{\cos\theta} \end{bmatrix} + \begin{bmatrix} -k_{v,\alpha}\dot{\alpha}^2\dot{x} \\ \frac{-k_{v,\alpha}\dot{\alpha}^2\dot{z}}{\cos\theta} \end{bmatrix} \tag{2.6}$$

The second vector in (2.6) is the additional nonlinear coupling term added to the PD controller to adapt the controller for a more complex system.

This approach to nonlinear system control is advantageous in terms of the simplicity of its theory and model and ease of implementation. However, it is heavily reliant on the model developed by the user and its robustness in real life application where other dynamics affecting the system may be present has not been yet confirmed.

2.1.2 Feedback Linearisation (FBL)

Feedback linearisation is a common technique implemented in nonlinear system control. The primary objective of this approach is to linearise a nonlinear system by employing state feedback along with a nonlinear coordinate transformation derived through a geometric analysis of the system [19].

By eliminating the nonlinearities in the system, conventional linear control techniques can then be easily applied. The linearisation of the quadcopter-load transporting system enabled [20] to implement a Passivity Based Controller (PBC) that controls the overall energy of the system. The work of [21] demonstrated the ability of utilising feedback linearisation to produce a high level nonlinear controller on a Planar Vertical Take off and Landing (PVTOL) aircraft with a suspended load.

There are many feedback linearisation techniques that can be utilised. However, this is beyond the scope of this thesis. For an extensive review of the most common feedback linearisation techniques refer to [19].

2.1.3 Optimal Control

The fundamental concept of any optimal controller is the computation of a control function that optimizes some performance metric or cost function [22]. Optimal control allows the designer to specify the dynamic model of the system as well as the desired outcomes. It is then up to the controller to compute the optimised control. Some widely used optimal control techniques include Linear Quadratic Regulators (LQRs), Model Predictive Control (MPC) and RL.

Linear Quadratic Regulators (LQRs)

The simplest class of optimal control problems is that of Linear Time Invariant (LTI) systems which is solved using the LQR approach. In this method, the cost J is quadratic in the states $\boldsymbol{x}$ and controls $\boldsymbol{u}$ [23]. This form of the cost function was employed in [10] and [24].

$$J(\boldsymbol{x}, \boldsymbol{\phi}) = \int_0^\infty \left[\boldsymbol{x}^T(t)\boldsymbol{Q}\boldsymbol{x}(t) + \boldsymbol{u}^T(t)\boldsymbol{R}\boldsymbol{u}(t)\right] dt \tag{2.7}$$

where $\boldsymbol{u}(t)$ is in a class $\boldsymbol{\Phi}$ of permissible controls, and $\boldsymbol{Q}$ and $\boldsymbol{R}$ are positive semidefinite and definite matrices known as the weight matrices that penalise the error from equilibrium and the control effort.

Recalling that the dynamics of an LTI system are governed by the equation $\dot{\boldsymbol{x}} = \boldsymbol{AX} + \boldsymbol{Bu}$, then for a stabilizable LTI system, the optimal control law is a linear function of $\boldsymbol{x}$ [22].

$$\boldsymbol{u} = -\boldsymbol{Kx} = -\left(\boldsymbol{R}^{-1}\boldsymbol{B}^T\boldsymbol{S}\right)\boldsymbol{x} \tag{2.8}$$

To obtain the optimal control value shown in (2.8), the matrix $\boldsymbol{S}$ must be obtained numerically by solving the algebraic Riccati equation (2.9).

$$\boldsymbol{A}^T\boldsymbol{S} + \boldsymbol{SA} - \boldsymbol{SBR}^{-1}\boldsymbol{B}^T\boldsymbol{S} + \boldsymbol{Q} = 0 \tag{2.9}$$

In [10] a trajectory PID controller computes the desired roll and pitch angles and feeds them to the LQR algorithm, which computes the error in roll, pitch, yaw and altitude and stores those as the states $\boldsymbol{x}$ for the cost function. The optimal action computed $\boldsymbol{u}$ is a vector of the roll, pitch and yaw torques as well as the total thrust which can then be converted to the corresponding motor speeds for each motor. The LQR controller showed superior performance when it came to attitude control, outperforming the FBL and PID controllers. However it behaved extremely poorly when it came to altitude control stabilising the quadcopter at an altitude that was 0.45m lower than the desired hover.

The LQR controller designed by [24] was a higher level controller which computes the desired total thrust as well as the desired roll and pitch as the control inputs $\boldsymbol{u}$ which will then rely on the onboard Paparazzi autopilot software for attitude tracking and motor mixing. The algorithm was able to successfully track the desired x, y and z trajectories. This is likely due to the control architecture of the system in which the higher level, more complex controllers were cascaded into lower level PID controllers which ensured the systems stability.

In the work presented by [25], the model was linearised using the Jacobian linearisation method (also seen implemented by [12]) before undertaking the LQR control approach. The cost function chosen slightly differed from the ones implemented in the previously discussed works as it additionally penalises the product of the states and control actions.

$$J(\boldsymbol{x}, \boldsymbol{\phi}) = \int_0^\infty \left[\boldsymbol{x}^T(t)\boldsymbol{Qx}(t) + \boldsymbol{u}^T(t)\boldsymbol{Ru}(t) + \boldsymbol{x}^T(t)\boldsymbol{Nu}(t)\right] dt \tag{2.10}$$

Here, $\boldsymbol{N}$ is the weight matrix in charge of penalising the product of the state and control of the system which is then used in computing the optimal control gain $\boldsymbol{K} = \boldsymbol{R}^{-1}\left(\boldsymbol{B}^T\boldsymbol{S} + \boldsymbol{N}^T\right)$. For the version of the cost function presented in (2.10) the Ricatti equation (2.9) must also include the additional weight matrix:

$$\boldsymbol{A}^T\boldsymbol{S} + \boldsymbol{SA} - (\boldsymbol{SB} + \boldsymbol{N})\,\boldsymbol{R}^{-1}\left(\boldsymbol{B}^T\boldsymbol{S} + \boldsymbol{N}^T\right) + \boldsymbol{Q} = 0 \tag{2.11}$$

In this work, the control input was chosen to be the motor toques $\boldsymbol{u} = \begin{bmatrix} U_1 & U_2 & U_3 & U_4 \end{bmatrix}$ which can then be converted to motor speeds or total thrust force and torques acting on the UAV. The controller was successful in controlling the UAV in all axes along with attenuation of the load swing. The LQR was successful in controlling the UAV in the x and y axes but did exhibit some small deviation in the z axis which is the same issue encountered in the LQR controller presented in [10].

One of the shortcomings of LQRs is the linearisation of the system. A clear example is the weighting of the action weight matrix $\boldsymbol{R}$ to avoid causing instability due to deviation from the linearised model. Taking the example of [24], the control inputs of the roll and pitch angles were

penalised heavily and the integral states were penalised lightly so that the quadcopter does not preform any sharp maneuvers or break the set bounds, keeping it within the linearised region:

$$\boldsymbol{u}^T\boldsymbol{R}\boldsymbol{u} = \begin{bmatrix} f \\ \phi_d \\ \theta_d \end{bmatrix}^T \begin{bmatrix} 1 & 0 & 0 \\ 0 & 10^4 & 0 \\ 0 & 0 & 10^4 \end{bmatrix} \begin{bmatrix} f \\ \phi_d \\ \theta_d \end{bmatrix}$$

$$\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} = \begin{bmatrix} x_{error} \\ y_{error} \\ z_{error} \end{bmatrix}^T \begin{bmatrix} 0.001 & 0 & 0 \\ 0 & 0.001 & 0 \\ 0 & 0 & 0.001 \end{bmatrix} \begin{bmatrix} x_{error} \\ y_{error} \\ z_{error} \end{bmatrix}$$

Furthermore, LQR algorithms can call for controls whose values are not realizable by the system and hence the control inputs must be manually bound by the user before sending them to the system. A similar methodology that incorporates the constraints of the system is Model Predictive Controllers (MPCs).

Model Predictive Controllers (MPCs)

For stabilization and tracking tasks it is often more desirable to solve the optimal control problem with infinite horizon due to the nature of the controller's goal [26]. However, with limited computing power this is often very difficult to implement unless one assumes no constraints and implements an LQR as presented in (2.7). As mentioned previously, implementing an LQR to a system requires the user to constraint the control inputs as it is common for a linear quadratic regulator to exceed the system's constraints. To take the constraints into account the optimal control problem can be turned from an infinite to a finite horizon problem and an MPC can take the place of the LQR. Another advantage that MPC presents is that it is a time varying control law which conducts repeated optimization of the cost function at every time instant k as opposed to an LQR which is a static control law.

Since the optimal control problem for an MPC is one of finite horizon with N steps its cost function must take into account the terminal condition :

$$J(\boldsymbol{x}, \boldsymbol{\phi}) = p\left(\boldsymbol{x}_N, \boldsymbol{u}_N\right) + \sum_{i=0}^{N} l\left(\boldsymbol{x}_{k+i|k}, \boldsymbol{u}_{k+i|k}\right) \tag{2.12}$$

where p is the terminal cost and l is the stage cost.

In [24], the MPC chosen was the simplest possible form, a linear MPC. A linear MPC is implemented on an LTI system and has a quadratic cost function much like an LQR. The cost function for a quadcopter payload system is the forementioned work is defined as:

$$J(\boldsymbol{x}, \boldsymbol{\phi}) = \boldsymbol{x}_N^T\boldsymbol{P}\boldsymbol{x}_N + \sum_{i=0}^{N} \boldsymbol{x}_i^T\boldsymbol{Q}\boldsymbol{x}_i + \boldsymbol{u}_i^T\boldsymbol{R}\boldsymbol{u}_i \tag{2.13}$$

where $\boldsymbol{P}$ $\boldsymbol{Q}$ and $\boldsymbol{R}$ are the weights relating to the terminal state, stage states and inputs consecutively.

Unlike LQR, the MPC can be constrained to produce control inputs between certain values. When comparing the LQR and MPC controllers developed in [24], the LQR controller was for the most part able to stay within the system control input limits, however there were multiple instances in which it exceeded the limits. That being said, when implementing an MPC, the

control input was always bound to the user defined constraints of the system and hence proved to be a more reliable control method when it comes to ensuring the system remains in within the linearised region. Both linear control methods suffer from a slow response with a settling time of 9 seconds for the MPC and 10 seconds for the LQR.

To remedy the slow response due to linearisation, the MPC was coupled with a Δu formulation where the state vector of the UAV-slung payload system is augmented with the previously computed control inputs as additional states and the deviation in the control inputs as the new inputs to the system. This greatly improved the performance of the MPC by nearly halving the settling time while indirectly incorporating the capability for disturbance estimation.

The MPC presented in [27] sets itself apart from all the previously discussed optimal control methods in that it seeks to maximise, rather than minimise, some criteria. In this work the MPC controller chooses an action $\boldsymbol{a}_t$ based on maximising the highest predicted reward using a developed dynamics model as shown in (2.14). Refer to Section 4.1.1 for a thorough explanation of rewards and the reward signal.

$$\boldsymbol{a}_t^* = \underset{\boldsymbol{a}_t}{\operatorname{argmax}} \left[\max_{\boldsymbol{a}_{t+1:t+H}} \sum_{\tau=t}^{t+H} \mathrm{E}_{\boldsymbol{s}_\tau \sim p_\theta} [\boldsymbol{s}_\tau, \boldsymbol{a}_\tau] \right] \tag{2.14}$$

Here, the states $\boldsymbol{s}_\tau$ are recursively sampled from the model initialised at $\boldsymbol{s}_\tau \leftarrow \boldsymbol{s}_t$ and defined by the following probability $\boldsymbol{s}_{\tau+1} \sim p_\theta(\boldsymbol{s}_{\tau+1}|\boldsymbol{s}_\tau, \boldsymbol{a}_\tau)$. Once the optimization is solved, the first action is chosen to be applied as the control input to the system. The work of [27] focuses more on the development and training of the dynamics model utilised by the MPC controller. Hence the MPC algorithm itself is fairly straightforward but relies heavily on the estimation provided by the dynamic model. That being said, it is important to note that the MPC controller presented in the work does not rely on the linearisation of the system and hence can deal with the nonlinear nature of the UAV-payload system at hand.

To generate an accurate estimate of the rewards, the model of the system was not generated through mathematical analysis. Rather, [27] utilises a Neural Network (NN) to solve a reinforcement learning problem.

Reinforcement learning problems are situations in which a system achieves control through learning from experiences with its environment. It does so through updating a mapping of states to actions, called a *policy*, which dictates the actions to be taken depending on the state that the system finds itself in. The updates to this policy are dependant on the reward $r(s)$, which is a function that relays information to the system regarding how advantageous its current state is, along with a value $\widehat{V}(s)$ which is a function estimating the sum of discounted rewards that the system is likely to collect by the time it reaches the goal (or a terminal state) from state s. Much like the policy, the value function must also be updated as the system explores its environment. Similar to the policy, the value function is a mapping of the estimated values to the states and is the system's key to deciphering what states are 'good' or 'bad' states to be in. To solve an RL problem, the system must have explored enough to allow the value function and policy estimates to both converge to their true values enabling the system to realise the value of being in each state along with the correct control action to take to reach the goal. Refer to 4.1.1 for more thorough review on the elements of RL.

Neural Actor-Critic Algorithms

Neural networks are a method in artificial intelligence that seeks to solve a machine learning problem by simulating neuron activation in the brain. While NNs are usually used to solve supervised learning problems, they can also be used to solve RL problems. For the specific control problem at hand neural actor-critic algorithms prove to be the popular option. For background information on actor-critic neural networks refer to Appendix A. The research in [28] and [29] employ a TD neural network architecture to control a quadcopter. In [30] a similar approach is utilised for controlling a UAV-load system. Lastly, [31] used NN to implement a Monte Carlo RL algorithm for UAV control.

The work presented by [28] utilises three neural networks to control a quadrotor: two actor networks, one to produce an action $\boldsymbol{\mu}_t$ from policy π_θ and one to produce an adversary action $\bar{\boldsymbol{\mu}}$ from adversary $\bar{\pi}_{\bar{\theta}}$, and a critic network that produces the action-value function $Q_\phi(\boldsymbol{s}_t, \boldsymbol{\mu}_t)$. The input to the actor networks are the states of the UAV which are then used to compute activation values for two layers of 64 neurons each using the tanh activation function. The final outcome of the actor neural networks is the action that each policy has chosen. Here the optimal RL agent responsible for updating the policy π_θ aims to maximise the sum of rewards of the system by choosing the appropriate optimal action $\boldsymbol{\mu}_t$. On the other hand, the adversary policy $\bar{\pi}_{\bar{\theta}}$ aims to minimise the sum of rewards by its choice of an action to act as a disturbance to the system $\bar{\boldsymbol{\mu}}$. Both the policies are trained using the Deep Deterministic Policy Gradient (AR-DDPG) algorithm in an off-policy manner where the value of the optimal policy is learned independent of the agent's actions. Note here that the subscripts θ,$\bar{\theta}$ and ϕ are the weights to be tuned for optimal policy, adversarial policy and state-action value function respectively.

The action implemented to the system is a combination of the chosen optimal action $\boldsymbol{\mu}_t$ and the disturbance $\bar{\boldsymbol{\mu}}$ with a probability of α. The purpose of the adversary policy is to add robustness to the system such that it can operate in uncertain conditions. Lastly, the critic network takes the states of the system along with the chosen optimal policy to update the action value function $Q(\boldsymbol{s}_t, \boldsymbol{\mu}_t)$.

The controller developed by [29] is very similar to the one seen in [28], with the major difference being the lack of an adversary policy. Rather, this work suggests simply two neural networks: an actor and a critic. Here, the critic produces a value function $V(\boldsymbol{s})$ which means it criticises the action based on the state that the system ended up in without knowledge of what the action is. The actor NN is a Gaussian probability density function that evaluates the policy π_θ. Note that the standard deviation for this Gaussian distribution is taken to be some small constant σ. The gradient descent in this work is implemented using the Proximal Policy Optimization (PPO) method which is based on the actor critic model. The NN architecture for this work consisted of two hidden layers similar to the previously mentioned controller. The paper also delves into the investigation of the choice of the reward function and how it affects the performance of the UAV. It was found that punishing the control input slowed the dynamics of the system. Furthermore omitting the control input from the reward function allows for the system to converge at a higher value V. The work also states that it took a total of 12 hours to train the networks to be able to control the UAV.

The work in [31] took a slightly different approach in constructing the RL algorithm by utilising the Monte Carlo method. The MC method differs from TD learning in that it does not use bootstrapping to estimate the value function. Rather the value function is only updated once the UAV has either reached the goal or a terminal state. The NN architecture in this paper

is much the same as the one in [28] as it utilises two hidden layers with 64 tahn nodes. It is explicitly stated that the NN structure was not optimised in any form and it was simply assumed that it would be able to deal with the problem at hand. The policy optimization in the actor network was conducted using a natural gradient descent algorithm. The policy resulted in a failure rate of 4% when compared to an MPC policy implemented on the same system with a failure rate of 71% in simulation. The resulting trained algorithm was implemented on a real system, discussed in Section 2.3 of this work, however training was not transferred to the system.

The only paper covered which utilised an RL method to control a quadrotor-payload system is the one presented by [30]. Here, a nonlinear controller is developed based on the full nonlinear dynamics of the system. The control objective in this work can be mathematically described as making the position of the payload equal to the desired pay load position $\lim_{t\to\infty} \boldsymbol{p}_l = \boldsymbol{p}_{ld}$ as well as bringing the errors of the load's velocity, the cable's attitude and angular velocity errors, and the attitude and angular velocity errors of the UAV down to zero as time approaches infinity $\lim_{t\to\infty} \begin{bmatrix} \boldsymbol{e}_v^T & \boldsymbol{e}_q^T & \boldsymbol{e}_\omega^T & \boldsymbol{e}_R^T & \boldsymbol{e}_\Omega^T \end{bmatrix} = 0_{1\times 15}$.

The RL actor-critic networks K^μ and Q^θ in this controller design are unique in that they cascade with the aforementioned full nonlinear controller. The input to the actor network is simply the state load vector $\boldsymbol{s} = \begin{bmatrix} \boldsymbol{e}_x^T & \boldsymbol{e}_v^T & \theta_x & \theta_y & \dot{\theta}_x & \dot{\theta}_y \end{bmatrix}$ where θ_x and θ_y are the swing angles of the cable. Unlike the previous actor-critic networks, the actor in this algorithm doesn't generate the control input to directly control the quadcopter. Rather it seeks to parameterise the control gains for the position portion of the nonlinear controller $\boldsymbol{a} = \begin{bmatrix} K_{px} & K_{py} & K_{pz} \end{bmatrix}$. The critic networks inputs are the states mentioned previously as well as the reward computed at the corresponding state $r_t = -||\boldsymbol{e}_x(t)||^2$.

The actor obtains the optimal policy by implementing a Deterministic Policy Gradient (DPG) algorithm similar to the one seen in [28] and the critic updates the action value function is using TD error. The network structures were generated using Tensorflow with 2 hidden layers each. The structures for the actor and critic networks are $10 \times 64 \times 128 \times 3$ and $13 \times 64 \times 128 \times 1$ respectively. This structure was chosen through testing to develop an NN controller which was large enough to learn the dynamics of the system while not presenting computational burdens. The proposed method was compared to energy-based and backstepping control methods where it outperformed them in every category of testing verifying its adaptability and fast response when compared to other widely accepted control methods.

The downfall of NN controllers is their training and run times. Unless a network structure is meticulously optimised for computation efficiency it is almost impossible to be implemented in a fast dynamics system like a quadrotor. This can be seen in [27] in which the MPC had to be run at 4hz to allow the off-board NN enough time to execute. In [31] it was explicitly stated that the simulated dynamics run time of the system were notably faster than the computational time for the neural network. Furthermore training was limited to the model and did not continue on the system, and hence major dynamical differences became apparent between the model used for training and the actual dynamics of the system, especially in areas of more complex aerodynamic phenomenon.

2.2 Modelling Methods

Dynamic modeling and system identification are an important step when it comes to system control, since for most control design purposes, it is necessary to have a mathematical model that reveals the dynamic behavior of a system [32]. Dynamic models are simplified representations or real-life systems in the form of mathematical equations or computer code [33]. These models illustrate the mathematical connection between a robot's control inputs and its movement [34].

In the field of aerial vehicle control the Euler-Newton and the Euler-Lagrange techniques are the most adopted model development methods [35]. Other less popular modelling techniques such as Kane's method and the Boltzmann-Hamel method have also been utilised for robot modelling in [36] and [37]. However for the purpose of this work only the two most favoured will be examined to determine what modelling method is best for modelling the quadcopter-slung load system.

2.2.1 Euler-Lagrange Modelling Method

The Euler-Lagrange method efficiently obtains the equations describing the motion of complex systems. For multi-body systems, this method is especially advantageous as it automatically incorporates constraints arising from the connections between bodies, without requiring the knowledge of the manipulator's internal forces [38]. The adoption of this approach in studies [15], [18], [20], [21], [25], [39], [40] and [41] reinforces its convenience in modeling a quadcopter-payload system due to its multi-body nature.

The Euler-Lagrange method subtracts the total potential energy of the system V from the total kinetic energy of the system T to obtain the *Lagrangian* L [42].

$$L = T - V \tag{2.15}$$

The Lagrangian describes the dynamics of the system and depends on the generalized coordinates $\boldsymbol{q} = (q_1, q_2, ..., q_i)$ and their derivatives $\dot{\boldsymbol{q}} = (\dot{q}_1, \dot{q}_2, ..., \dot{q}_i)$.

The goal of the Euler-Lagrange method is to find the equations of motion that govern the behavior of the system through applying the *principal of stationary action* by minimising the action integral S which is defined as the integral of the Lagrangian over time [43]:

$$S = \int_{t_1}^{t_2} L(\boldsymbol{q}, \dot{\boldsymbol{q}}, t)dt \tag{2.16}$$

The equations of motion are obtained by applying the calculus of variations to the action integral. The resulting equations take the form shown in (2.17). The full derivation of this equation can be found in pages 222-223 of [43].

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\boldsymbol{q}}}\right) - \frac{\partial L}{\partial \boldsymbol{q}} = 0 \tag{2.17}$$

Equation (2.17) is applicable only to conservative systems, where no energy is added to or removed from the system. In other words, the function $q_i(t)$ leads to a local minimum, maximum, or saddle point of $S(q_i(t))$ [43]. However, in the case of non-conservative systems like the quadcopter-payload system, the external forces acting on the system must also be considered. This results in a modified form of the Euler-Lagrange equation [42]:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\boldsymbol{q}}}\right) - \frac{\partial L}{\partial \boldsymbol{q}} = \boldsymbol{f}_{ext} \tag{2.18}$$

Here, $\boldsymbol{f}_{ext}$ represents the vector of generalized forces and moments corresponding to the external forces acting on the system. The process of deriving the Euler-Lagrange equation, as given in (2.18), can be found in pages 196-202 of [42].

The Euler-Lagrange equation (2.18) incorporates the system's constraints, even in cases where the specific constraints are unknown. This characteristic proves to be highly advantageous for robotic modeling and presents a clear benefit of using the Euler-Lagrange method [42]. However, if desired, the Euler-Lagrange equation can be modified to explicitly include the known constraints. In such cases, the resulting Euler-Lagrange equation incorporates the system's constraints, denoted as ϕ, through the utilization of Lagrange multipliers $\lambda(t)$ [44].

$$\frac{\partial L}{\partial \boldsymbol{q}} - \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\boldsymbol{q}}}\right) = \sum_{j=1}^{m} \lambda_j(t) \frac{\partial \phi_j}{\partial \boldsymbol{q}} \tag{2.19}$$

Note that this thesis does not extensively discuss the process of defining constraints, as it is not essential when modeling robotic systems with undetermined constraints. For a comprehensive understanding and practical application of the constrained Euler-Lagrange equation the reader is referred to pages 89-97 of [44].

The drawback of Euler-Lagrange models is that they can be computationally complex due to their use of rotation matrices to convert body frame coordinates to generalised coordinates (explained in detail is Appendix B) as well as their incorporation of the Coriolis and centrifugal forces [45]. This makes Euler-Lagrange models computationally inefficient for real-time applications.

2.2.2 Newton-Euler Modelling Method

In contrast to the Euler-Lagrange modelling method, the Newton-Euler modelling method does not consider Coriolis and centrifugal forces. This simplification reduces computation time, making Newton-Euler models suitable for time-sensitive control situations [45]. The most notable difference is that in the Lagrangian formulation, the system is considered as a whole and the Lagrangian is computed for the whole system. On the other hand, the Newton-Euler formulation addresses each member of the body separately and establishes equations describing its linear and angular motion. Since the members are interconnected, these equations incorporate coupling forces and torques that also appear in the equations for neighboring links [42].

In single body systems like a simple quadcopter, the Newton-Euler method proves to be the more favourable modelling technique due to its straightforward nature and simplicity as shown in [10], [11], [28], and [31]. That being said, it can also be seen applied to model a quadcopter-load system as in works [6], [12], and [46].

The formulation of the equations relies on three Newtonian mechanics facts [42]:

1. *Every action has an equal and opposite reaction.*
   (a) That is, if body A applies force f and torque τ on body B, then body B applies a force $-f$ and torque $-\tau$ on body A.
2. *The rate of change of the linear momentum $p = mv$ is equal to the total force applied to the body f*
3. *The rate of change of the angular momentum $H_0 = I_0\omega_0$ is equal to the total torque applied to the body τ_0*

Applying the second statement yields Newtons equation (2.20) where m is the mass of the body, f is the sum of external forces acting on the body, and $\boldsymbol{v}$ and $\boldsymbol{a}$ are the velocity and acceleration of the center of mass of the body with respect to some inertial frame of reference [42]:

$$\boldsymbol{f} = \frac{dp}{dt} = \frac{d(m\boldsymbol{v})}{dt} = m\boldsymbol{a} \tag{2.20}$$

Similarly, by implementing the third statement to the rotational dynamics of a body yields

$$\boldsymbol{\tau_0} = \frac{dH_0}{dt} = \frac{d(\boldsymbol{I_0}\boldsymbol{\omega_0})}{dt} \tag{2.21}$$

where $\boldsymbol{I_0}$ is the moment of inertia of the body about a fixed inertial frame whose origin is located at the its center of mass, $\boldsymbol{\omega_0}$ is the body's angular velocity, and $\boldsymbol{\tau_0}$ is the sum of external moments acting on the body.

The rotational dynamics of the body are measured about a body fixed axis that rotates with the body to maintain a constant moment of inertia matrix I [42].

$$\boldsymbol{\tau} = \boldsymbol{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times (\boldsymbol{I}\boldsymbol{\omega}) \tag{2.22}$$

Here, $\boldsymbol{I}$ is the constant inertia matrix of the body with respect to the body frame, and $\boldsymbol{\omega}$ and $\boldsymbol{\tau}$ are the angular velocity and total sum of moments of the body also expressed in the body-fixed frame [42].

This modelling method explicitly describes the translational and rotational dynamics of each body. This offers a high level of clarity in terms of the flow of forces and torques through the system as well as the interaction of the multiple components with one another.

2.2.3 Modelling Assumptions for a UAV Carrying a Slung Load

There are some assumptions regarding the quadcopter-payload model that appear to be widely accepted due to their frequent use amongst researchers in the field, as well as the proven success of the controllers built with these assumptions in mind:

- A massless cable with no slack
- The payload is approximated as a point mass
- The suspension point of the payload is located directly at the centre of mass of the UAV
- The suspension is assumed to be frictionless
- The aerodynamic effects on the payload are negligible

These simplifications can be seen utilised when developing control methods by [12], [18], [20], [21], [25], [24], [30], [39], [40] and [46].

Some of the literature opted to eliminate some assumptions in an effort to create a more accurate model for more robust control. The model in [15] does not assume a mass-less rigid cable. Rather, the cable is split into n segments with known masses. The direction unit vectors and angular velocities $\boldsymbol{q}_i$ and $\boldsymbol{\omega}_i$ of each segment of the cable are then utilised in the controller proposed by the research. The dynamic system created in [6] assumes an offset from the suspension to the center of mass of the quadcopter which allows for different linkage points along the UAV.

Lastly, the model developed in [41] takes the aerodynamic effects on the payload into account when developing the model such that the performance of the controller on the actual UAV more closely matches the one proven in simulation.

2.3 Implementation on the Physical System

Controller design is only the first step to controlling a system. The next step following verification in simulation is the implementation of the controller on the physical system. This process consists of obtaining the system, finding a way of acquiring the required data for the controller, and finally executing the controller on the system.

The most commonly used autopilots are *ArduPilot* and *PX4* which are both open source software that can be accessed through Github and edited and uploaded to the UAV. Furthermore due to the endless adaptability of a self-assembled UAV, the controller code can also be optimised for faster running speeds more easily than it would be for a ready made UAV.

The works of [24], [27], [31], and [40] opted to use a Parrot AR 2.0, DJI, Ascending Technologies Astec, and an Astec Hummingbird Drone respectively. Of these, the only controller implemented fully online was that in [31]. This was achieved through the addition on an on-board computer to perform the controller's computations. The rest of the papers utilising commercially available quadrotors opted to carry out the computations on a separate machine and send the commands to the quadcopters.

The research efforts of [12] and [30] opted to build custom drones. Both researchers employed a Pixhawk autopilot along with some companion computer. This is because the Pixhawk is not well suited for general computing tasks or any task that requires rigorous computation. Instead it is recommended by PX4, the manufacturers of the Pixhawk, to run those intensive tasks on a separate companion computer [47]. It is important to note that both works that chose to assemble the UAV themselves were also able to achieve controllers that ran fully on board and required no additional external computers.

To obtain the state information required for their respective controllers all the research groups utilised some form of motion capture system. The work of [27] used an externally mounted RGB camera using OpenCV to record the states which were chosen to be the pixel locations and size of the payload. The remainder of the experiments chose to employ commercially available motion capture cameras and software with [31] and [40] choosing to use Vicon, and [12] and [24] using Optitrack. The research of [40] used vicon to measure the trajectories of both the quadcopter and payload, [31] also used Vicon but fused the attitude data obtained from it with the sensor data from the on board Inertial Measurement Unit (IMU). Finally [12] and [24] both utilised Optitrack for position data of the quadcopter and load.

The reviewed works suggest that implementation of a controller on a UAV in a lab setting calls for the integration of a motion capture system to relay the required state information. In RL applications, this only becomes relevant after training the agent on a model of the system to achieve convergence in simulation.

Chapter 3

Quadcopter-Payload Dynamic Model

This chapter focuses on how the states of the quadrotor and slung payload are measured relative to some earth fixed inertial frame $\{I\}$ through the implementation of body fixed frames on both the quadcopter and the load which are used to build a full model of the system. This covers the construction and application of rotation matrices to convert from one frame to another, as well as the Euler angles and Euler sequence to measure orientations of a particular body. Additionally, the operation principles of a quadcopter are discussed. This includes concepts of torque and force balancing and how they can manipulate the attitude and position of the quadrotor.

Finally, by adopting the Euler-Lagrange some method along with modelling assumptions, a full six Degree-Of-Freedom (DOF) mathematical model governing the position and orientation of the quadcopter along with the swing angles of the payload is obtained. Due to the symmetry of the aircraft, the model is simplified to achieve the final $X - Z$ planar model for computational efficiency when training the proposed controller.

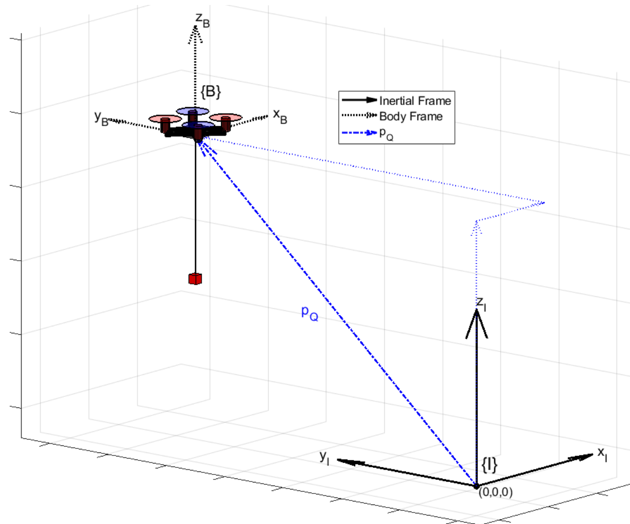


Figure 3.1: The quadcopter body frame $\{B\}$ shown relative to the inertial body frame $\{I\}$. The position of the quadcopter is measured by the position vector $\boldsymbol{p}_Q$ from the inertial to the body frame.

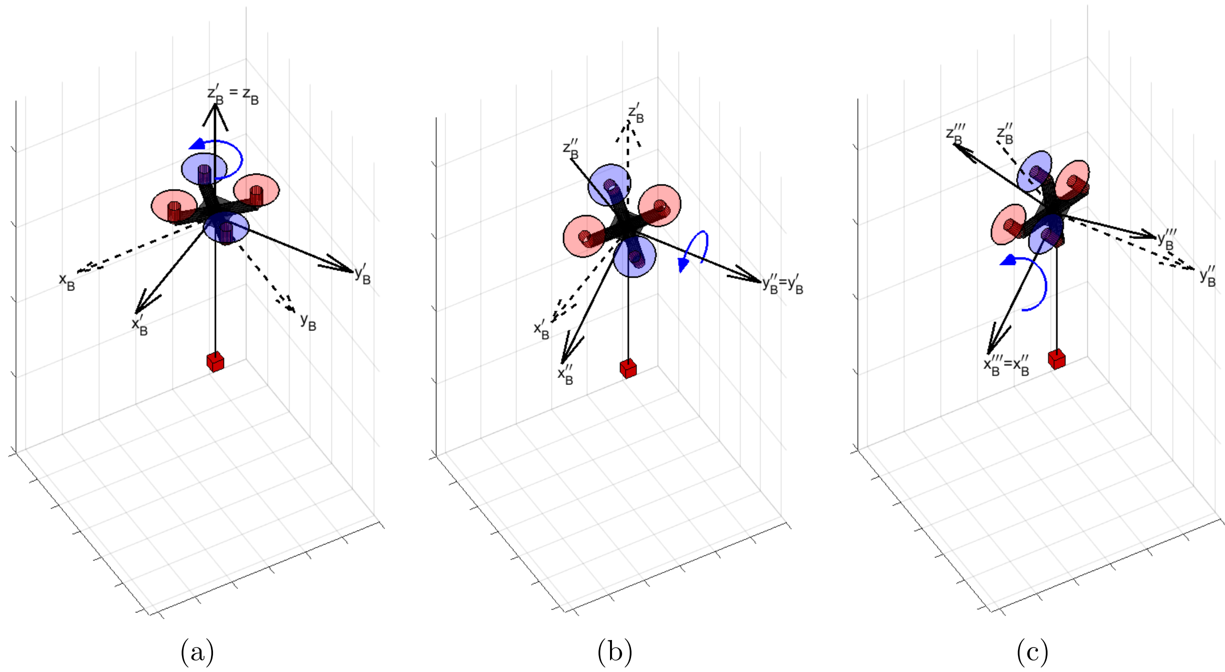
3.1 Frames of Reference & The Rotation Matrix

A quadrotor manages its attitude $\boldsymbol{\zeta} = [\phi, \theta, \psi]$ by manipulating the roll, pitch, and yaw angles. In the case of an autonomous quadcopter, the control system calculates and utilizes the necessary angle values. Consequently, the control system must possess the capability of measuring these angles, which entails employing two coordinate systems.

To determine the position and orientation of the quadrotor, the two right-handed *North-West-Up (NWU)* frames of reference shown in Figure 3.1 were chosen. The inertial frame $\{I\}$ is the earth fixed frame which the quadcopter's position and orientation are measured with respect to. The body frame of reference $\{B\}$ is the second NWU frame located at the center of mass of the quadcopter and is fixed with respect to the quadcopter's body, moving and rotating with it. The $\{B\}$ frame is used to represent the quadcopter itself, and the difference in orientation and position between it and the $\{I\}$ frame provides the states of the quadcopter used for control. Figure 3.1 also shows the position vector $\boldsymbol{p}_Q = \begin{bmatrix} x & y & z \end{bmatrix}$ used to obtain the position states of the quadcopter.

3.1.1 Quadcopter Position and Rotation Matrix

For a UAV where the attitude's Euler angles are non-zero, a certain order of rotations needs to be followed to visualise the orientation of the quadcopter $\{B\}$ in terms of the frame $\{I\}$, called the *Euler Sequence*. This is achieved through rotating about the z-axis by the yaw angle ψ, followed by a rotation around the current y-axis by a pitch angle θ, and finally rotating about the current x-axis by a roll angle ϕ as shown in Figure 3.2 [42]. This is a key concept for modelling any rotating system since any force, moment, or position measured in the body frame can also be rotated in the order of the Euler sequence to achieve its orientation in the inertial frame. Mathematically, this can be achieved through the implementation of the rotation matrix presented in (3.1). This matrix can be multiplied to any three dimensional vector represented in the $\{B\}$ frame to obtain its representation in the $\{I\}$ frame. For a complete derivation of



(a) (b) (c)

Figure 3.2: The Euler sequence of rotations to convert from the body frame $\{B\}$ to the inertial frame {I}. Start with rotating about the z-axis, then the new y-axis and then the final x-axis to achieve the resultant orientation.

(3.1) refer to Appendix B.

$$
{}^{I}_{B}R = \begin{bmatrix} C_\psi C_\theta & C_\psi S_\theta S_\phi - S_\psi C_\phi & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\psi C_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix} \tag{3.1}
$$

Here S_x and C_x denote $\sin x$ and $\cos x$ respectively.

3.1.2 Payload Position and Rotation Matrix

To quantify the position of the payload in the inertial frame, its swing angles must be measured. To do this, another coordinate system $\{S\}$ is created. This new coordinate system is another NWU frame located at the end of the cable of length λ as shown in Figure 3.3. The position of the load relative to the inertial frame is measured using the standard spherical coordinates $[\alpha, \beta, \lambda]$. The spherical coordinate angles α and β follow the standard spherical angle measurements in which α is measured from the z_I axis on the $Y - Z$ plane and β is measured about the z_I axis on the $X - Y$ plane. Both these angles are measured using the right hand notation as shown in Figure 3.4. To determine the position of the load from the spherical angles, two rotation matrices must be constructed. The first rotation matrix will describe the orientation of the load relative to the inertial frame following an α swing. Similarly, the second rotation matrix will describe the orientation of load to the $\{I\}$ frame following a β swing.

Using the same logic presented for constructing the rotation matrix for the quadrotor present in Appendix B, the rotational relationship from the $\{S\}$ frame to the $\{I\}$ frame can be represented

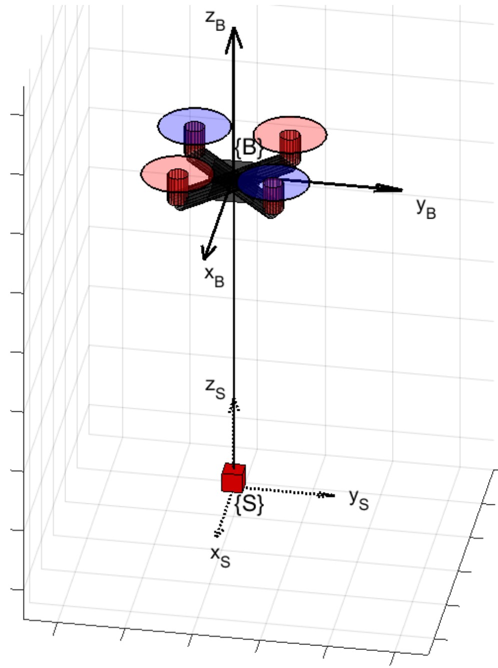


Figure 3.3: The payload fixed frame $\{S\}$

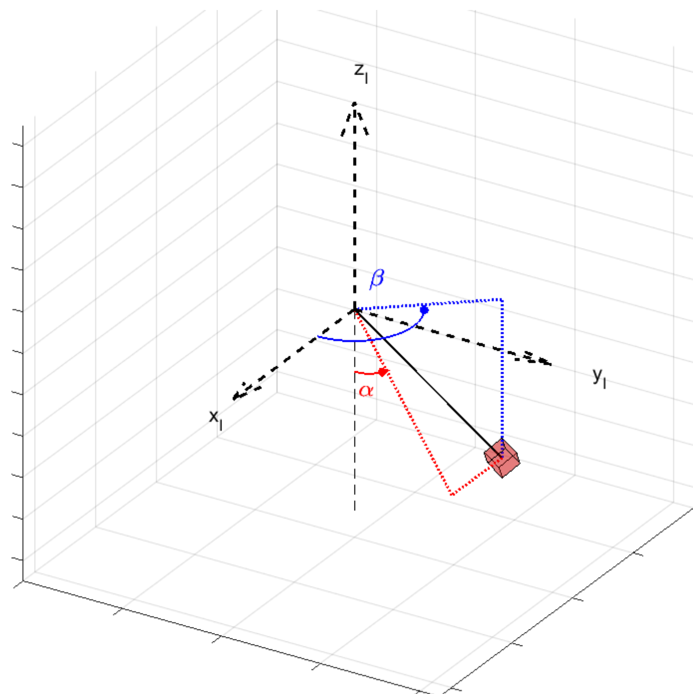


Figure 3.4: The angles used to measure the position of the cable relative to the inertial frame. α denotes the swing angle of the cable measured from the z_I axis and β denotes the rotational position of the cable measured about the z_I axis. Both angles are measured using the right hand notation.

using an x right hand rotation matrix for α and a z right handed rotation matrix for β.

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & C_\alpha & -S_\alpha \\ 0 & S_\alpha & C_\alpha \end{bmatrix} R_z(\beta) = \begin{bmatrix} C_\beta & -S_\beta & 0 \\ -S_\beta & C_\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

Unlike the UAV which always rotates with respect to its current frame, the cable rotates about the fixed inertial frame for both rotations. That is, the cable rotates about inertial x_I by α, then around the inertial z_I by β. When rotating about a fixed frame, the multiplication succession of the rotation matrices is reversed [42]. That means that in this case where the cable first rotates about x_I then about z_I, the rotation matrices are multiplied in the order $R_z(\beta)R_x(\alpha)$ as shown in (3.3). This does not reverse the order of the rotations but simply takes into account rotating about a constant frame rather than about a changing one.

$${}^I_S R = R_z(\beta)R_x(\alpha) = \begin{bmatrix} C_\beta & -S_\beta & 0 \\ S_\beta & C_\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & C_\alpha & -S_\alpha \\ 0 & S_\alpha & C_\alpha \end{bmatrix} = \begin{bmatrix} C_\beta & -S_\beta C_\alpha & S_\beta S_\alpha \\ S_\beta & C_\beta C_\alpha & -C_\beta S_\alpha \\ 0 & S_\alpha & C_\alpha \end{bmatrix} \tag{3.3}$$

The next step is to relate the position of the payload $\boldsymbol{p}_L = [x_L, y_L, z_L]'$ to the position of the quadcopter $\boldsymbol{p}_Q$. It can be said that the position of the payload is simply the position of the quadcopter with the addition of the distance of the rigid cable λ rotated using the spherical angle rotation matrices presented in (3.3).

$$\boldsymbol{p}_L = \boldsymbol{p}_Q +^I_S R \begin{bmatrix} 0 & 0 & \lambda \end{bmatrix}' \tag{3.4}$$

$$\longrightarrow \begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} C_\beta & -S_\beta C_\alpha & S_\beta S_\alpha \\ S_\beta & C_\beta C_\alpha & -C_\beta S_\alpha \\ 0 & S_\alpha & C_\alpha \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -\lambda \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} -S_\beta S_\alpha \lambda \\ C_\beta S_\alpha \lambda \\ -C_\alpha \lambda \end{bmatrix} \tag{3.5}$$

After determining the method of measuring the attitude of the UAV along with the positions of both the aircraft and the payload, the discussion moves to how the quadcopter achieves changes in attitude and position.

3.2 Quadcopter Operation Principles

Quadcopters have two possible configurations, the + (plus) configuration and the × (cross) configuration. The names are based on the direction in which the aircraft deems as forward and consequently, the way in which the body axis is aligned. The + configuration aligns the quadcopter along the body's x-axis such that the axis runs directly through the ClockWise (CW) propellers as shown in Figure 3.5a. In the × configuration, the x-axis runs directly between two propellers as illustrated in Figure 3.5b [48]. Although the flight mechanisms of a + configuration rotorcraft are easier to follow, the quadcopter chosen for implementation was a × configuration quadcopter due its superior stability characteristics [49].

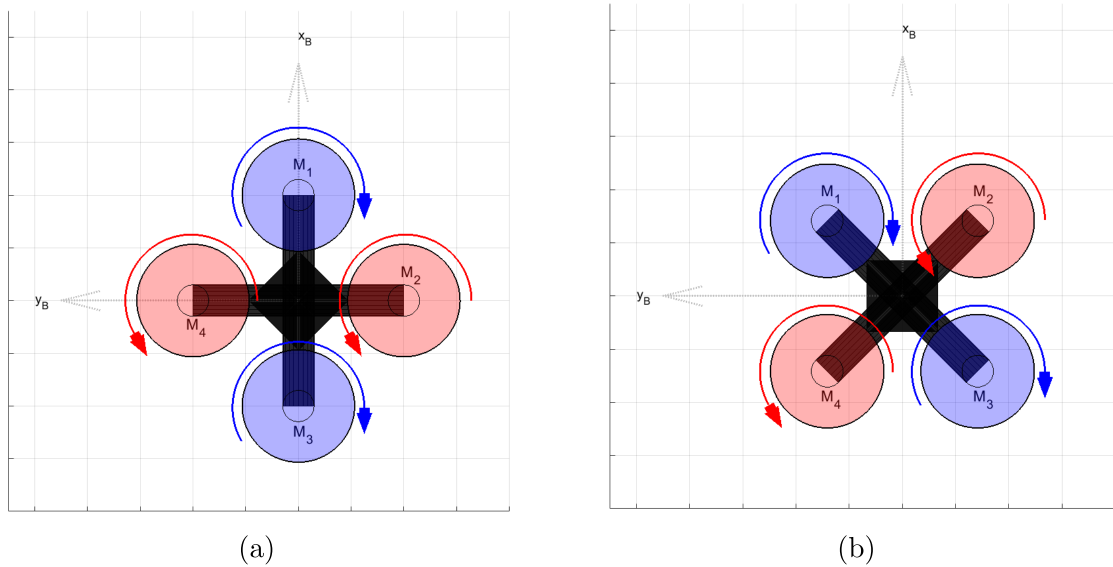


Figure 3.5: Possible quadcopter configurations a) +-configuration b) ×-configuration

As a system, a quadcopter is said to be an under-actuated system as it has six degrees of freedom and four control inputs in the form of lift forces f_1, f_2, f_3, f_4 from the four motors M_1, M_2, M_3 and M_4 that determine its motion in space [10]. In any case, this combination of inputs will output an overall thrust T, and moments about the x, y and z axes.

The lift force per blade is given by (3.6) [50]:

$$f_{blade} = \frac{1}{2}\rho r_{blade}^2 \Omega_i^2 S C_L \tag{3.6}$$

Similarly, the drag force per blade is computed using (3.7) [50]:

$$d_{blade} = \frac{1}{2}\rho r_{blade}^2 \Omega_i^2 S C_D \tag{3.7}$$

where ρ is the air density, r_{blade} is the radius of the propeller blade, Ω_i is the angular velocity of the ith motor, S is the total surface area of the blades, and C_L and C_D are the coefficients of lift and drag.

Since the propellers chosen were bi-blade, (3.6) and (3.7) are used to find the lift and drag

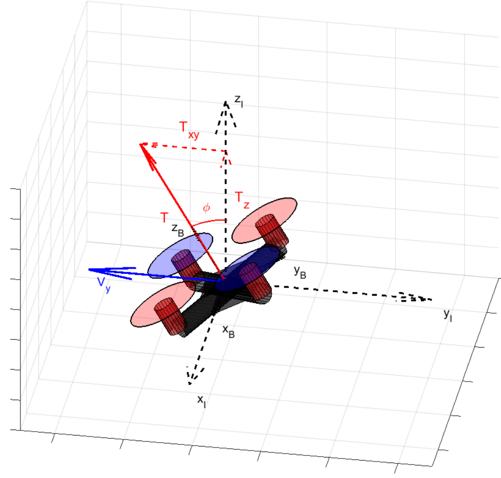


Figure 3.6: A quadcopter rolling by an angle ϕ about the inertial x-axis x_I to move in the $-y$ direction using the vector of the total thrust force T_{xy}

generated by each ith propeller.

$$f_i = 2\left[\frac{1}{2}\rho r_{blade}^2 \Omega_i^2 S C_l\right] = \rho r_{blade}^2 \Omega_i^2 S C_L \tag{3.8}$$

$$d_i = 2\left[\frac{1}{2}\rho r_{blade}^2 \Omega_i^2 S C_D\right] = \rho r_{blade}^2 \Omega_i^2 S C_D \tag{3.9}$$

Furthermore, upon further analysis of (3.8) and (3.9), the drag can be expressed in terms of the lift as shown in (3.10).

$$d_i = \frac{f_i}{C_L} C_D = c f_i \tag{3.10}$$

where $c = \frac{C_D}{C_L}$.

Finally, the total lift force is simply obtained by summing the forces produced by all four propellers as shown in (3.11).

$$T = f_1 + f_2 + f_3 + f_4 \tag{3.11}$$

The UAV moves along the $X - Y$ plane by manipulating its attitude to vector the total lift force T such that it has an additional component propelling the aircraft in the direction of motion as shown in Figure 3.6.

The total thrust vector is naturally always along the positive z_B axis. That is, it is always pointing upwards relative to the UAV's frame of reference. However, much like the quadrotor's position, the force vector must also be represented in the inertial frame. This is done through the utilisation of the rotation matrix presented in (3.1) to achieve the force vector in (3.12).

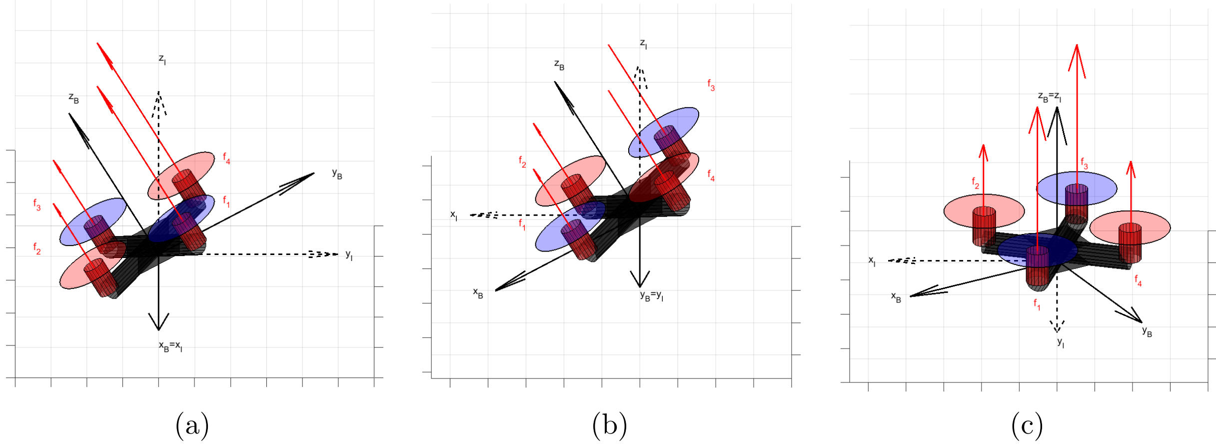
$$\boldsymbol{F}^I =_B^I R \boldsymbol{F}^B$$

$$\begin{bmatrix} F_x^I \\ F_y^I \\ F_z^I \end{bmatrix} = \begin{bmatrix} C_\psi C_\theta & C_\psi S_\theta S_\phi - S_\psi C_\phi & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\psi C_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix} \begin{bmatrix} F_x^B \\ F_y^B \\ F_z^B \end{bmatrix}$$

$$\begin{bmatrix} F_x^I \\ F_y^I \\ F_z^I \end{bmatrix} = \begin{bmatrix} C_\psi C_\theta & C_\psi S_\theta S_\phi - S_\psi C_\phi & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\psi C_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix}$$

$$\begin{bmatrix} F_x^I \\ F_y^I \\ F_z^I \end{bmatrix} = \begin{bmatrix} (S_\psi S_\phi + C_\psi S_\theta C_\phi)\, T \\ (S_\psi S_\theta C_\phi - C_\psi S_\phi)\, T \\ C_\theta C_\phi T \end{bmatrix} \tag{3.12}$$

From (3.8) and (3.9), it follows that the forces generated by a motor are directly proportional to its angular velocity. Therefore, the lift and drag generated by a single propeller will vary when varying the angular velocity to generate a moment about the center of the quadcopter altering its orientation in space.



(a) (b) (c)

Figure 3.7: Figure showing changes in forces required to achieve the three types of moments a quadcopter can experience. a) A quadcopter rolling by an angle ϕ by increasing forces f_1 and f_4 and decreasing forces f_2 and f_3. b) A quadcopter pitching by an angle θ by increasing forces f_3 and f_4 and decreasing forces f_1 and f_2. c) A quadcopter yawing by an angle ψ by increasing forces f_1 and f_3 and decreasing forces f_2 and f_4 .

Increasing the angular velocities of the propellers to the left of the x_B-axis f_1 and f_4 while decreasing those to the right of the axis f_2 and f_3 leads to a force imbalance resulting in a positive roll moment ϕ of the aircraft as shown in Figure 3.7a. Mathematically, this is described by (3.13).

$$\tau_\phi = [(f_1 + f_4) - (f_2 + f_3)]\, l \tag{3.13}$$

where l is the horizontal distance between the centers of the motors.

Similarly, as seen in Figure 3.7b, by increasing the angular velocities of the motors to the right of the y_B-axis f_3 and f_4 and simultaneously decreasing f_1 and f_2 leads the quadcopter to experience a positive pitching moment θ about the y_B-axis which is mathematically computed by (3.14).

$$\tau_\theta = [(f_3 + f_4) - (f_1 + f_2)]\, l \tag{3.14}$$

Finally, increasing the angular velocities of the clockwise motors 1 and 3, and decreasing those of the counter-clockwise motors 2 and 4 causes the counter-clockwise drag forces d_1 and d_3 to increase and the clockwise drag forces d_2 and d_4 to decrease. This causes the UAV to perform a counter-clockwise positive yaw ψ about the z_B axis as seen in Figure 3.7c. This torque can be described in terms of the thrust by implementing the relationship illustrated in (3.15).

$$\tau_\psi = [(f_1 + f_3) - (f_2 + f_4)]\, lc \tag{3.15}$$

Similar to the forces, the moments explained in this section must be transformed from the body to the inertial frame as shown in (3.16).

$$\boldsymbol{M}^I = {}_B^I R \boldsymbol{M}^B$$

$$\begin{aligned}
\begin{bmatrix} \tau_\phi^I \\ \tau_\theta^I \\ \tau_\psi^I \end{bmatrix} &= \begin{bmatrix} \left(S_\psi S_\phi + C_\psi S_\theta C_\phi\right)\tau_\phi \\ \left(S_\psi S_\theta C_\phi - C_\psi S_\phi\right)\tau_\theta \\ C_\theta C_\phi \tau_\psi \end{bmatrix} \\
&= \begin{bmatrix} \left(S_\psi S_\phi + C_\psi S_\theta C_\phi\right)\left[(f_1 + f_4) - (f_2 + f_3)\right] l \\ \left(S_\psi S_\theta C_\phi - C_\psi S_\phi\right)\left[(f_3 + f_4) - (f_1 + f_2)\right] l \\ C_\theta C_\phi \left[(f_1 + f_3) - (f_2 + f_4)\right] lc \end{bmatrix}
\end{aligned} \tag{3.16}$$

The forces and moments discussed in this section can now be utilised to build a dynamic model of the quadcopter-slung payload system using the Euler-Lagrange formulation.

3.3 Euler-Lagrange Model Formulation

Building a model is an essential step when developing a model-free control algorithm. It allows for simulation, testing, and training prior to implementation on the real system. This is both an economical and methodical approach to testing a model-free control algorithm.

The two most prevalent methods of obtaining the dynamic model of a system are the Newton-Euler and Euler-Lagrange formulations [51]. The Newton-Euler method is based on directly interpreting Newton's second law for the system. This law represents the system in terms of forces and momentum [51]. The Euler-Lagrange method, on the other hand, obtains the equations of motion through the system's work and energy [51].

Since the quadcopter-slung payload model can be seen as two rigid bodies connected to one another with a cable, the equations of motion would be extremely complex to derive using the Newton-Euler method. However, since the kinetic and potential energies of both the quadcopter and slung payload are fairly simple to compute, in this work, the dynamic model was created through deriving the non-linear equations of motion using the Euler-Lagrange method.

3.3.1 Modelling Assumptions

Prior to building the model, the following standard slung-payload modeling assumptions were considered:

- The origin of the body-fixed frame $\{B\}$ is located directly at the centre of mass of the quadcopter
- The body of the quadcopter is assumed to be symmetric and rigid.
- The quadrotor thruster arms are rigid and the thrusters produce a force which is parallel to the rotor axis.
- It is assumed that the cable is rigid and massless.
- The payload is taken to be a point-mass.
- The suspension point of the mass is assumed to be at the origin of the body-fixed frame $\{B\}$.
- The suspension point is taken to be a spherical frictionless point.

3.3.2 Euler's Kinematic Equations

Similarly to the position of the UAV, the desired angular velocities of the quadopter also need to be expressed in terms of the inertial frame. The readings of the angular velocities received by the Inertial Measurement Unit (IMU) onboard will be expressed in the body frame $\{B\}$ denoted as shown in (3.17).

$$\boldsymbol{\omega}_B = \begin{bmatrix} p & q & r \end{bmatrix}' \tag{3.17}$$

The angular velocities measured in the inertial frame $\{I\}$ are expressed in (3.18)

$$\dot{\boldsymbol{\zeta}} = \begin{bmatrix} \dot{\phi} & \dot{\theta} & \dot{\psi} \end{bmatrix}' \tag{3.18}$$

One must consider intermediate frames between the inertial and body frames when following the Euler sequence as shown in Figure 3.2. The quadcopter first starts in an aligned configuration

with the inertial frame $\{I\}$. It then yaws to reach frame $\{1\}$ as shown in 3.2a. From there, Figure 3.2b shows when the UAV pitches to orient itself to frame $\{2\}$ and finally, it rolls to achieve the final alignment with the body-fixed frame $\{B\}$ illustrated in Figure 3.2c.

Consider (3.19) [52]:

$$\boldsymbol{\omega}_{B/I} = \boldsymbol{\omega}_{B/2} + \boldsymbol{\omega}_{2/1} + \boldsymbol{\omega}_{1/I} \tag{3.19}$$

where $\boldsymbol{\omega_{B/I}}$ is the angular velocity of the {B} with respect to {I}, $\boldsymbol{\omega_{B/2}}$ is the angular velocity of {B} with respect to {2}, and so on.

To complete the derivation, one must consider $\boldsymbol{\omega_{B/I}}$ expressed in the $\{B\}$ frame, $\boldsymbol{\omega^{B}_{B/I}}$. To compute this, the right hand components in (3.19) must also be expressed in the $\{B\}$ frame, resulting in (3.20) shown below [52]:

$$\boldsymbol{\omega}^B_{B/I} = \boldsymbol{\omega}^B_{B/2} + \boldsymbol{\omega}^B_{2/1} + \boldsymbol{\omega}^B_{1/I} = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{3.20}$$

To move from the inertial frame to the body-fixed frame, the rotation matrices used are simply the transpose of the rotation matrices in (B.2) used to move from the body-fixed frame to the inertial frame:

$$R'_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & C_\phi & S_\phi \\ 0 & -S_\phi & C_\phi \end{bmatrix} R'_y(\theta) = \begin{bmatrix} C_\theta & 0 & -S_\theta \\ 0 & 1 & 0 \\ S_\theta & 0 & C_\theta \end{bmatrix} R'_z(\psi) = \begin{bmatrix} C_\psi & S_\psi & 0 \\ -S_\psi & C_\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.21}$$

An important point is that $\boldsymbol{\omega_{n/m}}$ for any two frames $\{n\}$ and $\{m\}$ is equal whether it is expressed in frame $\{n\}$ or frame $\{m\}$. This can be applied to the right hand side of (3.20) to result in the following:

$$\boldsymbol{\omega}^B_{B/2} = \boldsymbol{\omega}^2_{B/2} = \begin{bmatrix} \dot{\phi} & 0 & 0 \end{bmatrix}' \tag{3.22}$$

$$\boldsymbol{\omega}^2_{2/1} = \boldsymbol{\omega}^1_{2/1} = \begin{bmatrix} 0 & \dot{\theta} & 0 \end{bmatrix}' \tag{3.23}$$

$$\boldsymbol{\omega}^1_{1/I} = \boldsymbol{\omega}^I_{1/I} = \begin{bmatrix} 0 & 0 & \dot{\psi} \end{bmatrix}' \tag{3.24}$$

From there, using the rotation matrices shown in (3.21), (3.20) can be expanded to obtain the following [53]:

$$\boldsymbol{\omega}^B_{B/I} = \boldsymbol{\omega}^B_{B/2} + R'_x \boldsymbol{\omega}^2_{2/1} + R'_x R'_y \boldsymbol{\omega}^1_{1/I} = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{3.25}$$

$$\longrightarrow \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & C_\phi & S_\phi \\ 0 & -S_\phi & C_\phi \end{bmatrix} \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & C_\phi & S_\phi \\ 0 & -S_\phi & C_\phi \end{bmatrix} \begin{bmatrix} C_\theta & 0 & -S_\theta \\ 0 & 1 & 0 \\ S_\theta & 0 & C_\theta \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} \dot{\phi} - S_\theta \dot{\psi} \\ C_\phi \dot{\theta} + S_\phi C_\theta \dot{\psi} \\ C_\phi C_\theta \dot{\psi} - S_\phi \dot{\theta} \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

$$\longrightarrow \begin{bmatrix} 1 & 0 & -S_\theta \\ 0 & C_\phi & S_\phi C_\theta \\ 0 & -S_\phi & C_\phi C_\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

$$\longrightarrow W \dot{\boldsymbol{\zeta}} = \boldsymbol{\omega}^B$$

Therefore, the angular velocity of the quadrotor measured in the inertial frame $\dot{\boldsymbol{\zeta}}$ can be found by multiplying the measured angular velocities $\boldsymbol{\omega}^B$ by the inverse of the W matrix.

$$\dot{\boldsymbol{\zeta}} = W^{-1}\boldsymbol{\omega^B} = \begin{bmatrix} 1 & S_\phi T_\theta & C_\phi T_\theta \\ 0 & C_\phi & -S_\phi \\ 0 & S_\phi / C_\theta & C_\phi / C_\theta \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{3.26}$$

The same methodology can be implemented on the linear velocities of the UAV measured in the body frame $\boldsymbol{v}^B$ to obtain their measurements in the inertial frame $\dot{\boldsymbol{p}}_Q$. The quadrotor's linear velocities in the $\{B\}$ frame are denoted by the following:

$$\boldsymbol{v}^B = \begin{bmatrix} u & v & w \end{bmatrix}' \tag{3.27}$$

In the $\{I\}$ frame, the velocities are assigned the following variables:

$$\dot{\boldsymbol{p}}_Q = \begin{bmatrix} \dot{x} & \dot{y} & \dot{z} \end{bmatrix}' \tag{3.28}$$

The relationship between $\dot{\boldsymbol{p}}_Q$ and $\boldsymbol{v_B}$ is described by (3.29) [53].

$$\dot{\boldsymbol{p}}_Q = {}^I_B R \boldsymbol{v}^B \tag{3.29}$$

The conversions of both the angular and linear velocities shown in (3.26) and (3.29) can be combined in matrix form for clarity as shown in (3.30).

$$\begin{bmatrix} \dot{\boldsymbol{p}}_Q \\ \dot{\boldsymbol{\zeta}} \end{bmatrix} = \begin{bmatrix} {}^I_B R(\phi, \theta, \psi) & 0 \\ 0 & W^{-1}(\phi, \theta) \end{bmatrix} \begin{bmatrix} \boldsymbol{v}^B \\ \boldsymbol{\omega}^B \end{bmatrix} \tag{3.30}$$

3.3.3 Equations of Motion

Prior to determining the equations of motion of the system, the states required must be complied in a state vector. In the quadcopter-load system, the state vector is taken to be the linear and angular positions of the quadcopter along with the swing angles of the payload.

$$\boldsymbol{q} = \begin{bmatrix} x & y & z & \alpha & \beta & \phi & \theta & \psi \end{bmatrix}^T \tag{3.31}$$

From observing the state vector (3.31) it is apparent that the system will possess both translational and angular energies.

The kinetic energy of the system stems from the equations of translational and angular kinetic energies $T_{translational} = \frac{1}{2}mv^2$ and $T_{angular} = \frac{1}{2}I\omega^2$ as shown in (3.32).

$$T = \frac{1}{2}\dot{\boldsymbol{p}}_Q^T M_Q \dot{\boldsymbol{p}}_Q + \frac{1}{2}\dot{\boldsymbol{p}}_L^T M_L \dot{\boldsymbol{p}}_{\boldsymbol{L}} + \frac{1}{2}\boldsymbol{\omega}_{\boldsymbol{B}}^T I \boldsymbol{\omega}_{\boldsymbol{B}} \tag{3.32}$$

where $M_Q = diag(m_Q, m_Q, m_Q)$ and $M_L = diag(m_L, m_L, m_L)$ are diagonal matrices of the mass of the quadrotor and payload respectively, and $I = diag(I_{xx}, I_{yy}, I_{zz})$ is a diagonal matrix of the moments of inertia of the quadcopter.

The potential energy of the system is based on $V = mgz$ as presented in (3.33).

$$V = m_Q g z_Q + m_L g z_L \tag{3.33}$$

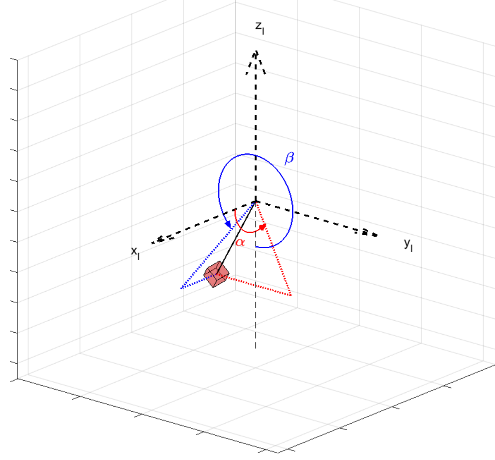


Figure 3.8: The modified angles used to measure the position of the cable relative to the inertial frame to prevent the occurrence of a singularity at the equilibrium. α denotes the swing angle of the cable measured from the x_I axis and β denotes the rotational position of the cable measured about the x_I axis from the $-z_I$ axis. Both angles are measured using the right hand notation.

The external forces and moments acting on the quadcopter are the propeller thrusts and induced moments, the weights of the quadcopter and payload, and any external disturbances. These forces and moments can be found detailed in the Section 3.2. Furthermore, note that all these forces and moments must be expressed in the inertial frame which can be achieved by utilising the rotation matrix (3.1).

The Lagrangian of the system is obtained by subtracting the total potential energy from the total kinetic energy of the system as shown in (3.34).

$$L = T - V = \left[\frac{1}{2}\dot{\boldsymbol{p}}_Q^T M_Q \dot{\boldsymbol{p}}_Q + \frac{1}{2}\dot{\boldsymbol{p}}_L^T M_L \dot{\boldsymbol{p}}_{\boldsymbol{L}} + \frac{1}{2}\boldsymbol{\omega}_{\boldsymbol{B}}^T I \boldsymbol{\omega}_{\boldsymbol{B}}\right] - [m_Q g z_Q + m_L g z_L] \tag{3.34}$$

Each equation of motion is then obtained through taking derivatives of (3.34) with respect to each variable in $\boldsymbol{q}$ and its time derivative $\dot{\boldsymbol{q}}$ in turn to obtain the terms required to compute the equation of motion in the form of (2.18) on Page 14.

An issue occurs when observing the equation of motion obtained for the β swing angle shown in (3.35).

$$\ddot{\beta} = \frac{1}{C_\alpha^2 \lambda - \lambda}\left[\frac{\tau_\beta}{-m_L \lambda} - S_\alpha \left(-2\dot{\beta}\dot{\alpha}C_\alpha \lambda + \ddot{y}S_\beta + \ddot{x}C_\beta\right)\right] \tag{3.35}$$

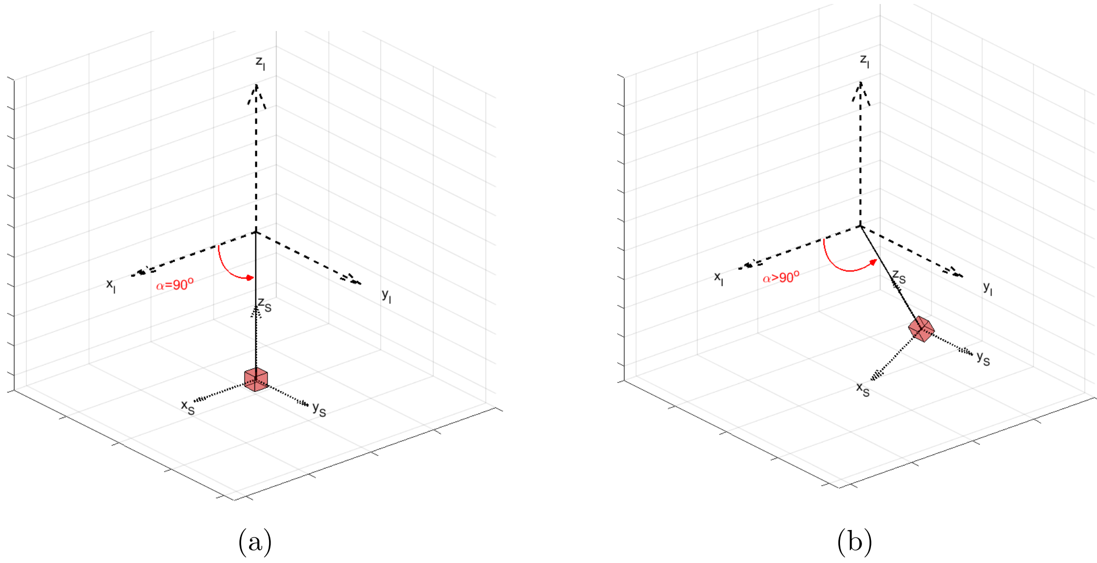
At equilibrium when $\alpha = 0 \rightarrow C_\alpha = 1$. At this condition, $C_\alpha^2 \lambda - \lambda = 0$ will cause $\ddot{\beta} \rightarrow \infty$, leading to model failure. The state of $\alpha = 0$ is frequently visited as the payload swings from side to side while the quadcopter is maneuvering.

To mediate the singularity issue present in (3.35), the spherical coordinates are altered such that they are measured about the x_I axis rather than the z_I axis as shown in Figure 3.8. This results in $\alpha = 90^\circ$ at equilibrium, preventing $\ddot{\beta}$ from going to infinity at the equilibrium point [54]. The rotation matrices for the new spherical angles to convert the payload's position are also different. The α angle can be viewed as a right handed rotation about the y_I axis and β about the x_I axis. This means that the required rotation matrix is now ${}^I_S R = R_x(\beta) R_y(\alpha)$.

Using this modified method for measuring the spherical coordinate angles changes the $R_y(\alpha)$ rotation matrix. When $\alpha = 90^\circ$, the $\{S\}$ and $\{I\}$ frames' axes are aligned as shown in Figure

3.9a. The frames only become offset when $\alpha \neq 90^{\circ}$ as seen in Figure 3.9b. This means that the right hand rotational offset between frames $\{I\}$ and $\{S\}$ is $\alpha + 90^{\circ}$, resulting in the ${}^{I}_{S}R$ rotation matrix shown in (3.36).

$$
\begin{aligned}
{}^{I}_{S}R = R_x(\beta)R_y(\alpha) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & C_\beta & -S_\beta \\ 0 & S_\beta & C_\beta \end{bmatrix} \begin{bmatrix} C_{\alpha+90^{\circ}} & 0 & S_{\alpha+90^{\circ}} \\ 0 & 1 & 0 \\ -S_{\alpha+90^{\circ}} & 0 & C_{\alpha+90^{\circ}} \end{bmatrix} \\
\longrightarrow {}^{I}_{S}R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & C_\beta & -S_\beta \\ 0 & S_\beta & C_\beta \end{bmatrix} \begin{bmatrix} S_\alpha & 0 & -C_\alpha \\ 0 & 1 & 0 \\ C_\alpha & 0 & S_\alpha \end{bmatrix} &= \begin{bmatrix} S_\alpha & 0 & -C_\alpha \\ -S_\beta C_\alpha & C_\beta & -S_\beta S_\alpha \\ C_\beta C_\alpha & S_\beta & C_\beta S_\alpha \end{bmatrix}
\end{aligned} \tag{3.36}
$$



(a) (b)

Figure 3.9: Figure showing the alignment of frames $\{S\}$ and $\{I\}$ at $\alpha = 90^{\circ}$ and $\alpha > 90^{\circ}$

Note that this change in the rotation matrix does not apply to the $R_x(\beta)$ rotation matrix since defining $\beta = 0$ when $\{S\}$ is aligned with $\{I\}$ presents no complications and hence requires no alterations.

Since ${}^{I}_{S}R$ has changed, $\boldsymbol{p}_L$ is now re-evaluated accordingly in (3.37).

$$
\begin{aligned}
\boldsymbol{p}_L &= \boldsymbol{p}_Q + {}^{I}_{S}R \begin{bmatrix} 0 & 0 & -\lambda \end{bmatrix}' \\
\longrightarrow \begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix} &= \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} S_\alpha & 0 & -C_\alpha \\ -S_\beta C_\alpha & C_\beta & -S_\beta S\alpha \\ C_\beta C_\alpha & S_\beta & C_\beta S_\alpha \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -\lambda \end{bmatrix} \\
\longrightarrow \begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix} &= \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} C_\alpha \lambda \\ S_\beta S_\alpha \lambda \\ -C_\beta S_\alpha \lambda \end{bmatrix}
\end{aligned} \tag{3.37}
$$

Following the adjustment of α, β and $\boldsymbol{p}_L$, the equations of motions were recalculated using the previously described procedure. The new $\ddot{\beta}$ equation, shown in (3.38), no longer contains a singularity at the equilibrium point which is now $\alpha = 90^{\circ}$.

$$
\ddot{\beta} = \frac{1}{-C_\alpha^2 \lambda + \lambda} \left[\frac{\tau_\beta}{\lambda m_L} - S_\alpha \left(2\dot{\beta}\dot{\alpha} C_\alpha \lambda + S_\beta g + \ddot{z} S_\beta + \ddot{y} C_\beta \right) \right] \tag{3.38}
$$

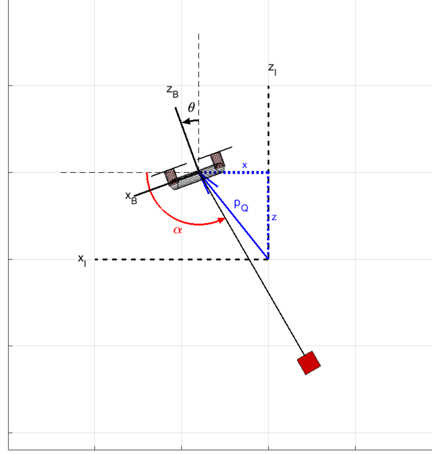


Figure 3.10: Figure of the planar model of the quadcopter in the $X - Z$ plane.

Although there is no longer a singularity at the equilibrium, there is still a singularity as $-C_{\alpha}^{2}\lambda + \lambda \rightarrow 0$ when $\alpha = 0^{\circ}$. For the purpose of this project, this does not pose issues as reaching this singularity requires the payload to swing aggressively, implying unstable or extremely fast UAV maneuvers which will not be applied in this work.

3.3.4 Simplified Equations of Motion For Training

Due to the symmetric nature of the quadcopter, the full model was able to be reduced to the planar model shown in Figure 3.10. This simplifies the control strategy and increase the computational efficiency of the training procedure, greatly reducing the complexity of tuning and building the reinforcement learning agent for x and y waypoint tracking purposes. The simulation portion of this thesis focuses on achieving waypoint tracking in the x-axis by controlling the pitch θ of the UAV. To achieve this two dimensional system of equations, y, ϕ, ψ and β and all their time derivatives are set to zero to achieve the model shown below.

$$\ddot{x} = \frac{1}{m_L + m_Q}\left[S_{\theta}T + m_L\lambda\left(\dot{\alpha}^2 C_{\alpha} + S_{\alpha}\ddot{\alpha}\right)\right] \tag{3.39}$$

$$\ddot{z} = \frac{1}{m_L + m_Q}\left[C_{\theta}T - m_L\lambda(\dot{\alpha}^2 S_{\alpha} - \ddot{\alpha}C_{\alpha}) - g\left(m_L + m_Q\right)\right] \tag{3.40}$$

$$\ddot{\alpha} = \frac{1}{\lambda}\left[\ddot{x}S_{\alpha} + C_{\alpha}\left(g + \ddot{z}\right)\right] \tag{3.41}$$

$$\ddot{\theta} = \frac{\tau_{\theta}}{I_{yy}} \tag{3.42}$$

Since the control input to the system was chosen to be θ, (3.42) was omitted from the model and (3.39) - (3.41) were utilised for training the controller.

Runge Kutta Differential Equation Solver

Equations (3.39) - (3.41) return the second order derivative of the states. However, interest lies in the position and velocity states rather than the acceleration computed by the second derivatives. To obtain x, z, α and their velocities, a *Fourth Order Runge Kutta (RK4)* Ordinary

Differential Equation (ODE) solver was implemented. The details of the solver can be found in Appendix C.

Chapter 4

Control Algorithm Development

This chapter provides an overview of the components within an RL agent. It explores the function of a policy, guiding the agent in maximizing cumulative rewards, and introduces the concept of a value function to quantify these rewards. The potential incorporation of a dynamic model for the environment is also briefly discussed.

In RL, a Markov Decision Processes (MDPs) is used to convey the environment to the agent. This encompasses the ideas of the return G_t and the expected return in the form of a value function $V(s)$. The overview of the derivation of Bellman and Bellman optimality equations for state-value and state-action value functions is covered before moving onto the RL algorithm that can be utilised to solve them.

For this thesis, the chosen algorithm was a Temporal Difference (TD) algorithm due to its model-free application and fast learning capabilities. This section explains the theory behind TD and presents one-step *TD* algorithms for prediction of a value function for some policy π, and control through converging to some optimal state-action value function Q_*.

Since the quadcopter will be operating in infinite state and action spaces, approximate solution methods which prompt the RL agent to generalise are discussed. The idea of using Semi Gradient Descent (SGD) and Ascent (SGA) methods for updating a weight vector $\boldsymbol{w}$ and a parameter vector $\boldsymbol{\theta}$ that enable the value function and policy to generalise over the entire state space are presented. Furthermore, the method of utilising Fourier basis to approximate the value function along with a Gaussian distribution to act as a policy which facilities action selection is described,

Finally, the algorithm constructed for the experiment is presented using all the previously discussed theories. This algorithm can be classified as a policy gradient $TD(0)$ actor-critic algorithm with a Fourier cosine basis linear value function approximation method.

The objective of the controller is to lead the x position of the quadcopter to the goal x position x_{goal} without the aid of a dynamic model. Mathematically, this can be described as the limit of the x position error e_x approaching zero as time t goes to infinity, as shown in (4.1). This is visually represented in Figure 4.1.

$$\lim_{t\to\infty} e_x = \lim_{t\to\infty} (x - x_{goal}) \to 0 \tag{4.1}$$

In addition, the controller must operate fully on board with no external computers performing controller computations making the autonomous UAV is fully self-sufficient.

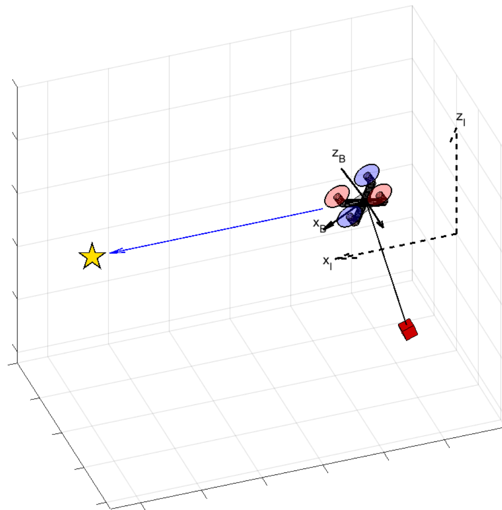


Figure 4.1: Figure of the path the quadcopter must take to reduce the x position error. Here, the x position goal is represented using a yellow star.

Reinforcement learning based controllers quantify the position error using a reward function $R = f(e_x)$ which is then applied to produce the required control input. To employ the reward signal, multiple reinforcement learning components are required to work interactively with the goal of reducing the error. The next section of this thesis covers components of RL components and their implementations to solve the control problem at hand.

4.1 Reinforcement Learning Review [55]

Machine Learning (ML) can be classified into three broad types: Supervised Learning, Reinforcement Learning and Unsupervised Learning. Supervised learning is a method based on learning from a set of labeled examples. It is expected that in any given situation the system will look back to the provided examples to generalise and take the best action even if its not explicitly provided in the training set. This type of learning has its benefits; however, in unfamiliar territory, it is near impossible to find representative and accurate data from a set of provided examples. In these cases the system would have to learn from its own experience. This requires the system to use reinforcement learning. RL also has advantages over traditional non-linear control methods due to its ability to adapt to complex environments and unpredictable dynamics, its continuous improvement through experience, its capability to generalise, and the ability to transfer an agent's learned capabilities to the next.

Reinforcement learning is the process by which a system trains on the optimal action at a given state. In other words, RL is the process of learning how to map states to actions to maximise the reward. Unlike supervised learning, the agent, which acts as the brain of the RL algorithm, is not told what actions to take, rather it must discover and deduce the action that results in the highest sum of rewards through trial and error. Rewards are numerical values that quantify how far the system is from its goal. This trial and error feature along with an estimate of the possible future rewards distinguishes RL from other learning methods. For reinforcement learning, a clear distinction between the problem and solution methods is extremely important. The problem in large is to enable a system to explore its environment enough such that it realises what actions it should take to achieve a set goal. The solution to this is to employ an agent that has the ability to sense its environment such that it can explore and perform actions accordingly to achieve learning. The agent must decide between exploring new options and exploiting known information to maximize overall rewards. Exploration involves trying out different actions to uncover unknown aspects of the environment, offering the potential of discovering better strategies. However, exploration comes with the risk of receiving suboptimal rewards as the agent ventures into uncharted territory. Exploitation entails choosing actions known to yield high rewards based on the agent's current knowledge, maximizing short-term gains. Excessive exploitation may lead to a suboptimal long-term strategy if better options remain undiscovered. Finding the right balance between exploration and exploitation is crucial for effective reinforcement learning algorithms, influencing the agent's ability to adapt to dynamic environments and converge towards an optimal policy.

It is important to distinguish between RL and unsupervised learning. Unsupervised learning is a type of learning associated with finding patterns in unlabeled data and using this information to make decisions. On the other hand, RL starts with no assumptions about its environment or data. Reinforcement learning agents must learn their environment through pure exploration and use that data to determine appropriate actions. This makes RL an invaluable tool in systems operating in complex environments or exhibiting unpredictable dynamics like slung load systems.

4.1.1 Elements of Reinforcement Learning

There are four main elements in Reinforcement Learning: a *policy*, a *reward signal*, a *value function*, and an optional *model of the environment*.

The Policy

An RL policy dictates to an agent which action to take at a given time. A policy can be thought of as a mapping from states to actions by defining the action the agent takes depending on its perceived state.

Policies take many forms such as simple functions, lookup tables, or prompts that involve heavy computations. The policy can be seen as the core component of the RL agent as it alone can determine the behaviour of the system. In the example of a self driving car, the policy would be the sense and avoid system: the sensors will send back information regarding obstacles to the policy and the policy will then determine the best course of action. For example, if there is an obstacle ahead of the car, the policy will prompt the car to move around it.

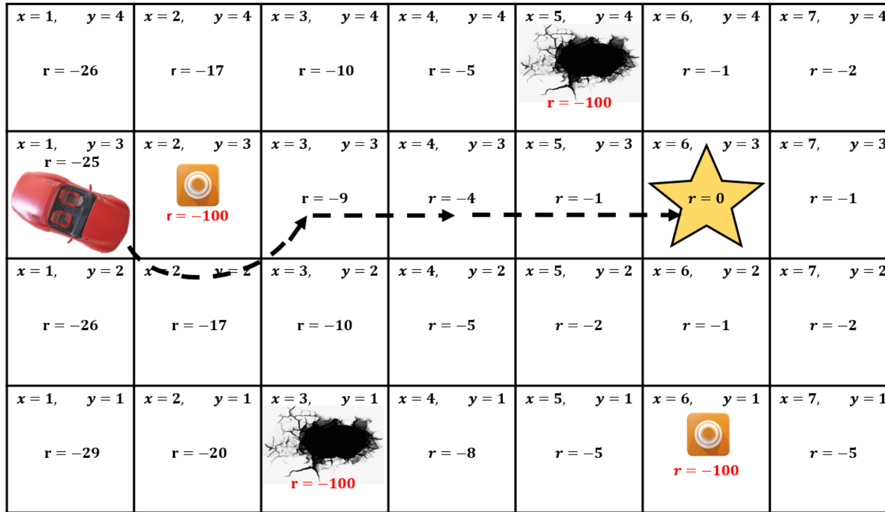


Figure 4.2: Reward values around the vicinity of a goal state for a self driving car.

The Reward Signal

The reward signal's purpose is to define and quantify the goal of the reinforcement learning problem. It provides the agent with a reward based on its state. Rewards are an immediate defining feature of the problem at hand. If the system is in an advantageous state the agent is awarded with a high reward value. Alternatively, if the system is in a disadvantageous state the agent is awarded with a low reward.

After receiving a reward from the reward signal it is then left to the agent to follow the path that leads to the goal and simultaneously maximise the sum of these rewards. This makes the reward signal the primary source to fulfill exploitation as well as policy improvement of the agent. Policy improvement can be achieved in many ways; however, in its simplest form: If at a given state the action selected yields a low reward, the policy is then altered such that the agent will choose a different action if it arrives at that same state in the future. Relating back to a self driving car, the reward can be a function of the distance from the goal state. For example: $R(s) = -\left[(x - x_{goal})^2 + (y - y_{goal})^2\right]$ and any states with obstacles are associated with an extremely low reward of $r = -100$. Using this function, the closer the state of the car is to the goal state, the higher the reward unless there is an obstacle. This will reward the car greater as it gets closer to the goal while prompting it to avoid obstacles as illustrated in the Figure 4.2.

The Value Function

Like reward signals, value functions also feed reward information to the agent. However, unlike the reward signal which only provides information on the current reward, the value function attempts to provide the sum of the rewards until reaching the terminal state given the current state and chosen action. While the reward signal indicates to the agent what is immediately the best action to take, the value function will indicate to the agent the best action to take for the long run. An immediate action could lead to a low reward at this time but it could make future states with considerably high rewards accessible.

Going back to the self driving car example, assume that the car is given a goal which is located somewhere ahead of it while it is in the middle of a drive. It then has to choose between two

Figure 4.3: Two one-way paths that a self driving car can take to reach its goal. The red path has a higher value than the blue path.

one-way roads it can take to reach the goal: a longer road whose entrance is located ahead represented as a solid blue path, or a shorter road whose entrance is located some distance behind the car represented as a red dashed path as shown in Figure 4.3.

If the car favours immediate rewards it will always exploit and choose the action corresponding to the highest reward at any state. This means that the car will not back-up to take the red path as that will require the car to initially choose the lower reward of -45 to move in the direction of the shorter path's entrance. This means that the car will opt for the blue path. If the car favours higher values, the car will back up and take the red path. In this case, assuming that the car is operating in an environment lacking uncertainty, the value function can simply be obtained by summing up the rewards of both paths. This results in a value of -548 for the red path and -1152 for the blue path.

In terms of action choices, actions are taken to maximise the value and not necessarily the reward. This is to ensure that the agent achieves a high reward in the long run, ensuring the RL agent achieves its final goal in the best possible manner. Since rewards are simply fed by the environment to the agent, it is significantly easier to obtain rewards than it is to obtain values. This is because values are an estimate of all future rewards, and in stochastic (real-life) environments, explained in Appendix D, taking a certain action does not necessarily guarantee a particular outcome, making value functions notably difficult to estimate. As a consequence, values are estimated and iteratively re-estimated through the observations that the agent makes over its lifetime.

The Model

The final and optional component of an RL algorithm is the dynamical model of the system itself. A dynamical model allows for predictions of the system and its environment to be made. Given a state and action, the model will make a prediction and provide the possible next state which will be used to compute the next reward.

Models are used by the RL agent to plan. The agent will use the model to determine a course of action based on the possible future scenarios provided by the model prior to taking an action. These types of methods are called model-based reinforcement learning methods.

The other type of RL algorithms are model-free. These types of algorithms will learn based on trial and error. They will essentially attempt to estimate their own environment in the form of a Markov Decision Process (MDP) and use that to plan.

4.1.2 Finite Markov Decision Processes [56] [57]

A Markov decision process is a formal way of conveying the environment for RL. For the sake of simplicity, an assumption is made that the agent has full observability of its environment. That is, the agent can determine its exact state through sensing its surroundings. To understand MDPs one must first begin with the fundamental concept of the Markov property then add layers of complexity to obtain Markov Processes (MPs), followed by Markov Reward Processes (MRPs) and finally Markov Decision Processes (MDPs).

The Markov Property

A state is considered Markov if *"The future is independent of the past given the present"*. In other words, the current state is enough of a statistic to determine the future state. For a state S_t to be Markov the probability of the next state being S_{t+1} relies solely on the value of S_t and doesn't require the history $S_1, ..., S_t$. Mathematically, this can be described by the following equation:

The Markov Property

A state S_t is Markov if and only if:

$$\mathbb{P}\left[S_{t+1}|S_t\right] = \mathbb{P}\left[S_{t+1}|S_1, ..., S_t\right] \tag{4.2}$$

Markov Processes (MPs)

A Markov process, also known as a Markov chain, is simply a memoryless, random chain of sequential states $S_1, S_2, S_3, ...$ where all the states obey the Markov property. To obtain a Markov chain, a current state and a way to quantify the next state through determining the dynamics of a system are required. Hence, a Markov chain can be described as a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$ where $\mathcal{S}$ is the agent's state space and $\mathcal{P}$ is the system's dynamics. It can be said that a Markov chain is a method of describing the dynamics of a system. To achieve a clearer understanding of $\mathcal{P}$, refer to Appendix E which discusses the dynamics of a system as a simple state transition matrix.

A sample *episode* from a Markov process is a possible sequence of Markov states that ends in a terminal state: $S_1, S_5, S_8, S_7, S_{10}, ..., S_{terminal}$. A terminal state can be determined by either the length of the episode (e.g. after taking 1000 steps terminate the episode) or by reaching a terminal state (e.g. when the goal is reached terminate the episode).

Markov Process

A Markov process (chain) is a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$ where :
- $\mathcal{S}$ is a finite set of states ($s \in \mathcal{S}$) defining the state space.
- $\mathcal{P}$ determines the dynamics of the system by returning the next stat s' given the current state s.
  $\mathcal{P}_{ss'} = \mathbb{P}\left[S_{t+1} = s' | S_t = s\right]$

$$S_1, S_2, S_3, S_4, ...$$

Markov Reward Processes (MRPs)

A Markov reward process is a Markov chain with a reward function associating a reward r to every state. The tuple in this case is $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where $\mathcal{R}$ is a reward function and γ is the discount factor. This introduces the whole essence of reinforcement learning, which is that the agent aims to maximise the cumulative sum of the rewards.

Markov Reward Process

A Markov reward process is a tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where:
- $\mathcal{S}$ is a finite set of states ($s \in \mathcal{S}$) defining the state space.
- $\mathcal{P}$ determines the dynamics of the system by returning the next state s' given the current state s.
  $\mathcal{P}_{ss'} = \mathbb{P}\left[S_{t+1} = s' | S_t = s\right]$
- $\mathcal{R}$ is the reward function computing a reward at every state visited given the current state s.
  $\mathcal{R}_s = \mathbb{E}[r_t | S_t = s]$
- $\gamma \in [0, 1]$ is the discount factor

The sum of rewards is computed as a weighted sum of all the rewards of an episode, better known as the *return*:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{4.3}$$

The return can also be written in terms of the next step's return G_{t+1}.

$$\begin{aligned} G_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ... \\ &= r_t + \left(\gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ...\right) \\ &= r_t + \gamma \left(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...\right) \\ G_t &= r_t + \gamma G_{t+1} \end{aligned} \tag{4.4}$$

The discount factor γ will determine whether more weight is put on current or future rewards. A discount factor of $\gamma = 0$ will result in the agent being "myopic" and only chasing immediate rewards. On the other hand, a discount factor close to $\gamma = 1$ will produce a "far-sighted" agent which will consider future rewards heavily.

Discounting the return signal serves many purposes both computationally and performance wise. An undiscounted return ($\gamma = 1$) in cases where an episode is not guaranteed to be finite can result in an infinite return in cyclic Markov chains where the agent is unable to reach a

terminal condition. There may also be uncertainty regarding the future, so weighing all rewards equally can adversely effect the agent in reaching its goal when future rewards do not turn out as predicted. Furthermore, when trying to mimic animalistic behaviours, it is beneficial to weigh immediate rewards higher as that is the biological tendency of most species. That being said, in Markov chains in which termination is guaranteed, one can implement an undiscounted return for mathematical convenience.

In stochastic environments (refer to Appendix D.2) starting at the same start state s will result in different episodes and therefore different returns G_t. The simplest method for approximating the return of the state s is to generate many episodes that start at s and take the average of all the returns. This expected return is the *value function* $V(s)$:

$$V(s) = \mathbb{E}[G_t | S_t = s] \tag{4.5}$$

In deterministic environments, the value and return of a state will be equal since all episodes generated from a given starting state will be equal.

Much like the return in the form given in (4.4), the value function can also be decomposed into the immediate reward r_t and the next step discounted value $\gamma V(S_{t+1})$, this form of the value function shown in (4.6) is called the *Bellman equation*.

$$V(s) = \mathbb{E}\left[r_t + \gamma V(S_{t+1}) | S_t = s\right] \tag{4.6}$$

Equation (4.6) can be analysed further by looking one step ahead to the next state s'. The current state value $V(s)$ is then expressed as a function of the immediate reward r_t as well as the discounted average of all the possible outcomes of the future sum of rewards using the probability distribution from the dynamics model $\gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s)V(s')$.

$$\begin{aligned} V(s) &= r_t + \gamma \left[\mathcal{P}_{ss_1} V(s_1) + \mathcal{P}_{ss_2} V(s_2) + ... + \mathcal{P}_{ss_n} V(s_n)\right] \\ &= r_t + \gamma \sum_{s' \in S} \mathcal{P}_{ss'} V(s') \end{aligned} \tag{4.7}$$

Moreover, the Bellman equation (4.7) can also be written in matrix form as presented in (4.8).

$$V = \mathcal{R} + \gamma \mathcal{P} V \tag{4.8}$$

where V and $\mathcal{R}$ are column vectors of one entry per state which correspond to the value and reward associated to each state.

For clarity (4.9) shows the matrices in (4.8) in their actual matrix form.

$$\begin{bmatrix} V(s_1) \\ \vdots \\ V(s_n) \end{bmatrix} = \begin{bmatrix} r(s_1) \\ \vdots \\ r(s_n) \end{bmatrix} + \gamma \begin{bmatrix} \mathbb{P}[s_1|s_1] & \dots & \mathbb{P}[s_n|s_1] \\ \vdots & & \\ \mathbb{P}[s_1|s_n] & \dots & \mathbb{P}[s_n|s_n] \end{bmatrix} \begin{bmatrix} V(s_1) \\ \vdots \\ V(s_n) \end{bmatrix} \tag{4.9}$$

The Bellman equation can be solved directly for V to give:

$$V = (I - \gamma \mathcal{P})^{-1} \mathcal{R} \tag{4.10}$$

This direct solution method is only possible for smaller Markov reward processes. For larger reward chains iterative RL methods like dynamic programming (explained in Appendix F), Monte-Carlo (MC) (explained in Appendix G), and Temporal Difference (TD) learning (explained in Section 4.1.3) are required.

Markov Decision Processes (MDPs)

Markov decision processes are MRPs in which the agent must make action decisions based on its environment. The MDP tuple is $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. Where $\mathcal{A}$ is a finite set of actions. In this case, the state probability matrix $\mathcal{P}$ and the reward function $\mathcal{R}$ will no longer depend only on the state, but will also be dependent on the action that the agent decides to take.

Markov Decision Process

A Markov decision process is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where:

- $\mathcal{S}$ is a finite set of states ($s \in \mathcal{S}$) defining the state space.
- $\mathcal{A}$ is a finite set of actions ($a \in \mathcal{A}$) defining the action space.
- $\mathcal{P}_{ss'}$ determines the dynamics of the system by returning the next state s' given the current state s and chosen action a.
  $\mathcal{P}^a_{ss'} = \mathbb{P}\left[S_{t+1} = s' | S_t = s, A_t = a\right]$
- $\mathcal{R}$ is the reward function computing a reward at every state visited given the current state s and chosen action a.
  $\mathcal{R}^a_s = \mathbb{E}[r_t | S_t = s, A_t = a]$
- $\gamma \in [0, 1]$ is the discount factor

In MDPs there are policies as mentioned in section 4.1.1. A policy π is simply *"a distribution over actions given states"*. This means that a policy will assign a probability to each action given the state that the agent finds itself in, fully dictating its behaviour and performance.

$$\pi(a|s) = \mathbb{P}[A_t = a | S_t = s] \tag{4.11}$$

Since MDPs abide by the Markov property, they are only dependent on the current state, This means that the policies are also time-independent.

Since the state transition dynamics (see Appendix E) and reward signal are directly linked to the behaviour of the agent, they are also dependent on the policy. The dynamics and rewards are therefore taken to be the average over the policy for all transition dynamics and reward probabilities respectively.

$$\mathcal{P}^{\pi}_{s,s'} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}^a_{s,s'} \tag{4.12}$$

$$\mathcal{R}^{\pi}_{s} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}^a_{s} \tag{4.13}$$

The value function for an MDP is also effected by the implementation of a policy π. This value function is now the expected return for starting at a state s and following the policy π.

$$\begin{aligned} V_{\pi}(s) &= \mathbb{E}_{\pi}[G_t | S_t = s] \\ &= \mathbb{E}_{\pi}[r_t + \gamma V_{\pi}(S_{t+1}) | S_t = s] \end{aligned} \tag{4.14}$$

With the introduction of the action space $\mathcal{A}$ comes a new definition, the *action-value function* Q. The action-value function is the expected return from starting at a state s, taking an action a and then following a policy π. It can also be decomposed similarly to the state value function into its own form of the Bellman equation.

$$\begin{aligned} Q_{\pi}(s, a) &= \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_{\pi}[r_t + \gamma Q_{\pi}(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \end{aligned} \tag{4.15}$$

The equation of the state value function $V_\pi(s)$ in terms of the state-action value function $Q_\pi(a, s)$ can be obtained when considering the action distribution $a_1, a_1, ..., a_m$ over a state s as illustrated in Figure 4.4. The state value function can be obtained by taking the sum of the probability distribution of the possible actions multiplied by their corresponding action-value function as shown in (4.16).

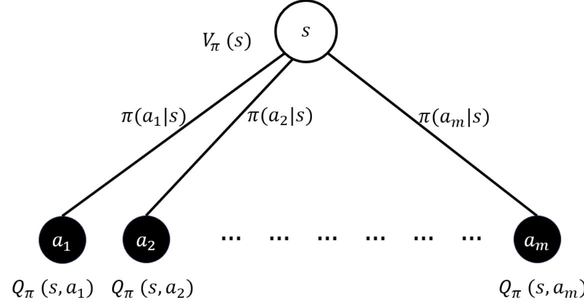


Figure 4.4: Figure showing the action distributions dictated by policy π when starting from state s.

$$\begin{aligned} V_\pi(s) &= \pi(a_1|s)Q_\pi(s, a_1) + \pi(a_2|s)Q_\pi(s, a_2) + ... + \pi(a_m|s)Q_\pi(s, a_m) \\ &= \sum_{a\in\mathcal{A}} \pi(a|s)Q_\pi(s, a) \end{aligned} \tag{4.16}$$

Figure 4.5 shows the possible resulting states $s_1, s_2, ..., s_n$ from being in state s and taking action a. The relationship between $V_\pi(s)$ and $Q_\pi(s, a)$ can be achieved as $Q_\pi(s, a)$ in terms of $V_\pi(s)$. $Q_\pi(s, a)$ can be computed by adding the immediate reward r_S^a to the discounted sum of the probability distribution of the possible states multiplied by their corresponding value function as shown in (4.17).

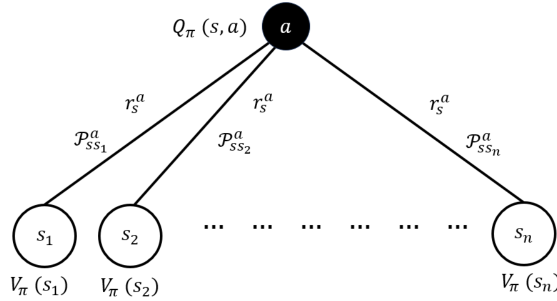


Figure 4.5: Figure showing the state distribution dictated by system dynamics $\mathcal{P}$ when starting from state s and taking action a.

$$\begin{aligned} Q_\pi(s, a) &= r_s^a + \gamma \left[\mathcal{P}_{ss_1}^a V_\pi(s_1) + \mathcal{P}_{ss_2}^a V_\pi(s_2) + ... + \mathcal{P}_{ss_n}^a V_\pi(s_n)\right] \\ &= r_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a V_\pi(s') \end{aligned} \tag{4.17}$$

By substituting (4.16) into (4.17) and vice versa the V_π and Q_π recursive Bellman expectation equations (4.18) and (4.19) are obtained. These equations are ultimately what are used to solve MDPs in more complex environments.

$$V_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left(r_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a V_\pi(s') \right) \tag{4.18}$$

$$Q_\pi(s,a) = r_s^a + \gamma \sum_{s' \in \mathcal{S}} \left[\mathcal{P}_{ss'} \left(\sum_{a' \in \mathcal{A}} \pi(a'|s') Q_\pi(s',a') \right) \right] \tag{4.19}$$

Optimal Value Function

There exists a value function which is optimal over all policies. This will specify the best performance for the MDP.

Optimal Value Function

The optimal state value function is the maximised value function over all policies

$$V_*(s) = \max_\pi V_\pi(s)$$

The optimal action-state value function is the maximised action-value function over all policies

$$Q_*(s) = \max_\pi Q_\pi(s)$$

The optimal value function is considered to be the solution for the MDP, signaling a resolved MDP.

Optimal Policy

As mentioned previously, the policy is ultimately what determines the behaviour of the system within its environment. Therefore, to optimise the behaviour of the system, the optimal policy π_* must be computed. This can be achieved through analysing the value function. If the value of being in state s and following policy a π is greater than the value of being in the same state and following a policy π' then it can be implied that policy π is better than policy π' if this applies for all the states in the state space.

$$\pi \geq \pi' \text{ if } V_\pi(s) \geq V_{\pi'}(s), \forall s$$

For any MDP there is a best policy π_* that outperforms or is equally as good as any other policy π. Following said optimal policy achieves the optimal state and state-action value functions.

Optimal Policy

The optimal policy is one is which is the best among all possible policies, ensuring that it either performs better than or matches the performance of any other policy.

$$\pi_* \geq \pi, \forall \pi$$

Every optimal policy attains the optimal state and state-action value functions.

$$V_{\pi_*}(s) = V_*(s)$$
$$Q_{\pi_*}(s) = Q_*(s)$$

Finding the optimal policy can be achieved through maximising over $Q_*(s,a)$.

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q_*(s,a) \\ 0 & otherwise \end{cases} \tag{4.20}$$

This implies that if $Q_*(s,a)$ is known, then the optimal policy can be immediately found. Finally, the Bellman optimality equations for obtaining the optimal value functions are shown in (4.21) and (4.22), stemming from (4.18) and (4.19).

$$V_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_*(s') \tag{4.21}$$

$$Q_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} Q_*(s') \tag{4.22}$$

These Bellman optimality equations are nonlinear and have no general solution, requiring specific iterative methods to solve them. These methods are value and policy iteration methods, Q-learning and State-Action-Reward-State-Action (SARSA). The type of iterative solution which can be implemented depends on the RL algorithm chosen by the user depending on the system at hand.

The three main types of RL algorithms are Dynamic Programming, Monte Carlo and Temporal Difference. Each set of methods has its own applications and uses. That being said, TD is most often what is implied when speaking of RL, specifically for time-sensitive control purposes. In this thesis the control methods covered will be strictly on-policy as they are the type relevant for this research. That is, the policy improved is the same one used to make action choices. The alternative would be off-policy control where the policy updated is different than the one used to take actions.

4.1.3 Temporal Difference Learning

The main objective of all RL algorithms is to compute the optimal state or state-action value function as described by (4.21) and (4.22) which then facilitates computing the optimal policy π_*. This can be split into two separate problems: policy evaluation (prediction) and policy improvement (control). Policy evaluation seeks to estimate the state value V_π or state-action value Q_π of a given policy π, quantifying how good the policy is. Policy improvement aims to enhance a policy with the knowledge of its value from the policy evaluation step to converge to some optimal policy π_*.

Dynamic programming is an umbrella term for RL algorithms that solve for the optimal policy assuming a perfect model of the environment is provided as a finite MDP. These types of algorithms are categorised as model-based RL algorithms and are not applicable for many real life implementations as in most cases a perfect model cannot be provided. However, DP introduces the ingenious concept of *bootstrapping* in which the estimated value of the next state $V_k(s')$ is used to improve the prediction of the value of the current state $V_{k+1}(s)$.

$$V_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(r_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_k(s') \right) \tag{4.23}$$

For a detailed overview of DP algorithms refer to Appendix F.

In applications where the MDP of the system is not provided, the system must learn from its own experience without any previous knowledge of the environment. Monte Carlo methods use samples of encountered state sequences, actions and rewards to deduce the optimal value and policy. These algorithms sample and average returns like the one in (4.3) for every state-action pair. This enables the reinforcement learning algorithm to reach the optimal value function and policy without the aid of a model. The major drawback of Monte Carlo methods is that it they only update the value function and policy at the end of an episode, resulting in a slow convergence to the optimal policy which is not suitable for real time learning in systems with fast dynamics.

Temporal Difference learning combines the beneficial concepts encompassed in both the DP and MC methods. TD methods, like their MC counterparts, learn from experience and do not require a model of the environment. They also update the value function using the bootstrapping technique like DP algorithms which allows value function updates to occur within the episode rather than only taking place at the end of the episode enable it to cope with rapid dynamics.

Temporal Difference Prediction $TD(0)$

The TD prediction portion of the algorithm focuses on the estimation of the value function V_π given some policy π without an MDP. The difference between MC and TD prediction algorithms is that the Monte Carlo update requires for the episode to terminate such that the actual reward G_t can be computed. This return G_t is called the MC target since the value function is updated to move towards it by some small learning step size α.

$$V(s_t) \leftarrow V(s_t) + \alpha\left(\underline{G} - V(s_t)\right)$$

Unlike MC methods, TD learning only waits for a certain number of steps to update the value function. Since in the middle of an episode the actual value of the return G_t is unknown, the best the agent can do without the dynamics model is to analytically estimate it. Using the reward r_t and value function estimate of the next state $V(s_{t+1})$, a new target (underlined in (4.24)) is used. This new target is appropriately named the TD target and acts in place of the return G_t. This update of $V(s_t)$ utilises the bootstrapping method since the TD target is constructed using the value of the next state. Another metric is the TD error denoted by δ. This is simply the difference between the TD target and the value estimate of the current state $V(s_t)$, which is the part in the square brackets in update equation below. Shrinking δ over time indicates successful learning of the system due to the decrease in the difference between what is expected to be the value (the TD target) and what is the current value estimate ($V(s_t)$).

$$V(s_t) \leftarrow V(s_t) + \alpha\left[\underline{(r_t + \gamma V(s_{t+1}))} - V(s_t)\right] \tag{4.24}$$

The policy evaluation method shown in Algorithm 1 is called the $TD(0)$ method and is a special form of the $TD(n)$ and $TD(\lambda)$ methods as it updates the value function estimate at every step.

The other TD prediction algorithms are out of scope of this thesis. Information on the $TD(n)$ and $TD(\lambda)$ algorithms can be found in Chapters 7 and 12 of [55].

Temporal Difference Control (SARSA)

Since the model is not provided, a state-action value function must be utilised for policy improvement such that the policy is greedy, choosing the maximum $Q(s, a)$ across all actions. This can be achieved through the implementation of an ϵ - greedy policy.

Algorithm 1: *TD*(0) for estimating $V \approx V_\pi$

input : Policy π, update step size α, and discount factor γ
output : Estimate of the state value function V_π

```
1  Initialise V(s) for all states s ∈ S arbitrarily with V(terminal) = 0
2  for i = 1, 2, ..., no.episodes do
3      Initialise a starting state s_0
4      s ← s_0
5      continue_episode ← True
6      while continue_episode do
7          a ← action chosen by π(s)
8          Take action a and observe r, s'
9          // Update the value function estimate
10         V(s) ← V(s) + α [(r + γV(s')) − V(s)]
11         if s' is terminal then
12             continue_episode ← False
13         else
14             s ← s'
15         end
16     end
17 end
```

For the agent to truly realise what is the best action to take at a given state s it must have discovered all the possible actions in that state. If the policy is simply greedy, once the agent discovers an action which leads to a large enough value it will always choose that action when returning to that state. Due to this, there might be a better action choice which will remain undiscovered. To ensure that the agent explores all the possible actions before converging to the optimal policy, the ϵ-greedy policy prompts the agent to choose a random action with a probability of ϵ, as shown in (4.25)), encouraging the agent to explore the possible actions.

$$\pi(a|s) = \begin{cases} \epsilon/|A(s)| + 1 - \epsilon & \text{if } a^* = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s,a) \\ \epsilon/|A(s)| & \text{otherwise} \end{cases} \tag{4.25}$$

Since at the start of its learning the agent is unaware of the possible actions, ϵ is chosen to have a value of $\epsilon = 1$ such that it always chooses a random action. The value of ϵ decays as learning continues and the agent realises more actions. Decaying ϵ at the appropriate rate results in a random action probability of $\epsilon \approx 0$ once the agent has sufficiently explored, resulting in the optimal state-value function Q_* and consequently the optimal policy π_*

The prediction of $Q(s, a)$ is achieved through the implementation of Algorithm 2. This is defined for every state-action pair unless the next state s' is terminal in which case $Q(s', \cdot)$ is taken to be zero.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[(r_t + \gamma Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t) \right] \tag{4.26}$$

The update equation, shown in (4.26), utilises the starting State, Action, and Reward (s_t, a_t, r_t) along with the next step State and Action (s_{t+1}, a_{t+1}) which is why it is dubbed SARSA. Algorithm 2 shows the structure of a general TD SARSA algorithm which obtains the optimal

state-action value function Q_*. This can be utilised to achieve the optimal policy π_* through constructing a greedy policy with respect to the optimal value function.

Algorithm 2: SARSA (on-policy TD Control) for Estimating $Q \approx Q_*$

input : Update step size α and small random action probability $\epsilon > 0$
output : Estimate of the optimal state-value function Q_*

1 Initialise $Q(s, a)\ \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ arbitrarily with $Q(s_{terminal}, \cdot) = 0$
2 **for** $i = 1, 2, ..., no.episodes$ **do**
3 Initialise a starting state s_0
4 $s \leftarrow s_0$
5 *continue_episode* $\leftarrow$ True
6 **while** *continue_episode* **do**
7 $a \leftarrow$ action chosen by ϵ-greedy policy $\pi(s)$
8 Take action a and observe r, s'
9 Choose action a' using policy derived from $Q(s, a)$
10 `// Update the value function estimate`
11 $Q(s, a) \leftarrow Q(s, a) + \alpha\left[(r + \gamma Q(s', a')) - Q(s, a)\right]$
12 **if** s' *is terminal* **then**
13 $Q(s', \cdot) = 0$
14 *continue_episode* $\leftarrow$ False
15 **else**
16 $s \leftarrow s'$
17 **end**
18 **end**
19 **end**

Thus far the explanation has only covered tabular methods where the state and action spaces are finite and the boundaries are clearly defined. For the quadcopter control problem that is not the case. The quadcopter has an infinite state space along with an infinite choice of actions. This case calls for the implementation of approximate solution methods.

4.1.4 Approximate Solution Methods (Continuous State and Action Space)

In scenarios in which reinforcement learning is applied to infinitely large state and action spaces is it essentially impossible to find the optimal value function and policy. The best approach then is to approximate an adequate value function and policy to solve the MDP. In infinite spaces the key concept is generalisation. Since it is impractical for the agent to visit an infinite number of states, the prediction and evaluation steps of the RL algorithm must be able to generalise around the visited areas in the state and action spaces. Fortunately the problem of generalisation has been studied extensively and can be applied through function approximation.

Continuous State Space: Linear Value Function Approximation

In the previous section, the value function was in the format of a look-up table in which the states and actions were discrete, meaning they were finite and had clear boundaries from one

another. This implies that an update can be implemented to a state or a state-action pair without having an effect on any other states or actions. However, with infinite spaces one is not able to have a clear split-up of the space. With the generalisation objective, performing an update on a state must also affect the states around it. This suggests that in an infinite space problem the value function can no longer longer be adequately represented by a table. Rather, the value function is now approximated to be some mathematical function parameterised by a weight vector $\boldsymbol{w}$ such that the Value Function Approximation (VFA) step results in some approximate value function $\widehat{V}(s, \boldsymbol{w}) \approx V_\pi$. Generally, the number of weights d is much smaller than the number of states $d << |\mathcal{S}|$. Hence, changing the value of one weight changes the value estimate of many states, satisfying the generalisation requirement. In this work $\widehat{V}(s, \boldsymbol{w})$ is taken to be a linear function of features of the state $\boldsymbol{\chi}(s)$ and the feature weight vector $\boldsymbol{w}$ as shown in (4.27). The value function approximation problem can be broken down into two parts: approximating the weights $\boldsymbol{w}$, and constructing the features vector $\boldsymbol{\chi}(s)$.

$$\begin{aligned}\widehat{V}(s, \boldsymbol{w}) &= w_1\chi_1(s) + w_2\chi_2(s) + w_3\chi_3(s) + ... + w_d\chi_d(s) \\ &= \sum_{i=1}^{d} w_i\chi_i(s) \\ &= \boldsymbol{w}^T\boldsymbol{\chi}(s) \end{aligned} \tag{4.27}$$

Each state in the state space can be encoded into a features vector $\boldsymbol{\chi}(s)$ in-charge of relaying it to the RL agent. This features vector is achieved through a process which maps a state to a real numbered vector of the same length as the weight vector $\boldsymbol{\chi}(s) = [\chi_1(s), \chi_2(s), \chi_3(s), ..., \chi_d(s)]$. The features vector can be directly computed using a variety of methods. The method chosen for implementation in this research is the Fourier cosine basis features vector, discussed in later in this thesis, due to its ease of implementation and proven successful integration with TD learning methods. Information on other methods of constructing the features vector can be found in Section 9.5 of [55].

Unlike the features vector, the weight vector cannot be directly computed since the value function is unknown. Hence, just as the tabular value function was updated using a TD target, the weight vector $\boldsymbol{w}$ is continuously updated using the TD target.

In a sense an update $s \mapsto u$ where s is the state being updated, and u is the TD target which the estimate of the value $V(s)$ is being moved towards, can be viewed as the desired input-output behaviour of the value function. That is, the estimate value of the state s, $V(s)$, should be more like the the update target u. Machine learning processes that to learn to mimic said input-output relationships, where the output u is a number, are called *function approximation processes*. Function approximation is a supervised learning method, meaning that it expects to receive a training example of the ideal input-output behaviour of the function it is trying to approximate. In terms of value function approximation, the sample is simply $s \mapsto u$. The approximate value function produced by this training sample is what is taken to be the estimated value function.

Looking at each update as a training sample enables many function approximation methods to be used to achieve an approximation of value function. Not all methods for function approximation are suitable for reinforcement learning applications. In RL it is essential that the learning takes place online while the agent is operating within its environment and incrementally acquiring the data.

Stochastic Gradient Methods One highly reputable set of methods which satisfy the reinforcement learning requirements for VFA are the Stochastic Gradient Descent (SGD) methods. In SGD methods, the weight vector is updated at a series of discrete time steps chosen by the user. Even when assuming that the actual correct value for every encountered state $V_\pi(s_t)$ is given, the correct weights that represent said function are still challenging to obtain. This is because there are not enough weights to allow for the values at every state to be correct. A well respected approach in this scenario is to apply a SGD update to $\boldsymbol{w}$ which shifts the weight vector in the direction of the gradient shown in (4.28) to step towards reducing the error between the approximated value function $\widehat{V}(s_t, \boldsymbol{w}_t)$ and the actual value function $V_\pi(s_t)$.

$$\boldsymbol{w}_{t+1} \doteq \boldsymbol{w}_t - \frac{1}{2}\alpha\nabla\left[V_\pi(s_t) - \widehat{V}(s_t, \boldsymbol{w}_t)\right]^2 \tag{4.28}$$

$$= \boldsymbol{w}_t + \alpha\left[V_\pi(s_t) - \widehat{V}(s_t, \boldsymbol{w}_t)\right]\nabla\widehat{V}(s_t, \boldsymbol{w}_t) \tag{4.29}$$

Here, α is some small step size and $\nabla\widehat{V}(s_t, \boldsymbol{w}_t)$ is the gradient of the approximated value function with respect to the weight vector.

$$\widehat{V}(s, \boldsymbol{w}) = \left[\frac{\partial\widehat{V}(s, \boldsymbol{w})}{\partial w_1}, \frac{\partial\widehat{V}(s, \boldsymbol{w})}{\partial w_2}, ..., \frac{\partial\widehat{V}(s, \boldsymbol{w})}{\partial w_d}\right]$$

For linear value function approximation the features vector is taken to be the gradient $\nabla\widehat{V}(s, \boldsymbol{w}) \leftarrow \boldsymbol{\chi}(s)$. Applying this to (4.29) results in (4.30).

$$\boldsymbol{w}_t = \boldsymbol{w}_t + \alpha\left[V_\pi(s_t) - \widehat{V}(s_t, \boldsymbol{w}_t)\right]\boldsymbol{\chi}(s_t) \tag{4.30}$$

Both (4.29) and (4.30) assume that the actual value function is known. However, that is not the case. This problem can be solved by utilising the TD target in place of the true value function.

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\left[(r_t + \gamma\widehat{V}(s_{t+1}, \boldsymbol{w}_t)) - \widehat{V}(s_t, \boldsymbol{w}_t)\right]\boldsymbol{\chi}(s_t) \tag{4.31}$$

An important consideration is the SGD learning step-size α. Figure 4.6 shows a function approximation using Fourier basis (explained in Section 4.1.4) of order 15 with two different learning step sizes. The left column shows the function approximation using $\alpha = 1 \times 10^{-3}$ while the right column uses $\alpha = 1 \times 10^{-4}$. It is apparent that the larger step size is able to provide a viable approximation in a timely manner, where as the smaller α was not able to do so. This is because the larger learning rate allows for bigger steps to be taken towards the target where-as the conservative learning rate is not taking big enough steps to achieve the results in the time given. The smaller time step will eventually reach the desired function as the $no.steps \to \infty$. However, that isn't viable for time sensitive applications. Conversely, choosing an α that is too large will cause the update to overshoot or undershoot the target function, resulting in an inaccurate estimate of the function. This is evident in Figure 4.7 which aimed to estimate the same function as the one in Figure 4.6 using $\alpha = 1.5 \times 10^{-2}$.

Having discussed the method for obtaining the weight vector $\boldsymbol{w}$, the next step is to develop the features vector $\boldsymbol{\chi}(\boldsymbol{s})$, completing the components required to achieve a value function estimate.

Features Vector Construction for Linear $\widehat{V}$: Fourier Basis A limitation of assuming a linear value function is that it cannot relate the interaction between states to one another. This

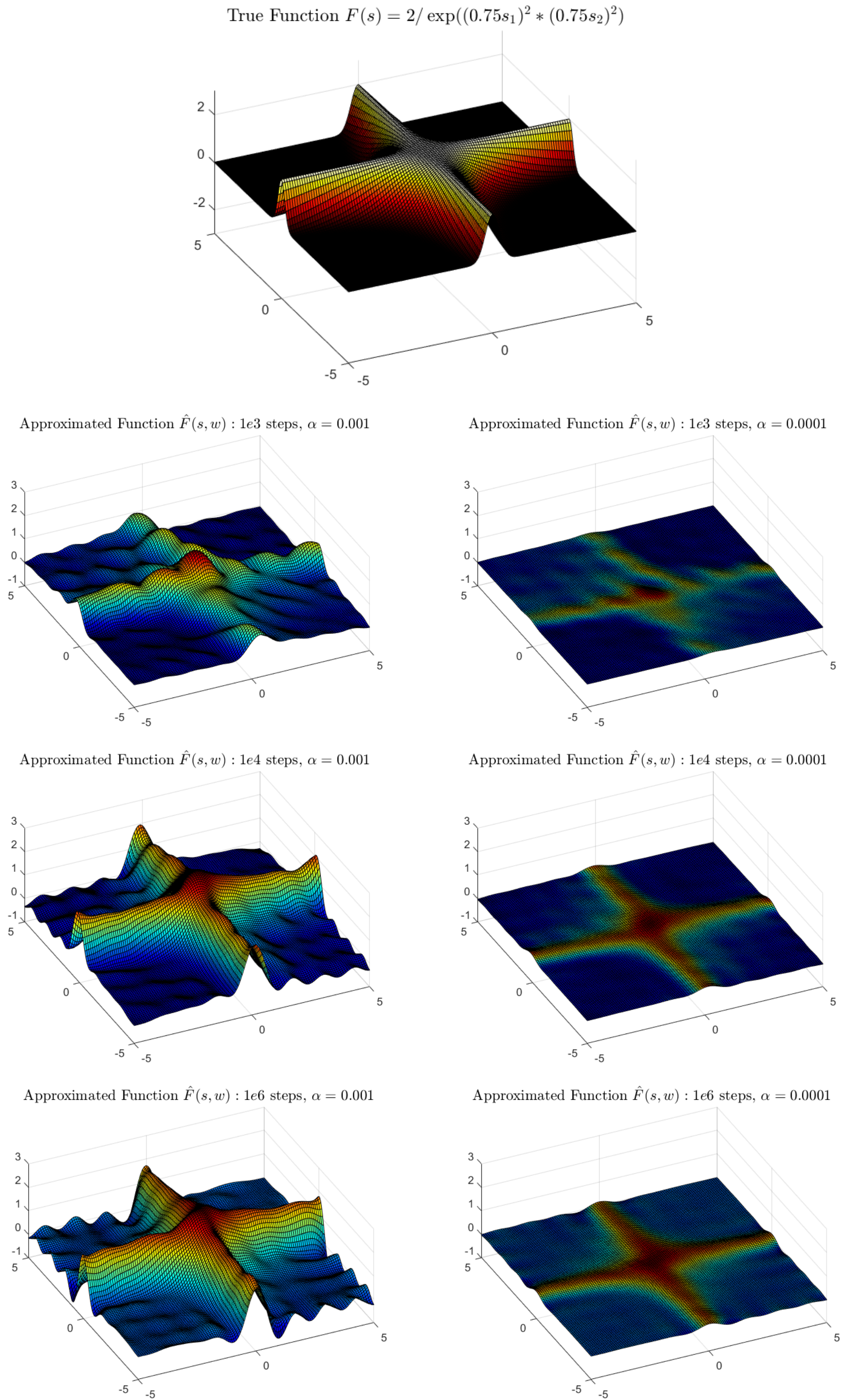


Figure 4.6: Comparison of the effect of the training duration and step-size choice α on a Fourier basis function approximation ($n = 15$).

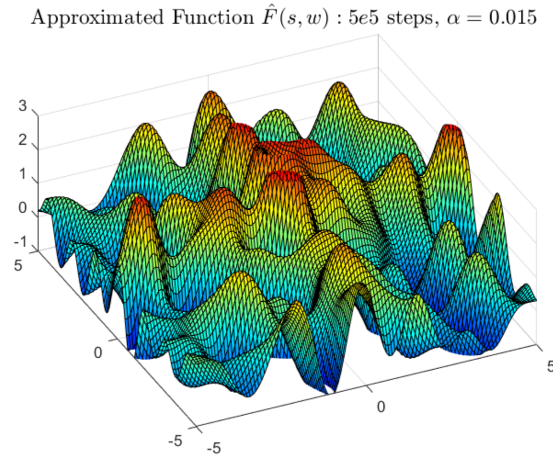


Figure 4.7: Fourier series approximation of the function shown in Figure 4.6 with an excessively step size $\alpha = 1.5 \times 10^{-2}$ showing over and under shoot of the actual function.

can be crucial especially in robot control applications. For example, a robot travelling at a high velocity to track a waypoint can be either advantageous or disadvantageous. If the robot is far away from the goal and moving fast towards the goal that is an advantageous state to be in. However, if it is moving fast away from the goal or is close to the goal and moving towards it at a high speed then those are disadvantageous states. To enable the value function to make these kinds of differentiations, the features vector $\boldsymbol{\chi}(\boldsymbol{s})$ needs to be built with a combination of all the states within the state vector.

One way to construct the features vector is through the use of *Fourier Basis* based on the implementation of the Fourier series. Fourier series are a widely utilised method for expressing a function in terms of a weighted sum of sine and cosine functions of different frequencies. Consider the following step function.

$$f(x) = \begin{cases} 1 & \text{if } x < 0.5 \\ 0 & \text{if } x = 0.5 \\ -1 & \text{if } x > 0.5 \end{cases} \tag{4.32}$$

This can be represented using the following Fourier series.

$$f(x) \approx \frac{4}{\pi}\cos\pi x - \frac{4}{3\pi}\cos 3\pi x + \frac{4}{5\pi}\cos 5\pi x - \frac{4}{9\pi}\cos 9\pi x + ... - \frac{4}{n\pi}\cos n\pi x \tag{4.33}$$

where n is the maximum odd number.

From Figure 4.8 it is evident that with the appropriate value for n, the step function described by (4.32) can be represented using Fourier series. This implies that with enough basis, many types of functions can be approximated. In applications where the function being estimated is known, Fourier series are a desirable option as the weights can easily be obtained using simple formulas. Although in reinforcement learning the value function is unknown, Fourier series have proven to be a viable option for their ability to prosper in a multitude of RL applications and their ease of implementation.

Fourier basis can be used to approximate the linear value function of the RL problem through constructing a Fourier cosine series feature vector $\boldsymbol{\chi}(\boldsymbol{s})$. It is mathematically more convenient

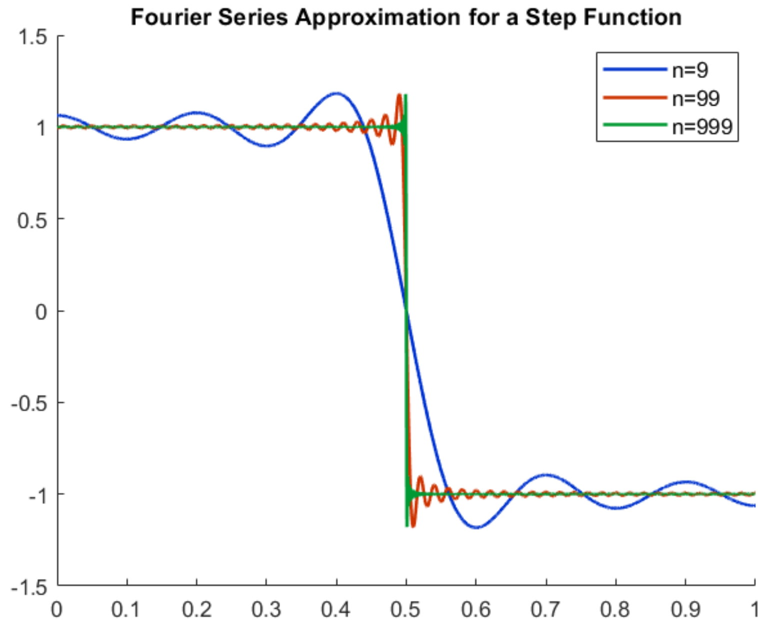


Figure 4.8: Fourier series approximation of a step function with varying order n.

to use a cosine basis as opposed to sine basis due to the discontinuous nature of odd functions at the origin. For more details on this refer to Section 9.5.2 of [55].

As mentioned previously, since the value function is approximated to be linear, the features vector is designed with the entire state vector of k-dimensions $\boldsymbol{s} = (s_1, s_2, ..., s_k)$ embedded within the Fourier cosine basis. This allows for the agent to realise the relationship between the elements within the state vector. The processes begins with the choice of a Fourier order n. A vector $\boldsymbol{c}$ of all possible k length combinations of $(1, 2, 3, ..., n)$ is created. This vector will have a length of $(n + 1)^k$ which dictates the number of features and coincidentally the number of weights in $\boldsymbol{w}$.

$$\boldsymbol{c}^i = \left(c_1^i, c_2^i, ...c_k^i\right)^T \tag{4.34}$$

where $c_j^i \in \{0, 1, ..., n\}$ for $j = 1, 2, ..., k$ and $i = 0, 1, ..., (n + 1)^k$.

To create the features vector, any given state vector is scaled such that each element within $\boldsymbol{s}$ is bound between zero and one $s_{1,2,...,k} \in [0, 1]$. The cosine basis features are then computed using this scaled state vector $\boldsymbol{s}$ along with the vector $\boldsymbol{c}$.

$$\chi_i(s) = \cos\left(\pi \boldsymbol{s}^T \boldsymbol{c}^i\right) \tag{4.35}$$

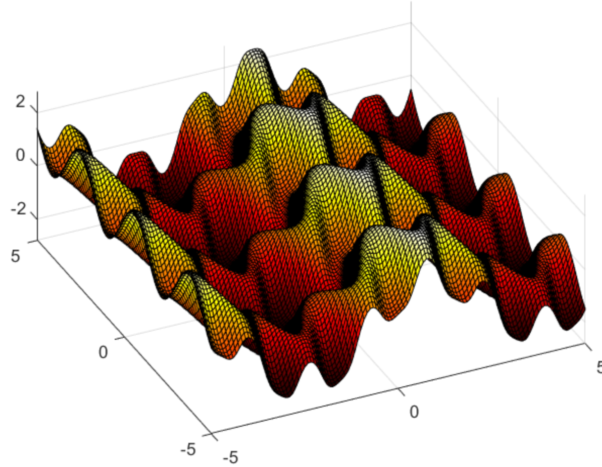
In the inner product, the $\boldsymbol{c}$ vector assigns some integers $\{0, ..., n\}$ to each element within $\boldsymbol{s}$ which will determine the frequency of that feature along that dimension.

A Fourier cosine basis features vector has proved to provide satisfactory value function approximation results when paired with both the SARSA control and $TD(0)$ prediction algorithms making it ideal for implementation in robot control problems. One disadvantage of Fourier series is the difficulty it has in estimating discontinuities. This is because it is difficult to avoid oscillations about the discontinuity point unless really high frequency basis functions are selected. This is demonstrated in Figure 4.8 where as the value of n increases the ringing about the discontinuity is not as evident.

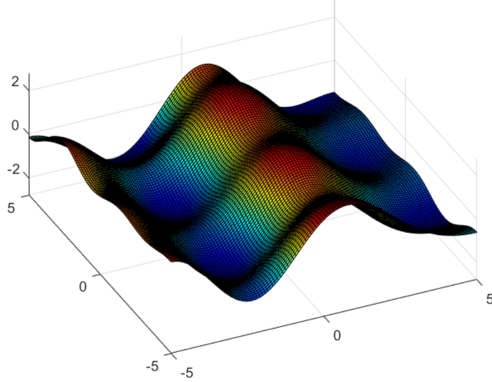
Figure 4.9 shows the semi-gradient weight update in (4.30) with a Fourier basis features vector used to approximate a function $F(\boldsymbol{s} = (s_1, s_2)) = \sin(s_1) + \cos(s_1) - \sin(2s_2 - \cos(4s_1))$ with increasing Fourier orders n. Increasing the Fourier order moves the estimate of the function closer to the true value as evident by the decrease in error. That being said, increasing the order also significantly affects the computation time as it exponentially increases the weight vector which needs to be updated at every step. However, in applications like robot control, the weight update step must be computed in a timely manner. The weight update step takes place as the robot moves between one state to another. This means that no control inputs or state measurements can take place while the weight vector is being updated. The longer this update takes, the longer the agent is unaware of its state and is unable to command a control input which can be crucial for fast-moving systems.

The changes in the approximation of a function are subtle after a certain order. From Figure 4.9 it can be seen that the difference in the function approximation using an order $n = 10$ and $n = 15$ is significant. However, the changes in the function approximation between using an order $n = 15$ and $n = 25$, although visible, are not as significant. That being said, the increase of the computational requirements between a function approximation of order $n = 15$ and $n = 25$ is notable. Function approximation using a Fourier order of 15 must update 256 weights at every step, while one with a Fourier order of 25 has to update 676. This brings forth the consideration of the trade-off between performance and accuracy when utilising Fourier basis in RL applications.

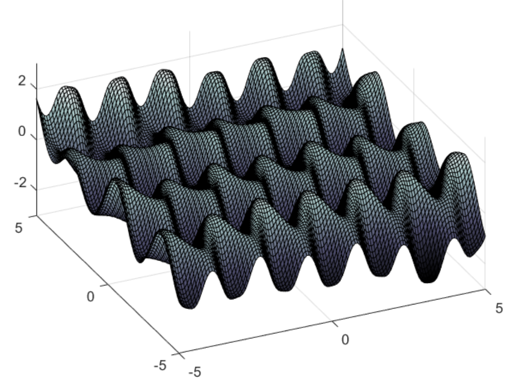
The Fourier basis features vector can be used in tandem with a SGD update of $\boldsymbol{w}$ to achieve a full linear VFA algorithm to estimate the value V_π of a given policy. An example of a full SGD $TD(0)$ Fourier basis VFA algorithm is outlined in Algorithm 3.



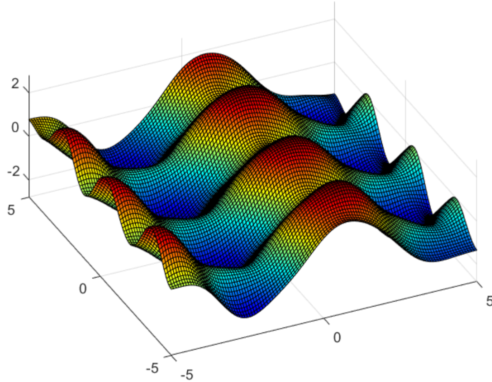
(a)



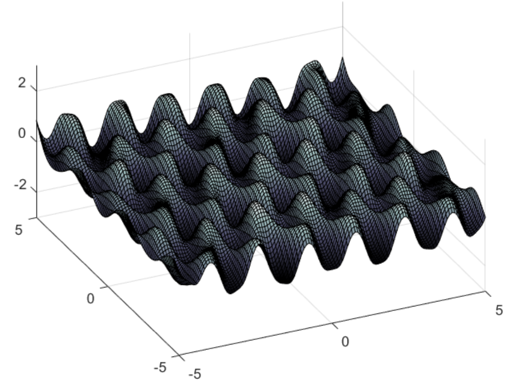
(b)



(c)



(d)



(e)

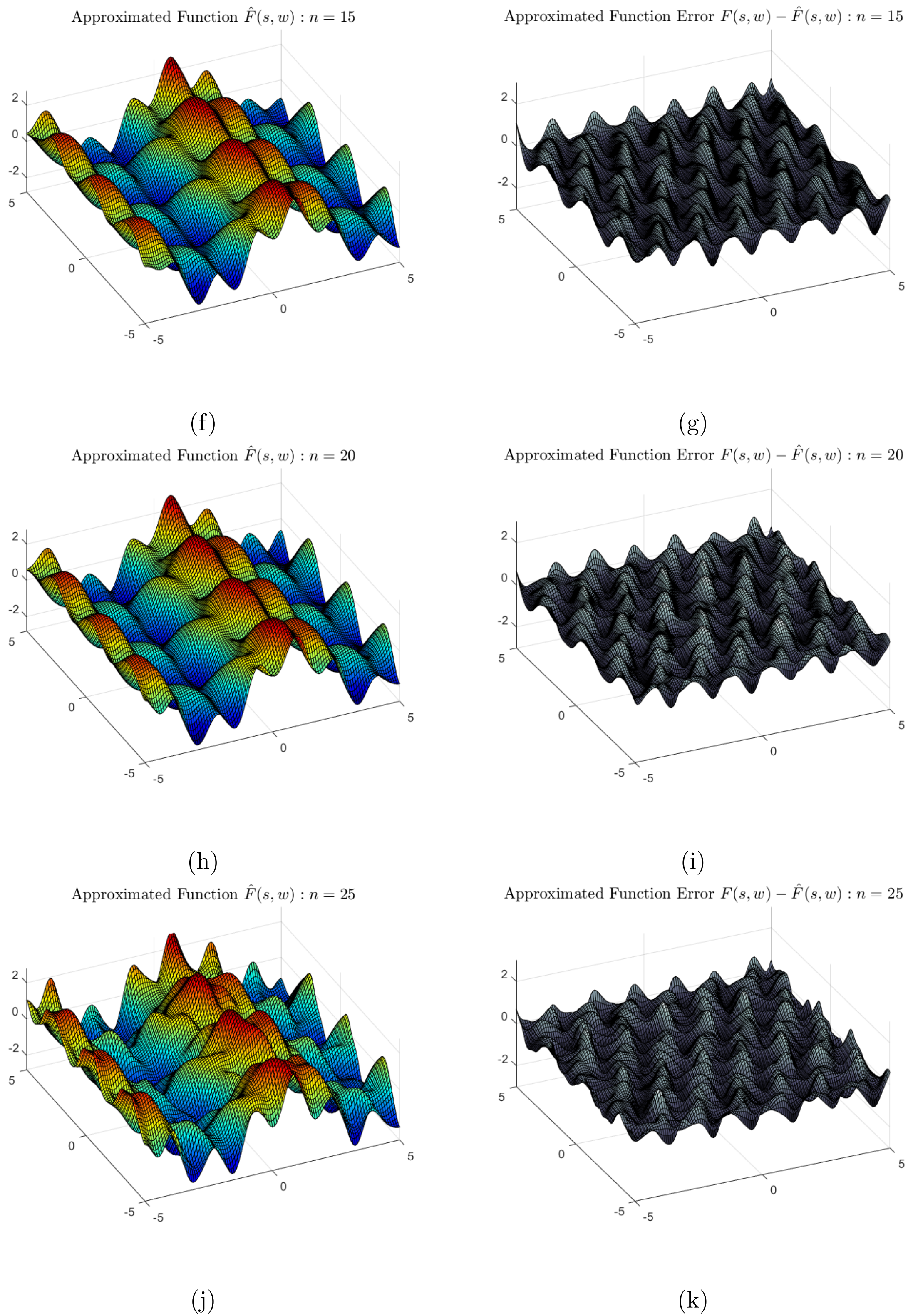


Figure 4.9: Comparison of the affect of the Fourier order n on the function approximation after 6×10^4 updates.

Algorithm 3: Semi-Gradient $TD(0)$ Value Function Approximation Using a Fourier Basis Features Vector for estimating $\widehat{V} \approx V_\pi$

input : Policy π, update step size α, Fourier order n, and discount factor γ
output : Estimate of the state value function V_π

1 $k \leftarrow$ number of dimensions in the state space $d \leftarrow (n+1)^k$
2 Create the $\boldsymbol{c}$ vector
3 Initialise the value function weight vector $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily
4 Initialise the features vector $\boldsymbol{\chi} \in \mathbb{R}^d$ arbitrarily
5 **for** $i = 1, 2, ..., no.episodes$ **do**
6   `// Initialise a starting state` s_0
7     $\boldsymbol{s} \leftarrow \boldsymbol{s}_0 = (s_{1(0)}, s_{2(0)}, ..., s_{k(0)})$
8   Scale the state vector such that $s_{1,2,..,k} = [0, 1]$
9   `// Construct the Fourier basis vector`
10   **for** $m = 1, 2, ..., d$ **do**
11         $\chi_m(\boldsymbol{s}) = \cos(\pi \boldsymbol{s}^T \boldsymbol{c}^m)$
12   **end**
13   **for** $j = 1, 2, ..., no.steps$ **do**
14   `// Compute the linear VFA for the state` s
15         $\widehat{V}(\boldsymbol{s}, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{\chi}(\boldsymbol{s})$
16         $a \leftarrow$ action chosen by $\pi(s)$
17   Take action a and observe $r, \boldsymbol{s'}$
18   Scale the state vector such that $s'_{1,2,..,k} = [0, 1]$
19   `// Construct the Fourier basis vector for the state` s'
20   **for** $m = 1, 2, ..., d$ **do**
21             $\chi_m(\boldsymbol{s'}) = \cos(\pi \boldsymbol{s}^T \boldsymbol{c}^m)$
22   **end**
23   `// Compute the linear VFA for state` $\boldsymbol{s}'$
24         $\widehat{V}(\boldsymbol{s'}, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{\chi}(\boldsymbol{s'})$
25   `// Update the value function weight vector`
26         $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \left[(r + \gamma \widehat{V}(\boldsymbol{s'}, \boldsymbol{w})) - \widehat{V}(\boldsymbol{s}, \boldsymbol{w}) \right] \chi(\boldsymbol{s})$
27   **end**
28 **end**

Continuous Action Space: Policy Gradient Methods

Thus far the discussed control efforts have been achieved through the estimation and maximisation of the state-action value function. This however is not a viable approach when the action space is extremely large or infinite. A relevant example is the action space of a UAV. A UAV changes its position by taking the action of altering its roll, pitch, or yaw angles. Although most UAVs are bound to a certain pitch or roll angle limit for safety, they are still able to choose any of the infinite possible angles within that limit. In these scenarios, methods employing a *parameterised policy* are utilised. Parameterised policies do not require the value function, rather the learned parameter $\boldsymbol{\theta}$ is used to choose the appropriate action without the knowledge of the value function at the time of action selection.

Policy Gradient Theorem: Actor-Critic $\boldsymbol{\theta}$ Update The policy for infinite action spaces is taken to be a function which produces an action given a state. The policy must be able to choose an action without consulting a value function which maximises the value along with being able to generalise the chosen action to surrounding states. Both of these requirements are met through the implementation of a learned policy parameter vector $\boldsymbol{\theta}$.

The way that the policy is able to choose actions that maximise the value is through the integration of the value function in the policy's parameter learning process. The policy is the probability of action a being taken at time t given that the agent is at state s with parameter vector $\boldsymbol{\theta}$

$$\pi(a|s,\boldsymbol{\theta}) = \mathbb{P}\left[a_t = a|s_t = a, \boldsymbol{\theta}_t = \boldsymbol{\theta}\right]$$

The policy parameter $\boldsymbol{\theta}$ is learned based on some estimate of the gradient of a scalar performance measure J with respect to itself. This follows the same procedure as the SGD approach but instead of stepping towards a minimum it is stepping towards the maximum. Hence (4.36) can be seen as a *Stochastic Gradient Ascent* (SGA) update equation.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{4.36}$$

All methods that follow this type of methodology for their policy estimate are called *policy gradient methods*. Furthermore, methods that learn both their value function and policy are called *actor-critic* methods. In actor-critic algorithms the 'actor' is the policy gradient which chooses the action and the 'critic' is the value function approximator which criticises the actor's choice.

In the episodic case, the performance measure $J(\boldsymbol{\theta})$ can be taken to be the true value function for some parameterised policy $\pi_{\boldsymbol{\theta}}$.

$$J(\boldsymbol{\theta}) \doteq V_{\pi_{\boldsymbol{\theta}}}(s)$$

When the value function is approximated, updating the policy parameter in a manner that guarantees policy improvement appears to be a challenge. The system's performance is fully dictated by $\boldsymbol{\theta}$ which is in-charge of action selections and generalisation to the states in which these actions are chosen. When the agent is at a state s, it is simple to determine the effect on actions and therefore the rewards given knowledge of the parameterisation of the policy. That being said, the effect of the parameterised policy on the state distribution is not as intuitive or straight forward. The difficulty lies in the fact that the estimation of the gradient of the performance measure with respect to the policy parameter depends on this unknown effect of the

policy on the state distribution. The *policy gradient theorem* provides an analytical expression for $\nabla J(\boldsymbol{\theta})$ in terms of $\boldsymbol{\theta}$ that does not rely on the derivative of the state distribution.

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &\propto \sum_{s\in\mathcal{S}} \mu(s) \sum_{a\in\mathcal{A}} Q_\pi(s,a)\nabla\pi(a|s,\boldsymbol{\theta}) \\ &= \mathbb{E}\left[\sum_{a\in\mathcal{A}} Q_\pi(s_t,a)\nabla\pi(a|s_t,\boldsymbol{\theta})\right] \\ &= \mathbb{E}\left[\sum_{a\in\mathcal{A}} \pi(a|s_t,\boldsymbol{\theta})Q_\pi(s_t,a)\frac{\nabla\pi(a|s_t,\boldsymbol{\theta})}{\pi(a|s_t,\boldsymbol{\theta})}\right] \\ &= \mathbb{E}\left[Q_\pi(s_t,a_t)\frac{\nabla\pi(a_t|s_t,\boldsymbol{\theta})}{\pi(a_t|s_t,\boldsymbol{\theta})}\right] \\ &= \mathbb{E}\left[G_t\frac{\nabla\pi(a_t|s_t,\boldsymbol{\theta})}{\pi(a_t|s_t,\boldsymbol{\theta})}\right] \end{aligned} \tag{4.37}$$

where μ is the on policy distribution under π. The full proof of this theorem can be found in Section 13.2 of [55]. The expression illustrated in (4.37) can be incorporated in the parameter update in (4.36) to achieve the following update equation:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla\pi(a_t|s_t,\boldsymbol{\theta})}{\pi(a_t|s_t,\boldsymbol{\theta})} \tag{4.38}$$

Each update increment is directly proportional to the return G_t along with the vector $\frac{\nabla\pi(a_t|s_t,\boldsymbol{\theta})}{\pi(a_t|s_t,\boldsymbol{\theta})}$. This vector is in the direction in the parameter space which increases the probability of taking action a_t in the future if state s_t is visited again. The update in (4.38) increases the parameter vector by α in the direction of the gradient proportional to the return so that it pushes $\boldsymbol{\theta}$ in a direction which increases the return G_t and inversely proportional to the action probability $\pi(a_t|s_t,\boldsymbol{\theta})$ to ensure that updates are conducted more evenly for all actions.

The update in (4.38) can provide slow policy convergence properties. Hence to enhance the performance a bias $b(s)$ is incorporated into the policy gradient theorem resulting in the following update equation.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\left(G_t - b(s_t)\right) \frac{\nabla\pi(a_t|s_t,\boldsymbol{\theta})}{\pi(a_t|s_t,\boldsymbol{\theta})} \tag{4.39}$$

This baseline can be any value that does not rely on the action chosen. Thus one logical choice is the state-value function approximation $\widehat{V}(s_t,\boldsymbol{w})$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\left(G_t - \widehat{V}(s_t,\boldsymbol{w})\right) \frac{\nabla\pi(a_t|s_t,\boldsymbol{\theta})}{\pi(a_t|s_t,\boldsymbol{\theta})} \tag{4.40}$$

Although the update in (4.40) uses an estimate of the value function, it is still not considered an actor-critic method. This is because the value function is only used as a baseline and not actually used as a critic. Criticism of the quality of the value function estimate is only introduced once when the value function baseline is used in a bootstrapping algorithm which significantly accelerates the learning process. One algorithm that utilises bootstrapping is the actor-critic *TD*(0) algorithm. The *TD*(0) update on $\boldsymbol{\theta}$ employs the TD target in place of the actual return.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\left[\left(r_T + \gamma\widehat{V}(s_{t+1},\boldsymbol{w})\right) - \widehat{V}(s_t,\boldsymbol{w})\right] \frac{\nabla\pi(a_t|s_t,\boldsymbol{\theta})}{\pi(a_t|s_t,\boldsymbol{\theta})} \tag{4.41}$$

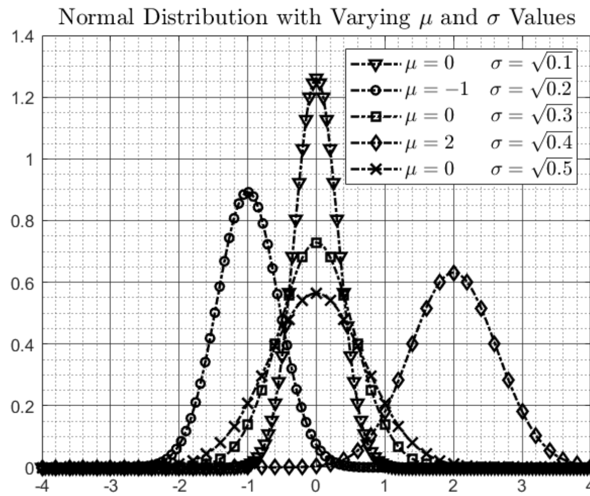


Figure 4.10: Figure of normal distributions with varying mean μ and standard deviation σ values.

This update provides a straight forward and fast converging method of learning the parameter of the policy. However, the policy is no longer for some small finite action space. Rather, the problem of infinite action space arises. A way of dealing with this kind of problem is to resort to representing the policy as a probability distribution along the action space rather than computing the probability for each separate action.

Policy Parameterisation for Continuous Actions: Gaussian Probability Distribution
As previously mentioned, in infinitely large action spaces it is no longer practical to compute the probability for each of the actions at each state. The better approach would be to learn the statistics as some probability distribution. One of the most commonly used probabilities is the Gaussian (normal) distribution.

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{4.42}$$

Here, μ and σ are the mean and standard deviation of the distribution. Figure 4.10 illustrates the effect of μ and σ on a normal distribution. The parameterised policy is defined as the normal probability density over some action, with $\mu(s, \boldsymbol{\theta})$ and $\sigma(s, \boldsymbol{\theta})$ being function approximators that depend on the state.

$$\pi(a|s, \boldsymbol{\theta}) = \frac{1}{\sigma(s, \boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(a-\mu(s, \boldsymbol{\theta}))^2}{2\sigma(s, \boldsymbol{\theta})^2}\right) \tag{4.43}$$

The parameter $\boldsymbol{\theta}$ is broken down into two parts: $\boldsymbol{\theta}_\mu$ and $\boldsymbol{\theta}_\sigma$. The subscripts of the parameter vectors indicate whether it is in-charge of estimating the mean μ or standard deviation σ.

$$\mu(s, \boldsymbol{\theta}) \doteq \boldsymbol{\theta}_\mu^T \boldsymbol{\chi}(s) \tag{4.44}$$

$$\sigma(s, \boldsymbol{\theta}) \doteq \exp\left(\boldsymbol{\theta}_\sigma^T \boldsymbol{\chi}(s)\right) \tag{4.45}$$

To improve the policy parameter vectors, $\boldsymbol{\theta}_\mu$ and $\boldsymbol{\theta}_\sigma$ are updated according to the chosen algorithm using (4.41).

The fraction $\frac{\nabla \pi(a_t|s_t,\boldsymbol{\theta})}{\pi(a_t|s_t,\boldsymbol{\theta})}$ corresponding to the updates of the μ and σ parameter vectors are given by the following equations.

$$\frac{\nabla \pi(a_t|s_t,\boldsymbol{\theta}_\mu)}{\pi(a_t|s_t,\boldsymbol{\theta})} = \frac{a-\mu(s,\boldsymbol{\theta})}{\sigma(s,\boldsymbol{\theta})^2}\boldsymbol{\chi}(s) \tag{4.46}$$

$$\frac{\nabla \pi(a_t|s_t,\boldsymbol{\theta}_\sigma)}{\pi(a_t|s_t,\boldsymbol{\theta})} = \left(\frac{a-\mu(s,\boldsymbol{\theta})}{\sigma(s,\boldsymbol{\theta})^2} - 1\right)\boldsymbol{\chi}(s) \tag{4.47}$$

Having determined the fractions given by (4.46) and (4.47) the TD SGA updates for the policy parameters are shown in (4.48) and (4.49).

$$\boldsymbol{\theta}_{\mu,t+1} = \boldsymbol{\theta}_{\mu,t} + \alpha_\mu \left[\left(r_T + \gamma \widehat{V}(s_{t+1},\boldsymbol{w})\right) - \widehat{V}(s_t,\boldsymbol{w})\right] \frac{a-\mu(s,\boldsymbol{\theta})}{\sigma(s,\boldsymbol{\theta})^2}\boldsymbol{\chi}(s) \tag{4.48}$$

$$\boldsymbol{\theta}_{\sigma,t+1} = \boldsymbol{\theta}_{\sigma,t} + \alpha_\sigma \left[\left(r_T + \gamma \widehat{V}(s_{t+1},\boldsymbol{w})\right) - \widehat{V}(s_t,\boldsymbol{w})\right] \left(\frac{a-\mu(s,\boldsymbol{\theta})}{\sigma(s,\boldsymbol{\theta})^2} - 1\right)\boldsymbol{\chi}(s) \tag{4.49}$$

These semi gradient ascent updates can be utilised in any RL TD method as a way to approximate a policy for some continuous action space.

4.2 Waypoint Tracking Algorithm

Recalling the controller's objectives:

1. Must push the x position error to zero: $\lim_{t\to\infty} e_x = x - x_{goal} \to 0$
2. Accomplish the objective stated in 1. without knowledge of the model of the system
3. Be fully on-board such that it requires no external intervention

The first step in the process of creating an RL algorithm is determining what states are necessary for the controller. The states should be chosen such that the agent is able to complete the task without the addition of any unnecessary states that may slow down the computation time of the algorithm. When controlling a system with fast dynamics like a quadcopter the consideration of frequent state sampling coupled with fast overall computation is of great importance, especially since it will be running fully on board.

Recalling the objective described in the first point, it is natural to incorporate the position error e_x into the state space. In addition, for the RL agent to realise the relationship between the velocity and the position error of the quadrotor, as explained in 4.1.4, the addition of the velocity error $\dot{e}_x = \dot{x} - \dot{x}_{goal}$ into the state space becomes useful. Since the goal is a waypoint tracking mission, the quadcopter is desired to come to a stop at the waypoint meaning that $\dot{x}_{goal} = 0 \to \dot{e}_x = \dot{x} - \dot{x}_{goal} = \dot{x}$. The state space of the UAV took the following form:

$$\mathcal{S} = \begin{bmatrix} e_{x,min} & e_{x,max} \\ \dot{x}_{min} & \dot{x}_{max} \end{bmatrix} \tag{4.50}$$

Hence, a sample of a state $\boldsymbol{s}$ would be in the form $\boldsymbol{s} = (e_x, \dot{x})$. This by definition is a infinite 2-dimensional state space with $k = 2$. This small k value enables all updates in the algorithm to be calculated with minimal delay and reduced computational complexity such that it can operate on the UAV without issues. The minimum and maximum values of e_x and $\dot{x}$ can be altered to suit the problem at hand.

Since the controller is model-free the choice of reinforcement learning method is restricted to either Monte Carlo methods or Temporal Difference learning. The chosen method must be able to rapidly update the value function and policy estimates to accommodate for a rapidly changing environment. The algorithm was first trained in simulation before being transferred to the actual aircraft. Hence, the controller must be able to quickly adapt to the discrepancy between the simulated and actual environments. When taking the acceleration in learning that bootstrapping provides, TD proves to be the clear choice for this application. The chosen TD method was the $TD(0)$ method which conducts its updates after each step.

Since the quadcopter will be operating in infinite state and control action spaces which unknown dynamics, both the value function and the policy must be approximated. This means that the algorithm is defined as an actor-critic $TD(0)$ algorithm.

4.2.1 Actor - Critic $TD(0)$ Algorithm for Waypoint Tracking

The value function was estimated as a linear value function as shown in (4.27) using the Fourier cosine basis presented by (4.35). The order of the Fourier basis was chosen through experimentation later in the thesis. This particular method is illustrated in Algorithm 3. The chosen starting reward function was the most intuitive form of the reward function for position

error reduction since the quadcopter will be rewarded more negatively when its further away from the goal. This form of the reward function can be seen implemented in the works of [28], [29], and [30].

$$r(s_t) = \beta - c\,(x_t - x_{goal})^2 \tag{4.51}$$

The affect of the constant c on the performance of the system is discussed in Section 6.1.3 of this thesis.

The policy was estimated as a Gaussian distribution with a constant small standard deviation σ and a SGA $TD(0)$ update on the mean μ shown in (4.52). The reasoning behind a constant standard deviation was that the testing bed for the quadcopter was a controlled space with minimal to no disturbances. Hence, to reduce unnecessary computation σ was chosen to be some small constant accommodating for possible small disturbances without the addition of unnecessary computations. This approach was proven to be viable by the work presented in [29].

$$\boldsymbol{\theta}_{\mu,t+1} = \boldsymbol{\theta}_{\mu,t} + \alpha \left[\left(r_t + \gamma \widehat{V}(s_{t+1}, \boldsymbol{w}) \right) - \widehat{V}(s_t, \boldsymbol{w}) \right] \frac{a - \mu(s, \boldsymbol{\theta})}{\sigma(s, \boldsymbol{\theta})^2} \boldsymbol{\chi}(s) \tag{4.52}$$

That being said, in environments where disturbances exist such that learning σ is deemed necessary, a SGA update on the standard deviation can be added to the algorithm.

The chosen control action for the quadcopter was the roll angle θ which directly controls its x position. This high level control input was chosen as opposed to a low level action, like motor PWMs, to allow the algorithm to be implemented on any aircraft that accepts a roll angle command. The logistics behind sending this command to an aircraft is explored in Section 5.3 of this thesis.

Algorithm 4 shows the complete policy gradient $TD(0)$ actor-critic with a Fourier cosine basis linear VFA method for waypoint tracking. This is the algorithm which was used for training in simulation and testing on the actual quadcopter.

Algorithm 4: Policy Gradient Actor-Critic *TD*(0) Method Using Linear VFA with Fourier Basis Features Vector

input : Update step sizes α_V, α_μ, Fourier order n, discount factor γ, and some small standard deviation σ

output : Estimate of the optimal state value function and policy V_*, π_*

1 Specify the minimum x and $\dot{x}$ values for the state space
2 $k \leftarrow 2$ (number of dimensions in the state space).
3 $d \leftarrow (n+1)^k$
4 Initialise the value function weight vector $\boldsymbol{w} \in \mathbb{R}^d$, the features vector $\boldsymbol{\chi} \in \mathbb{R}^d$, and the mean parameter vector $\boldsymbol{\theta}_\mu \in \mathbb{R}^d$ to zero
5 Create the $\boldsymbol{c}$ vector
6 **for** $i = 1, 2, ..., no.episodes$ **do**
7   Specify the goal for the episode: x_{goal}
8   Initialise a starting state s_0: $\boldsymbol{s} \leftarrow \boldsymbol{s}_0 = (e_{x,0}, \dot{x}_0)$
9   Scale the state vector $\boldsymbol{s}$ such that $e_x = [0,1]$, $\dot{x} = [0,1]$
10   // Construct the Fourier basis vector for state $\boldsymbol{s}$
11   **for** $m = 1, 2, ..., d$ **do**
12         $\chi_m(\boldsymbol{s}) = \cos(\pi \boldsymbol{s}^T \boldsymbol{c}^m)$
13   **end**
14   // Compute the linear VFA for state s
15     $\widehat{V}(\boldsymbol{s}, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{\chi}(\boldsymbol{s})$
16   **for** $j = 1, 2, ..., no.steps$ **do**
17   // Compute the mean for the policy
18         $\mu = \boldsymbol{\theta}_\mu^T \boldsymbol{\chi}(\boldsymbol{s})$
19         $\theta \leftarrow$ roll angle chosen by $\pi(\theta | s, \mu, \sigma)$
20   Roll by the angle θ and observe $\boldsymbol{s}' = (e_x', \dot{x}')$
21   // Compute the reward r
22         $r = \beta - c\,(x - x_{goal})^2$
23   Scale the state vector $\boldsymbol{s}'$ such that $e_x' = [0,1]$, $\dot{x}' = [0,1]$
24   // Construct the Fourier basis vector for the state s'
25   **for** $m = 1, 2, ..., d$ **do**
26             $\chi_m(\boldsymbol{s}') = \cos(\pi \boldsymbol{s}^T \boldsymbol{c}^m)$
27   **end**
28   // Compute the linear VFA for state $\boldsymbol{s}'$
29         $\widehat{V}(\boldsymbol{s}', \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{\chi}(\boldsymbol{s}')$
30   // SGD update on $\boldsymbol{w}$ and SGA update on $\boldsymbol{\theta}_\mu$
31         $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_V \left[\left(r + \gamma \widehat{V}(\boldsymbol{s}', \boldsymbol{w}) \right) - \widehat{V}(\boldsymbol{s}, \boldsymbol{w}) \right] \chi(\boldsymbol{s})$
32         $\boldsymbol{\theta}_\mu \leftarrow \boldsymbol{\theta}_\mu + \alpha_\mu \left[\left(r + \gamma \widehat{V}(\boldsymbol{s}', \boldsymbol{w}) \right) - \widehat{V}(\boldsymbol{s}, \boldsymbol{w}) \right] \frac{a - \mu(s, \boldsymbol{\theta})}{\sigma(\boldsymbol{s}, \boldsymbol{\theta})^2} \boldsymbol{\chi}(\boldsymbol{s})$
33   // Set the new $\widehat{V}(\boldsymbol{s}, \boldsymbol{w})$ and $\boldsymbol{\chi}(\boldsymbol{s})$
34         $\widehat{V}(\boldsymbol{s}, \boldsymbol{w}) \leftarrow \widehat{V}(\boldsymbol{s}', \boldsymbol{w})$
35         $\boldsymbol{\chi}(s) \leftarrow \boldsymbol{\chi}(\boldsymbol{s}')$
36   **end**
37 **end**

Chapter 5

Experimental Setup

This chapter discusses the setup of systems required to conduct the experiment, including the fabrication of the quadcopter. The assembly components and relevant design decisions are also mentioned.

Test bed setup and the specific location chosen for experimentation are outlined to provide an explanation of the testing environment. The integration of the OptiTrack motion capture system and its importance in capturing real-time, high-precision data is vital in ensuring accurate experimental results.A pivotal aspect of the setup involves the seamless exchange of data between the OptiTrack system, the controller, and the autopilot. This section discusses the intricacies of data transfer, describing the utilization of the UDP protocol through socket programming and MAVLink. The procedure for retrieving precise position data from the OptiTrack system, their conversion, and subsequent transmission to both the ArduPilot autopilot and the RL controller is explained. Furthermore, the logistics governing the transmission of attitude commands from the controller to the autopilot, executed through MAVLink commands, is clarified.

5.1 Experimental Platform

A quadcopter was constructed from the ground up to evaluate the feasibility of the reinforcement learning algorithm. This approach offers flexibility and ensures complete transparency in both software and hardware aspects of the project. From a hardware perspective, this customized quadcopter design provides the freedom to adapt to testing requirements.

The quadrotor uses a Raspberry Pi that operates a Linux operating system and facilitates seamless file transfers between a Linux machine and the quadcopter. In addition, this enables the use of Visual Studio Code for code editing and execution directly on the quadcopter's microcomputer, eliminating the complexities of file transfers and addressing operating system disparities.

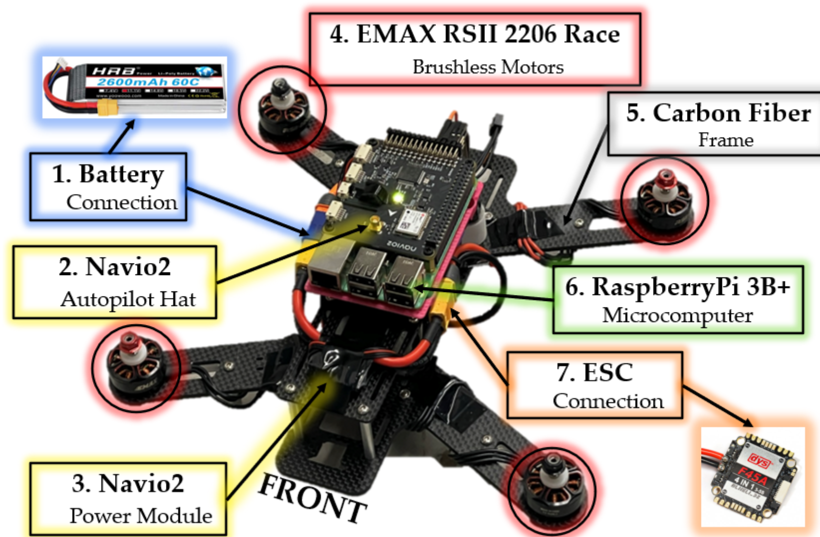


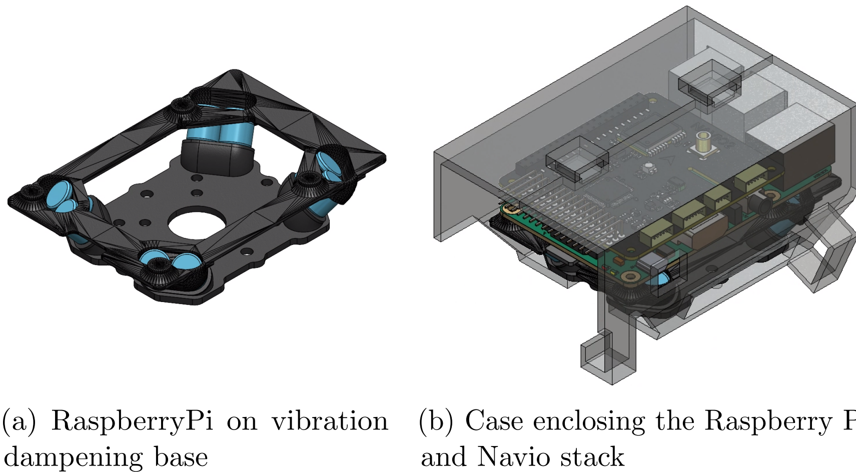
Figure 5.1: Picture of the quadcopter's components without the propellers or protective case.

The quadcopter's foundation is composed of a carbon fiber frame with dimensions $226l \times 130w \times 40h$ mm that is attached to four 75mm long legs. The selection of key components for the quadcopter's assembly relied on factors such as the frame's size, the estimated overall aircraft mass, and the desired performance outcomes. With these considerations, the *EMAX RSII 2206 Race Spec* brushless motors, coupled with 6040 bullnose bi-blade propellers was chosen. The brushless motors are engineered for high performance and align seamlessly with the project's specified criteria. The *DYS Aria F45A ESC* was chosen for the *Electronic Speed Controller* (ESC) and a 3S 2600mAh LiPo battery was selected as the power source. For an in-depth exploration of motor and ESC specifications and their performance characteristics in tandem with the propellers, refer to [54].

To implement the reinforcement learning controller on the quadcopter, a Raspberry Pi 3B+ was selected as the onboard companion computer and a Navio2 hat was integrated to implement the autopilot system. The Navio2 attaches as a Raspberry Pi shield incorporating critical components including a Global Navigation Satellite System (GNSS) receiver (U-blox M8N), dual IMUs (MPU9250 and LSM9DS1) for precise orientation and motion data, and a high-quality barometer (MS5611) for accurate altitude measurements. Moreover, it provides convenient access to the ADC, I^{2}C, and UART ports, facilitating the seamless integration of external sensors and communication modules. The Navio2 also offers PWM output channels, accommodating

control for up to 14 motors or servos.

The Raspberry Pi Operating System (OS) includes the ArduPilot software which can be coupled with a Ground Control Station (GCS) to provide the aircraft with real-time communication capabilities between the airborne platform and the ground-based team [58]. The Mission Planner software is a highly recommended option for a GCS due to its ArduPilot compatibility and user-friendly interface. The full autopilot stack was mounted on a vibration dampening base shown in Figure 5.2a and a 3D printed case was used to enclose the Pi-Navio stack for protection, as shown in Figure 5.2b.



(a) RaspberryPi on vibration dampening base

(b) Case enclosing the Raspberry Pi and Navio stack

Figure 5.2: 3D rendition of (a) vibration dampening base and (b) the full autopilot stack enclosed within the protective case

The slung payload was attached to the quadcopter using two brackets aligned to the x and y axes of the quadcopter and connected to one another using a cross beam. The ends of the beam were fitted with small frictionless bearings to allow the load to swing freely.

The fully assembled quadcopter with a mass of 900g can be seen in Figure 5.3 and the wiring for the quadcopter is illustrated in Figure 5.4. For more information on the wiring and the power module, refer to [59].

Figure 5.3: Picture of the fully assembled quadcopter

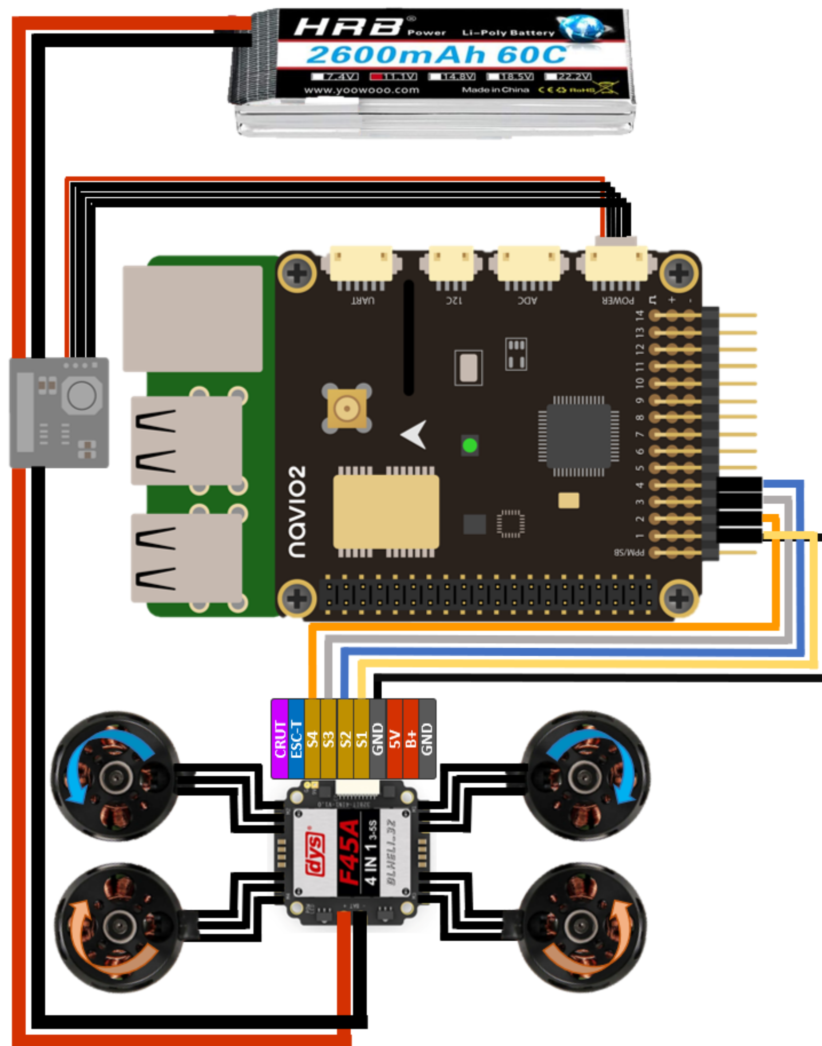
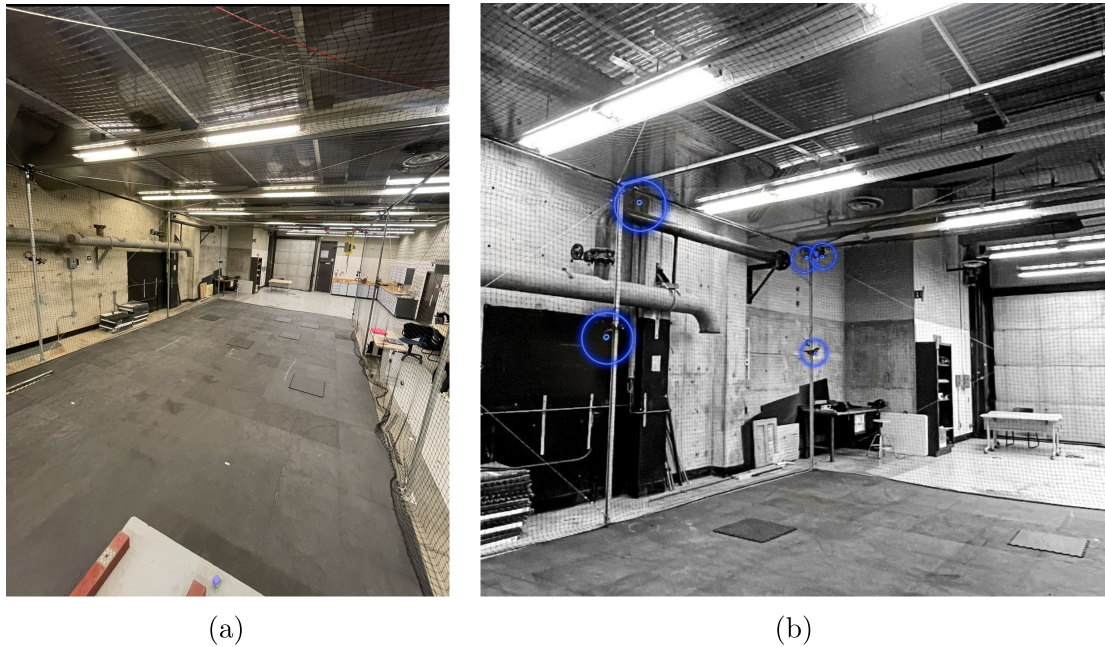


Figure 5.4: Figure of the wiring of Raspberry Pi, Navio2, the power module, the ESC, the motors, and the battery.

5.2 Testing Area Setup - Motion Capture

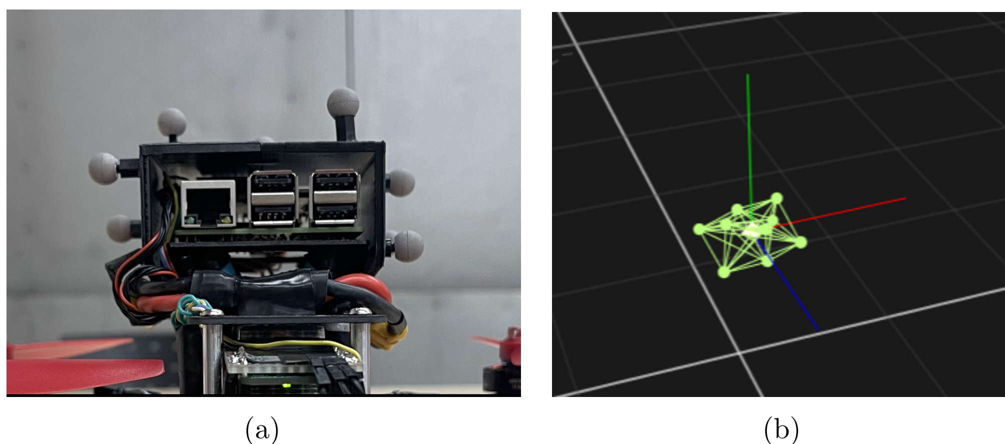
The quadcopter was tested in a secure area within a testing laboratory in the university shown in Figure 5.5a. An OptiTrack motion capture system was used to emulate the functionality of a GPS system and provide essential state information to the quadcopter's controller.



(a) (b)

Figure 5.5: Pictures of (a) The confined testing area in the robotics laboratory (b) The OptiTrack cameras.

Figure 5.6a shows the placement of the reflective markers on the quadcopter's protective case. The markers were placed on spacers with different heights to increase the tracking probability of the quadcopter. Motive's interpretation of the quadcopter as a cluster of these markers is shown in Figure 5.6b.



(a) (b)

Figure 5.6: (a) Photo of the markers positioned on the quadcopter's case (b) Figure of how OptiTrack interpreted the quadcopter.

By triangulating the markers' positions from multiple camera viewpoints, the motion capture system calculates the exact coordinates of the markers. The Motive software then processes this

raw marker data and translates it into the position and orientation of the quadcopter.

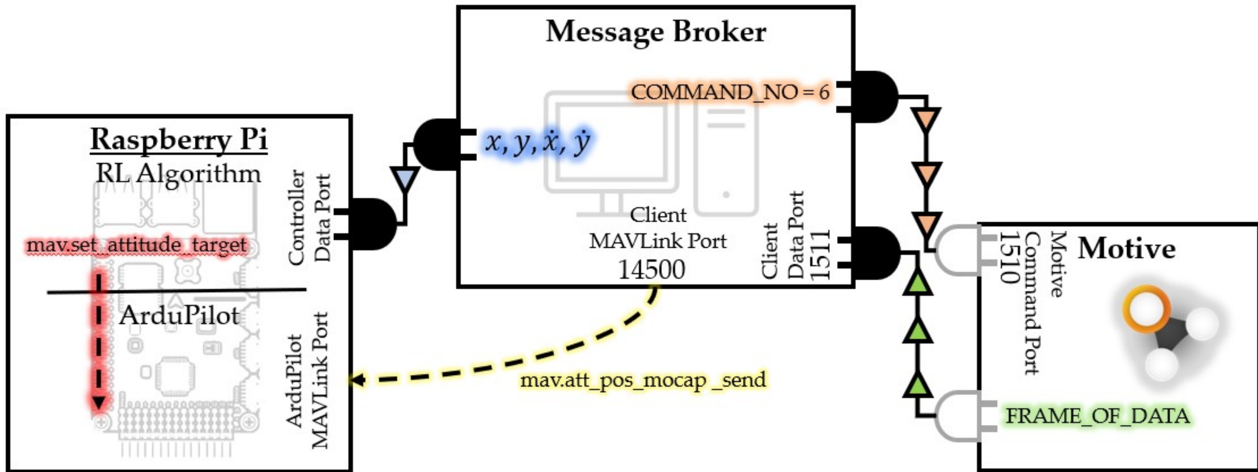


Figure 5.7: Schematic of the socket programming data transfer from the Motive software to the RL controller and autopilot on the UAV.

5.3 Data Transfer

Multiple considerations are necessary to test the RL algorithm on the UAV. Firstly, the UAV must be able to obtain its position and velocity, (x, y) and $(\dot{x}, \dot{y})$ to utilise the controller. This RL control algorithm must be executed fully on board the aircraft without external computational aid. Due to the use of OptiTrack in place of a GPS system, the autopilot must also receive the position and orientation data such that it has knowledge of the state of the UAV to operate. Finally, the autopilot must also be able to receive and execute the angle commands sent by the controller.

Once the necessary UAV position and orientation states $(x, y, z, q_x, q_y, q_z, q_w)$ required for either the RL controller or autopilot were configured in Motive, a method in which to process and send this data to each respective software was devised. Additionally, sending the angle commands from the controller to the autopilot was facilitated. The full diagram illustrating the data transfer between systems can be seen in Figure 5.7.

5.3.1 Receiving and Processing Motive's Data

Motive is configured to send frame data as a long string of bytes. To receive this frame of data, the client (receiver) must send a command to Motive through its command port requesting it. Once Motive receives this command it will send a frame of data back to the client at its data port. This transfer between the client and Motive server can be seen on the right hand side of Figure 5.7. To send the command and accept the data, a modified version of the NatNet SDK's Python sample was created. Refer to [60] for more information on the NatNet SDK client/server toolkit.

This transfer of data to and from the client and Motive server is achieved through *socket programming*. To establish this communication, both programs create a "socket", which is like a door through which data can enter or leave a program. Socket programming provides a set of rules and functions that enable these sockets to open, close, send, and receive data. Most motion capture systems are setup for sending data and receiving commands through UDP (User Datagram Protocol), which is a communication protocol commonly used in real time

applications for its low latency and simplicity [61].

Motive is equipped with two sockets: a command socket and a data socket. The data socket is dedicated to transmitting data, rendering port information unnecessary as nothing is being sent to it directly. Conversely, the command port is designed to receive instructions from the client and is bound to port 1510 of the Motive machine. Motive has the capability to send data directly to the Raspberry Pi, allowing it to unpack and compute the velocity before distributing the position and velocity data to the RL controller and ArduPilot. However, this approach burdens the Raspberry Pi's computational resources, given that Motive transmits data in lengthy byte sequences. Consequently, it falls upon the client to sift through the data packet and extract the relevant details.

To ease this strain on the Raspberry Pi, an intermediary message broker has been devised to act as the client for the Motive server. This message broker operates through two ports: one port (1511) for receiving the data transmitted by Motive, and another for sending commands to Motive's command port (1510). The broker sends a specific command with an associated command ID of 6 via the command socket, prompting Motive to dispatch a data frame with a designated message ID of 7 through the data socket. At the other end, the client awaits this data transmission.

Upon receiving the data frame, the message broker's responsibility is to extract the relevant information from the data string, compute the velocities, and send it to the corresponding program. Figure 5.8 shows bytes 0 to 96 of the data string which are the relevant bytes for this application. The client shifts the data to starts at byte 24 to obtain the position and orientation of the UAV and byte 62 for the position and orientation of the load both highlighted in the Figure 5.8.

For testing, the UAV was yawed 90°. This resulted in the axes shown in Figure 5.9 which also shows the motion capture's inertial frame and Ardupilot's NED frame. The message broker receives the positions in the motion capture inertial frame $\{M\}$. To send the data to the controller and autopilot, the message broker must convert them to frames $\{Q\}$ and $\{A\}$ respectively.

Sending the Positions and Velocities to the RL Controller

To convert from the motion capture position data from the Motive frame to the UAV frame, the message broker assigns a negated x_M to y_Q, and y_M to x_Q: $y_Q = -x_M \; x_Q = y_M$. It then computes the required velocities for the controller $(\dot{x}, \dot{y})$, packs the data and forwards this condensed data through another socket to the controller's data socket, specifying the appropriate port for seamless communication. This transfer of data can be seen in Figure 5.7 highlighted in blue. By providing the RL agent with the quadcopter's position and velocity states it can then generate the required features vector $\boldsymbol{\chi}(s)$ to enable TD updates of the value function and policy to occur in real time. This means that as the quadcopter flies the policy will continue to improve as it addresses model discrepancies.

Sending the Positions to the Autopilot

As the chosen autopilot software, ArduPilot was responsible for interpreting and handling MAVLink messages. MAVLink provides a set of predefined commands and messages that drones and ground control systems understand [62]. The drone can send information regarding its

Byte No.	Data
0–1	Message ID
2–3	Byte Count
4–7	Frame No.
8–11	No. of Marker Sets
12–15	No. of Unlabeled Markers
16–19	No. of Rigid Bodies
20–23	Rigid Body 1 ID
24–27	Rigid Body 1 x Position
28–31	Rigid Body 1 y Position
32–35	Rigid Body 1 z Position
36–39	Rigid Body 1 qx
40–43	Rigid Body 1 qy
44–47	Rigid Body 1 qz
48–51	Rigid Body 1 qw
52–55	Rigid Body 1 Mean Marker Error
56–57	RB1 Params
58–61	Rigid Body 2 ID
62–63	RB 2 x pos
64–65	RB 2 x pos
66–69	Rigid Body 2 y Position
70–73	Rigid Body 2 z Position
74–77	Rigid Body 2 qx
78–79	RB 2 qy
80–81	RB 2 qy
82–85	Rigid Body 2 qz
86–89	Rigid Body 2 qx
90–93	Rigid Body 2 Mean Marker Error
94–95	RB2 Params

Figure 5.8: Table of the bytes of data sent from motive and their corresponding information for bytes 0 to 95 to illustrate what data is being extracted to be sent to the controller.

attitude, sensors, position and many more to the controller. Likewise, the controller can use MAVLink to send commands to the drone's Ardupilot, instructing it to take-off, land, fly to a specific location, or return home among many others. These commands are packaged as MAVLink messages, ensuring that the drone comprehends the instructions correctly. In essence, MAVLink acts as a translator, allowing drones' autopilots and their controllers to talk to each other in a standardized way.

To relay the location and orientation of the UAV from Motive to the autopilot, the message broker uses MAVlink's ATT_POS_MOCAP message enabling the UAV to perform the pre-programmed ArduPilot functions in case of error. Before sending the positions and orientations to the autopilot it must be converted from the NWU $\{M\}$ frame to the NED $\{A\}$ frame. This is achieved for the position through applying the following: $x_A = x_M,\ y_A = -y_M,\ z_A = -z_M$. Similarly, for the quaternion: $q_{x,A} = q_{x,M},\ q_{y,A} = -q_{y,M},\ q_{z,A} = -q_{z,M},\ q_{w,A} = q_{w,M}$.

The ATT_POS_MOCAP message requires six inputs: the time in microseconds, the orientation q, the x position, the y position, the z position, and the covariance. The time can simply be obtained from the Raspberry Pi, the positions and quaternion are received and processed by the message broker, and the covariance is simply set to be 'NaN'. The transmission of this message can be seen in yellow in Figure 5.7.

For more information on this message and other available MAVLink messages refer to the MAVLink common messages in [63].

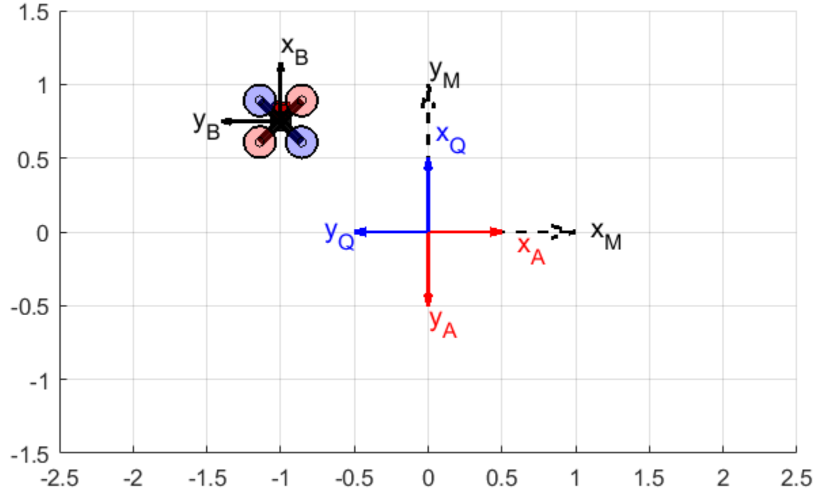


Figure 5.9: Figure showing the alignment of the UAV's frame once rotated $\{Q\}$, the MOCAP frame$\{M\}$ and the ArduPilot frame $\{A\}$ in the testing area.

5.3.2 Sending Attitude Commands to the AutoPilot

In addition to the x and y positions and velocities, the controller requires a method of sending the chosen command action to the autopilot.

To send attitude commands to the aircraft the MAVLink's SET_ATTITUDE_TARGET command was utilised. This command takes the desired roll, pitch and yaw in quaternion form along with the boot time, system ID, component ID, type mask and throttle. The boot time and system and component IDs are obtained directly from the Raspberry Pi and hence don't require any customisation. The quaternion is the most important part as it dictates the commanded attitude and can be achieved through mathematical conversion of the desired $[\phi, \theta, \psi]$ in the North-East-Down (NED) frame. The type mask is an eight bit string which is simply a way of conveying to the autopilot what it should ignore and what it should consider. Bits 1,2 and 3 relate to the roll, pitch and yaw rate while bits 7 and 8 relate to the throttle and attitude command. Assigning a bit to zero prompts the autopilot to ignore it while assigning it to one prompts the autopilot to consider it. For most applications including this experiment, the type mask is set to '00000111'. Finally the throttle dictates the climb rate of the UAV. For this work the throttle value was set to 0.5 which is associated with the aircraft not climbing. This means that when an attitude is commanded the UAV will adjust its thrust to maintain its z position at that new attitude.

The controller produces the desired roll ϕ_{RL} and pitch θ_{RL} in the NWU configuration. This needed to be converted to its NED equivalent by multiplying the NWU desired attitude vector $\boldsymbol{\zeta}_{RL}$ by the rotation matrix which converts from NWU to NED as shown in (5.1).

$$\begin{gathered}\boldsymbol{\zeta}_{NED} = {}^{NED}_{NWU}R\,\boldsymbol{\zeta}_{RL} \\ \begin{bmatrix}\phi_{NED}\\ \theta_{NED}\\ \psi_{NED}\end{bmatrix} = \begin{bmatrix}1 & 0 & 0\\ 0 & -1 & 0\\ 0 & 0 & -1\end{bmatrix}\begin{bmatrix}\phi_{RL}\\ \theta_{RL}\\ \psi_{NWU}\end{bmatrix}\end{gathered} \tag{5.1}$$

Since the controller is being trained to produce a pitch angle θ if used to control the roll, the ϕ command must be negated before being sent to the autopilot. This is because a positive ϕ angle

and a positive θ angle effect their corresponding positions changes differently. A positive θ angle results in an increase of the x position in the positive direction, while a positive ϕ results in an increase of the y position in the negative direction. Hence, for the controller to work in the roll axis, the sign of ϕ must be negated before being sent over. The final form of the conversion before sending to ArduPilot can be seen in (5.2).

$$\begin{bmatrix} \phi_{ArduPilot} \\ \theta_{ArduPilot} \\ \psi_{ArduPilot} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} -\phi_{RL} \\ \theta_{RL} \\ \psi_{NWU} \end{bmatrix} \tag{5.2}$$

For this project, MAVLink served multiple purposes beyond transmitting state information and roll and pitch commands to Ardupilot. It enabled a fully autonomous flight by directing the UAV to take off and reach the starting position before instructing the user to initiate the RL algorithm. Additionally, MAVLink was employed for enhanced safety measures. In case of errors in the RL algorithm, MAVLink commands were issued to Ardupilot to bring the UAV back to the center of the test bed, where it would pause, awaiting user input to either land or resume flight.

Chapter 6

Results and Discussion

This chapter presents the results of replicating the dynamics of the quadcopter and the waypoint tracking algorithm both in simulation and on the actual system. Discussion of the simulation of the input-output dynamics of the autopilot's roll PID controller is explained. The replicated input and output signals are then integrated into the model of the quadcopter-slung load to train the RL algorithm on possible aircraft dynamics.

The choice of the reward function is then explained as a result of testing several reward functions. The chosen reward function was utilised to train the RL agent on MATLAB using the model. The benefits and effect of extended training along with the choice of the standard deviation σ on the behaviour of the quadcopter is showcased. The progressive improvement and growth of the value function estimate and policy along the state space as training continues is illustrated, and the final trained value function and policy are discussed in detail.

Lastly, the tracking waypoint capability of the actual quadcopter is presented and compared to the same test replicated in simulation.

6.1 Simulated Results

While the algorithm employed by the controller (as given in Algorithm 4) is model-free, the initial training was carried out using the model described by (3.39) to (3.41). In this approach, the pitch θ serves as the input to the system, making (3.42) unnecessary for calculating the pitch angle, as it is now directed by the controller.

Simulated training is conducted to facilitate preliminary training where virtual convergence to the weight and parameter vectors $\boldsymbol{w}$ and $\boldsymbol{\theta}_\mu$ occurs. This phase is vital since during this training stage, the quadcopter explores numerous actions and may venture beyond the testing area, potentially causing safety issues. Therefore, training the quadcopter in simulation such that it behaves in a predictable manner before porting the weight and parameter vectors to the real system is essential.

Moreover, training the quadcopter in simulation significantly expedites the training process. Once the simulated quadcopter demonstrates reliable behavior and the vectors are transferred over to the real system, and further training continues to address model uncertainty that was not captured in simulation.

Training the RL agent in simulation can be split into the following parts:

1. Simulating the quadcopter's response to angle commands to enhance the accuracy of the model.
2. Developing the reward function for the RL algorithm.
3. Commence training the RL agent on the quadcopter-slung payload model.

MAVLink sends the RL controller's angle command as an input signal (or command signal) to the PID controller on board the autopilot which achieves the desired command through altering the rotation speeds of the propellers. To avoid infinite jerk, the command received by the PID on the autopilot closely resembled a sigmoid rather than a perfect step as shown in Figure 6.1. To enhance the model's accuracy, an analysis was conducted on the relationship between the input signal received by the PID and the generated response of the quadcopter. The analysis aimed to replicate this input-output relationship within the model such that the agent is trained to expect such dynamics from the quadcopter.

Prior to analysing the input signal to the PID and the generated response of the quadcopter, the quadcopter was tuned in the pitch and roll axes using ArduPilot's autotune mode. This process is explained further in Appendix H.

6.1.1 Control Input Signal Modelling

MAVLink's preset SET_ATTITUDE_TARGET command (explained in Section 5.3) was used to send roll angle commands of 5, 10, 15 and 20 degrees to the PID from a roll of 0; this was then repeated with negative roll angles. The data of the input signal to the PID as well as the quadcopter's response was stored and studied.

To replicate the input signal, MATLAB's curve fitting tool was used to estimate the variables of the general sigmoid function shown in (6.1).

$$y(t) = \frac{|\Delta\phi|}{1 + \exp\left(b(t-c)\right)} + d \tag{6.1}$$

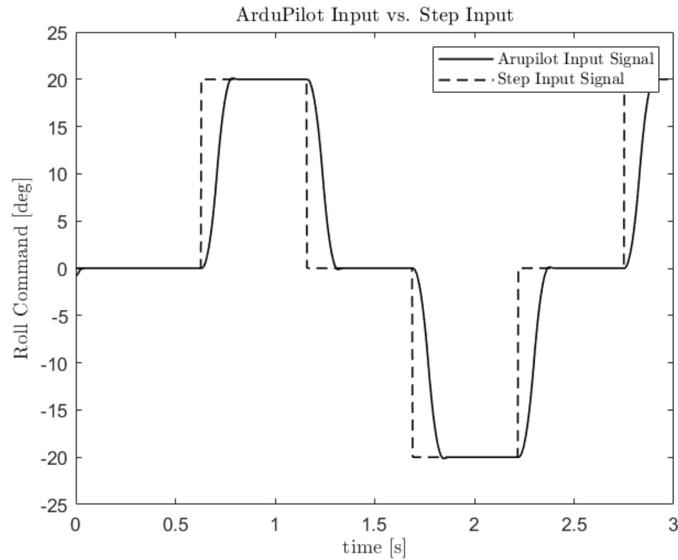


Figure 6.1: Figure comparing the ArduPilot input signal with a time constant of zero to a step signal for a 20 degree roll command.

The final estimated input signal equation takes the form shown in (6.2). For the full method of achieving this equation refer to Appendix I.

$$y(t) = \begin{cases} \frac{\Delta\phi}{1+\exp\left((-232.2/\sqrt{\Delta\phi})(t-0.0175\sqrt{\Delta\phi})\right)} + \phi_{current} & \text{if } \Delta\phi > 0 \\ \frac{|\Delta\phi|}{1+\exp\left((232.2/\sqrt{|\Delta\phi|})(t-0.0175\sqrt{|\Delta\phi|})\right)} + \phi_{commmanded} & \text{if } \Delta\phi < 0 \end{cases} \tag{6.2}$$

Figure 6.2 shows the control signal constructed using (6.2) for different roll angles compared with Ardupilot's command signal.

Each true input signal and roll response pair for the 5, 10, 15 and 20 degree tests were passed as arguments to MATLAB's *tfest* function. This resulted in four separate transfer functions each related to a specific angle command. The real transfer function of the aircraft was taken to be the average of all four transfer functions.

A simulated transfer function was created following the same method. However, instead of passing the true input signals, the inputs provided as arguments were the modelled signals of the 5, 10, 15 and 20 degree tests using (6.2). The four transfer functions were averaged to achieve the final simulated transfer function.

Figure 6.3 shows that the response generated using the transfer function created with the simulated command signals closely follows that of the true transfer function. This validates (6.2).

6.1.2 Aircraft Response Modelling

The transfer function generated using the modelled inputs is given mathematically by the continuous time transfer function shown in (6.3).

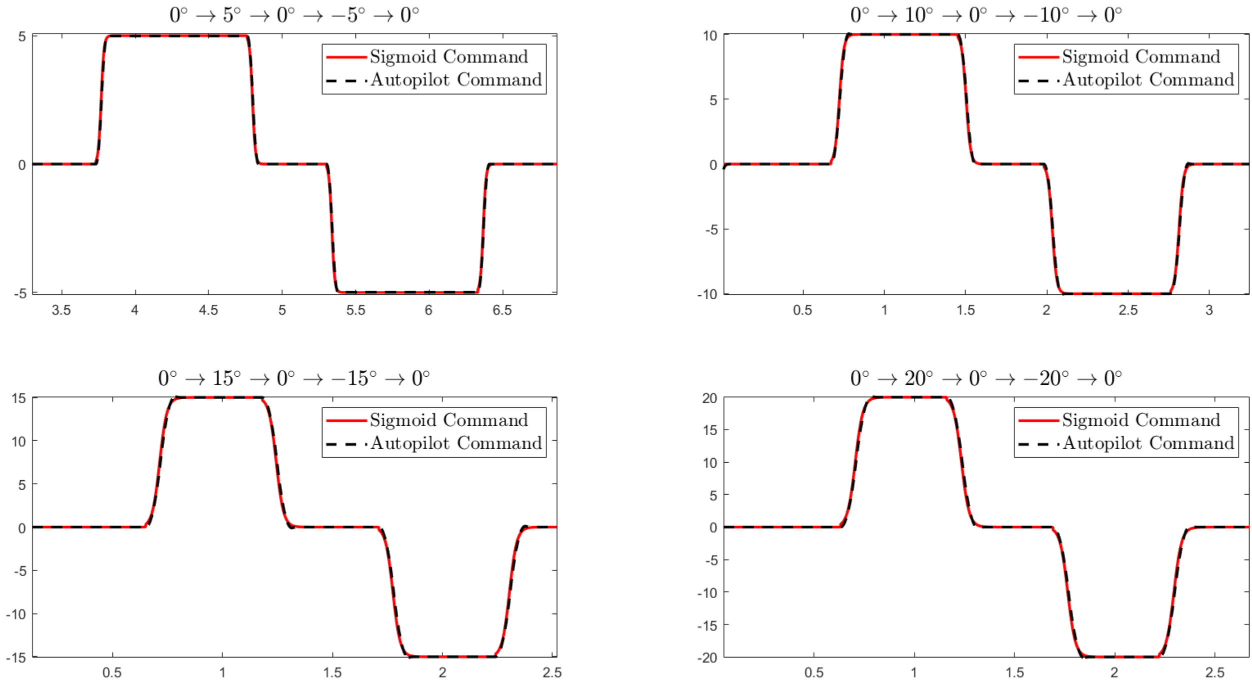


Figure 6.2: Figure showing the simulated command signals compared to the actual ArduPilot command signals.

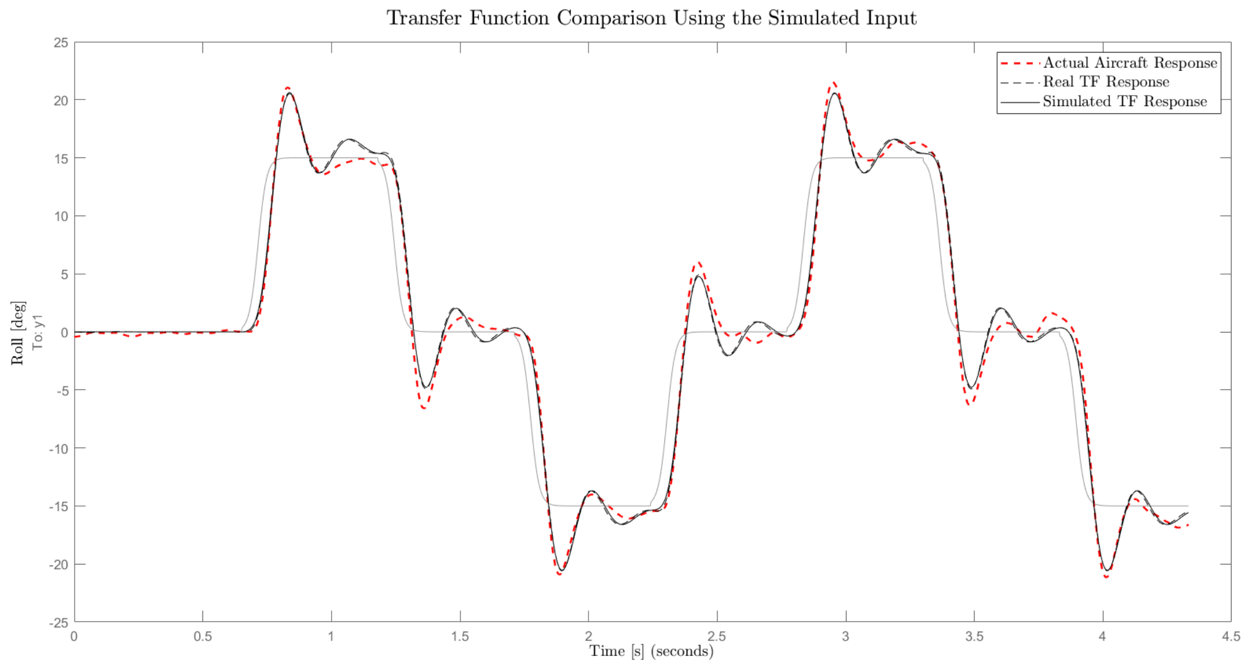


Figure 6.3: Figure showing a comparison between transfer functions created using the actual and simulated input.

$$TF(s) = \frac{\phi(s)}{y(s)} = \frac{880.7}{s^2 + 14.78 + 861.8} \tag{6.3}$$

To compute the roll angle ϕ at a specific time, the transfer function must be converted to the discrete time domain. At this point, the choice of the step size to train the RL agent had to be made and to compute the discrete transfer function accordingly. The RL time step was chosen to be 0.1 seconds to allow the RL algorithm sufficient time to produce a command and update the value function and policy estimates. That is, the quadcopter measures its current state, chooses an action, observes the reward and new state, computes the features vector and value function for that new state, and updates its weight and parameter vectors every 0.1 seconds.

Each RL step (0.1s) requires the model to provide the next state. For simulation, the model is applied using the fourth order Runge-Kutta (RK4) method (refer to Appendix C). For this method to provide accurate results it needs to compute the dynamics much faster than the speed at which the controller is commanding states. The RK4 model must compute the dynamics using a smaller time step and provide the final result to the controller. In this work, the RK4 time step was chosen to be 0.01 seconds. This meant that once the controller entered a state, the RK4 model would perform 10 successive computations at 0.01 seconds to provide the next state of the quadcopter after one RL step (0.1s) to the controller.

Using MATLAB's *c2d* function and the chosen Runge-Kutta time step, the discrete time transfer function was found to be the one shown in (6.4).

$$TF(z) = \frac{\phi(z)}{y(z)} = \frac{0.04164z + 0.03964}{z^2 - 1.783z + 0.8626} \tag{6.4}$$

$$\rightarrow \phi(z)\left(z^2 - 1.783z + 0.8626\right) = y(z)\left(0.04164z + 0.03964\right) \tag{6.5}$$

$$\rightarrow \phi(z)\left(1 - 1.783z^{-1} + 0.8626z^{-2}\right) = y(z)\left(0.04164z^{-1} + 0.03964z^{-2}\right) \tag{6.6}$$

$$\rightarrow \phi(t) - 1.783\phi(t-1) + 0.8626\phi(t-2) = 0.04164y(t-1) + 0.03964y(t-2) \tag{6.7}$$

$$\rightarrow \phi(t) = 0.04164y(t-1) + 0.03964y(t-2) + 1.783\phi(t-1) - 0.8626\phi(t-2) \tag{6.8}$$

Through rearranging the transfer function as shown in (6.5) to (6.7), the value of the roll of the aircraft ϕ at some time t can be calculated using (6.8) using the last two computed values of ϕ and y. Figure 6.4 shows the effect of integrating the aircraft's estimated system dynamics into the model.

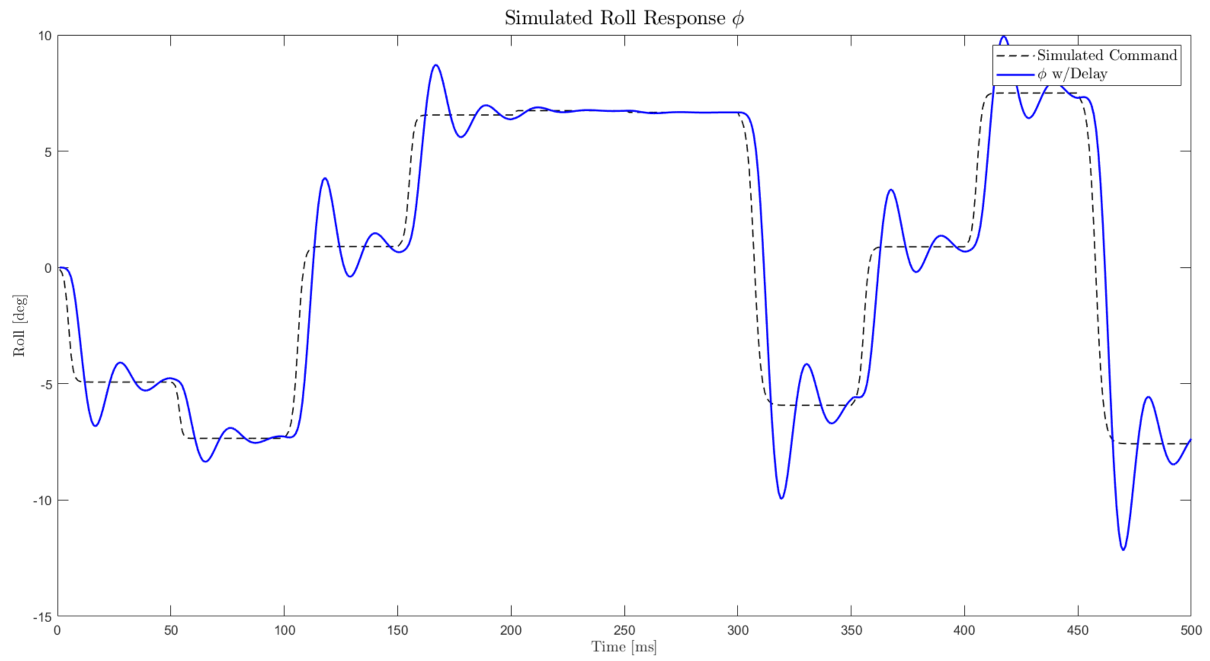


Figure 6.4: Figure showing the simulated roll response of the aircraft achieved through rearranging the validated transfer function.

6.1.3 Reward Function

Before starting the simulated training of the quadcopter, the appropriate reward function must be resolved. The values for the constant standard deviation σ, and the discount factor γ shown in Table 6.1 were chosen based on the control problem at hand. The values for the learning rates α_V and α_μ were tuned heuristically from initial learning step sizes of an inverted pendulum problem. The episodes were chosen to be time based with each episode being the equivalent to 10 seconds of flight time with no existing terminal states.

The base of the reward function used was in the exponential form shown in (4.51) on Page 64. Since there are no specified terminal states or updates, the β value for the reward function was chosen to be zero. Several values for c were tested for $1,000$ episodes of 100 steps each. Figure 6.5 shows that increasing the c constant speeds up the learning rate of the quadcopter.

Table 6.1: Table of the step sizes, standard deviation and discount factor used for the reinforcement learning controller

RL Controller Constants	
Value Function SGD Step Size α_V	1.5×10^{-3}
Gaussian Action Mean SGA Step Size α_μ	1.0×10^{-9}
Gaussian Action Standard Deviation σ	3.0×10^{-2}
Discount Factor γ	0.99
Fourier Basis Order n	25
Reinforcement Learning Time Step ts_{RL}	0.1s
Runge Kutta Time Step ts_{RK}	0.01s
Steps per Episode	100

Although $1,000$ episodes of training is not enough to achieve adequate performance, there is a significant improvement between a reward function with a constant $c = 1$ shown in Figure 6.5a in comparison with the reward functions with constant $c = 15$ shown in Figure 6.5c. As the constant c increases, the quadcopter is pushed faster towards the goal. This is due to the reward function becoming steeper with an increasing c, ultimately resulting in more aggressive semi-gradient updates of the value function and policy weight and parameter vectors.

This effect remains evident even after $5,000$ episodes of training as shown in Figure 6.6. While both cases show improvement when compared to the $1,000$ episode training case, the reward function with $c = 15$ shown in Figure 6.6b is more reliable in achieving the goal of driving the error to zero. This is because the reward is integrated into TD error $\delta = \Big(r + \gamma\widehat{V}(\boldsymbol{s}'\boldsymbol{w})\Big) - \widehat{V}(\boldsymbol{s}, \boldsymbol{w})$. Increasing the value of the constant c increases the overall TD error which increases the magnitude of the steps taken by the semi-gradient updates of the weight and parameter vectors relative to the error.

However, the constant c cannot be set to an extremely large number. Due to the nature of the Fourier basis value function approximator, problems arise when approximating large jumps or discontinuities. Since the reward function is an integral part of the value function semi-gradient descent update, increasing the c value creates excessively large jumps in the approximation process which adversely affects the value function approximation. This results in extreme behaviours and is illustrated in Figure 6.7 which shows the case when c is set to 200.

In Figure 6.7a, after 100 episodes of training, the quadcopter is starting to learn as it pushes

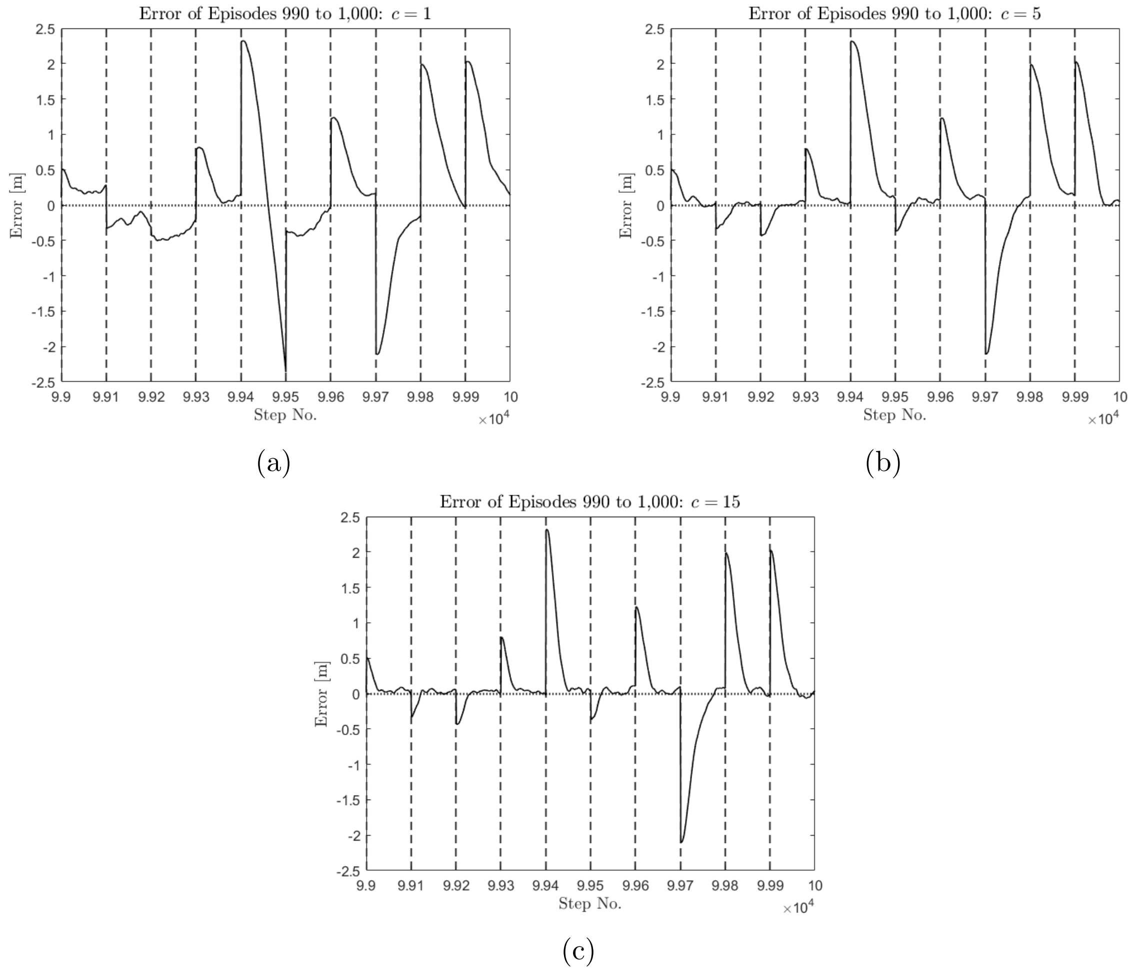


Figure 6.5: Comparison of the performance of the simulated quadcopter following 1,000 episodes of training with reward functions with a c constant of (a) 1. (b) 5. (c) 15.

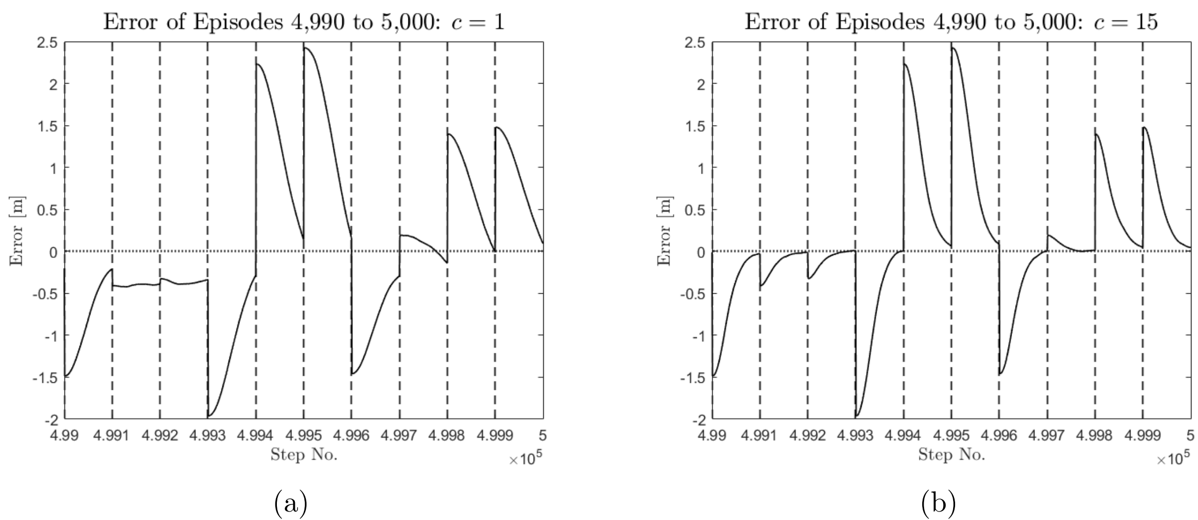
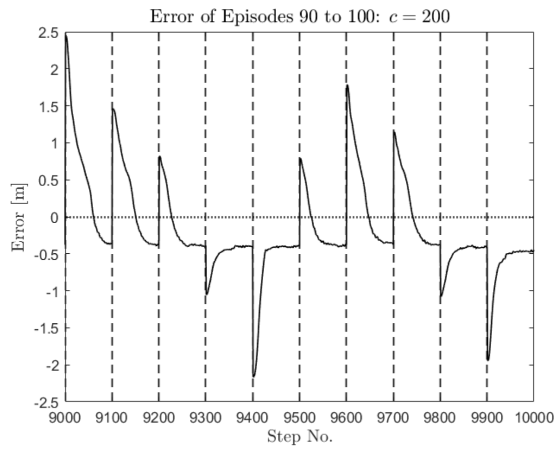
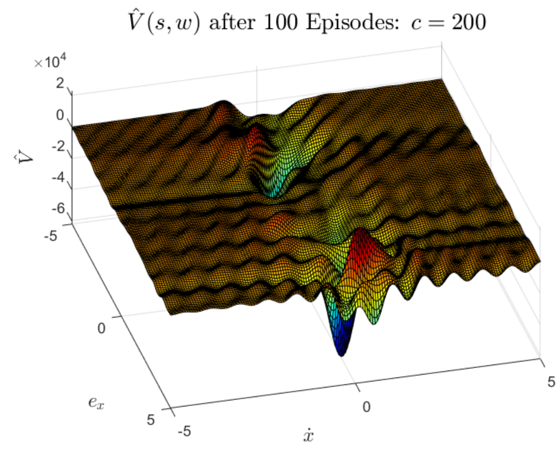


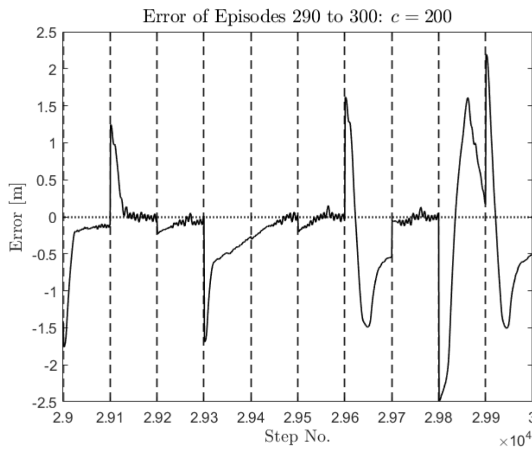
Figure 6.6: Comparison of the performance of the simulated quadcopter following 5,000 episodes of training with reward functions with a c constant of (a) 1. (b) 15.



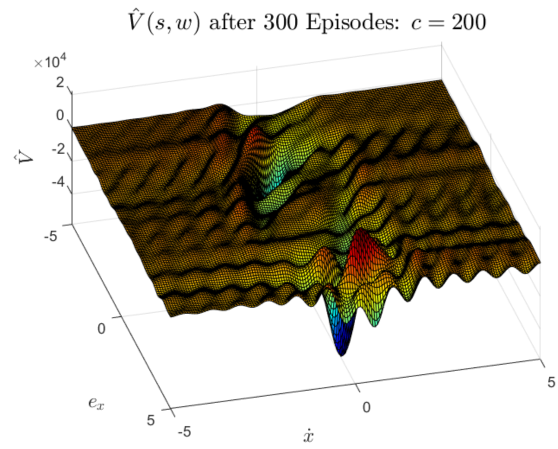
(a)



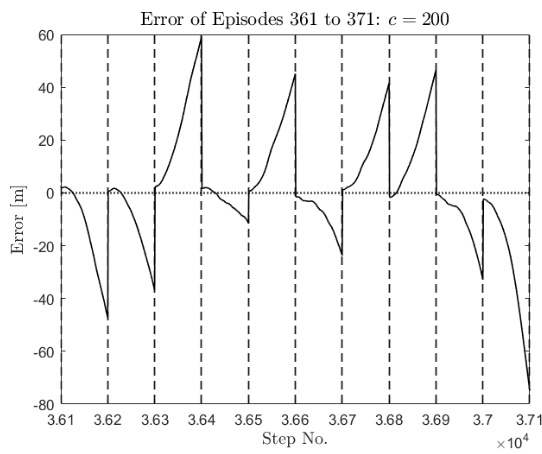
(b)



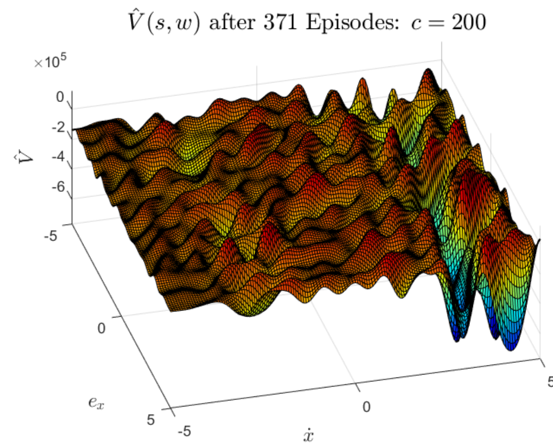
(c)



(d)

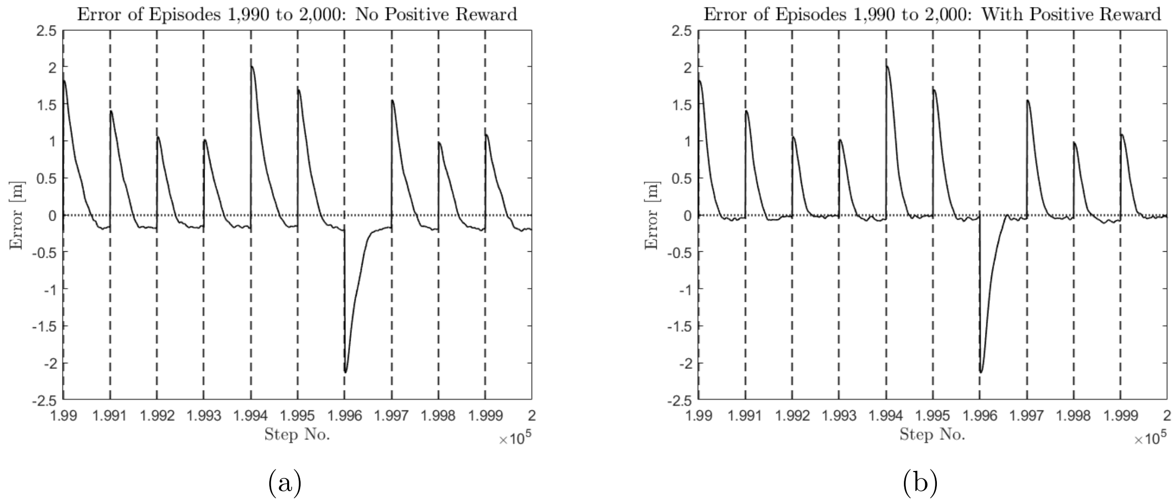


(e)



(f)

Figure 6.7: Comparison of the value function approximation and performance of a quadcopter with a reward function with $c = 200$



(a)

(b)

Figure 6.8: Figure comparing the waypoint tracking performance of a quadcopter after 2,000 episodes of training (a) without any positive reward. (b) with the introduction of positive reward when the error is < 0.2m.

the error closer to zero in most of the episodes. Similarly, in Figure 6.7b, the value function is starting to take form. After 300 episodes of training, Figures 6.7c and 6.7d show that the quadcopter is starting to exhibit unpredictable behaviour as it overshoots the goal and stabilises at different error values. The beginning breakdown in the value function approximation is also starting to spread. Finally, after 371 episodes, the aircraft completely loses control and shoots away from the goal as represented in Figure 6.7e. The value function has also been completely taken over by the oscillatory "ringing" as shown in Figure 6.7f. One can see that the area about $\dot{x} = 0$ where the value function was beginning to take shape in Figures 6.7b and 6.7d has been covered completely in oscillations that resemble random noise. This is expected from Fourier series approximation when evaluating frequent and large discontinuities resulting from an excessively large value of c. This is ultimately what leads to the failure of the controller.

Following recursive testing, the value of c was chosen to be 20, which resulted in the following reward function.

$$r(s_t) = -20(x_t - x_{goal})^2 \tag{6.9}$$

One issue that recurs was that once the quadcopter got close enough to the goal it would often overshoot or undershoot the goal and stabilise there as shown in Figure 6.8a. In the work of [29] a positive reward was given to the agent once it got close to the goal. Following further testing and excessive training, the choice of introducing a positive reward, equivalent to 10% of c, upon reaching an error of $|e_x| = 0.2$ was made, resulting in the following and final reward function.

$$r(s_t) = \begin{cases} -20(x_t - x_{goal})^2 & \text{if } |x_t - x_{goal}| \geq 0.2 \\ 2 - 20(x_t - x_{goal})^2 & \text{if } |x_t - x_{goal}| < 0.2 \end{cases} \tag{6.10}$$

The implementation of the positive reward on the same training set significantly improved the aircraft's approach to the goal as illustrated in Figure 6.8b. Note that Figure 6.8 shows training after 2,000 episodes which is far from convergence.

6.1.4 Simulated Quadcopter Training & Waypoint Tracking

The training of the simulated quadcopter was done with the replicated input signal (6.2), the response (6.8) and the final reward function (6.10). The mass for the quadcopter and payload were set to 900g and 40g respectively to match the true system. Training using the $TD(0)$ algorithm shown in Algorithm 4 was conducted on MATLAB with the constants shown in Table 6.1.

In reinforcement learning, the agent always starts learning from scratch with both the weight and parameter vectors $\boldsymbol{w}$ and $\boldsymbol{\theta}_\mu$ initialised to zero. In the beginning, the agent attempts to explore its environment and inputs by taking unexplored actions that may lead to new, unvisited states. Thus at the start, the quadcopter will exhibit unpredictable and possibly dangerous behaviours. This continues until the quadcopter has enough experience to realise which states and actions lead to desirable behaviours that achieve higher rewards. Allowing the RL agent to start learning on the real quadcopter poses safety hazards in addition to being impractical time-wise. By starting the RL agent's training in simulation, not only can the agent be trained without any safety concerns, but the training can be done much faster than on the real quadcopter with a 10 second episode in real-time taking only a fraction of a second to run on MATLAB. Consequently, convergence to appropriate vectors can occur in shorter time using a simulator. Additionally, training on the simulator allows for essentially infinite training space, allowing the quadcopter to realise more states further away from the goal than it could when trained on the actual system. Once the controller is trained on the model, it can then be transferred to continue learning on the real quadcopter.

The desired range for training the simulated quadcopter was chosen to have limits of -2.5m and 2.5m in the position error domain emulating the available testing space in the laboratory. However, the state space, explained in (4.50), was chosen to have upper and lower limits of -5m and 5m as shown in (6.11) to account for possibility of the quadcopter increasing the error rather than decreasing it in the beginning portion of the training.

$$\mathcal{S} = \begin{bmatrix} e_{x,min} & e_{x,max} \\ \dot{x}_{min} & \dot{x}_{max} \end{bmatrix} = \begin{bmatrix} -5 & 5 \\ -5 & 5 \end{bmatrix} \tag{6.11}$$

Training Process

At the start of every episode, the quadcopter is initialised to start at a zero position and velocity and a goal between $[-2.5, 2.5]$ meters is chosen randomly by MATLAB. Due to this, the frequency that states were visited by the quadcopter was randomised.

As previously mentioned, the vectors $\boldsymbol{w}$ and $\boldsymbol{\theta}_\mu$ were set to zero at the start of the training. This meant that the value function initially gives all states the same value until the aircraft had explored enough to distinguish what states were desirable. As for the policy's mean μ, the controller would start commanding small angles and continue taking steps towards increasing angle commands until it had discovered the appropriate commands for each state. For this particular case, a saturation limit was set such that the roll angle commands were limited between -20° and 20° due to safety concerns.

After $1,000$ episodes of training the waypoint tracking capabilities of the quadcopter were poor, with stabilisation occurring 0.15m away from the goal as shown in Figure 6.9a. In addition, the controller was only commanding maximum roll angles of around 8° as shown in Figure 6.9b. These low pitch θ commands are also apparent in Figure 6.9d where most of the state

to action mapping does not exhibit any extreme peaks since the plot lacks dark red and blue portions which indicate high θ commands. Furthermore, the value function shown in Figure 6.9c is asymmetrical with more negative values being present in the bottom half of the plot. This indicates that the quadcopter had not yet explored a large portion of the state space since the value function in this experiment is expected to be symmetric.

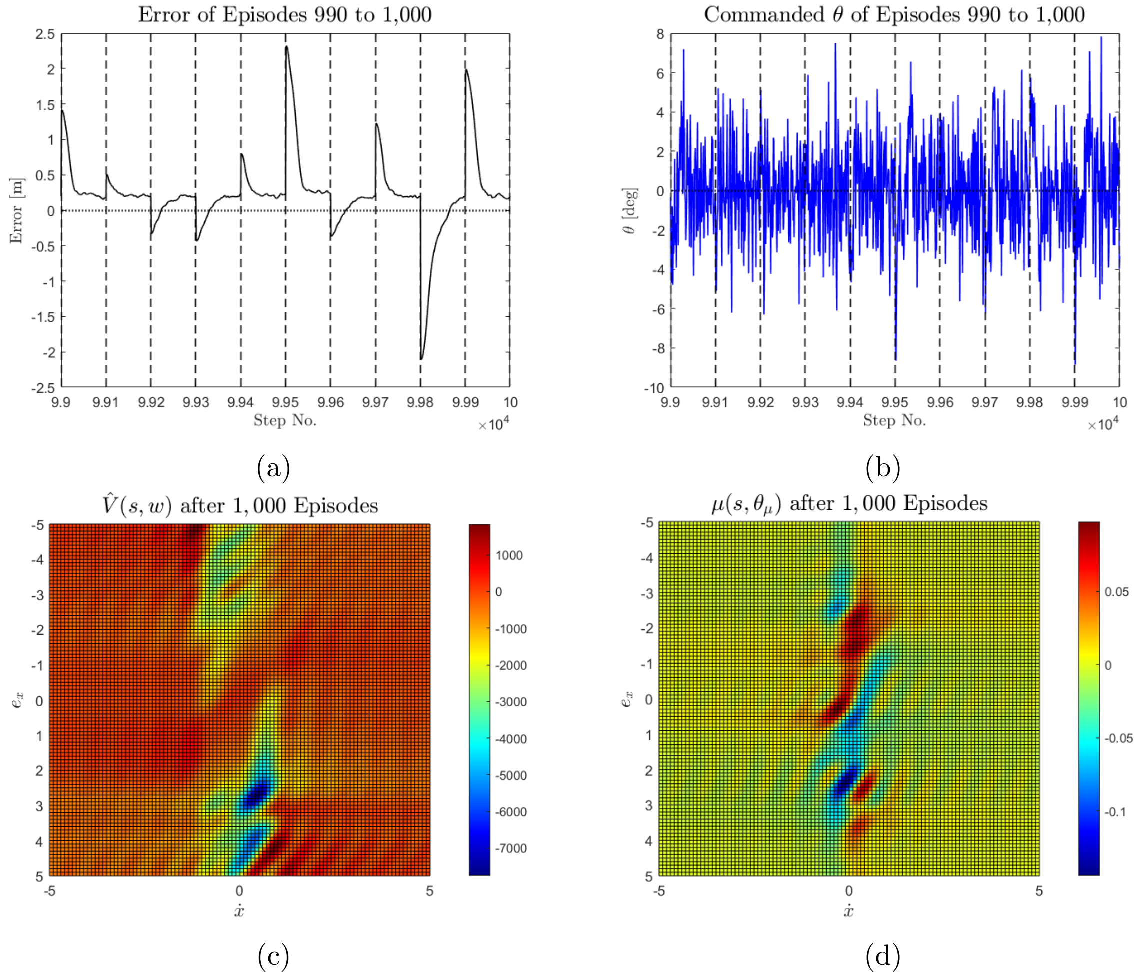


Figure 6.9: Figure showing the (a) Performance (b) Commanded roll of the simulated quadcopter (c) The value function estimate and (d) The policy's mean following 1,000 episodes of training.

After 10,000 episodes, the waypoint tracking abilities have become more reliable as they pushed the quadcopter to reach the goal rather than stabilising in the neighbourhood around the goal as shown in Figure 6.10a. The controller was commanding overall higher pitch angles, crossing the 10° range demonstrated by the commands in Figure 6.10b along with the expanded darker regions in the policy's mean shown in Figure 6.10d indicating higher and lower areas forming in the policy surface. Additionally, the symmetric property of the value function began to take form as expected with darker regions occurring diagonally from one another as shown in Figure 6.10c. It was also at this point that the realisation of the space that the quadcopter will be occupying started to become evident. The majority of the states in the error domain e_x were being visited as it can be seen visually that updates were occurring along that axis. However, the same could not be said for the velocity states. The quadcopter had only explored around the range of $[-1.20, 1.20]$ m/s in the $\dot{x}$ domain as indicated in Figure 6.11. However, this does

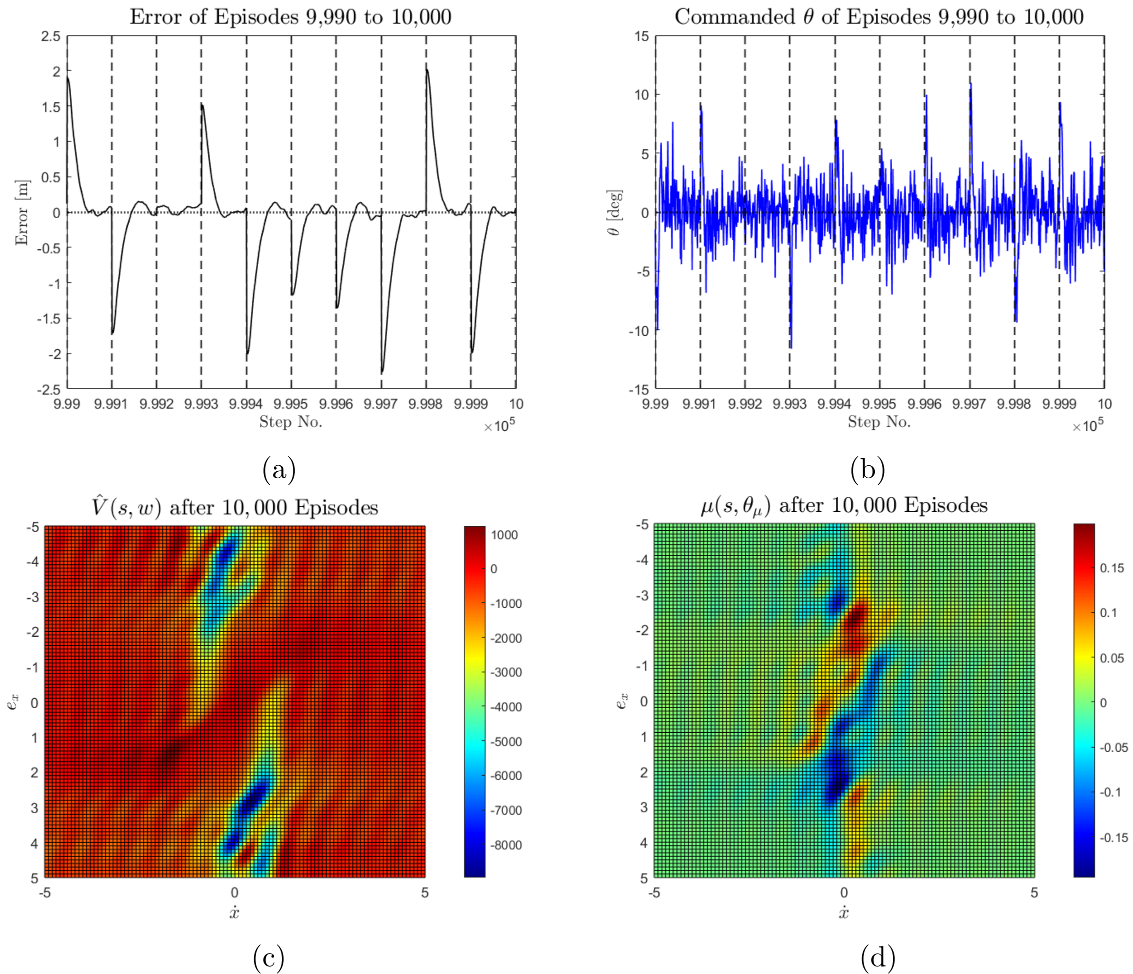


Figure 6.10: Figure showing the (a) Performance (b) Commanded roll of the simulated quadcopter (c) The value function estimate and (d) The policy's mean following 10,000 episodes of training.

not pose an issue since it simply indicates that the quadcopter is unable to reach higher speeds with the limits placed on its command.

The quadcopter was trained further, reaching 25,000 total episodes of training. Although there was no significant increase in the magnitude of the pitch commands apparent in Figure 6.15b,there was a notable improvement in performance with a decrease in the rise time.

Figure 6.12 shows the improvement in the controller between 10,000 and 25,000 episodes of training. By allowing the quadcopter to train for an additional 15,000 episodes the rise time dropped by 46%. Figure 6.15d shows that this improvement is related to the increase in the number of states in which the agent has updated the policy, resulting in overall higher pitch commands in a larger area in the state space. By examining the value function in Figure 6.15c, there has been an expansion of the darker red areas indicating that during this training period, the quadcopter had realised areas of high values which are usually located closer to the goal. This would indicate why the policy has not updated substantially since when the quadcopter is close to the goal, the pitch required would not be high.

It was determined that the standard deviation value chosen of 3.0×10^{-2} was too large for the system due to the presence of large oscillations in the chosen commands shown in Figure 6.15b even when the quadcopter was at the goal. Hence, the decision was made to reduce the

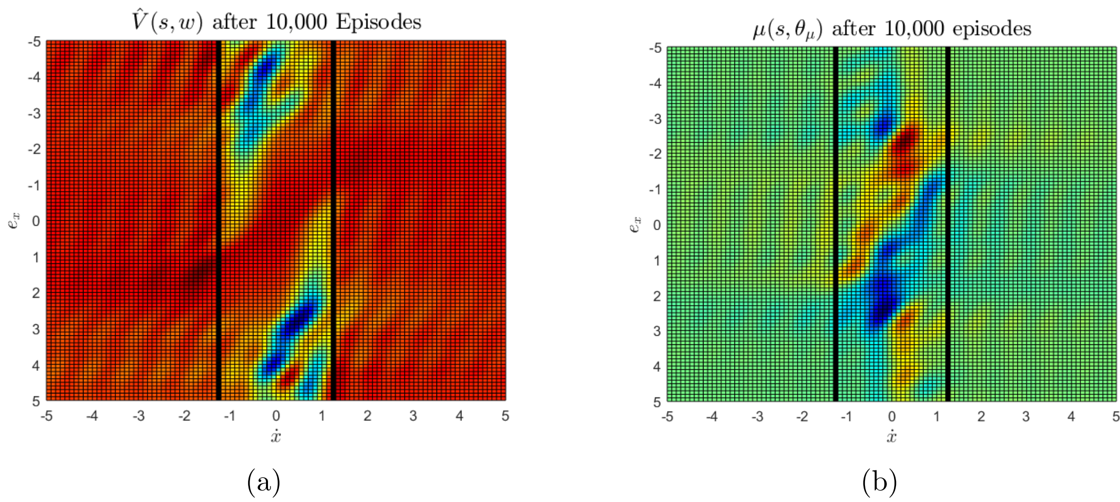


Figure 6.11: Figure indicating the updated portion of the (a) $\hat{V}$ (b) μ after 10,000 episodes of training

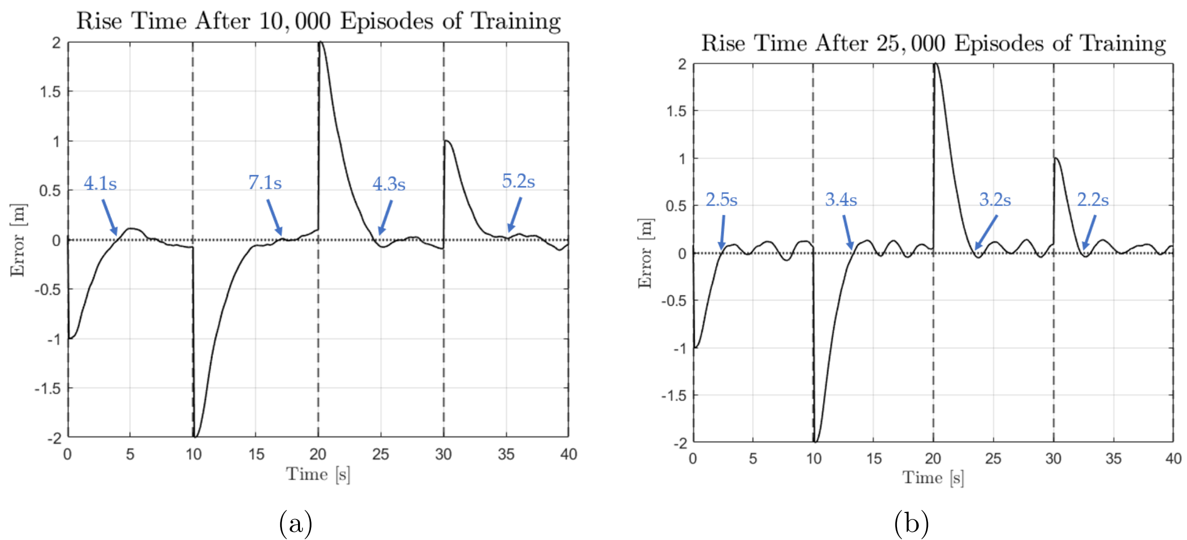


Figure 6.12: Figure presenting the rise times of the controller after (a)10,000 (b)25,000 episodes of training.

standard deviation σ to 3×10^{-3} and allow the quadcopter to continue training for 5,000 more episodes to account for the change.

The improvement of the RL agent can be visualised through the examination of its returns G_t and TD error δ shown in Figure 6.13. Both the return and TD error start out extremely high and taper off as training continues. This indicates that at the start, the value function estimate was poor leading the quadcopter to collect a large amount of highly negative rewards. As the agent continues to learn, the TD error starts to decrease indicating the improvement in the value function approximation resulting in the quadcopter achieving higher returns. An interesting phenomenon is the sudden spike in δ around the 15,000 episode mark indicating that even after 15,000 episodes there were areas in the state space which the quadcopter still had not visited. This is mirrored in the return where there is a sudden decrease indicating that the aircraft had collected low rewards in that time. The sudden spike did not interrupt the

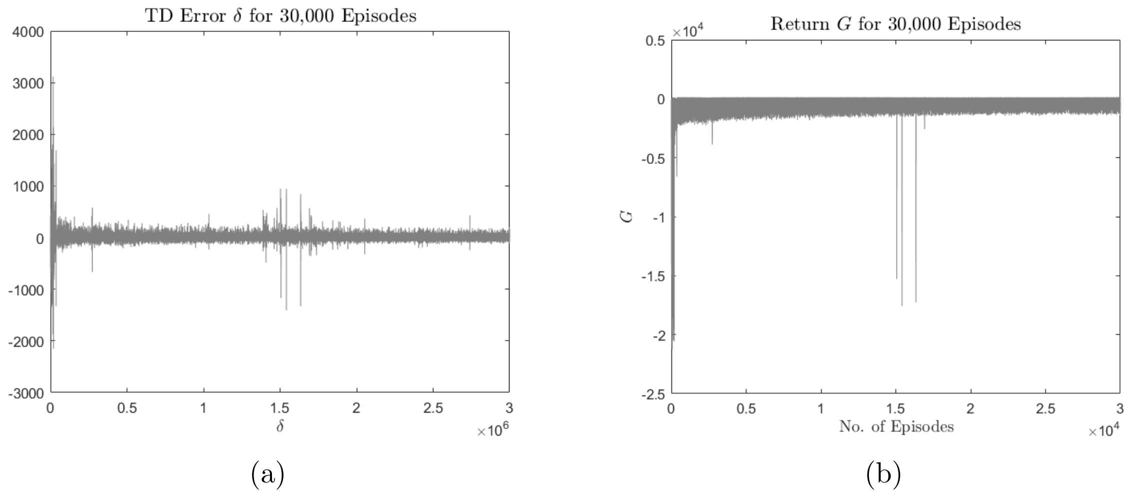


Figure 6.13: Figure illustrating the (a) TD error and (b) Return for 30,000 episodes of training.

agent's training as it continued to shrink both performance metrics.

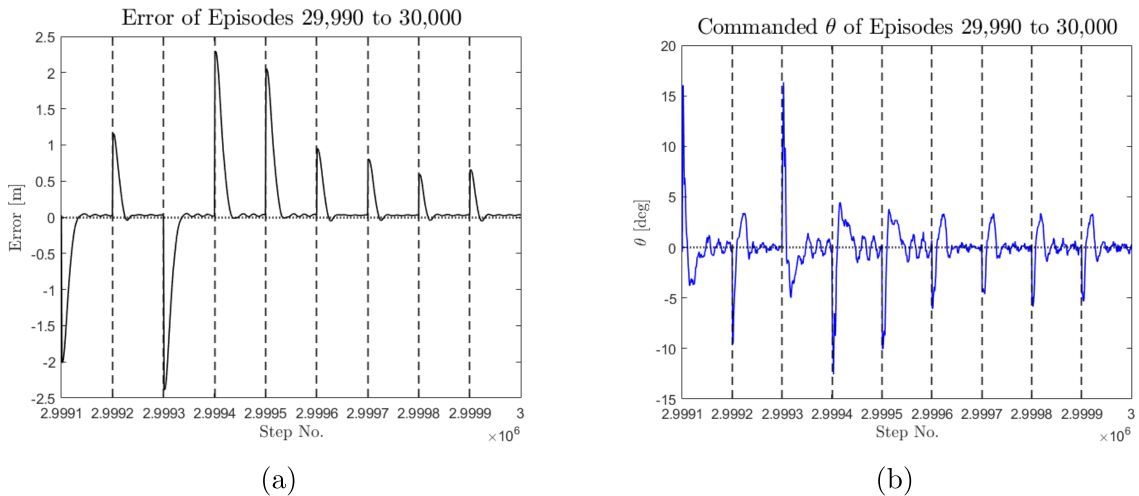


Figure 6.14: Figure illustrating the (a) Performance and (b) Commanded roll of the simulated quadcopter following the decrease in the standard deviation.

Figure 6.14 shows the final results following the reduction of σ followed by additional training. It is evident that the oscillations have been mitigated in both the commanded pitch θ and position about the goal. Satisfied with the performance of the quadcopter, the weight and parameter vectors following the 30,000 episodes were chosen for implementation on the actual quadcopter. The next section of results discusses the trained value function and policy.

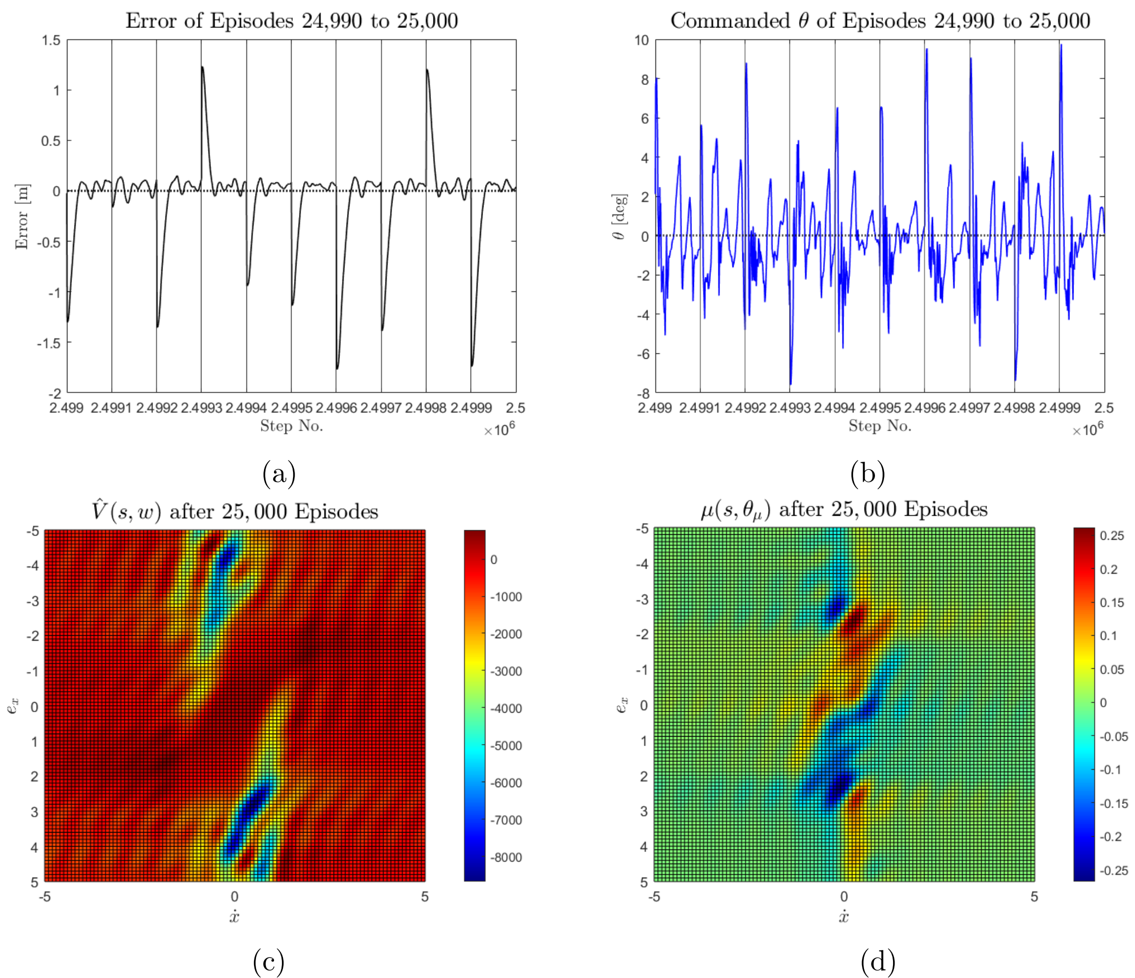


Figure 6.15: Figure showing the (a) Performance (b) Commanded roll of the simulated quadcopter (c) The value function estimate and (d) The policy's mean following 25,000 episodes of training.

Value Function and Action Selection Analysis

To understand the quadcopter's behaviour within the state space in more detail, the state space can be split into quadrants which can further be split into sub-quadrants as shown in Figure 6.16. Since the error e_x is defined at $x - x_{goal}$, opposing signs of e_x and $\dot{x}$ indicate that the quadcopter is moving towards the goal. This means that in quadrants 1 and 3 the quadcopter is moving closer to the desired waypoint whereas in quadrants 2 and 4 the quadcopter is moving further away from it. This understanding of what the quadcopter is actually doing within certain portions of the state space can help in understanding why the reinforcement learning agent might be choosing certain values and actions.

	Quadrant 2 -ve e_x, -ve $\dot{x}$		Quadrant 1 -ve e_x, +ve $\dot{x}$	
	2.2	2.1	1.2	1.1
	2.3	2.4	1.3	1.4
e_x	3.2	3.1	4.2	4.1
	3.3	3.4	4.3	4.4
	Quadrant 3 +ve e_x, -ve $\dot{x}$	$\dot{x}$	Quadrant 4 +ve e_x, +ve $\dot{x}$	

Figure 6.16: Figuring illustrating the split of the state space with labeled numbers used in the text.

- Sub-quadrants 1.3 and 3.1
  - $0 < |e_x| < 2.5$m, $0 < |\dot{x}| < 0.6$m/s, $sgn(e_x) \neq sgn(\dot{x})$
  - The aircraft is in close proximity to the goal and approaching slowly.
- Sub-quadrants 1.4 and 3.2
  - $0 < |e_x| < 2.5$m, $0.6 < |\dot{x}| < 1.2$m/s, $sgn(e_x) \neq sgn(\dot{x})$
  - The aircraft is in close proximity to the goal and approaching quickly.
- Sub-quadrants 1.1 and 3.3
  - $2.5 < |e_x| < 5$m, $0.6 < |\dot{x}| < 1.2$m/s, $sgn(e_x) \neq sgn(\dot{x})$
  - The aircraft is far away from the goal and approaching quickly.
- Sub-quadrants 1.2 and 3.4
  - $2.5 < |e_x| < 5$m, $0 < |\dot{x}| < 0.6$m/s, $sgn(e_x) \neq sgn(\dot{x})$
  - The aircraft is far away from the goal and approaching slowly.
- Sub-quadrants 2.4 and 4.2

  - $0 < |e_x| < 2.5$m, $0 < |\dot{x}| < 0.6$m/s, $sgn(e_x) = sgn(\dot{x})$
  - The aircraft is in close proximity to the goal and moving away slowly.
- Sub-quadrants 2.3 and 4.1
  - $0 < |e_x| < 2.5$m, $0.6 < |\dot{x}| < 1.2$m/s, $sgn(e_x) = sgn(\dot{x})$
  - The aircraft is in close proximity to the goal but moving away quickly.
- Sub-quadrants 2.1 and 4.3
  - $2.5 < |e_x| < 5$m, $0 < |\dot{x}| < 0.6$m/s, $sgn(e_x) = sgn(\dot{x})$
  - The aircraft is far away from the goal and moving away slowly.
- Sub-quadrants 2.2 and 4.4
  - $2.5 < |e_x| < 5$m, $0.6 < |\dot{x}| < 1.2$m/s, $sgn(e_x) = sgn(\dot{x})$
  - The aircraft is far away from the goal and moving away quickly.

The above explanation of the behaviour of the quadcopter in each sub-quadrant, simplifies the analysis of the results shown of the final plots of the value function and policy, especially after splitting the plots into the same sub-quadrants. The quadcopter was provided a state space of $e_x = [-5, 5]$ and $\dot{x} = [-5, 5]$, however the quadcopter did not utilise this entire space. To accurately analyse the results of the value function and policy the actual utilised state space must be determined.

Following the training in simulation, the quadcopter had performed updates along the entire error e_x's range. However, the RL agent did not update the value function and policy beyond the range of $\dot{x} = [-1.20, 1.20]$m/s. This is because the quadcopter could not visit any states with speeds greater than 1.20m/s due to the saturation limit of 20° placed on the controller's policy. Therefore, to achieve a clearer understanding of the controller's generated results, the trained $\hat{V}(s, w)$ and $\mu(s, \theta_\mu)$ surface plots were adjusted to the range of $e_x = [-5, 5]$ and $\dot{x} = [-1.2, 1.2]$.

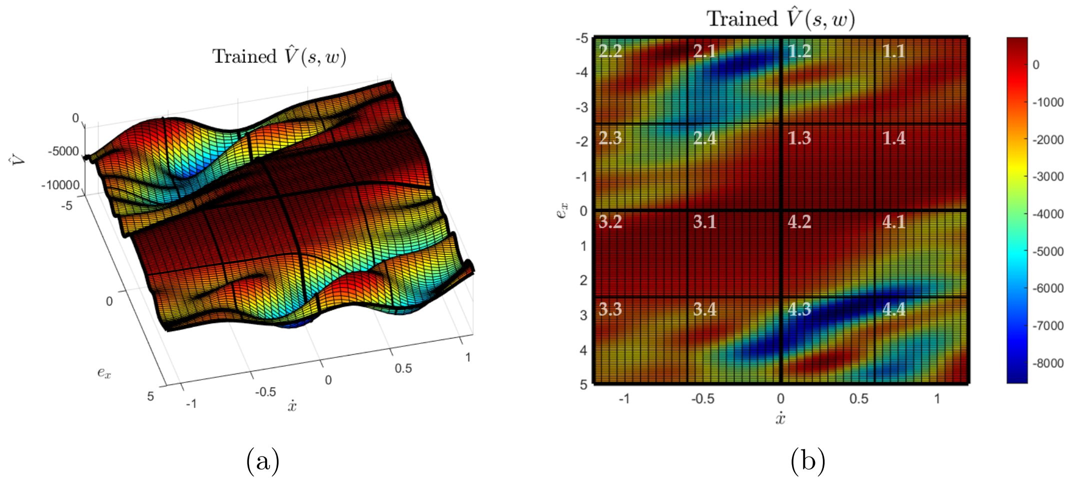


Figure 6.17: Figure illustrating the chosen value function split into the quadrants and sub-quadrants defined in Figure 6.16 from (a) an isometric view. (b) a top view.

Value Function Results Figure 6.17 shows the trained value function plot split into the sections described in Figure 6.16. The area of highest estimated value produced correlates

with an error and velocity of zero which is as expected since the quadcopter will only receive positive reward when remaining at an error $e_x < 0.2$ with the highest possible reward located at $e_x = 0$. As a general trend, the first and third quadrants contain higher values than the second and fourth. This is because the states in the first and third quadrants are associated with the quadcopter moving towards the goal, shrinking the error and increasing the reward it is receiving. The second and fourth quadrants contain states in which the aircraft is moving away from the goal, exponentially decreasing the reward it receives while simultaneously increasing the time it would take to reach the goal.

Furthermore, sub-quadrants 1.3, 1.4, 3.1, and 3.2 where the error $0 < |e_x| < 2.5$m have higher values than sub-quadrants 1.1, 1.2, 3.3, and 3.4 where the error $2.5 < |e_x| < 5$m. This is expected since the value of a state $V(s)$ is the accumulated sum of rewards from starting in a particular state to the end of the episode which is set to be 10s long. Hence, if the quadcopter finds itself in any of the sub-quadrants with $0 < |e_x| < 2.5$m, it is bound to accumulate less negative rewards and more positive rewards than if it were to start at a state with an error $2.5 < |e_x| < 5$m as it will spend less time travelling towards the goal and more time near the goal.

The same phenomenon is encountered in the second and fourth quadrants even though the quadcopter is moving away from the goal. Sub-quadrants 2.3, 2.4, 4.1 and 4.2 with error values of $0 < |e_x| < 2.5$m have higher values than than 2.1, 4.3 and 4.4 where $2.5 < e_x < 5$m. Assuming that the quadcopter realises it is moving away from the goal rather than towards it and decides to switch direction, it would take longer for the quadcopter to reach the goal from higher error values resulting in more negative rewards in the episode. Note that sub-quadrant 2.2 has unexpectedly high values. This is discussed later in this section.

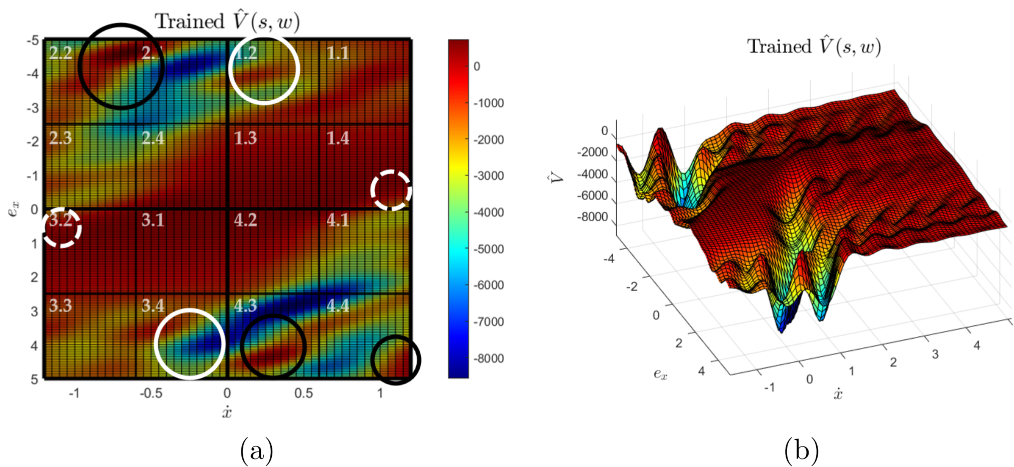


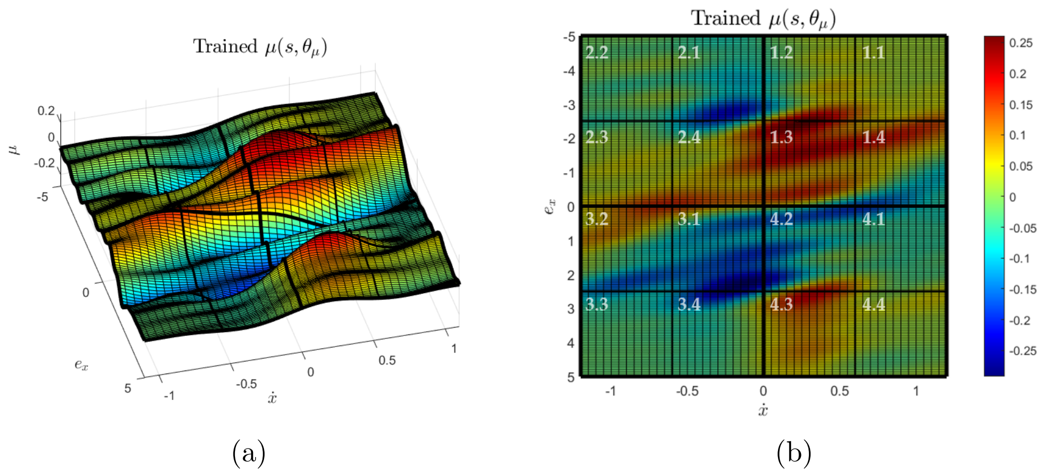
Figure 6.18: (a) Figure illustrating the areas of unexpected behaviour within the value function. The areas encircled in white show the complex learning capabilities of the RL agent and how it reflects basic intuitive reasoning. The enclosed areas in black indicate unexpected behaviour. (b) Figure showing 'ringing' behaviour occurring at the edges of low value updated states.

Although the general trend of the value function was predictable, Figure 6.18a shows areas which exhibited notable behaviour. The areas within the dashed white circles show decreased value areas within the region even though the aircraft is moving towards the goal. This is because the aircraft is in close proximity to the goal and moving towards it very fast which requires the quadcopter to react to stabilise at the goal, resulting in extended time away from

the goal waypoint which accumulates more negative rewards than if the quadcopter was slowing down on the approach. The portions enclosed in solid white circles indicate areas of lower value even though the quadcopter is moving towards the the goal. This is because the velocities in these areas are very low with values of $0 < |\dot{x}| < 0.4$m/s while the error is has high values of $3 < e_x < 5$m, resulting in a slow approach to the goal in areas where the reward is highly negative, adversely impacting the return of the episode.

The areas within the black circles indicate completely unexpected behaviour. These areas were expected to have the lowest value in the entire state space as it is the region in which the quadcopter is the furthest away from the goal and moving the fastest away from it. Instead, these areas have the some of the highest values in the entire state space. On reviewing the data it was determined that these areas were rarely visited by the quadcopter resulting in an unupdated portion of the state space. One would expect an unvisited state within the state space to have a value of zero as that is what the value function was initialised to at the start of training. However, due to the ringing characteristic of the Fourier basis, the large decrease in value of the neighbouring states in sub-quadrants 2.1, 4.3 and 4.4 caused a 'ringing' to occur, as shown in Figure 6.18b, consequently increasing the values of the circled states even though they were not visited. The test bed is a controlled space with minimal disturbance meaning that the states visited in simulation are likely going to be the states that the quadcopter will visit in the physical experiment. Since the RL agent was trained sufficiently in simulation it will not move away from the goal. Thus these states not being updated does not pose a problem to the experiment.

It is also important to note that the value function is the most established in the sub-quadrants where $0 < |e_x| < 2.5$m. This is because the simulation was conducted with the purpose of training the quadcopter for an error between -2.5m and 2.5m. Hence, as the training continues the quadcopter will stop veering away from the desired waypoint and stops visiting further away states. The lack of updates in the far away states is more apparent when examining the policy's mean $\mu(s, \boldsymbol{\theta}_m u)$.



(a)

(b)

Figure 6.19: Figure showing the trained policy's mean μ split into the quadrants and sub-quadrants defined in Figure 6.16 from (a) an isometric view. (b) a top view.

Policy's Mean Results Figure 6.19 shows the trained policy's mean μ split into the sub-quadrants described in Figure 6.16. Due to the formulation of the model, positive pitch ($\theta > 0$) leads to a positive velocity ($\dot{x} > 0$), and negative roll ($\theta < 0$) leads to negative velocity ($\dot{x} < 0$).

Since opposing signs of error and velocity lead the quadcopter to move towards the waypoint, it was expected that the mean μ would also have opposing signs to the error to direct the quadcopter to the desired waypoint.

For the most part, the majority of the positive pitch commands lie in the first and second quadrant which are associated with errors $e_x < 0$, whereas most of the negative commands are within the third and fourth quadrant where $e_x > 0$ as expected. There are large regions in which the policy did not get updated, specifically towards the edges of the state space where both e_x and $\dot{x}$ are high. This can be seen in Figure 6.20b where the edges of the surface can be seen tapering off to zero, indicating that the states towards the edges of the state space have not been visited enough such that an appropriate mean value is chosen. There are areas within the state space, namely in sub-quadrants 2.1 and 4.3 where the policy's mean had matching signs to the error $sgn(\mu) = sgn(e_x)$ which is opposite to what was expected. These are some of the areas within the policy's mapping of the mean which exhibited behaviours which were not anticipated.

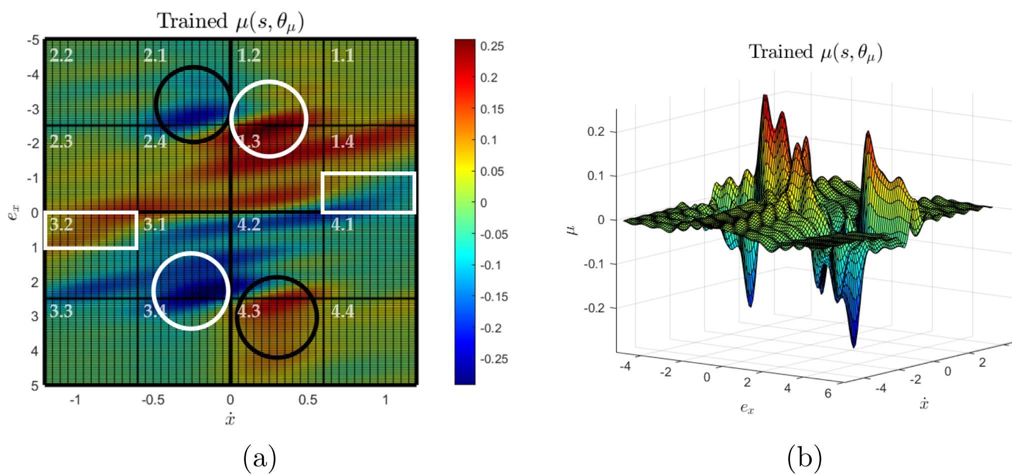


Figure 6.20: (a) Figure illustrating the areas of unexpected behaviour within the action selection. (b) Figure showing a clearer view of the unupdated portion of the action selection space.

Figure 6.20a highlights interesting action choices made by the policy. The areas enclosed within the white rectangles correspond to areas of μ commands opposite to what was expected due to an aggressive approach. As previously stated, it was expected that e_x and μ will posses opposing signs along the state space. However, in the areas enclosed by the rectangles, the policy commands a pitch with the same sign as the error $sgn(\mu) = sgn(e_x)$. Within these areas, the quadcopter is within close proximity to the goal $|e_x| < 1$m and the velocity in which the quadcopter is closing in is high $|\dot{x}| > 0.6$m/s. Hence, to rapidly slow down the quadcopter the policy commands a pitch θ in the opposite direction. Note that these areas were also the ones associated with lower values in the value function indicating that these states are less desirable than slowing down for the approach.

The states encompassed within the white circles are associated with aggressive action choices. These occur close to edge of the chosen training error range of -2.5 and 2.5m when the quadcopter is moving at a velocity $|\dot{x}| < 0.4$. These aggressive actions prompt the quadcopter to move towards the goal quickly. Although these types of aggressive actions would be expected in the sub-quadrants along the edges of the state space where the aircraft is far away from the goal, as alluded to previously, the quadcopter did not visit these states often enough to

adequately update the policy for the entirety of the state space resulting in extremely low roll angle commands in those regions. However, it does visit the region of $e_x = [-2.5, 2.5]$m and $\dot{x} = [-1.2, 1.2]$m/s in the first and third quadrants frequently which is why the policy is well defined in those areas.

The action choices shown within the black circles are completely opposite of what is expected within the area. They are located in the high error regions just past the training bounds of $|e_x| > 2.5$m with low velocity $|\dot{x}| < 0.6$. As stated previously, from examining the data, the quadcopter rarely finds itself in the region of the state space in which is it moving away from the goal $sgn(e_x) = sgn(\dot{x})$. This is because the training always starts with a position and velocity of zero $[x, \dot{x}] = [0, 0]$, which means unless the aircraft makes an active effort to move away from the goal it will stay operating withing the first and third quadrants. Additionally, the regions in question are outside of the error training bounds. That means, that for this experiment these commands will not come into play.

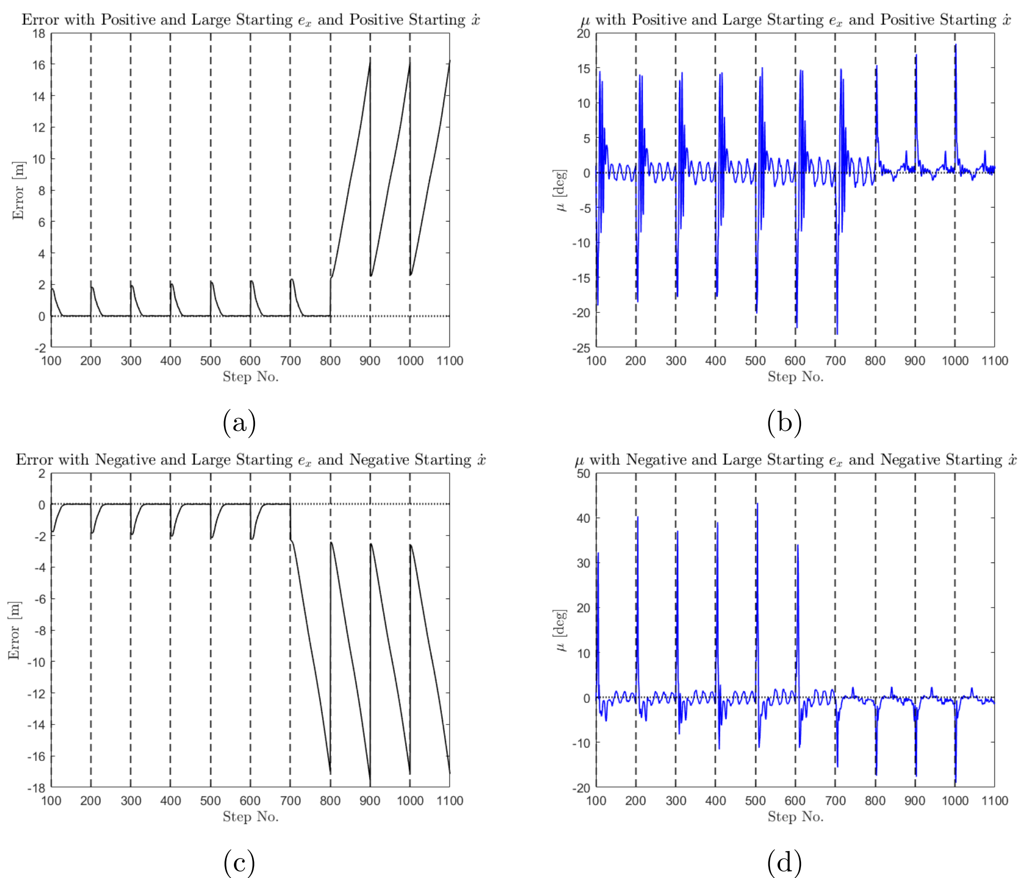


Figure 6.21: (a) Figure showing the error and action selection when starting at (a,b) positive e_x and positive $\dot{x}$. (c,d) negative e_x and negative $\dot{x}$.

Figure 6.21 shows how the quadcopter behaved when it was prompted to start its episodes from states within the regions encircled in black in Figure 6.20 with the goal values x_{goal} increasing with every episode. As expected, when forced to start within the circled region, the RL agent loses control of the quadcopter. This is because the quadcopter did not visit this region enough to achieve an accurate updated command angle choice even after 30,000 episodes of training. Again, if a state is not visited it would be expected that μ would have a value close to zero. However, due to the Fourier basis VFA method any region with strong jumps around it will

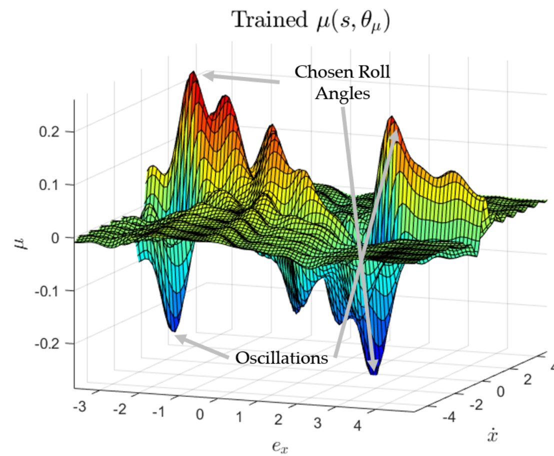


Figure 6.22: Figure illustrating the oscillatory behavior present in the action selection due to strong jumps in action choices paired with the Fourier basis approximation method.

encounter resultant oscillatory behaviour. Looking back at Figure 6.20a, the white circled regions have aggressive jumps which result in the Fourier 'ringing' which adversely effected the regions encompassed in the black circles. Since the states within the black circles are not visited often enough to remedy this, the final result has this region of invalid action choices which can clearly be seen in Figure 6.22. This region of unfavourable action choices had no effect throughout the entire duration of the training process and hence will not effect the performance of the quadcopter once the vectors are transferred to the physical aircraft in an area with no disturbances similar to the simulated training environment.

6.2 Real Quadcopter Waypoint Tracking Results

The training algorithm in Section 6.1.4 was coded in Python and uploaded to the Raspberry Pi on board the quadcopter. The only difference in the algorithms is that the one uploaded to the quadcopter does not require the model as it receives the position and velocity data $[x, y, \dot{x}, \dot{y}]$ directly from the OptiTrack system. This meant that even on the actual quadcopter, the controller continued learning using the $\boldsymbol{w}$ and $\boldsymbol{\theta}_\mu$ vectors trained in simulation as a starting point. This enables the agent to resolve model uncertainties and improve overall performance as the quadcopter continues to fly.

Since the quadcopter was symmetric, the controller could be applied to either the roll or pitch axis and should be able to perform as expected. To test this, the following four-part experiment was designed:

At the start of every test the quadcopter starts on the ground at $[0, 0, 0]$. Upon start up, the quadcopter runs through its initialisation sequence where the Raspberry Pi establishes connection with the Navio2 and sets up the required ArduPilot parameters before taking off to a position of $[x, y, z] = [0, 0, 2]$ and attitude of $[0, 0, -\frac{\pi}{2}]$ (NED) where it starts the RL controller. In this testing each episode was set to be 50 steps which equates to 5 seconds.

The controller program prompts the user for goals on the $x_M(= -y_Q)$ or $y_M(= x_Q)$ axis. This goal is then converted to the quadcopter's x_Q or y_Q axis accordingly to be used as the goal for the controller. Note that each test was performed six times in order to ensure repeatability of the experiment. Refer to Figure 5.9 for the coordinate systems.

- Tests with one agent: The controller is only applied to Motive's x_M axis, only generating a ϕ command. The desired attitude is sent through MAVLink as $[\phi_{desired}, \theta_{desired}, \psi_{desired}] = \left[-\phi_{RL}, 0, -\frac{\pi}{2}\right]$.
  - Test 1: The UAV starts and is returned to $[x, y, z] = [0, 0, 2]$ at the end of every episode where it is given a new x_M goal in order to replicate the training conducted in MATLAB and to mitigate any drift which may have occurred from having an uncontrolled y_M axis. This test is meant to replicate the exact planar training conducted in MATLAB.
- Tests with two agents: Two separate controllers are applied to Motive's x_M and y_M axes, generating ϕ and θ commands. The desired attitude is sent through MAVLink as $[\phi_{desired}, \theta_{desired}, \psi_{desired}] = \left[-\phi_{RL}, -\theta_{RL}, -\frac{\pi}{2}\right]$.
  - Test 2: The UAV starts and is returned to $[x, y, z] = [0, 0, 2]$ at the end of every episode where it is given a new x_M goal and the y_M goal is always set to zero in order to mitigate the drift. This test determines whether dual axis control is viable.
  - Test 3: The UAV starts at $[x, y, z] = [0, 0, 2]$ where it is given its first x_M goal and the y_M goal is always set to zero. At the end of each episode the UAV prompts the user for a new x_M goal without returning to the center. This test determines whether the controller is able to achieve one-dimensional point-to-point waypoint tracking.
  - Test 4: The UAV starts at $[x, y, z] = [0, 0, 2]$ where it is given its first (x_M, y_M) goal. At the end of each episode the UAV prompts the user for a new (x_M, y_M) goal without returning to the center. This test determines whether the controller is able to achieve full two-dimensional point-to-point waypoint tracking in the $X - Y$ plane.

Real life testing is susceptible to sources of error not encountered in simulation. One of the most frequently encountered sources of error is the calibration error. The IMUs on board the Navio2 were calibrated prior to testing by following the calibration procedure on the Mission Planner GCS. Any small movements during the IMU's calibration process can result in an offset in angle measurements causing the aircraft to incorrectly assume its attitude is at a certain angle. When an aircraft is perfectly calibrated it will remain in place if both ϕ and θ are zero. However any small offset will cause the aircraft to drift even if the IMU is measuring $[\phi, \theta] = [0, 0]$.

Another possible source of error is in OpiTrack's position data. The quadcopter had nine reflectors placed on it that were captured by the OptiTrack cameras. The Motive program was set to keep sending position data even if OptiTrack has lost track of up to five of these reflectors, reducing the accuracy.

Thirdly, the simulation did not take aerodynamic effects into account. Any drag or propeller down-wash acting on the payload causing excess swinging motion has not been accounted for. This unexpected swinging could affect the ability of the agent to perform waypoint tracking with the same accuracy shown in simulation.

Finally, the last source of error is related to the tuning of the PIDs and the replication of the input signal. The auto-tuning process was difficult to complete and required frequent pilot intervention to prevent crashes. Although manual testing was conducted to ensure smooth operation, the data retrieved from the quadcopter for replication of the input signal to the PID and response of the quadcopter was only for the roll commands and not the pitch commands of the aircraft. This is because it was assumed that the auto-tune function would tune both axes to the same performance ability. Hence, the controller was trained assuming the roll behaviour of the aircraft is applicable to both axes. However, if there are discrepancies between the behaviour of the roll and pitch PID controllers the RL controller might not be able to perform to the same ability on the quadcopter's pitch axis.

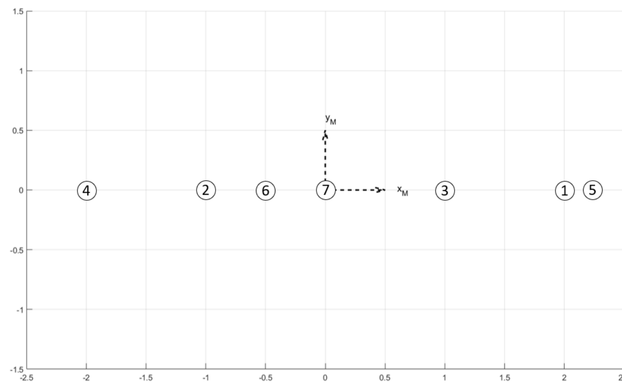


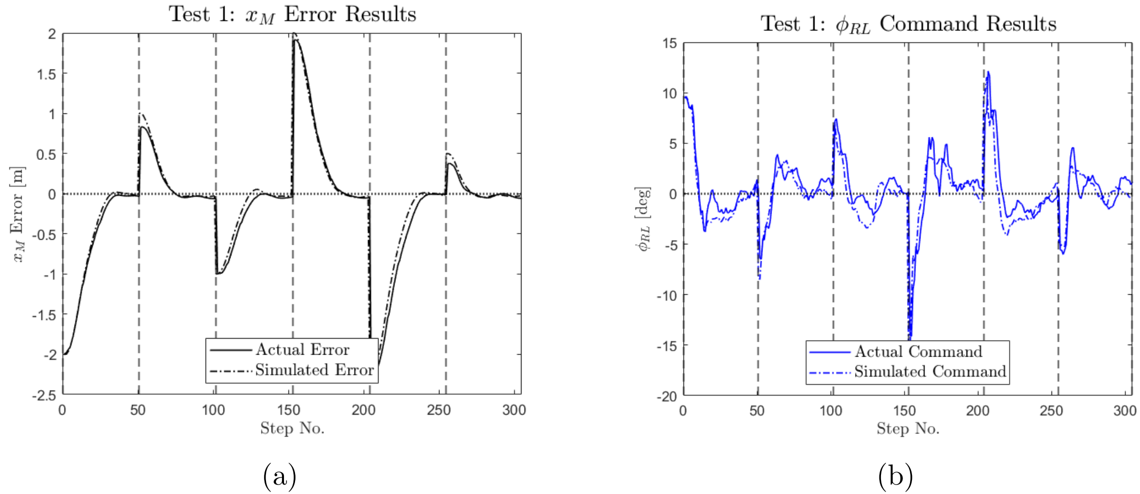
Figure 6.23: Figure illustrating the x_M goals chosen for Tests 1 and 2.

Test 1

The controller program prompts the user for a goal on the $x_M(= -y_Q)$ axis. Since the RL agent in this test is only applied to the y_Q axis, only ϕ commands will be generated. Therefore the commands sent to ArduPilot took the form of $\left[-\phi_{RL}, 0, -\frac{\pi}{2}\right]$. This leaves the y_M axis uncontrolled and prone to drift due to calibration error.

After completing an episode, the quadcopter is prompted to return to $[0, 0, 2]$ using ArduPilot's

waypoint tracking feature, where the RL controller asks the user for another goal to commence the next episode. The chosen x_M goals for this test were [2.00, −1.00, 1.00, −2.00, 2.25, −0.50] as shown in Figure 6.23.



(a) (b)

Figure 6.24: Figure comparing the actual and simulated performance of the quadcopter in Test 1 (a) Error in the $x_M(= -y_Q)$ axis. (b) ϕ command produced by the controller.

The test was conducted on the actual quadcopter and in simulation. The results are presented in Figure 6.24. It is apparent from Figure 6.24a that the quadcopter was able to successfully achieve full control in the x_M axis by driving the error down in every episode. Additionally, the simulation closely matched the actual quadcopter's performance in both the position and roll angle ϕ generation, validating that controller training on a simulation can be transferred to the actual system and achieve highly comparable performance.

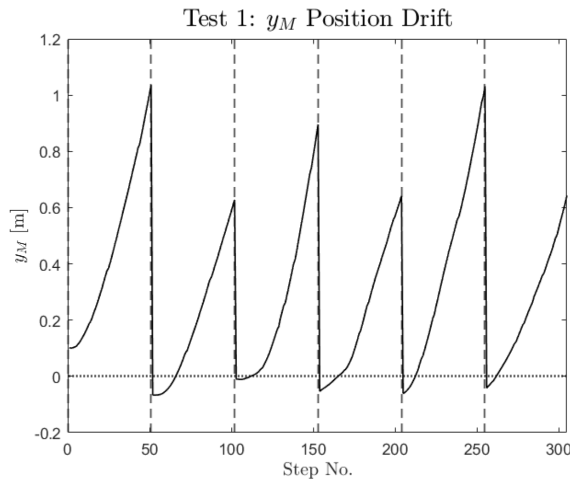


Figure 6.25: Figure showing the drift of the quadcopter in the y_M axis in Test 1.

As predicted, since the controller was not applied to the y_M axis, the quadcopter demonstrated notable drift in that axis as shown in Figure 6.25. This indicates that there was a calibration error in the quadcopter's IMU measurement of θ which lead it to measure $\theta = 0$ while $\theta > 0$ causing drift in the positive y_M axis direction.

Test 1 was repeated using a simple PID controller to compare with the RL controller, as shown in Figure 6.26. Although both controllers exhibit comparable approaches, the RL controller

shows superior performance at ranges closer to the goal. As the quadcopter attempts to come to a stop at the goal, the RL controller is able to handle the dynamics of the swinging payload whereas the PID is unable to do so because of its static parameters. In the first and fifth episodes, the PID error overshoots zero while the RL controller holds the quadcopter at the goal. In the last episode, the error from the PID controller exhibits a jump due to excessive swinging of the payload; the RL controller is able to handle and mitigate these disturbances. The error achieved by the RL controller consistently exhibits a steady state error 0.026m away from the goal. This error along with its possible mitigation is discussed in Test 2. The PID controller exhibits final error values ranging between -0.17m and 0.05m. At the end of the episode, the RL approach results in an average error that is 51% less than the error from the PID controller. Moreover, the quadcopter approach is more consistent and guarantees convergence to a known value.

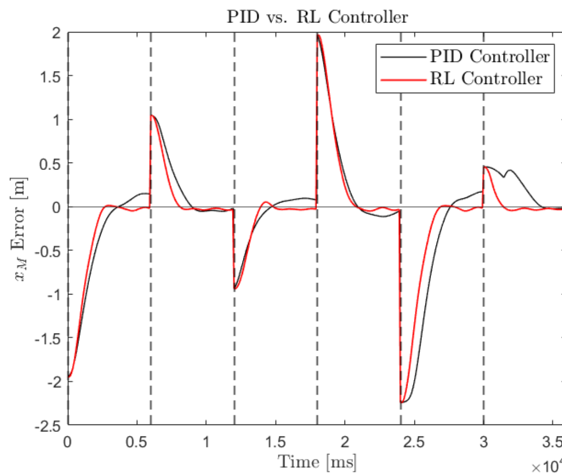


Figure 6.26: Figure comparing the a PID controller to the RL controller.

The next test aims to control this drift by applying the controller to both axes of the quadcopter.

Test 2

In Test 2, the program sets the y_M goal to be zero for all episodes in order to attempt to avoid the drift in the y_M axis that was present in the first test. The x_M goals for the second test were the same as those for the first test.

As anticipated, the results for the x_M axis for the second test, shown in Figure 6.27 match those for the first test. However, there is an immense improvement when it comes to the drift in the y_M axis as shown in Figure 6.28.

By applying the controller to the y_M axis, the quadcopter was able to hold itself consistently within 0.08m of the axis containing its drift. Upon closer examination of the x_M error plots, it was evident that the steady state error present in the y_M axis appears in the x_M axis as shown in Figure 6.29. It is interesting to note that in both cases the simulator also predicted a non-zero drift indicating the possible need for further training or adjustment of the step sizes α_V and α_μ. The steady state error of x_M had the same steady state error predicted by the simulator while the y_M axis' steady state error was 32% higher than what was predicted. This is linked to the difference between the pitch PID controller's performance and the roll PID which was used to train the RL controller.

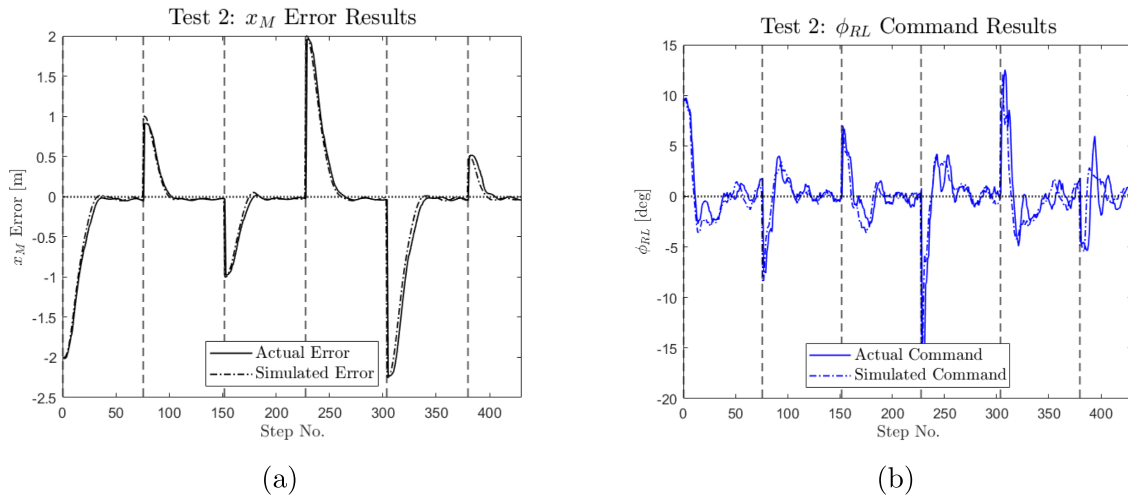


Figure 6.27: Figure comparing the actual and simulated performance of the quadcopter in Test 2 (a) Error in the $x_M(= -y_Q)$ axis. (b) ϕ command produced by the controller.

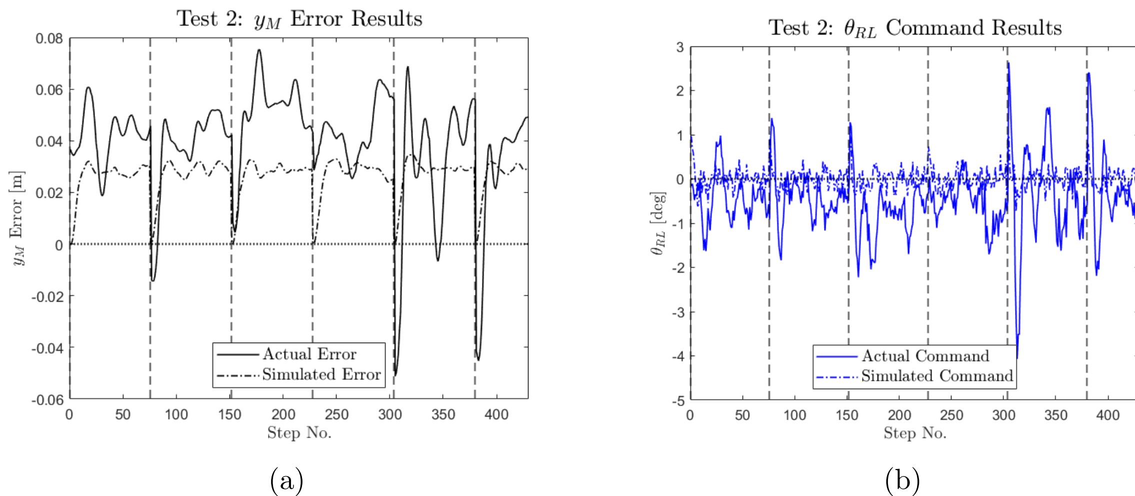


Figure 6.28: Figure comparing the actual and simulated performance of the quadcopter in Test 2 (a) Error in the $y_M(= x_Q)$ axis. (b) θ command produced by the controller.

This dissimilarity of the PID controllers causes the most notable difference between the quadcopter and simulation when controlling the y_M axis. The controller on board the quadcopter asked for higher and more fluctuating pitch inputs in order to avoid drifting as illustrated in Figure 6.28b. This again is because the controller was trained on the command response of ϕ and not θ meaning that the aggressiveness of the PID controller affects how the quadcopter will act when the RL controller sends a command. From Figure 6.28a one can see that the oscillations of the actual quadcopter are stronger than those predicted by the model. This indicates that the PID controller of θ is more aggressive, reaching the goal faster than the PID that the model was trained on. Thus the quadcopter will spend more time at the commanded θ making larger position changes than expected. Since the controller was not expecting the position to change as drastically, it will then attempt to remedy this by commanding a greater θ value in the opposite direction. This was an expected source of error since the quadcopter was trained for the dynamics of the roll PID and not the pitch. The next step was to determine how the controller will act when the quadcopter is not returned to a position of $[0, 0, 2]$ at the

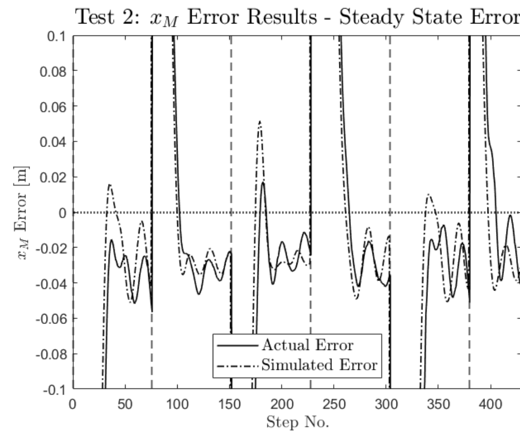


Figure 6.29: Figure illustrating the steady state error in the x_M axis of Test 2.

end of every episode.

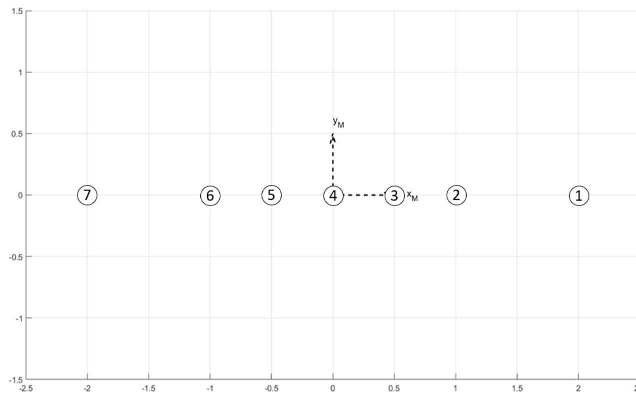


Figure 6.30: Figure illustrating the x_M goals chosen for Test 3.

Test 3

The third test was the first one to replicate a typical waypoint tracking algorithm in which the aircraft moves from one point to the next.

The x_M goals chosen for this test prompted the quadcopter to move in a straight line along the x_M axis by requesting the following goals [2.00, 1.00, 0.50, 0, −0.50, −1.00, −2.00] as shown in Figure 6.30.

Since the controller is error based, this test did not pose any problems to the controller. Rather, this test was conducted for safety to ensure that the controller is able to move from one point to the next without returning to the origin since the agent was trained to start from $[x, \dot{x}] = [0, 0]$. One can see from Figure 6.31 that the quadcopter was able to move from $x_M = 2.00$ to $x_M = -2.00$ by following waypoints without any undesirable behaviour. The results from the true system closely matched those generated by the simulator in both its waypoint tracking capabilities and commanded ϕ.

From Figure 6.32 similar results to the previous test in terms of holding the quadcopter to a position of $y_M = 0$ can be observed. Again, this discrepancy between the simulation and the

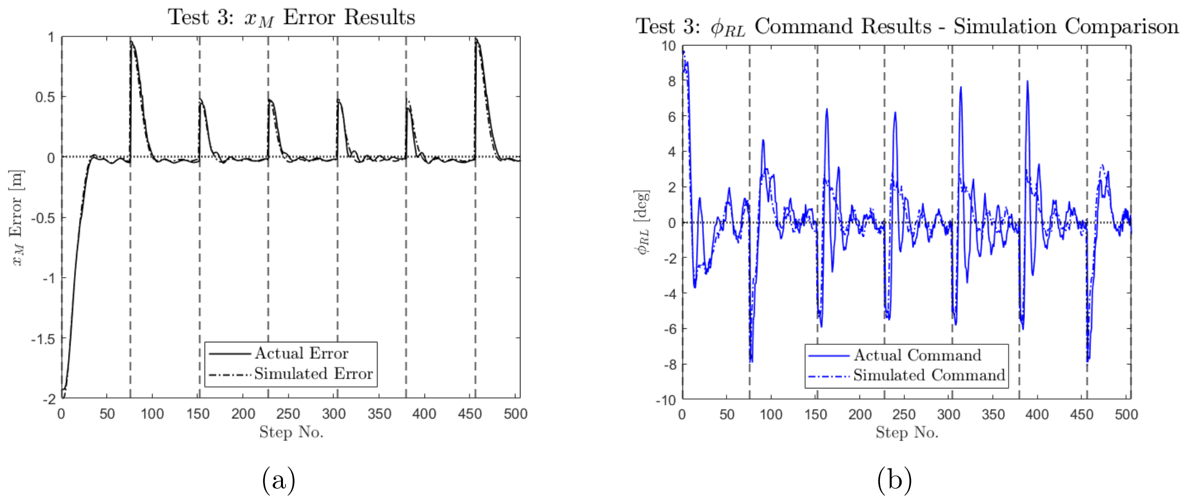


Figure 6.31: Figure comparing the actual and simulated performance of the quadcopter in Test 3 (a) Error in the $x_M(= y_Q)$ axis. (b) ϕ command produced by the controller.

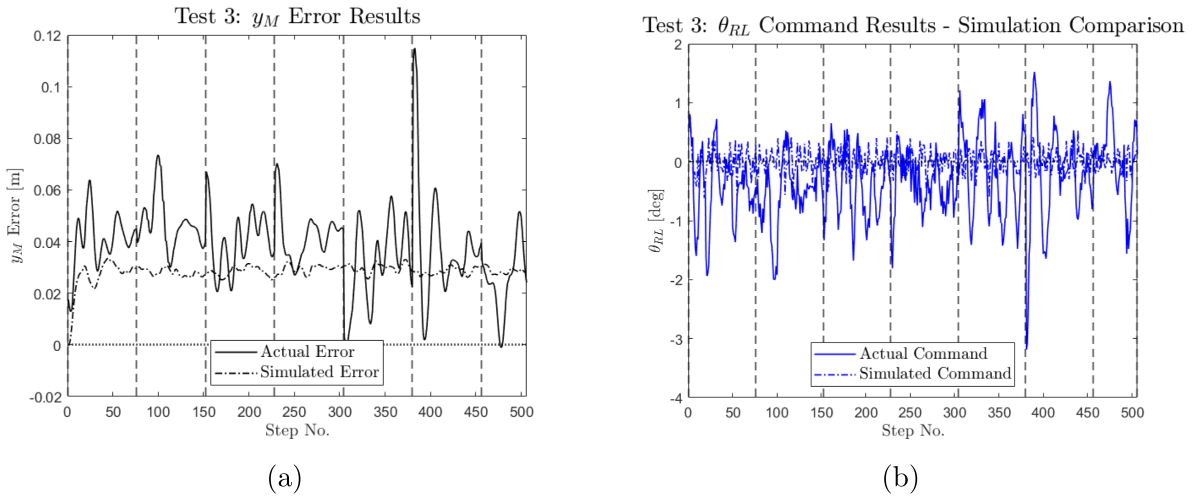


Figure 6.32: Figure comparing the actual and simulated performance of the quadcopter in Test 3 (a) Error in the $y_M(= x_Q)$ axis. (b) θ command produced by the controller.

actual aircraft is due to the the differences in the PID performance of the roll and pitch PID controllers.

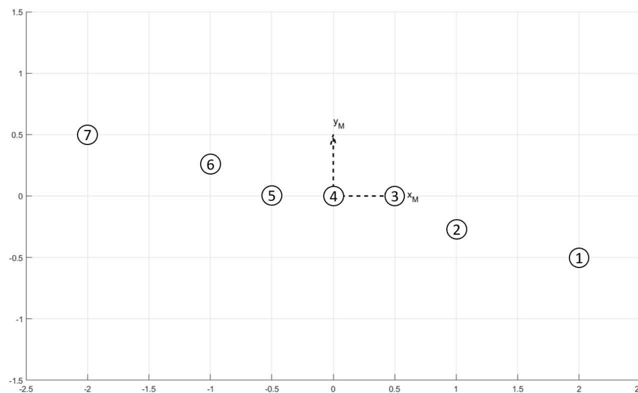


Figure 6.33: Figure illustrating the (x_M, y_M) goals chosen for Test 4.

Test 4

The fourth and final test took goal requests on both the x_M and y_M axes and did not return to the center after each episode.

The goals for the final test were chosen to be $[(2.00, -0.50), (1.00, -0.25), (0.50, 0.00), (0.00, 0.00), (-0.50), 0.00), (-1.00, 0.25), (-2.00, 0.50)]$ as illustrated in Figure 6.33.

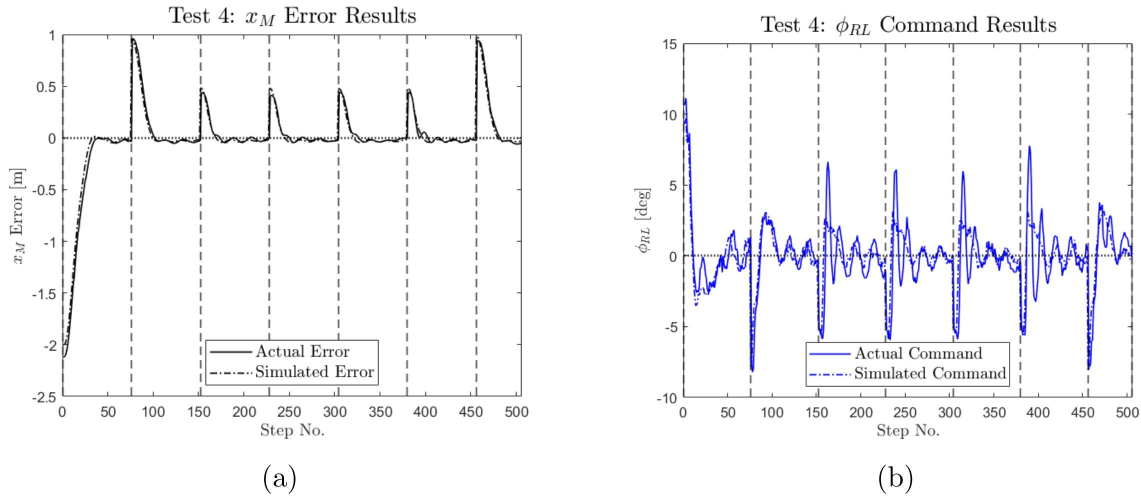


Figure 6.34: Figure comparing the actual and simulated performance of the quadcopter in Test 4 (a) Error in the $x_M(= -y_Q)$ axis. (b) ϕ command produced by the controller.

This test used the same x_M goals as the previous test yielding near identical results in the x_M axis.

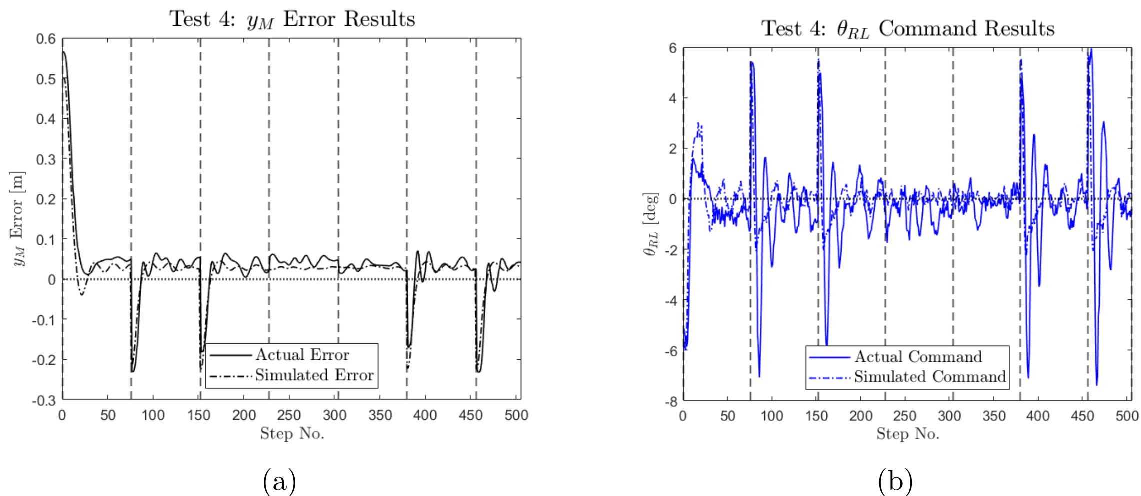


Figure 6.35: Figure comparing the actual and simulated performance of the quadcopter in Test 4 (a) Error in the $y_M(= x_Q)$ axis. (b) $-\theta$ command produced by the controller.

Figure 6.35 shows the y_M error and θ command results from the final test. The discrepancies in the model here are more pronounced as the steady state error is higher than predicted with the position's oscillations larger and more abrupt. The quadcopter was able to approach the goal with a mean steady state error of 0.035m which is significantly smaller than the starting errors thus fulfilling the purpose of the controller.

Although the RL controller implemented was successful at leading the quadcopter to the desired waypoint despite the perturbations of the payload, there was no analysis conducted on the controller to provide a stability guarantee. Stability of an RL algorithm cannot be guaranteed solely using data [64]. That being said, there have been recent research initiatives focused on establishing the stability of reinforcement learning control algorithms. The work of [64] uses Lyapunov's method as the critic in an actor-critic deep RL algorithm to ensure stability. In [65] the research focuses on utilising the $\mathcal{L}_2$ gain to apply a constraint on a neural network RL control algorithm to achieve a stability guarantee. Both of the mentioned works were successful at providing a stable RL algorithm, proving that reinforcement learning algorithms have the capability of achieving stability in terms of classical control criterion with the implementation of the appropriate analysis methodology.

Chapter 7

Future Improvements and Conclusions

In this chapter, the concluding segment encapsulates the essence of the control algorithm and presents a brief summary of the achieved results. It serves as a comprehensive wrap-up, synthesizing the key findings and paving the way for future developments in the realm of slung payload quadcopter control using Reinforcement Learning.

Attention is then turned toward potential avenues for enhancing the research presented in the thesis and finalises the outcomes of its work. A critical examination of the study's limitations is undertaken, and viable strategies for addressing these shortcomings are proposed. The suggested improvements span both the RL algorithm employed in the experiment and the hardware utilized for its execution.

7.1 Conclusions

The research described in this thesis aimed to achieve full waypoint control of a quadcopter carrying a slung payload in the $X-Y$ plane. The work began by using the Euler-Lagrange formulation to derive a planar model of the quadcopter-load system in the $X-Z$ plane. A TD learning RL algorithm capable of handling infinite state and action spaces was developed with the objective of leading the quadcopter to a specified waypoint in the $X-Z$ plane. The algorithm was then trained in MATLAB using the planar model until the desired waypoint tracking behaviour was exhibited in simulation. Due to the symmetric nature of the quadcopter, a trained RL agent on the $X-Z$ axis could also be used to achieve control on the $Y-Z$ axis. The trained weights and parameters associated with the value function and policy were ported from the simulated training to the real quadcopter system where training continued during testing to mitigate model uncertainty.

A number of tests were curated to determine the ability of the RL controller to achieve the goal objective in both the $X-Z$ and $Y-Z$ planes of the quadcopter. Testing illustrated that training the RL agent on the model of the system was a viable approach as the real quadcopter's performance closely matched simulated results. Furthermore, when compared to a standard PID controller the RL controller exhibited more reliable results showing that the agent has learned to expect the dynamics induced by the slung load.

Although the RL algorithm employed was able to successfully complete the required task, there are drawbacks that are important to note. RL algorithms are sensitive to their hyperparameters. This implies that these parameters, which are tedious to tune especially when there is no value to commence from, must be re-examined with any changes in the system or the environment in which it operates. Secondly, the ethical and safety concerns presented by implementing RL algorithms on robotic systems are significantly higher than other nonlinear control techniques. This is because stability guarantees of an RL controller are more difficult to obtain than a conventional controller. Finally, RL controllers can be prone to sample bias if the policy does not allow for enough exploration of certain areas of the state space. This means that although an RL controller can perform adequately in certain areas of the state space, this assumption cannot be imposed on the rest of the area.

7.2 Future Improvements

Overall, the RL controller was successful in achieving the goal of leading the quadcopter to a specified waypoint in both the x_M and y_M axes. This demonstrates that the reinforcement learning algorithm trained on an imperfect dynamic model of the system can still perform adequately and adapt on the actual system. Furthermore, this experiment proves that utilising a reinforcement learning approach is computationally suitable for integration with a Raspberry Pi microcomputer to enable real-time learning on multiple axes which is difficult to achieve with other forms of machine learning.

Although the quadcopter-slung load waypoint tracking reinforcement learning controller was able to be trained in simulation and seamlessly transferred to the real system there is always room for improvement within the work. The first and most obvious improvements to this works would be to incorporate disturbances in the training such that the quadcopter would be able to maneuver in areas where interference in the environment exist. This can be done by adding wind vectors in specific states that push the quadcopter in certain directions or by incorporating

an adversary action agent like the one seen in [28].

This leads into the enhancement of the standard deviation σ of the policy. In this research the standard deviation was chosen to be a small constant value since the environment in which the quadcopter operated in had virtually no disturbances. However, in environments with disturbances each state will require a specific σ value to cope. This calls for a semi gradient ascent approach to σ much like the one utilised for the mean μ in this thesis. Not only does this aid in accounting for disturbance but it also converges to the appropriate σ value for each state in the environment. This means that instead of arbitrarily choosing some σ and having to adjust it according to the performance results as done in this work, the SGA on σ will achieve the appropriate value without requiring user intervention.

Another short coming of this work was that some portions of the state space were not updated correctly due to the quadcopter not visiting certain states frequently enough. The policy's mean μ exhibited some adverse behaviours in locations in the state space in which the quadcopter was moving away from the goal at low speeds. Upon investigation it was determined that this was due to the ringing nature of the Fourier basis approximation along with the states not being visited often. This was illustrated clearly in Figure 6.21. This occurred because the training started each episode with $[x, \dot{x}] = [0, 0]$ meaning that the quadcopter always started from neutral conditions, avoiding the states in which the quadcopter is moving away from the goal, leading those states to remain unvisited. To remedy this, much like the goal was chosen randomly for the start of each episode, the position and velocity of the quadcopter can also be randomly chosen at the start the episode increasing the chances of the quadcopter updating states it wouldn't have otherwise.

Although the controller was able to control the position of the quadcopter, it had no knowledge of the dynamics of the payload since they were not included in the state space. This means that the payload was allowed to move freely as the quadcopter chose the quickest path to the goal. In fragile load delivery applications, where sudden movements of the payload are unfavourable, this would not be ideal. To address this, the payload's swing angles $[\alpha, \beta]$ and angular velocities $\left[\dot{\alpha}, \dot{\beta}\right]$ can be added to the state space of the controller. This enables α to be added to the reward function such that swing angles are penalised much like the current reward function penalises position error values. The addition of the the swing angles' states to the state space increases the computational complexity of the controller, increasing the training time and execution time in real-life applications. This must be taken into account either by increasing the RL time step to allow the agent more time to conduct its computations or by altering the VFA method from Fourier basis to a more time efficient method like tile-coding which only updates the relevant weights.

This leads into the reward function itself. In this work, not much has been done to optimise the reward function or the step sizes α_V and α_μ. These choices were made heuristically based on repeated testing and analysis of the performance of the quadcopter. That being said, this work could benefit greatly from the implementation of exact optimization methods over heuristic ones since those would guarantee the selection of optimal step sizes and reward function constants for the problem at hand. Additionally, the reward function itself can be explored to examine what adding different states to the reward function can achieve. An example of this could be adding the sign of velocity such that the quadcopter is penalised if it is moving away from the goal and rewarded if it is moving towards it.

Finally to truly test the feasibility of this controller in real life delivery applications, a beneficial

next step would be to remove the need for a motion capture system all together and utilise the GPS in the autopilot to retrieve position and velocity data for the controller. The Navio2 features a NEO-M8N GPS module with ±0.5m/s velocity accuracy and ±2.0m position accuracy [66]. This means that the quadcopter will not be able to perform as well as it has when using a motion capture system. That being said, additional component such at a GPS antenna can be added to help improve the position accuracy. This improvement requires for the quadcopter to be flown outside hence calling for additional security measures to be addressed. This includes finding an appropriate airspace to test in as well as filing any required Transport Canada documents regarding the quadcopter,

References

[1] S. Lund, A. Madgavkar, J. Manyika, S. Smit, and K. E. O. Robinson, "The future of work after COVID-19." https://www.mckinsey.com/featured-insights/future-of-work/the-future-of-work-after-covid-19, Feb. 2021.

[2] S. Zanzana and J. Martin, "Retail e-commerce and COVID-19: How online sales evolved as in-person shopping resumed." https://www150.statcan.gc.ca/n1/en/catalogue/11-621-M2023002, Feb. 2023.

[3] M. Ghajargar, G. Zenezini, and T. Montanaro, "Home delivery services: innovations and emerging needs," *IFAC-PapersOnLine*, vol. 49, no. 12, pp. 1371–1376, 2016. 8th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2016.

[4] Shipping and F. Resource, "7 challenges in last-mile delivery and steps to address them." https://www.shippingandfreightresource.com/7-challenges-in-last-mile-delivery-and-steps-to-address-them/, Oct. 2022.

[5] M. Misaros, O.-P. Stan, I.-C. Donca, and L.-C. Miclea, "Autonomous Robots for Services-State of the Art, Challenges, and Research Areas," *Sensors*, vol. 23, no. 10, 2023.

[6] S. K. I. Graham, "Development of a Quadrotor Slung Payload System," Master's thesis, Graduate Department of Institute for Aerospace Studies - University of Toronto, Toronto, ON, Canada, 2019.

[7] N. T. K. Chi, L. T. Phong, and N. T. Hanh, "The drone delivery services: An innovative application in an emerging economy," *The Asian Journal of Shipping and Logistics*, vol. 39, no. 2, pp. 39–45, 2023.

[8] FedEx, "FedEx Plans to Test Autonomous Drone Cargo Delivery with Elroy." https://newsroom.fedex.com/newsroom/global/elroyair, Mar. 2022.

[9] J. J. Craig, *Introduction to Robotics - Mechanics and Control.* Upper Saddle River, NJ, USA: Pearson Education, Inc., third ed., 2005.

[10] A. Sulficar, H. Suresh, A. Varma, and A. Radhakrishnan, "MODELING, SIMULATION AND COMPLETE CONTROL OF A QUADCOPTER.," tech. rep., Department of Mechanical Engineering -National Institute of Technology Karnatka Surathkal, Mangalore, India, May 2017.

[11] H. Elkholy and M. Habib, *Dynamic Modeling and Control Techniques for a Quadrotor.* 01 2015.

[12] L. Qian, S. Graham, and H. H.-T. Liu, "Guidance and Control Law Design for a Slung Payload in Autonomous Landing: A Drone Delivery Case Study," *IEEE/ASME Transactions on Mechatronics*, vol. 25, no. 4, pp. 1773–1782, 2020.

[13] A. Gautam, P. Sujit, and S. Saripalli, "Application of guidance laws to quadrotor landing," in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 372–379, 2015.

[14] A. Packard, K. Poolla, and R. Horowitz, "Dynamic Systems and Feedback Class Notes for ME 132- 19 Jacobian Linearizations, equilibrium points," Lecture 19, University of California, Berkeley CA, USA, 2002.

[15] S. Dai, T. Lee, and D. Bernstein, "Adaptive Control of a Quadrotor UAV Transporting a Cable-Suspended Load with Unknown Mass," vol. 2015, 12 2014.

[16] I. D. Landau, R. Lozano, M. M'Saad, and A. Karimi, *Adaptive Control: Algorithms, Analysis and Applications*, pp. 1–33. London: Springer London, 2011.

[17] Y. Rahman, A. Xie, J. B. Hoagg, and D. S. Bernstein, "A tutorial and overview of retrospective cost adaptive control," in *2016 American Control Conference (ACC)*, pp. 3386–3409, 2016.

[18] M. Guerrero-Sánchez, R. Lozano, P. Castillo, O. Hernández-González, C. García-Beltrán, and G. Valencia-Palomo, "Nonlinear control strategies for a UAV carrying a load with swing attenuation," *Applied Mathematical Modelling*, vol. 91, pp. 709–722, 2021.

[19] F. Garces, V. M. Becerra, C. Kambhampati, and K. Warwick, *Strategies for Feedback Linearisation: A Dynamic Neural Network Approach*, pp. 27–60. London, UK: Springer London, 2003.

[20] A. R. Godbole and K. Subbarao, "Nonlinear control of unmanned aerial vehicles with cable suspended payloads," *Aerospace Science and Technology*, vol. 93, p. 105299, 2019.

[21] I. H. B. Pizetta, A. S. Brandão, and M. Sarcinelli-Filho, "Modelling and control of a PVTOL quadrotor carrying a suspended load," in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 444–450, 2015.

[22] K. Hauser, "Robotic Systems," Lecture Notes 17, University of Illinois at Urbana-Champaign, Champaign, IL, United States.

[23] K. Hauser, "ECE410F – Control Systems: Chapter 6 LINEAR QUADRATIC OPTIMAL CONTROL," Lecture Notes 6, University of Toronto, Toronto, ON, Canada, 2008.

[24] R. P. K. Jain, "Transportation of Cable Suspended Load using Unmanned Aerial Vehicles-A Real-time Model Predictive Control Approach," Master's thesis, Delft Center for Systems and Control - Delft University of Technology, Delft, Netherlands, Aug. 2015.

[25] K. Us, A. Cevher, M. Sever, and A. Kirli, "On the Effect of Slung Load on Quadrotor Performance," *Procedia Computer Science*, vol. 158, pp. 346–354, 01 2019.

[26] L. Grüne, *Handbook of Model Predictive Control*, pp. 29–52. Cham: Springer International Publishing, 2019.

[27] S. Belkhale, R. Li, G. Kahn, R. McAllister, R. Calandra, and S. Levine, "Model-Based Meta-Reinforcement Learning for Flight with Suspended Payloads," *CoRR*, vol. abs/2004.11345, 2020.

[28] A. M. Deshpande, A. A. Minai, and M. Kumar, "Robust Deep Reinforcement Learning for Quadcopter Control," *CoRR*, vol. abs/2111.03915, 2021.

[29] Z. Jiang and A. F. Lynch, "Quadrotor motion control using deep reinforcement learning," *Journal of Unmanned Vehicle Systems*, vol. 9, no. 4, pp. 234–251, 2021.

[30] H. Hua, Y. Fang, X. Zhang, and C. Qian, "A New Nonlinear Control Strategy Embedded with Reinforcement Learning for a Multirotor Transporting a Suspended Payload," *IEEE/ASME Transactions on Mechatronics*, vol. 27, no. 2, pp. 1174–1184, 2022.

[31] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter, "Control of a Quadrotor with Reinforcement Learning," *CoRR*, vol. abs/1707.05110, 2017.

[32] F. L. Lewis, D. M. Dawson, and C. T. Abdallah, *Robot Manipulator Control: Theory and Practice*, pp. 107–124. Pearson, Jan. 2000.

[33] S. P. Ellner and J. Guckenheimer, *Dynamic Models in Biology*. Princeton, New Jersey, USA: Princeton University Press, Apr. 2006.

[34] S. Dong, Z. Yuan, and F. Zhang, "A Simplified Method for Dynamic Equation of Robot in Generalized Coordinate System," *Journal of Physics: Conference Series*, vol. 1345, 2019.

[35] S. Musa, "Techniques for Quadcopter Modelling & design: A review," vol. 5, May 2018.

[36] Z. Hussain and N. Z. Azlan, "Kane's Method for Dynamic Modeling," *IEEE International Conference on Automatic Control and Intelligent Systems*, Oct. 2016.

[37] E. Jarzebowska, "Modeling and control design using the Boltzmann-Hamel equations: A roller-racer example," vol. 39, pp. 236–241, 09 2006.

[38] B. Friedland, *Control System Design - An Introduction to State Space Methods*. Mineola, N.Y, USA: Dover Publications, Inc., 1986.

[39] A. S. Dimova, K. Y. Kotov, and A. S. Maltsev, "Trajectory control of a quadrotor carrying a cable-suspended load," in *2020 24th International Conference on System Theory, Control and Computing (ICSTCC)*, pp. 501–505, 2020.

[40] M. Guo, D. Gu, W. Zha, X. Zhu, and Y. Su, "Controlling a Quadrotor Carrying a Cable-Suspended Load to Pass Through a Window," in *Journal of Intelligent & Robotic Systems)*, p. 387–401, 2020.

[41] A. A. Rezaei Lori, M. Danesh, P. Amiri, S. Y. Ashkoofaraz, and M. A. Azargoon, "Transportation of an Unknown Cable-Suspended Payload by a Quadrotor in Windy Environment under Aerodynamics Effects," in *2021 7th International Conference on Control, Instrumentation and Automation (ICCIA)*, pp. 1–6, 2021.

[42] M. W. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Dynamics and Control*. Wiley, second ed., 2004.

[43] D. Morin, *Introduction to Classical Mechanics: with Problems and Solutions*. Cambridge, UK: Cambridge University Press, 2008.

[44] P. Mann, *Lagrangian and Hamiltonian Dynamics*. Oxford University Press, June 2018.

[45] K. S. Fu, R. C. Gonzalez, and C. G. Lee, *Robotics: Control Sensing, Vision and Intelligence*. New York, NY, USA: McGraw-Hill College, July 1987.

[46] S. Sadr, S. Ali, A. Moosavian, and P. Zarafshan, "Dynamics Modeling and Control of a Quadrotor with Swing Load," 2014.

[47] PX4, "Flight controller selection." https://docs.px4.io/main/en/getting_started/flight_controller_selection.html.

[48] R. Niemiec and F. Gandhi, "A Comparison Between Quadrotor Flight Configurations," in *Proceedings of the 42nd European Rotorcraft Forum*, (Lille, France), 2016.

[49] A. Hussein and R. Abdalla, *Autopilot Design for a Quadcopter*. PhD thesis, 10 2017.

[50] J. Seddon, *Basic Helicopter Aerodynamics*. Oxford, UK: BSP Professional Books, 1990.

[51] H. Asada, "Introduction to Robotics," lecture notes, Massachusetts Institute of Technology, Boston, Massachusetts, USA, 2005.

[52] C. Lum, "Computing Euler Angles: The Euler Kinematical Equations." https://www.youtube.com/watch?v=9GZjtfYOXao&ab_channel=ChristopherLum, Apr. 2020. Video Series.

[53] S. K. Phang, K. Li, B. M. Chen, and T. H. Lee, *Handbook of Unmanned Aerial Vehicles*, pp. 181–206. Dordrecht: Springer Netherlands, 2015.

[54] E. Sabourin, "Rotorcraft Slung Payload Stabilization Using Reinforcement Learning," Master's thesis, University of Ottawa, Ottawa, Canada, 2023.

[55] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: The MIT Press, 2018.

[56] D. Silver, "Lecture 2: Markov Decision Processes," Lecture 2, DeepMind, 2015. https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf.

[57] E. Brunskill, "CS234: Reinforcement Learning Winter 2019," Lecture 2, Stanford University, Stanford, CA, USA, 2019. Lecture 2: Making Sequences of Good Decisions Given a Model of the World, https://web.stanford.edu/class/cs234/slides/lecture2_ns.pdf.

[58] ArduPilot, "ArduPilot." https://ardupilot.org/. Accessed: 2023-04-12.

[59] emlid, "Navio2." https://docs.emlid.com/navio2/. Accessed: 2023-01-26.

[60] OptiTrack, "NatNet SDK." https://optitrack.com/software/natnet-sdk/. Accessed: 2023-07-03.

[61] W. Stevens, B. Fenner, and A. Rudoff, *UNIX Network Programming: The sockets networking API*. No. v. 1 in Addison-Wesley professional computing series, Addison-Wesley, 2004.

[62] MAVLink, "MAVLink Common Message Set." https://mavlink.io/en/messages/common.html. Accessed: 2023-07-03.

[63] MAVLink, "MAVLink Developer Guide." https://mavlink.io/en/. Accessed: 2023-07-03.

[64] M. Han, L. Zhang, J. Wang, and W. Pan, "Actor-Critic Reinforcement Learning for Control With Stability Guarantee," *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 6217–6224, 2020.

[65] M. Jin and J. Lavaei, "Stability-Certified Reinforcement Learning: A Control-Theoretic Perspective," *IEEE Access*, vol. PP, pp. 1–1, 12 2020.

[66] u blox, "NEO-M8: u-blox M8 concurrent GNSS modules - Data sheet." https://content.u-blox.com/sites/default/files/NEO-M8-FW3_DataSheet_UBX-15031086.pdf. Accessed: 2023-09-10.

[67] K. Mehrotra, C. Mohan, and S. Preface, "Elements of Artificial Neural Nets," 01 1997.

[68] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*, ch. Chapter 16. Integration of Ordinary Differential Equations. USA: Cambridge University Press, 1992.

[69] W. J. Teahan, *Artificial Intelligence – Agents and Environments*. Ventus Publishing, 2010.

[70] D. L. Poole and A. K. Mackworth, *Artificial Intelligence: Foundations of Computational Agents*. Cambridge, United Kingdom: Cambridge University Press, second ed., 2017.

Appendix A

Actor Critic Neural Networks

Neural networks attempt to apply machine learning techniques through imitating the way in which the human brain processes information. The process involves simulating numerous interconnected processing units, called nodes, that are meant to resemble the neurons in the brain. These nodes are arranged in layers: an input layer, hidden layers, and an output layer. The most common NN architecture is feedforward networks in which every node in the ith layer is connected to every node in the $i+1^{th}$ layer by some connection whose strength is determined by a numerical value called the weight [67]. An illustration of a neural network feedforward structure can be seen in Figure A.1. Each node produces some numerical value called an activation which is obtained through an activation function using the all the weights, biases, and activation values of the preceding layer's nodes.

$$a_0^{(i+1)} = f\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \end{bmatrix} \begin{bmatrix} a_0^{(i)} \\ a_1^{(i)} \\ \vdots \\ a_n^{(i)} \end{bmatrix} + b_0^{(i)}\right)$$

$$a_0^{(i+1)} = f\left(\boldsymbol{W}_0 \boldsymbol{a}^{(i)} + \boldsymbol{b}\right)$$

where n is the number of nodes in the ith layer, $\boldsymbol{a}^{(i)}$ are activations of in layer i with associated signal strengths $\boldsymbol{W}_0$ and bias for inactivity $b_0^{(i)}$.

This activation function can take many forms depending on how the user chooses to define the type of neural network. Furthermore there are many weights to be tuned in a neural network such that the outcome layer provides a suitable result. These weights are tuned using gradient descent while propagating backwards through the layers updating the weights depending on the final output of the NN. In offline training, this is repeated until the neural network is behaving sufficiently and the policy is retrieved and applied to the system. Alternatively, in online training the neural network is allowed to keep training and updating the weights as the system collects data, continuously updating and improving the policy.

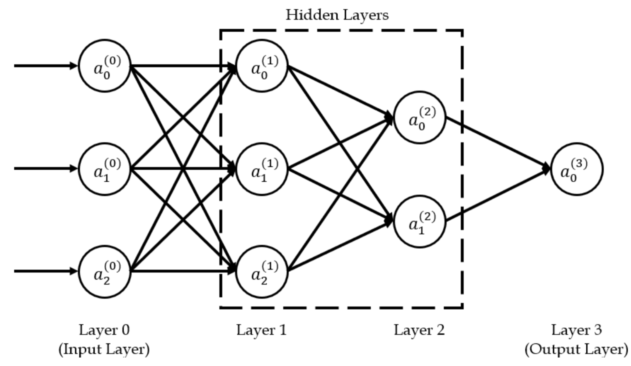


Figure A.1: The architecture of a feedforward neural network where each node of a layer is connected to every node in the succeeding layer.

Appendix B

Constructing The Rotation Matrix

To be able to track the orientation of the quadcopter, the quadcopter's rotation must be expressed in the inertial frame of reference since the body frame moves and rotates with the copter. To achieve this one must utilise the rotation matrix that will express the body frame orientation in terms of the inertial frame of reference.

The derivation begins with considering the positive rotations in terms of the frames. i.e. frame {B} and {I} start aligned in orientation. Then consider the quadcopter rolling or pitching or yawing resulting in the 1 frame from the original position at 0. This deviation from the original orientation is show below in Figure B.1.

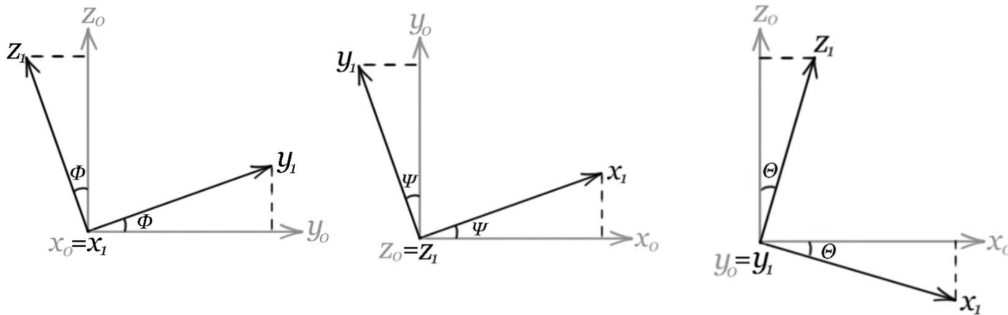


Figure B.1: Illustration of the rotations of the quadcopter frame

A 3-Dimensional rotation matrix is defined by the following equation [9]:

$$R = \begin{bmatrix} x_1 \cdot x_0 & y_1 \cdot x_0 & z_1 \cdot x_0 \\ x_1 \cdot y_0 & y_1 \cdot y_0 & z_1 \cdot y_0 \\ x_1 \cdot z_0 & y_1 \cdot z_0 & z_1 \cdot z_0 \end{bmatrix} \tag{B.1}$$

By applying Equation B.1 to the frames shown in Figure B.1, it can be seen that the rotation matrices for the roll about the x-axis, the pitch about the y-axis and the yaw about the z-axis are as follows:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} R_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{(B.2)}$$

The rotations of the aircraft are taken to be to be Z-Y-X Euler angles: the aircraft firstly yaws about the z-axis, followed by rolling about the new y-axis and finally pitching about the final x-axis. Using this information, a rotation matrix can be computed in order to express the final orientation of the quadcopter, or equivalently the body-fixed frame {B}, relative to the earth-fixed frame {I} using the following [9]:

$${}^I_BR = R_z(\psi)R_y(\theta)R_x(\phi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \quad \text{(B.3)}$$

$${}^I_BR = \begin{bmatrix} C_\psi C_\theta & C_\psi S_\theta S_\phi - S_\psi C_\phi & C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi C_\theta & S_\psi S_\theta S_\phi + C_\psi C_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi \\ -S_\theta & C_\theta S_\phi & C_\theta C_\phi \end{bmatrix} \quad \text{(B.4)}$$

Where C_x denotes $\cos x$ and S_x denotes $\sin x$.

It is also important to note that in order to describe orientations in the inertial frame in terms of the body frame the rotation matrix ${}^B_IR$ can be used where [9]:

$${}^B_IR = {}^I_BR^T \quad \text{(B.5)}$$

These rotation matrices can be used to convert coordinates from the body frame to the inertial frame and vice versa. Equation B.6 shows the the rotation matrix being utilised to transform a position vector measured in the body frame to the inertial frame.

$${}^I\boldsymbol{P} = {}^B_I R \; {}^B\boldsymbol{P} \quad \text{(B.6)}$$

Appendix C

Fourth Order Runge Kutta ODE Solver

For a comprehensive review of the RK4 method along with an example code refer to [68]. The RK4 method is a first order ODE solver. Hence, the RK4 algorithm is able to solve Equations (3.39) - (3.41) to obtain the velocities. That being said, the position states are also required resulting in the following set of Runge Kutta coefficients.

$$a(\dot{x}) = \dot{x} \tag{C.1}$$
$$b(\dot{z}) = \dot{z} \tag{C.2}$$
$$c(\dot{\alpha}) = \dot{\alpha} \tag{C.3}$$
$$d(\alpha, \dot{\alpha}) = \ddot{x} \tag{C.4}$$
$$e(\alpha, \dot{\alpha}) = \ddot{z} \tag{C.5}$$
$$f(\alpha) = \ddot{\alpha} \tag{C.6}$$

Where a, b, c, d, e and f are associated with solving for $x, z, \alpha, \dot{x}, \dot{z}$ and $\dot{\alpha}$ respectively. Equations (C.4) - (C.6) used to compute the second derivatives of x z and α are the rearranged form of (3.39) - (3.41) such that they are expressed solely in terms of the 2D states and their first derivatives to satisfy the first order solving constraint of the RK4 method. This results in $\ddot{x}$ and $\ddot{z}$ being functions of α and $\dot{\alpha}$, and $\ddot{\alpha}$ being a function of α.

In simulation, the controller feeds the current 2D state vector $\boldsymbol{q} = \begin{bmatrix} x_i & z_i & \alpha_i & \dot{x}_i & \dot{z}_i & \dot{\alpha}_i \end{bmatrix}^T$ along with the chosen command pitch θ to the RK4 solver at every time step ts. The solver then computes the first set of Runge Kutta coefficients $a_1, b_1, c_1, d_1, e_1, f_1$ using the states provided by the controller. Subsequently, following the RK4 method, the second set of coefficients are computed using the first set of coefficients along with the Runge Kutta time step h. Similarly, the third set is computed using the second set along with the time step and so on as shown in (C.7).

$$
\begin{bmatrix} x \\ z \\ \alpha \\ \dot{x} \\ \dot{z} \\ \dot{\alpha} \end{bmatrix} \rightarrow \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \\ e_1 \\ f_1 \end{bmatrix} = \begin{bmatrix} a\left(\dot{x}\right) \\ b\left(\dot{z}\right) \\ b\left(\dot{\alpha}\right) \\ d\left(\alpha, \dot{\alpha}, \theta\right) \\ e\left(\alpha, \dot{\alpha}, \theta\right) \\ f\left(\alpha, \theta\right) \end{bmatrix} \rightarrow \begin{bmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \\ e_2 \\ f_2 \end{bmatrix} = \begin{bmatrix} a\left(\dot{x} + \frac{h}{2}d_1\right) \\ b\left(\dot{z} + \frac{h}{2}e_1\right) \\ b\left(\dot{\alpha} + \frac{h}{2}f_1\right) \\ d\left(\alpha + \frac{h}{2}c_1, \dot{\alpha} + \frac{h}{2}f_1, \theta\right) \\ e\left(\alpha + \frac{h}{2}c_1, \dot{\alpha} + \frac{h}{2}f_1, \theta\right) \\ f\left(\alpha + \frac{h}{2}c_1, \theta\right) \end{bmatrix}
$$

$$
\rightarrow \begin{bmatrix} a_3 \\ b_3 \\ c_3 \\ d_3 \\ e_3 \\ f_3 \end{bmatrix} = \begin{bmatrix} a\left(\dot{x} + \frac{h}{2}d_2\right) \\ b\left(\dot{z} + \frac{h}{2}e_2\right) \\ b\left(\dot{\alpha} + \frac{h}{2}f_2\right) \\ d\left(\alpha + \frac{h}{2}c_2, \dot{\alpha} + \frac{h}{2}f_2, \theta\right) \\ e\left(\alpha + \frac{h}{2}c_2, \dot{\alpha} + \frac{h}{2}f_2, \theta\right) \\ f\left(\alpha + \frac{h}{2}c_2, \theta\right) \end{bmatrix} \rightarrow \begin{bmatrix} a_4 \\ b_4 \\ c_4 \\ d_4 \\ e_4 \\ f_4 \end{bmatrix} = \begin{bmatrix} a\left(\dot{x} + hd_3\right) \\ b\left(\dot{z} + he_3\right) \\ b\left(\dot{\alpha} + hf_3\right) \\ d\left(\alpha + hc_3, \dot{\alpha} + hf_3, \theta\right) \\ e\left(\alpha + hc_3, \dot{\alpha} + hf_3, \theta\right) \\ f\left(\alpha + hc_3, \theta\right) \end{bmatrix} \tag{C.7}
$$

After obtaining the four sets of coefficients, the solver is able to produce the dynamics of the system after the passing of time h using (C.8) - (C.13). Note that this time-step h is considerably smaller than the controller time-step ts in order to ensure accurate model dynamics.

$$x_{i+h} = x_i + \frac{h}{6}\left(a_1 + 2a_2 + 2a_3 + a_4\right) \tag{C.8}$$

$$z_{i+h} = z_i + \frac{h}{6}\left(b_1 + 2b_2 + 2b_3 + b_4\right) \tag{C.9}$$

$$\alpha_{i+h} = \alpha_i + \frac{h}{6}\left(c_1 + 2c_2 + 2c_3 + c_4\right) \tag{C.10}$$

$$\dot{x}_{i+h} = \dot{x}_i + \frac{h}{6}\left(d_1 + 2d_2 + 2d_3 + d_4\right) \tag{C.11}$$

$$\dot{z}_{i+h} = \dot{z}_i + \frac{h}{6}\left(e_1 + 2e_2 + 2e_3 + e_4\right) \tag{C.12}$$

$$\dot{\alpha}_{i+h} = \dot{\alpha}_i + \frac{h}{6}\left(f_1 + 2f_2 + 2f_3 + f_4\right) \tag{C.13}$$

The RK4 solver repeats this process to produce the dynamics after each h time step until the dynamics for the next step of the controller $\boldsymbol{q}_{i+1} = \begin{bmatrix} x_{i+1} & z_{i+1} & \alpha_{i+1} & \dot{x}_{i+1} & \dot{z}_{i+1} & \dot{\alpha}_{i+1} \end{bmatrix}$ are obtained . This process is more clearly illustrated in Algorithm 5.

Algorithm 5: Fourth Order Runge Kutta ODE Solver

input : Initial state vector $\boldsymbol{q}_i = \begin{bmatrix} x_i & z_i & \alpha_i & \dot{x}_i & \dot{z}_i & \dot{\alpha}_i \end{bmatrix}$ and command pitch θ

output : Next step state vector $\boldsymbol{q}_{i+1} = \begin{bmatrix} x_{i+1} & z_{i+1} & \alpha_{i+1} & \dot{x}_{i+1} & \dot{z}_{i+1} & \dot{\alpha}_{i+1} \end{bmatrix}$

1 $x \leftarrow x_i$; $z \leftarrow z_i$; $\alpha \leftarrow \alpha_i$; $\dot{x} \leftarrow \dot{x}_i$; $\dot{z} \leftarrow \dot{z}_i$; $\dot{\alpha} \leftarrow \dot{\alpha}_i$;

2 **for** $j \leftarrow 1$ **to** $\frac{ts}{h}$ **do**

3 `// Calculate the four sets of RK4 coefficients`

4 $[a_1, b_1, c_1, d_1, e_1, f_1] = EOMS\,(x, z, \alpha, \dot{x}, \dot{z}, \dot{\alpha}, \theta)$;

5 $[a_2, b_2, c_2, d_2, e_2, f_2] = EOMS\left(x + \frac{h}{2}a_1, z + \frac{h}{2}b_1, \alpha + \frac{h}{2}c_1, \dot{x} + \frac{h}{2}d_1, \dot{z} + \frac{h}{2}e_1, \dot{\alpha} + \frac{h}{2}f_1, \theta\right)$;

6 $[a_3, b_3, c_3, d_3, e_3, f_3] = EOMS\left(x + \frac{h}{2}a_2, z + \frac{h}{2}b_2, \alpha + \frac{h}{2}c_2, \dot{x} + \frac{h}{2}d_2, \dot{z} + \frac{h}{2}e_2, \dot{\alpha} + \frac{h}{2}f_2, \theta\right)$;

7 $[a_4, b_4, c_4, d_4, e_4, f_4] = EOMS\,(x + ha_3, z + hb_3, \alpha + hc_3, \dot{x} + hd_3, \dot{z} + he_3, \dot{\alpha} + hf_3, \theta)$;

8 `// Calculate the new 2D states (after one RK4 time step)`

9 $x \leftarrow x + \frac{h}{6}\left(a_1 + 2a_2 + 2a_3 + a_4\right)$;

10 $z \leftarrow z + \frac{h}{6}\left(b_1 + 2b_2 + 2b_3 + b_4\right)$;

11 $\alpha \leftarrow \alpha + \frac{h}{6}\left(c_1 + 2c_2 + 2c_3 + c_4\right)$;

12 $\dot{x} \leftarrow \dot{x} + \frac{h}{6}\left(d_1 + 2d_2 + 2d_3 + d_4\right)$;

13 $\dot{z} \leftarrow \dot{z} + \frac{h}{6}\left(e_1 + 2e_2 + 2e_3 + e_4\right)$;

14 $\dot{\alpha} \leftarrow \dot{\alpha} + \frac{h}{6}\left(f_1 + 2f_2 + 2f_3 + f_4\right)$;

15 **end**

16 `// Assign the computed states as the next step states`

17 $x_{i+1} \leftarrow x$;

18 $z_{i+1} \leftarrow z$;

19 $\alpha_{i+1} \leftarrow \alpha$;

20 $\dot{x}_{i+1} \leftarrow \dot{x}$;

21 $\dot{z}_{i+1} \leftarrow \dot{z}$;

22 $\dot{\alpha}_{i+1} \leftarrow \dot{\alpha}$;

Appendix D

Environment Classification for Reinforcement Learning

An environment is formally defined as everything that surrounds the agent apart from the agent itself [69]. It is where the agent resides and is confined for operational purposes. It has certain characteristics from the point of view of the agent which are essential to ascertain prior to algorithm implementation.

Reinforcement learning can be carried out on a plethora of systems, whether solely operating in a software environment or on physical systems functioning in the real world. The appropriate RL algorithm is chosen depending on the environment in which the agent is employed, making the classification of an RL problem's environment an essential milestone. Since the problem being solved in this paper is a single quadrotor control problem (i.e. single agent) no multi-agent environment classifications will be considered in this section.

D.1 Fully Observable vs. Partially Observable:

An agent can only be called an agent if it has the ability to sense its environment to some extent. In other words, the environment must have some degree of observability. In some cases the agent can determine the exact state of the world around it from a set of stimuli meaning that the agent has no uncertainty regarding the environment it is operating in. In these cases the environment is deemed to be *fully observable* [70]. An example of a fully observable environment is the chess board for a chess playing agent. The agent is able to observe exactly where all the chess pieces are as well as all of the opponent's plays.

In environments where one set of stimuli can mean being in one of several states the agent experiences some degree of uncertainty. These environments are called partially observable as the agent is not directly able to observe the whole state of the world around it [70]. Under these circumstances the best an agent can do is have a distribution of probabilities over the set of all possible states. An example of a partially observable environment is a card-game agent as shown in Figure D.1. Unlike the chess agent which is able to observe the location of all the chess pieces and its opponents moves, the card-game agent is only able to observe its own cards, the cards in play, and its opponents moves, limiting the full picture of its environment since it has no knowledge of the opponents cards or the cards out of play.

Figure D.1: An illustration of card game mid-play as a representation of a partially observable state space for a card game playing agent.

In Figure D.1 the agent is shown playing a card game. There is a total of 54 cards in a deck including jokers. Each player is given 10 cards, there are five cards currently in play, and the remainder of the cards are out of play. In this scenario there are 15 observable cards (the agent's own deck and the cards in play) and 39 non-observable cards (the opponent's deck and the cards out of play). The agent is 100% certain of the states of the observable cards. However the probability of a non-observable state being any particular concealed card is $\frac{1}{\text{no. of unknown cards}} \times 100\% = \frac{1}{39} \times 100\% = 2.56\%$. As the game continues, more cards are shown decreasing the number of unknown cards and increasing the observability of the agents environment. This will also increase the probability of the agent knowing what cards the opponent is keeping hold of and hence aiding in predicting best actions to take.

D.2 Deterministic vs. Stochastic:

If the agent can predict that taking a particular action given its state guarantees a specific result then the environment is *deterministic*. More formally, it can be said that an environment is deterministic when the current state resulting from an action is dependent solely on that action and the previous state [70].

In many real-life applications is is difficult to guarantee the effects of taking an action. In other words, unlike deterministic environments, *stochastic* environments exhibit some degree of randomness. In this scenario, the most an agent can do is have a probability distribution over the possible resulting states from taking an action. For example, take a self driving car operating in winter conditions. During the winter season roads can be unpredictable and result in unexpected dynamics. For instance the car attempting to take a sharp right turn from a starting state s_0 into the right-most lane of a three-lane road can have multiple results: the car can succeed in performing the turn and end up in the desired lane (s_1), or the car can skid on ice and end up in either the middle (s_2) or the left-most lane (s_3) as shown in Figure D.2.

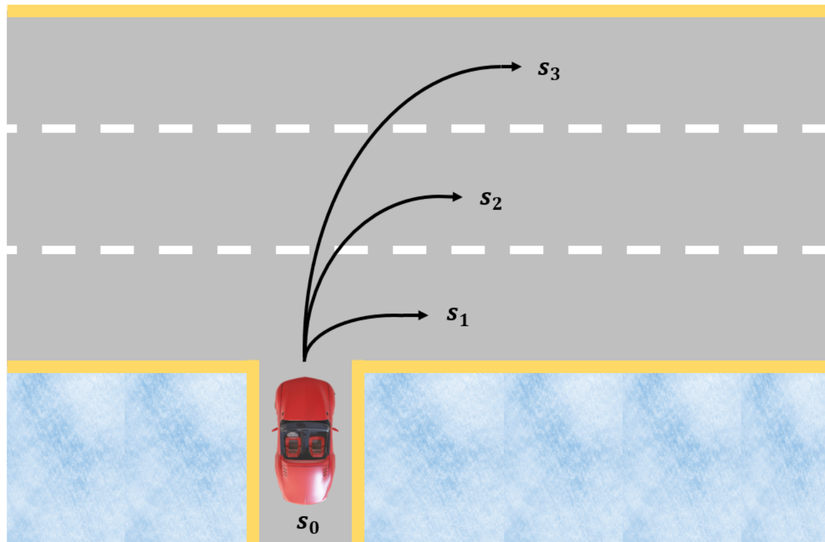


Figure D.2: Figure showing the possible states that could result from a car attempting to turn into the right-most lane in icy conditions.

D.3 Static vs. Dynamic

Environments can also be classified as static or dynamic. *Static* environments are simply environments that do not change over time [69]. An example of this is a self-driving car in an empty parking lot. As the car drives around its environment is not changing as nothing else is moving within the parking lot. If other cars are allowed into the parking lot, the environment is now a *dynamic* environment. This is because as the other cars move around the parking lot the environment is continuously changing around the agent [69].

D.4 Discrete vs Continuous

Environments can additionally be classified based on the state space bounds and the way it is divided. State spaces that can be divided into clear sections and have finite bounds are called *discrete* environments. An example of a discrete environment a chess board. A chess board has clear state divisions with each square being a state and is finite because it is bound to being an 8x8 square board. Hence a chess-board is a discrete environment with 64 clearly defined states.

On the other hand if a state space has infinite bounds either because of it has infinite possibilities or is infinitely large it is called a *continuous* environment. An example of this is the self-driving car's environment. Assuming that the self-driving car is not bound to a particular region it has an infinite $X - Y$ plane to move about, create an infinite state space. That being said, even if the car is bound to a particular region it still has an infinite number of possible locations as with a finite set of x and y values there exists an infinite number of x and y possibilities.

D.5 Episodic vs. Sequential

The final categorisation is related to the task or goal of the agent. A task can be categorised as either episodic or sequential. If the agent has multiple goals that don't require the agent's past performance to complete, then it can be said that the environment the agent is operating

in is *episodic*. For example a self-driving car performing a way-point following algorithm is an episodic task system as the past actions taken by the car to reach the previous waypoint do not effect the car's ability to reach the next waypoint.

Sequential tasks on the other hand are tasks that cannot be divided into episodes as each action effects the agent's possibility of reaching its final goal. A good example of this is a game of chess as every play the agent makes can either enhance of hinder its ability to win the match.

Having discussed the types of environments in which an single agent can be operating in, it is a natural next step to examine the way in which it will be presented to the RL agent. Markov Decision Processes (MDPs) are a key concept as they are the method in which the environment is communicated to the agent.

Appendix E

Markov Decision Process - The State Transition Matrix

In a finite state space, the Markov property describes the probably of transitioning from any initial state $S_t = s$ to any next state $S_{t+1} = s'$, this probability is called a *state transition probability*:

$$\mathcal{P}_{s,s'} = \mathbb{P}\left[S_{t+1} = s' | S_t = s\right] \tag{E.1}$$

All the possible state transition probabilities can be arranged in a matrix called the *state transition matrix*. This is essentially a matrix which contains all probabilities of ending up in any successor state s' given any previous state s.

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix} \tag{E.2}$$

Where n is the number of states. For example, $\mathcal{P}_{1n}$ is the probability of ending up in state $s' = s_n$ when starting in state $s = s_1$, and $\mathcal{P}_{n1}$ is the probability of ending up in state $s' = s_1$ when starting in state $s = s_n$. Note that the nth row in the state transition matrix is a probability distribution of ending up in states 1 through n having started in state i. Hence the sum of each row will have the value 1. Figure E.1 [57] shows an example of a state transition probability matrix for a mars rover moving with seven states. Note that the number above each arrow indicates the probability of ending up going in that direction.

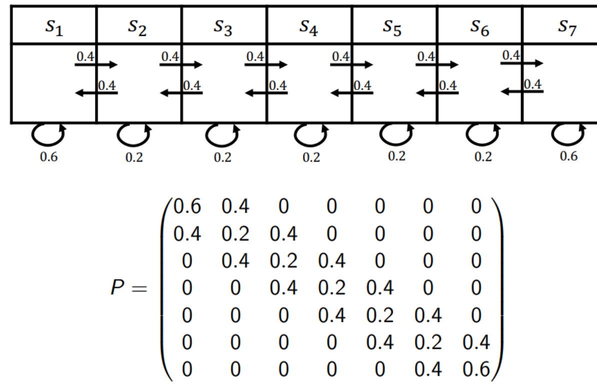


Figure E.1: Figure of the Markov chain and state transition probability matrix of a simple mars rover that can go left or right [57].

Appendix F

Dynamic Programming (DP)

This appendix goes through the main DP algorithms. Since DP algorithms are considered building blocks for the remainder of the RL algorithms, it will cover some essential concepts for reinforcement learning. For a complete review of DP in reinforcement learning please refer to Chapter 4 of [55].

F.1 Policy Evaluation

Policy evaluation is the first step of any RL algorithm. This step's goal is to predict the state value function V_π for some arbitrary policy given the policy π along with system's MDP. Solving for the state value function given a policy is a straight-forward update method in which the new estimate of value for current state $V_{k+1}(s)$ is updated through using the old estimate of the value for the successor state $V_k(s')$. This step is achieved using the state value function Bellman expectation equation shown in (4.18). Starting with some arbitrary initial value function V_0 (usually initialised to all zeros), the value function is updated by applying (F.1) to all the states in the state space in order to obtain the new value function V_1. This process in which the expected values of the next state are used to enrich the prediction of the value of the current state is called *Bootstrapping*.

$$V_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(r_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_k(s') \right) \tag{F.1}$$

Repeating this iterative process continuously improves the estimate of the state value function for the policy $V_2, V_3, V_4, ...$ until the estimated value function converges to the true value function for that policy V_π. For implementation purposes, convergence is considered achieved when the difference between V_{k+1} and V_k is less than some user specified margin θ. The full DP policy evaluation algorithm is clearly presented in Algorithm 6.

F.2 Policy Improvement

The reason for employing a policy evaluation step to compute V_π for some arbitrary policy π is ultimately to determine whether said policy can be improved upon. The policy improvement step of DP algorithms focuses on refining the policy π such that the optimal policy π_* is achieved.

Algorithm 6: Iterative Policy Evaluation for Estimating $V \approx V_\pi$

input : Policy π, small threshold θ for determining the accuracy of the estimation and MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

output : Estimate of the state value function V_π

1 Initialise $V(s)$ for all states $s \in \mathcal{S}$ arbitrarily with $V(terminal) = 0$
2 **while** $\Delta \geq \theta$ **do**
3 $\quad\Delta \leftarrow 0$
4 $\quad$**for** $\forall s \in \mathcal{S}$ **do**
5 $\quad\quad$`// Assign new` V_k `to be the last` V_{k+1}
6 $\quad\quad V_k \leftarrow V_{k+1}$
7 $\quad\quad$`// Compute the new` V_{k+1} `using the Bellman expectation equation`
8 $\quad\quad V_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(r_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_k(s') \right)$
9 $\quad\quad$`// Compute the difference between the value function estimates`
10 $\quad\quad \Delta \leftarrow \max(\Delta, |V_k(s) - V_{k+1}(s)|)$
11 $\quad$**end**
12 **end**

Following the policy evaluation step, the system is only aware of the value of being in some state s and following the policy π. A way to determine whether there exists some action a in state s which is better than the action that the policy assigned to that state a_π is to take the action a then continue to follow the policy for the rest of the episode. The value of this behaviour is computed by the state-action value function $Q_\pi(s, a)$ using (4.15). If the computed $Q_\pi(s, a)$ is greater than $V_\pi(s)$, then according to the *policy improvement theorem* it can be said that taking action a in state s is better than following policy π. The policy improvement theorem states that for any two deterministic policies π and π' for all $s \in \mathcal{S}$ if

$$Q_\pi(s, \pi'(s)) \geq V_\pi(s)$$

Then policy π' must be as good as or better than π for all $s \in \mathcal{S}$. This means that the value obtained by following π' is also greater than or equal to that achieved from following π.

$$V'_\pi(s) \geq V_\pi(s)$$

The full proof of this theorem can be found in section 4.2 of [55].

Having determined which policy guarantees a higher return, policy π is updated to π' such that action a is chosen rather than a_π when the system reaches s. When considering changes to all states and actions the policy improvement selects at each state the action that appears to be the best according to $Q_\pi(s, a)$. Therefore it can be said that the new *greedy* policy π' simply seeks to maximise $Q_\pi(s, a)$ through appropriate action selection.

$$\begin{aligned} \pi' &= \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q_\pi(s, a) \\ &= \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left[r_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_\pi(s') \right] \end{aligned} \tag{F.2}$$

This introduces the previously alluded to *policy iteration* algorithm.

F.3 Policy Iteration Algorithm

A policy π_0 can be improved using V_{π_0} to obtain π_1 which can again be improved using V_{π_1} to obtain π_2 and so on until optimal policy convergence.

$$\pi_0 \xrightarrow[Eval.]{Policy} V_{\pi_0} \xrightarrow[Improv.]{Policy} \pi_1 \xrightarrow[Eval.]{Policy} V_{\pi_1} \xrightarrow[Improv.]{Policy} \pi_2 \; ... \xrightarrow[Improv.]{Policy} \pi_* \xrightarrow[Eval.]{Policy} V_*$$

This iterative process of policy evaluation using Algorithm 6 and greedy policy improvement using (F.2) is the entire essence of the policy iteration DP algorithm which is presented in detail in Algorithm 7. Convergence is reached when the Bellman optimality equation in 4.21 is satisfied.

F.4 Value Iteration Algorithm

A disadvantage of the policy iteration algorithm is the policy evaluation step which is computationally intensive as it sweeps through all the possible states within the state space over and over until achieving convergence on V_π for the current policy being evaluated. The value iteration algorithm provides a solution for this problem by truncating the policy evaluation step while still maintaining the convergence guarantee on V_π.

The value iteration algorithm ceases the policy evaluation after one full update over the states then proceeds with the policy improvement step. The truncated policy evaluation and improvement are both combined in the Bellman optimality equation shown in (4.21) if it is assumed that V_* is known. Hence the value iteration algorithm simply updates and improves the value function using 4.21 with the current value function until converging to an optimal value function V_π.

$$V_0 \xrightarrow[Optimality]{Bellman} V_1 \xrightarrow[Optimality]{Bellman} V_2 \xrightarrow[Optimality]{Bellman} ... \xrightarrow[Optimality]{Bellman} V_*$$

One the optimal value function is obtained, the optimal greedy policy π_* can calculated using (F.2). The full DP value iteration algorithm is presented in Algorithm 8.

F.5 Asynchronous Dynamic Programming

As previously mentioned the major drawback of DP is that it must perform updates for every state in $\mathcal{S}$. One way to overcome this is to implement asynchronous updates such that the DP algorithm is not stuck in a states space sweep for an extended amount of time. The policy evaluation and improvement steps are implemented on the states in no order at all. That being said, in order to ensure convergence to the true optimal policy the asynchronous algorithm must ensure that all the states are constantly being updated. Note that this does not reduce the computational complexity of dynamic programming but it does reduce the time it spends within loops. For more information on asynchronous DP please refer to section 4.6 of [55].

Algorithm 7: Policy Iteration Using Iterative Policy Evaluation for Estimating $\pi \approx \pi_*$

input : Small threshold θ for determining the accuracy of the estimation and MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

output : Estimate of the optimal policy π_* and optimal state value function V_*

```
1  // 1. Initialisation
2  Initialise V(s) and π(s) ∈ A(s) for all states s ∈ S arbitrarily with V(terminal) = 0
3  // 2. Policy Evaluation
4  while Δ ≥ θ do
5      Δ ← 0
6      for ∀s ∈ S do
7          // Assign new V_k to be the last V_{k+1}
8          V_k ← V_{k+1}
9          // Compute the new V_{k+1} using the Bellman expectation equation
10         V_{k+1}(s) = Σ_{a∈A} π(a|s) (r_s^a + γ Σ_{s'∈S} P_{ss'}^a V_k(s'))
11         // Compute the difference between the value function estimates
12         Δ ← max(Δ, |V_k(s) − V_{k+1}(s)|)
13     end
14 end
15 // 3. Policy Improvement
16 policy_eopitmal ← True
17 for ∀s ∈ S do
18     // Assign the old policy action for state s to the variable a_old
19     a_old ← π(s)
20     // Update the policy at state s by choosing the greedy action
21     π(s) ← argmax_a [r_s^a + γ Σ_{s'∈S} P_{ss'}^a V_π(s')]
22     if a_old ≠ π(s) then
23         policy_opitmal ← False
24     end
25 end
26 if policy_opitmal then
27     return π ≈ π_*, V ≈ V_*
28 else
29     Return to 2. Policy Evaluation
30 end
```

Algorithm 8: Value Iteration for Estimating $\pi \approx \pi_*$

input : Small threshold θ for determining the accuracy of the estimation and MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

output: Estimate of the optimal policy π_* and optimal state value function V_*

```
1  Initialise V(s) for all states s ∈ S arbitrarily with V(terminal) = 0
2  while Δ ≥ θ do
3      Δ ← 0
4      for ∀s ∈ S do
5          // Assign new V_k to be the last V_{k+1}
6          V_k ← V_{k+1}
7          // Compute the new V_{k+1} using the Bellman optimatlity equation
8          V_{k+1}(s) = max_a R_s^a + γ Σ_{s'∈S} P_{ss'}^a V_k(s')
9          // Compute the difference between the value function estimates
10         Δ ← max(Δ, |V_k(s) − V_{k+1}(s)|)
11     end
12 end
```

Line 8: $V_{k+1}(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V_k(s')$

Line 10: $\Delta \leftarrow \max(\Delta, |V_k(s) - V_{k+1}(s)|)$

Appendix G

Monte Carlo Methods

The following appendix briefly introduces the main ideas of Monte Carlo (MC) methods along with it's basic algorithms. For a better understanding of the Monte Carlo Methods as well as its variations please refer to Chapter 5 of [55].

G.1 Monte Carlo Prediction

As illustrated by (4.14). The state value function for some policy π is simply the expected return from starting in the state s and following π. A clear way of obtaining this is by actually going through the episodes and taking the average the returns from starting in s to obtain $V_\pi(s)$ for all the states in the space. The more a state is visited by the agent, the more accurate the estimation of its value is, as it will have more returns to average.

The MC prediction method has two variations: first visit and every visit. In a given episode a state s can be visited multiple times. The first visit MC method estimates $V_\pi(s)$ by averaging the returns only following the agent's first visits to s, while every-visit MC methods estimate the value using the average of the returns following every visit of s. The difference between the two approaches is that the first visit algorithm must keep track of whether this is the agent's first time in s within the episode whereas the every visit one does not need to. Algorithm 9 shows the full breakdown of the first-visit MC prediction as it is the more widely studied approach in this field.

The main concept behind the MC prediction algorithm is to achieve the value function through using an incremental mean update.

$$V(s_t) = V(s_t) + \frac{1}{N(s_t)}(G_t - V(s_t))$$

Where $N(s_t)$ is the number of first visits to state s_t. Here, the actual return G_t is called the target since the value function is updated to move in the direction of it. This is a method of computing the mean of the returns for all episodes without having to append the returns to some vector. This method enables faster computations and lower storage requirements.

In non-stationary environments (refer to Appendix D for information on environment classification for RL) where the value function is not constant, it may be beneficial to take a small step α towards the return rather than calculating the actual mean.

$$V(s_t) = V(s_t) + \alpha(G_t - V(s_t))$$

This makes the steps towards the targets smaller to avoid overshoot of the dynamic value function. The step size α can be tuned by the user depending on the nature of the operating environment.

Algorithm 9: First-Visit Monte Carlo Prediction for Estimating $V \approx V_\pi$

```
input  : Policy π and update step size α
output : Estimate of the state value function V_π
1  Initialise V(s) ∀s ∈ S arbitrarily with V(terminal) = 0
2  N(s) ← an empty list, ∀s ∈ S
3  Check(s) ← an empty list, ∀s ∈ S
4  for i = 1, 2, ..., no.episodes do
5      Generate an episode following π: s_0, a_0, r_0, s_1, a_1, r_1, ..., s_T, a_T, r_T
6      G ← 0
7      Check(s) ← 0, ∀s ∈ S // Reset the first visit checking array
8      // for all states in the episode
9      for t = T, T − 1, T − 2, ..., 0 do
10         G ← r_t + γG // Compute the return for s_t
11         Check(s_t) = Check(s_t) + 1 // Increment the first visit checker for s_t
12         // If this is the first visit for s_t in the episode
13         if Check(s_t) = 1 then
14             N(s_t) = N(s_t) + 1 // Update the first step counter
15             // Incremental mean update of V(s_t)
16             V(s_t) = V(s_t) + 1/N(s_t) (G − V(s_t)) // Stationary problem
17             V(s_t) = V(s_t) + α (G − V(s_t)) // Non-stationary problem
18         end
19     end
20 end
```

G.2 Monte Carlo Control

The next step is to then estimate the optimal policy π_*. In circumstances where the MDP is not provided, an estimate of the state value function is usually not enough to obtain an accurate estimate of the optimal policy. Without the model the algorithm must explicitly estimate the value of taking each action in order for the values to be sufficient enough to be used for obtaining π_*. Hence MC methods are more concerned with obtaining a state-action value estimate $Q_\pi(s, a)$ rather than the state value function $V_\pi(s, a)$. MC methods for obtaining π_* follow the same methodology as the ones presented in the DP policy iteration algorithm in Appendix F.

$$\pi_0 \xrightarrow[Eval.]{Policy} Q_{\pi_0} \xrightarrow[Improv.]{Policy} \pi_1 \xrightarrow[Eval.]{Policy} Q_{\pi_1} \xrightarrow[Improv.]{Policy} \pi_2 \ \dots \xrightarrow[Eval.]{Policy} Q_{\pi_*} \xrightarrow[Improv.]{Policy} \pi_*$$

The policy evaluation step follows the steps provided in Algorithm 9. The policy improvement step is applied through constructing a greedy policy with respect to the current value function. And since the value function being approximated is the state-action value function, a model is

not required to choose the greedy action.

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s,a)$$

That being said, the greedy policy implementation for MC methods halts the systems exploration of other possible actions. Once the system takes an action and it provides some reward which is greater than the other actions, the agent will always opt to choose that action when returning to the state again. This ceases the agent from exploring other options which may actually be the optimal action for the state. Hence in order to remedy this an ϵ-greedy exploration algorithm is utilised for improving the policy.

$$\pi(a|s) = \begin{cases} \epsilon/|A(s)| + 1 - \epsilon & \text{if } a^* = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s,a) \\ \epsilon/|A(s)| & \text{otherwise} \end{cases} \tag{G.1}$$

The policy improvement in (G.1) ensures that all actions in the action space $\mathcal{A}(s)$ are chosen with some none zero probability by selecting a random action with probability ϵ and the greedy action with probability $1 - \epsilon$. Furthermore, the MC policy evaluation step runs into the same issue as the DP policy evaluation where to truly approach $Q_\pi(s,a)$ the number of episodes must approach infinity. This is obviously impractical. Hence the policy evaluation step is terminated following one episode. This still provides enough of an improvement to the state-action value approximation to ultimately achieve convergence to the optimal value without being held up in the full evaluation process. This full Monte Carlo algorithm is illustrated in Algorithm 10

Algorithm 10: First-Visit MC Control for ϵ-soft Policies for Estimating $\pi \approx \pi_*$

input : Update step size α and small random action probability $\epsilon > 0$
output : Estimate of the optimal policy π_*

```
1  Initialise π as some arbitrary ε-soft policy
2  Initialise Q(s, a) arbitrarily ∀s ∈ S, a ∈ A(s)
3  N(s, q) ← an empty list, ∀s ∈ S, a ∈ A(s)
4  Check(s, a) ← an empty list, ∀s ∈ S, a ∈ A(s)
5  for i = 1, 2, ..., no.episodes do
6      Generate an episode following π: s_0, a_0, r_0, s_1, a_1, r_1, ..., s_T, a_T, r_T
7      G ← 0
8      Check(s, a) ← 0, ∀s ∈ S, a ∈ A // Reset the first visit checking array
9      // for all states in the episode
10     for t = T, T − 1, T − 2, ..., 0 do
11         G ← r_t + γG // Compute the return for s_t
12         Check(s_t, a_t) = Check(s_t, a_t) + 1 // Increment the first visit checker for
               the state-action pair s_t, a_t
13         // If this is the first visit for the pair s_t, a_t in the episode
14         if Check(s_t, a_t) = 1 then
15             N(s_t, a_t) = N(s_t, a_t) + 1 // Update the first step counter
16             // Incremental mean update of Q(s_t, a_t)
17             Q(s_t, a_t) = Q(s_t, a_t) + 1/N(s_t, a_t) (G − Q(s_t, a_t)) // Stationary problem
18             Q(s_t, a_t) = Q(s_t, a_t) + α (G − Q(s_t, a_t)) // Non-stationary problem
19             // Update the policy π
20             A_* = argmax_{a∈A} Q(s_t, a)
21             π(a|s_t) = { ε/|A(s)| + 1 − ε   if a = A_*
                          { ε/|A(s)|           if a ≠ A_*
22         end
23     end
24 end
```

Line 17: $Q(s_t, a_t) = Q(s_t, a_t) + \frac{1}{N(s_t,a_t)}\left(G - Q(s_t, a_t)\right)$

Line 20: $A_* = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s_t, a)$

Line 21: $\pi(a|s_t) = \begin{cases} \epsilon/|A(s)| + 1 - \epsilon & \text{if } a = A_* \\ \epsilon/|A(s)| & \text{if } a \neq A_* \end{cases}$

Appendix H

Auto-Tuning ArduPilot's PID Controllers

The autotune mode aims to optimize the PID controller's response to the aircraft's attitude inputs. It does this by deliberately moving the aircraft around the tuning axis, observing its impulse and step responses, and adjusting parameters such as 'Stabilise P,' 'Rate P,' and 'Rate D', as shown in Figure H.1, until the desired response without significant overshoot it achieved. Following the autotune process the ArduPilot input time constant parameter ATC_INPUT_TC was changed to zero in order to push the input signal of the autopilot as close to a step input as possible.

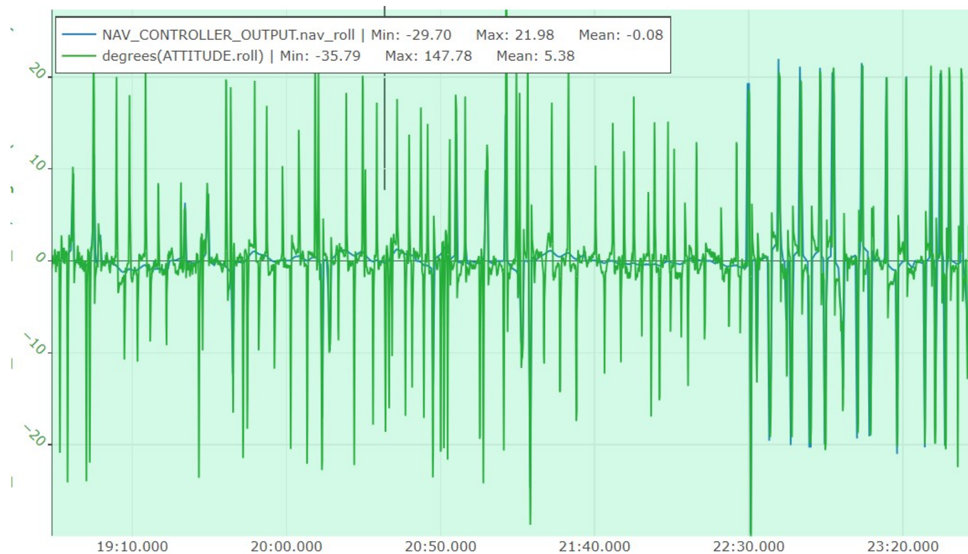


Figure H.1: Figure of the twitching behaviour of the roll of the UAV during the autotune process.

Due to the confined space of the test bed along with the drift of the aircraft when set in auto-tune mode, the auto-tuning task proved to be extremely tedious to execute especially in the roll axis as it called for frequent manual intervention in order to ensure that the aircraft did not crash. However the aircraft was flown following the completion of the auto-tuning task to ensure smooth and reliable control characteristics.

Appendix I

PID Input Signal Modelling

The general equation for a sigmoid is in the following form:

$$y(t) = \frac{|\Delta\phi|}{1 + \exp\left(b(t - c)\right)} + d \tag{I.1}$$

The numerator of (I.1) is the absolute value of the commanded change in angle $\Delta\phi = \phi_{command} - \phi_{current}$, and the variable d is the location of the bottom of the sigmoid. When the autopilot commands a positive $\Delta\phi$, d is the starting angle. On the other hand, when $\Delta\phi$ is negative, d is the commanded angle. Both cases are described mathematically in (I.2) and shown in Figure I.1.

$$d = \begin{cases} \phi_{current} & \text{if } \Delta\phi > 0 \\ \phi_{commanded} & \text{if } \Delta\phi < 0 \end{cases} \tag{I.2}$$

The variables b and c are responsible for the shape and direction of the sigmoid curve. Equation (I.1) was produced by the curve fitting tool for all the roll input values and the b and c values were recorded. The trends of b and c vs $\Delta\phi$ were analysed to find equations which fit the obtained values. Again, using the curve fitting app (I.3) and (I.4) were obtained to approximate the values of variables b and c for any given $\Delta\phi$.

$$b(\Delta\phi) = \begin{cases} -232.2/\sqrt{\Delta\phi} & \text{if } \Delta\phi > 0 \\ 232.2/\sqrt{|\Delta\phi|} & \text{if } \Delta\phi < 0 \end{cases} \tag{I.3}$$

$$c(\Delta\phi) = 0.0175\sqrt{|\Delta\phi|} \tag{I.4}$$

The final estimated input signal is the following:

$$y(t) = \begin{cases} \frac{\Delta\phi}{1+\exp\left((-232.2/\sqrt{\Delta\phi})(t-0.0175\sqrt{\Delta\phi})\right)} + \phi_{current} & \text{if } \Delta\phi > 0 \\ \frac{|\Delta\phi|}{1+\exp\left((232.2/\sqrt{|\Delta\phi|})(t-0.0175\sqrt{|\Delta\phi|})\right)} + \phi_{commmanded} & \text{if } \Delta\phi < 0 \end{cases} \tag{I.5}$$

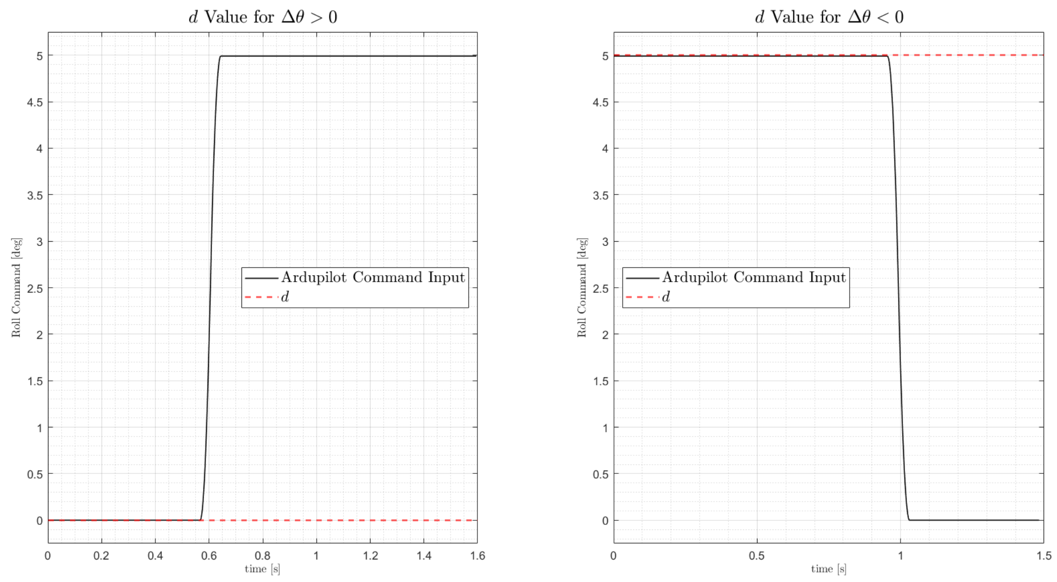


Figure I.1: Figure showing the d variable value of the sigmoid function (I.1) for an increasing and decreasing roll command.