ROTORCRAFT SLUNG PAYLOAD STABILIZATION USING REINFORCEMENT LEARNING

Eleni Sabourin

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

Department of Mechanical Engineering
Faculty of Engineering
University of Ottawa

© Eleni Sabourin, Ottawa, Canada, 2024

Abstract

In recent years, the use of rotorcraft uninhabited aerial vehicles (UAVs) for cargo delivery has become of particular interest to private companies and humanitarian organizations, namely due to their reduced operational costs, ability to reach remote locations and to take off and land vertically. The slung configuration, where the cargo is suspended below the vehicle by a cable, is slowly being favoured for its ability to transport different sized loads without the need for the vehicle to land. However, such configurations require complex control systems in order to stabilize the swing of the suspended load. The goal of this research is to design a control system which will be able to bring a slung payload transported by a rotorcraft UAV back to its stable equilibrium in the event of a disturbance.

A simple model of the system is first derived from first principles for the purpose of simulating a control algorithm. A controller based in model-free, policy-gradient reinforcement learning is then derived and implemented on the simulator in order to tune the learning parameters and reach a first stable solution for load stabilization in a single plane. An experimental testbed is then constructed to test the performance of the controller in a practical setting. The testbed consists of a quadcopter carrying a weight suspended on a string and of a newly designed on-board load-angle sensing device, to allow the algorithm to operate using only on-board sensing and computation. While the load-angle sensing design was found to be sensitive to the aggressive manoeuvres of the vehicle and require reworking, the proposed control algorithm was found to successfully stabilize the slung payload and adapt in real-time to the dynamics of the physical testbed, accounting for model uncertainties. The algorithm also works within the framework of the widely-used, open-source autopilot program ArduCopter, making it straightforward to implement on existing rotorcraft platforms. In the future, improvements to the load-angle sensor should be made to enable the algorithm to run fully on-board and allow the vehicle to operate outdoors. Further studies should also be conducted to limit the amount of vehicle drift observed during testing of the load stabilization.

Acknowledgements

I would like to thank my supervisor, Dr. Eric Lanteigne, for his continued guidance, assistance and advice throughout this project and throughout my studies. I would also like to thank my parents who have supported me for my entire university career and allowed me to succeed in a way that they can be proud of. I would also like to thank my good friends and colleagues, Nourah and Ahmed, for the laughs, lively discussions, blackboard brainstorming sessions and online games which made the long days in the lab fun, and the time spent doing online learning a lot more enjoyable.

Table of Contents

List of Figures

List of Tables

Nomenclature

a	Control action
G	Return (sum of rewards from current time to the horizon)
g	Gravitational constant ($9.81 m/s^2$)
I	Inertia tensor of the quadcopter in body frame coordinates
J	Inertia tensor of the quadcopter in generalized coordinates
L	Lagrangian
m	Mass of the quadcopter
m_p	Mass of the slung payload
n	Number of tilings
Q^π	Action-value function
$\mathbf{q}$	Vector of generalized coordinates
R	Rotation matrix
r	Reward
s	System state
T	Total propeller thrust force
V^π	Value function
$\mathbf{w_V}$	Value function parameter/weight vector
$\mathbf{w}_\mu$	Action mean parameter/weight vector
$\mathbf{w}_\sigma$	Action standard deviation parameter/weight vector
x	Position of the quadcopter center of mass in the inertial x direction
y	Position of the quadcopter center of mass in the inertial y direction

z	Position of the quadcopter center of mass in the inertial z direction
α	Load polar angle
α_{enc}	Load polar angle measured by the on-board sensing device
β	Load azimuthal angle
β_{enc}	Load azimuthal angle measured by the on-board sensing device
γ	Discount factor
δ	Temporal-difference error
ε_V	Step size for the SGD update of $\mathbf{w_V}$
ε_μ	Step size for the SGA update of $\mathbf{w}_\mu$
ε_σ	Step size for the SGA update of $\mathbf{w}_\sigma$
η	Attitude "vector" of the quadcopter
θ	Pitch angle
λ	Slung load cable length
μ	Gaussian distribution mean for action selection
$\xi_\mathbf{B}$	Position vector of the quadcopter center of mass in the inertial frame
$\xi_\mathbf{P}$	Position vector of the slung payload in the inertial frame
$\xi_\mathbf{P/B}$	Position vector of the slung payload relative to the vehicle center of mass, in the inertial frame
π	Control policy
σ	Gaussian distribution standard deviation for action selection
τ_θ	Quadcopter pitching moment
τ_ϕ	Quadcopter rolling moment
τ_ψ	Quadcopter yawing moment
$\mathbf{\Phi}$	Features vector

ϕ Roll angle

ϕ_{cmd} Roll angle command requested by the RL agent, after saturation limit

$\phi_{cmd,filt}$ Roll angle command filtered by ArduPilot's sigmoid filter

ψ Yaw angle

$\mathbf{\Omega_B}$ Angular velocity vector of the quadcopter in the body frame

{B} Body reference frame

{E} Inertial reference frame

{P} Payload reference frame

1. Introduction

1.1 Background

In recent years, uninhabited aerial vehicles (UAVs) have found applications in diverse areas and fields, owing to an increased public awareness and acceptance, clearer government regulations and reduced hardware costs. By removing the need for a human operator on board, UAV missions have significantly lower operational costs, and pose no hazard to a human pilot, enabling tasks such as dangerous rescue operations or manoeuvring in hostile environments to be completed with little risk [1]. The use of rotorcraft UAVs has become of particular interest to private companies and humanitarian organizations due to their ability to vertically take off and land, removing the need for runways or launch mechanisms. This makes them particularly suited for cargo delivery operations to urban or remote locations which may otherwise be inaccessible [1].

Cargo may be loaded onto an aircraft in one of two ways: the first is in a grasped configuration by which the load is rigidly fixed to the rotorcraft body, and the second is by way of a cable which suspends the load beneath the rotorcraft body ("slung load") [2]. While the first method results in simpler dynamics and therefore easier controller design, it also results in an increased rotational inertia of the vehicle, slowing down its response to disturbance and making manoeuvring with agility more difficult. The second approach is therefore favoured when agility is of particular concern [2], [3]. Furthermore, there are several cases where a grasped load is impossible to achieve, for example if the load is far too large to be housed in the payload bay, or when the aircraft is simply unable to land in order to secure the load to the aircraft body. The latter may occur in the case of a rescue operation [4], or in hostile terrain. Increasing interest in the slung load approach requires sophisticated control algorithms to be designed, owing to the complex, strongly-coupled, under-actuated dynamics of such systems [3]. Control becomes particularly important when the load in question is delicate or sensitive to motion or disturbance, requiring an additional level of complexity to ensure it remains level or stable throughout flight. Stabilizing the swing of the slung payload is also crucial for the delivery process itself, where the parcel might need to be set down smoothly at a precise location.

1.2 Objectives

The complex dynamics and unknown disturbances inherent to slung-load aircraft systems suggest that purely model-based approaches to control design, such as those stemming from modern control theory, may not be optimal for this application: In a purely model-based approach, the performance of the controller is entirely dependent on the model's ability to reflect the true dynamics of the system. Unfortunately, models are never able to perfectly capture the behaviour of any real system, and this is especially true if the system in question has complex interactions and dynamics, such as the one studied herein. With industrial systems becoming increasingly intricate, and performance requirements also increasing, data-driven, model-free approaches are being favoured over modern control approaches for such applications [5]. Model-free approaches will adapt their control laws to changing or unmodelled dynamics. As such, these approaches can be especially useful when the system to be controlled has complex interactions, or when an accurate model of the environment, which is extremely difficult to obtain, is required for control [6].

Also, in order to ensure a controller will be able to operate in any outdoor environment, it must fully rely on on-board sensing and processing in order to complete its task. The goal of this research is therefore to develop a model-free control algorithm whose objective is to stabilize a slung load transported by rotorcraft in the event of an unforeseen disturbance such as a wind gust, which may result from unknown environment dynamics. The proposed solution should:

1. run fully on-board the vehicle;
2. not rely on any actuators besides the rotorcraft propulsion system;
3. integrate seamlessly with existing autopilot technologies, making it simple to implement in practice;
4. be computationally-efficient enough to enable real-time adaptation, and enable the controller to account for behaviours which are difficult to model, resulting from the actuators, power systems, sensors on board the aircraft or changes in system parameters.

1.3 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 presents a review of existing solutions and approaches to the slung load stabilization problem; Chapter 3 presents a review of the model-free reinforcement learning theory which is relevant to this project; Chapter 4 describes the derivation of a simple mathematical model of the slung-load system, for the purpose of simulating the performance of a model-free control algorithm and appropriately selecting its learning parameters; Chapter 5 outlines the proposed model-free control scheme which will be used to accomplish the goal; Chapter 6 describes the experimental setup and approach to verify and improve upon the simulated control algorithm; Finally, results are presented and discussed, and conclusions are drawn, in Chapter 7.

1.4 Notes on Terminology

While the term "uninhabited aerial vehicle" (UAV) will be used in the remainder of this thesis, such systems have been known to go by many different names. These include "drones", "uninhabited aircraft systems", "unmanned aerial vehicles" and "uncrewed aircraft systems", to name a few. Official Canadian government regulations refer to them as "remotely-piloted aircraft systems" (RPAS) [7]. These terms all refer to aerial vehicles which do not have a human pilot on board, and can be used interchangeably.

2. Literature Review

2.1 Slung Payload Control

Existing solutions to slung payload control on aerial vehicles can be divided into two main approaches: model-based controllers and model-free controllers. A brief review of existing model-based solutions will first be discussed, in order to contrast with existing model-free approaches, which will be discussed afterwards.

2.1.1 Model-Based Approaches

Many model-based solutions to the task of controlling aircraft with slung loads have been proposed in the last few years. However, most of these are concerned with the trajectory tracking of the aircraft, while treating the slung load dynamics as an external disturbance to the stability of the aircraft [8]–[10]. While these solutions can provide insight into the dynamics of such systems, the goal of this project is to stabilize the position of the load itself under unforeseen disturbances. To the latter effect, a few model-based methods have previously been proposed, utilizing some aspect of a mathematical model in order to derive the control law.

For instance, conventional, linear approaches such as proportional-derivative (PD) control have been proposed to stabilize a slung load transported by a quadcopter, with disturbances to the load introduced by manually striking it [11]. In this approach, a virtual point is placed directly above the payload, at the same altitude as the vehicle, and a position error is generated by subtracting the position of the virtual point from that of the vehicle center of mass. The vehicle is then instructed, using PD control, to follow this point in order to force the payload to come directly beneath the vehicle. A few additional approaches use nonlinear controllers based in geometric control theory in order to track a desired load position or trajectory, also using a quadcopter [12]–[14]. In each of these approaches, the control development begins by demonstrating that the system is differentially-flat. That is, all system states and inputs can be expressed as nonlinear functions of a finite set of "flat" outputs, using first principles and model equations. In this case, the load position and quadcopter yaw angle are found to be flat outputs. If the yaw is held constant, this property enables the generation of complex trajectories for the load position. Given a desired load position

(and its derivatives), a set of system states and inputs can be computed from the nonlinear functions. The proposed controllers then uses geometric definitions of the configuration errors to achieve the computed system state, by driving these errors to zero using something similar to PD control with added feedforward terms. In [2], the differential-flatness assumption is again used to generate desired system states which can be used to follow a desired load trajectory. However, a modified linear-quadratic regulator (LQR) is then used to track the desired states. The load dynamics are first linearized about the desired trajectory in order to yield a linear-time-variant system, and the tracking problem is reduced to the classical LQR optimization problem. The required control inputs are computed by solving the continuous-time Riccati equation for the quadcopter thrust and moments. In [15], payload swing is minimized during flight for a helicopter which is given a separate waypoint tracking task. An "ideal" vehicle trajectory which minimizes load swing, yet also reaches a desired position efficiently, is first derived following the path planning outlined in [16]: A smooth S-shaped function is selected as the ideal trajectory for path planning from one waypoint to another, and the vehicle velocity and acceleration are bounded so as to not disturb the slung payload. Next, an energy-shaping approach is used to follow the generated trajectory. Energy-shaping approaches introduce a control signal which will force the system to have a certain, desired, energy function. This energy function has a minimum which corresponds to the desired system trajectory [17]. In this case, the energy function was defined as the kinetic energy error added to the system's potential energy and to additional error terms. The control law was obtained by finding the minimum of the defined function. It was demonstrated that all tracking errors, as well as the load swing angle, approached zero in the limit.

Several methods based in reinforcement learning (RL) have also been proposed. While it is possible to use RL to control in a model-free manner, many approaches applied to aircraft-slung-load systems have used model-based RL. In [18], a controller based in dynamic programming, a RL technique, is proposed to generate "swing-free" quadcopter trajectories. First, the system model is converted to a discrete-time system, with piecewise-linear dynamics. A quadratic objective function, which incorporates the payload state, quadcopter attitude, velocity and acceleration, as well as the vehicle position error, is then defined. A series of optimal quadcopter accelerations are then pre-generated by dynamic programming, by selecting those which minimize the objective function, and therefore stabilize the slung payload while also reaching the given waypoint target. The sequence of desired quadcopter accelerations is then followed using a feedback-linearizing controller designed with pole-placement. In [19], a

similar approach is used: the system is linearized about a desired vehicle and payload trajectory, and an objective function is defined which, when minimized, corresponds to the desired trajectory being followed. A dynamic-programming approach known as iterative linear quadratic gaussian (iLQG) is then used to generate an optimal sequence of quadcopter inputs (thrust and moments) which will manoeuvre the load according to the desired trajectory, all the while considering the effect of Gaussian random noise in the measurements [20].

While dynamic programming is powerful at generating trajectories which are optimal for the given task, it is a RL technique which requires an exact model of the full system and may behave poorly when subjected to disturbances or in stochastic environments [6], [20], [21].

2.1.2 Model-Free Approaches

Approaches based in model-free control encompass those where the control law does not rely on the dynamics of the system in any way. It instead relies on "learning" from input and output data in order to select appropriate control signals [5]. There are several data-driven model-free approaches to controlling nonlinear systems, for example model-free adaptive controllers based on some form of dynamic linearization [5]. However, the vast majority of model-free controllers used for slung payload control or stabilization are based in model-free reinforcement learning (RL). While model-free RL has existed for decades, it has garnered a plethora of attention very recently, along with other machine learning techniques [22]. This rise in popularity is mainly owed to the recent availability of large amounts of data though the internet, and the significant rise in the computational power of modern computers [23]. A review of the aspects and theory of model-free RL which are relevant to the current project is presented in Chapter 3.

In [21], [24]–[26], model-free RL is used to generate flight trajectories which minimize payload swing for a quadcopter carrying a slung load. Linear value function approximation is used to learn the control policy's value functions for a continuous state-space. The continuous states are taken to be the vectors of vehicle position and velocity, as well as the vectors of payload angle and payload angular velocity. In [21], [24], [25], the state features for value function approximation are obtained by squaring the magnitudes of each of the vectors corresponding to these quantities. In [26], however, the features are instead taken to be a series of Chebyshev polynomials

constructed from the positions and velocities. The RL algorithms used are all variations of an action-value method, and therefore the allowable action space is limited to the discrete case. The discrete action space is generated by taking the allowable interval for the control action, and sampling it at a given number of sub-intervals. While [21], [24], [25] use the quadcopter's accelerations as the control action, [26] instead applies the selected action by increasing the commanded pitch and roll angles of the vehicle in response to changes in the payload state. While [25] presents results from simulations, [21], [24], [26] present data from an experimental testbed, by first generating the swing-free trajectories on a simulator, and then applying the trajectories to the testbed. Since the trajectories were not learned from interaction with the true environment, and rather from interaction with a simulator, the authors of [24] note that model uncertainties may cause a significant amount of discrepancy between the trajectory taken by the testbed and the optimal trajectory predicted by the simulator. This is especially true as trajectories become more complex.

One approach in [27] attempts to use a variation of a policy-gradient method to solve the payload trajectory tracking problem, making use of the continuous action-space of the quadcopter. It again uses the squared magnitudes of the state vectors as its features, and the quacopter acceleration vector as its control action. The approach assumes that the action-value function is a quadratic function of the action a, rather than a discrete value to be computed for each discrete action. The optimal control action is found by locating the maximum of the action-value function using Lagrangian interpolation, and selecting the action which corresponds to this maximum. The approach is tested in simulation, with environmental stochasticity modelled as random Gaussian external forces applied to the aircraft.

To the best of our knowledge, the problem has yet to be approached using artificial neural networks (ANNs) for value function approximation. One paper does use an ANN, but its purpose is for system identification, i.e. the ANN is charged with identifying the dynamics of the slung-load system, such that a linear, PID controller may then be designed to stabilize the aircraft itself [28]. In another approach, an ANN is used to learn the gains of a non-linear controller, where the structure of the controller is designed based on prior knowledge of the system dynamics [29].

The model-free RL approaches found in the literature present a few shortcomings which could be addressed by the controller proposed herein. While quadratic basis functions, i.e. basis functions consisting purely of some state characteristic squared,

appear to be a popular choice [21], [24], [25], [27], they are limited in that they can only be used to approximate quadratic value functions. While the assumption of a quadratic value function is required for the policy-based algorithm derived in [24], it is limiting in that, if the true value function is not quadratic, which could be due to the choice of reward function or to the system dynamics, this type of approximation may not be accurate enough to generate the truly optimal control policy. A large amount of varied orthogonal basis functions are sometimes preferred, such as the Chebychev polynomials used in [26] which are able to approximate a wider variety of functions. Furthermore, most of the proposed approaches rely on discretizing the action space, and storing a value function for each action. Not only does this limit the controller by forcing it to choose between actions from a discrete set, but it also requires hundreds if not thousands of value functions to be stored on-board the vehicle for adequate control. Finally, for the approaches which present results on an experimental testbed, all of the training appears to be performed on a simulator with no adaptation taking place on the experimental testbed itself. While most of the training can be performed in simulation, the agent should be allowed to learn in some capacity from the true environment and leverage the potential of the model-free control system, in order to account for model uncertainty, complex interactions in the system, stochasticity in the environment, etc.

2.2 Modelling of Slung Payload Systems

As mentioned, simulators can be extremely useful tools for designing model-free control algorithms. While no mathematical model is entirely accurate, models facilitate the selection of an appropriate algorithm and learning parameters, and provide insight into the basic dynamics of the proposed system. Luckily, aircraft systems carrying slung payloads have been modelled and studied extensively. The approaches typically used to obtain their equations of motion can be separated into two categories: Newton-Euler approaches and Euler-Lagrange approaches. Both approaches first require that the concept of frames of reference be defined, in order to properly describe the position and orientation of the aircraft and payload in three-dimensional space.

2.2.1 Euler Angles and the Aerospace Rotation Sequence

The information required to fully locate a rigid object in three-dimensional space is its linear position ("position") and angular orientation ("attitude"), yielding six

degrees of freedom. To this effect, a set of right-handed coordinate axes, referred to as the body frame and usually denoted by "B", is typically fixed to the center of mass of the object and allowed to rotate with it. The object's position and attitude are then determined by comparing the position and attitude of the body frame to those of *stationary* reference axes, i.e. to an inertial reference frame, often denoted by "E" [30]. For smaller-scale applications such as small UAVs, the inertial reference frame is usually taken to be fixed to the earth's surface with its horizontal axes parallel to the ground, and its vertical axis aligned with the direction of gravity. Accelerations of the inertial frame due to the earth's spin and revolution are ignored at this scale [31]. Using this approach, the aircraft's position can be located by a three-dimensional position vector beginning at the origin of the inertial frame and ending at the origin of the body frame, while the aircraft's attitude can be described by a 3×3 rotation matrix, relating the orientations of both frames:

$$ {}^{E}_{B}R = \begin{bmatrix} c_\theta c_\psi & -c_\phi s_\psi + s_\phi s_\theta c_\psi & s_\phi s_\psi + c_\phi s_\theta c_\psi \\ c_\theta s_\psi & c_\phi c_\psi + s_\phi s_\theta s_\psi & -s_\phi c_\psi + c_\phi s_\theta s_\psi \\ -s_\theta & s_\phi c_\theta & c_\phi c_\theta \end{bmatrix} \tag{2.1} $$

Here s_i and c_i are shorthand notation for $\sin(i)$ and $\cos(i)$, respectively. The subscript B and superscript E indicate that this matrix describes the rotation of the body frame with respect to the inertial frame, and ϕ, θ and ψ are the so-called Euler angles of the aircraft. The above rotation matrix allows vectors expressed in the body-frame to be mapped to the inertial frame, by pre-multiplication. If the opposite is required, i.e. mapping from the inertial frame to the body-frame, the above matrix is simply inverted to yield the opposite mapping. Since rotation matrices are orthogonal, the inverse is simply reduced to its transpose [30]. The full theory and derivation for the rotation matrix is detailed in Appendix A.1.

2.2.2 Newton-Euler Approach

Newton-Euler mathematical models are obtained using a vectorial approach. The equations of motion are derived for each mass in a system by applying Newton's Second Law to the system's translational coordinates, and Euler's Equations for the system's rotational coordinates [32].

Newton-Euler models are simple to understand intuitively, since they stem from the fundamental principles of rigid body dynamics. However, the approach requires that each body be analyzed in isolation, often yielding unknown reaction forces in the

equations of motion of an interconnected system. These require additional constraint equations in order to provide a unique solution [33]. Furthermore, Newton's Laws only hold for absolute accelerations, that is, accelerations taken with respect to an inertial frame of reference, while Euler's Equations hold in rotating reference frames, specifically when the body frame axes are aligned with the object's principal inertia axes [32]. The latter considerations can make Newton-Euler approaches slightly more difficult to follow when the system becomes complex, for example involving multiple interconnected masses or multiple non-inertial reference frames to describe the motions of those masses.

For systems such as aircraft carrying slung payloads, the Newton-Euler approaches detailed in the literature are all very similar, with the equations first applied to the aircraft alone, yielding the equations of motion for its position and attitude:

$$\sum \mathbf{F} = m\mathbf{a} \tag{2.2}$$

$$\sum \tau = I\dot{\mathbf{\Omega}} + \mathbf{\Omega} \times I\mathbf{\Omega} \tag{2.3}$$

where $\mathbf{F}$ is the sum of external forces on the aircraft, m is the mass of the aircraft, $\mathbf{a}$ is its acceleration in the inertial frame, τ is the sum of external torques on the aircraft, I is its inertia tensor and $\mathbf{\Omega}$ is its angular velocity about its principal axes [8], [34]–[36]. Since the aircraft is taken in isolation, the external forces and moments must include the tension force and torque exerted by the payload cable. Torque due to tension only arises if the suspension point is offset from the vehicle center of mass, which was not always found to be the case for models in the literature.

The equations of motion for the payload are then typically obtained using the same approach: its acceleration in the inertial frame is first derived from its position vector using vector kinematics, and Newton's Second Law is used to obtain the equations of motion. The payload is most often taken as a point mass, i.e. a mass with no rotational inertia, reducing Equation 2.3 to $\sum \tau = 0$, and consequently reducing the number of equations of motion required to characterize its dynamics. As mentioned, unknown reaction forces, in this case the tension in the payload cable which relates both masses, must be solved for with an additional constraint equation. This is most often accomplished by forcing torque equilibrium about the suspension point [9], [10], [18], [37].

2.2.3 Euler-Lagrange Approach

Euler-Lagrange mathematical models are obtained using a scalar, systematic approach which considers the system in its entirety, meaning that any external forces and moments in the equations of motion are those external to the system as a whole. The system's generalized coordinates are first defined, that is, a set of independent variables able to fully characterize the state. The total kinetic and potential energies of the system are then determined as functions of these coordinates, and the *Lagrangian* is defined as the total potential energy subtracted from the total kinetic energy. Note that the kinetic and potential energies in the Lagragian must be absolute, requiring that they be written from the perspective of a stationary reference frame. It is therefore practical to force the generalized coordinates to be those of a stationary frame of reference [33].

Systematic application of the *Euler-Lagrange equation* to each coordinate in the Lagrangian yields the equation of motion for each:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\boldsymbol{q}}}\right) - \frac{\partial L}{\partial \boldsymbol{q}} = \boldsymbol{Q} \tag{2.4}$$

where L is the Lagrangian, $\boldsymbol{q}$ is the vector of generalized coordinates, and $\boldsymbol{Q}$ is the vector of forces and torques external to the system which act in each of the coordinate directions. These are most often referred to as the generalized forces.

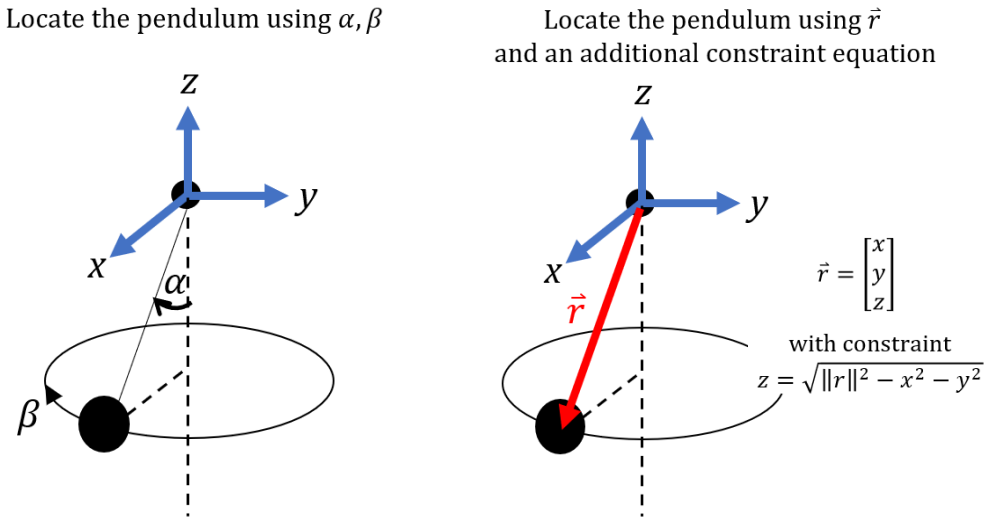
Euler-Lagrange approaches can be easily modified to account for any system constraints, and they effectively neglect any internal forces, making them well-suited to complex, multi-bodied, interconnected systems, such as aircraft with slung payloads. Models of the latter mostly begin by establishing the vector of generalized coordinates fully describing the positions and orientations of the aircraft and payload [15], [34], [38]. Other, highly complex dynamical models have been derived using this approach, for example where the payload cable is modelled as a series of n rigid links with mass, approximating a flexible cable [14]. By contrast, modelling the same type of system using a Newton-Euler approach requires an additional n equations which must be solved iteratively for the tension in each segment of the cable [2].

Alternatively, a combination of both approaches has also been used, where the aircraft is modelled using Euler-Lagrange formalism, while the payload is modelled using Newton-Euler formalism [10].

2.2.4 Choice of Payload Coordinates

The slung load, whether attached to a series of rigid links or to a single rigid cable, is likened to a spherical pendulum with a mobile attachment point (or a series of connected spherical pendulums in the case of a series of links). The analogous pendulum system makes the dynamics of the slung load easier to analyze and understand.

The simplest single pendulum system has two degrees of freedom, and its state is typically specified using two angles in spherical coordinates, or occasionally, using a three-component vector in cartesian coordinates. These representations are illustrated in Figure 2.1.



(a) Spherical coordinates (b) Cartesian coordinates

Figure 2.1: Typical payload coordinate choices.

Any representation using two angles such as that illustrated in Figure 2.1a, while simplest to implement, leads to singularities at the sphere poles (when one of the angles is zero), making numerical simulation notoriously troublesome [39], [40]. Cartesian representations such as the one illustrated in Figure 2.1b, on the other hand, require twice as many equations to characterize the dynamics: one for each vector component, and an additional equation to constrain the pendulum to the surface of a sphere, making derivation of the equations of motion a relatively cumbersome task [39]. These shortcomings appear to rarely be acknowledged in the literature for models of aircraft carrying slung payloads, except in [14] and [12] where a coordinate-free form of the Euler-Lagrange equation is used, modified specifically for mechanical systems constrained to the surface of a sphere [13], [39]. Alternatively, coordinate transfor-

mations can be used to eliminate singularities, as was observed for other types of systems involving spherical pendulums [41].

2.3 Payload State Sensing

While theoretical and simulation-based control algorithms for aircraft carrying slung payloads are prevalent in the literature, testing on experimental platforms is less widespread, but far more valuable for the validation of the control algorithm. Accurate knowledge of the payload state, that is, its position and velocity, is usually required for the stabilization algorithm. The vast majority of experimental platforms described in the literature use motion capture setups such as a *Vicon* or *Optitrack* motion capture system to estimate the position of the slung load [18], [24], [42]–[46]. An Optitrack setup is shown in Figure 2.2, and consists of several high-speed, infrared cameras transmitting to a central computer running a triangulation algorithm in order to locate the robot. These types of setups are not usually practical nor feasible for outdoor field operations, and control algorithms which successfully make use of on-board payload sensing are far more valuable.

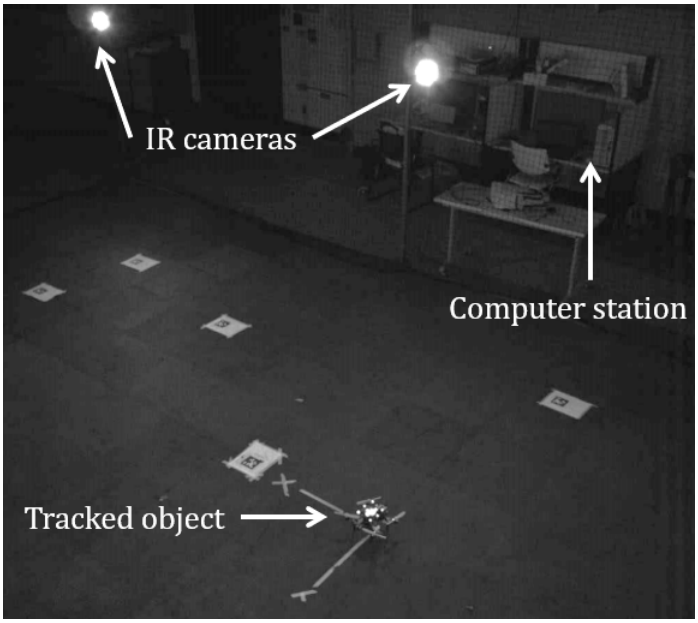


Figure 2.2: Capture of an Optitrack setup through one of the camera lenses.

A few experiments have successfully been conducted using on-board sensing, where the payload position is determined by data from one or several on-board cameras [12], [47], [48]. Camera data is sometimes combined with data from an inertial measurement unit (IMU) or indoor motion capture system to yield accurate sensing [12].

An obvious drawback for the latter setup is again the dependence on motion capture. Furthermore, vision-based sensing in general has been known to be computationally expensive, require larger amounts of battery power, have high-latency and have low transmission frequency, all of which are not ideal for small UAVs attempting to manoeuvre in an agile manner, with relatively high frequency controllers [3], [49]. For example, an extremely efficient image processing algorithm running on a Raspberry Pi microcomputer may take around 16 ms to process the location of a moving object using pattern recognition, and using a mobile camera (e.g. one mounted to the underside of a quadcopter). This does not include the time required to capture the image or transmit it to the microcomputer [50]. By contrast, a Raspberry Pi reading data from a sensor over a relatively slow I2C bus is able to read and process 8 bytes of data in approximately 0.16 milliseconds [51]. Novel sensor setups have recently been developed to estimate the payload state, such as cable angle sensors consisting of low-latency magnetic encoders. However, the data from these encoders must be fused with data from a higher-latency camera [49] or must be used in combination with control by a human operator to achieve a desired trajectory with minimal payload swing [52].

3. Background on Model-Free RL

Model-free RL is characterised by an impressive versatility and ability to devise solutions to a wide variety of exceedingly complex problems. In the particular case of control of physical systems, model-free RL can be used to generate optimal solutions for a control objective, without any modelling assumptions or prior knowledge of the system or its environment, and even if the system is not fully observable [6]. A summary of the theory on model-free RL is presented herein and outlines the classic RL framework devised by Richard Sutton, described further in [6].

3.1 Introduction

The goal of RL is to ensure that the controller, called the *RL agent*, is able to take a sequence of optimal or favourable actions to reach a desired goal, even under uncertainty or stochastic conditions. The sequence of actions that the agent takes is dictated by its *control policy*, denoted $\pi(a \mid s)$. The control policy is essentially an instruction for the agent on which action a it must take if it finds itself in a given state s. The model-free RL agent is expected to "learn" the optimal sequence of actions by updating its control policy for every possible state s, based on experience and interaction alone. Many RL-based controllers adapt their control policies in real-time, as data is sampled from the environment. These are called *online* RL controllers.

Most RL algorithms operate on the assumption that all states s possess the *Markov property*. That is, the state contains all the information that is necessary for the RL agent to make informed decisions about the future, regardless of past states. As an example, consider an autonomous car navigating to a specific point in a parking garage with multiple identical levels. The car happens to only possess vision sensors that map its immediate surroundings, and the latter are used to construct its state. For the controller to be "aware" of which level the car is located on, it would need to retain information about where the car has previously been (past states), in addition to the information provided by the vision sensors. For example, knowing the car began on the first floor and took an upward ramp, it could infer which level it is located on. For the previous example, the current state s_t, as provided by the vision

sensors, does not obey the Markov property, since it cannot provide all the information required to adequately navigate the parking garage and accomplish the task, without including some information about past states [53]. Note that the Markov assumption is not strictly necessary to the use of RL, but it does allow faster and more efficient algorithms to be applied.

The controlled dynamic system obeying the Markov property is said to behave as a Markov decision process (MDP), formally defined as follows:

$$s_{t+1} = f(s_t, a_t) \tag{3.1}$$

That is, the next state s_{t+1} is a function of s_t which contains all necessary information about the system in its current state, and of a_t, the current control action.

MDPs may be deterministic (only one next state is possible, given the current state and input) or stochastic (multiple next states are possible, given the current state and input). Stochastic MDPs may be more useful in the context of physical systems, since they are able to account for external disturbances or control uncertainty by assuming an element of randomness exists in the dynamics.

As an example, a car driving on a dirt road may be treated as a stochastic MDP: assuming its state s to be measured position and velocity, then s_{t+1} depends only on s_t, and on the magnitude of the control inputs (a_t) such as the throttle and steering angle. Furthermore, there may be a small probability of the car accidentally getting stuck in mud or being faced with a large wind gust, for example. In this case, a given throttle position and steering angle may result in the car ending up in a different position than if it was driving without any obstacles or disturbances. This idea is illustrated in Figure 3.1, where the action taken is throttling forward with no steering angle, but the state reached is different in each scenario. This probability of ending up in different "next states" given the same current state and inputs can be treated as stochasticity in the dynamics, and allows the RL agent to deal with environmental disturbances among other types of unpredictability, such as sensor noise, interference or malfunctioning hardware [27].

3.2 Value functions and policy evaluation

At each time step of a MDP, the RL agent takes an action dictated by its control policy, observes the state reached after taking that action, and assigns it a reward or

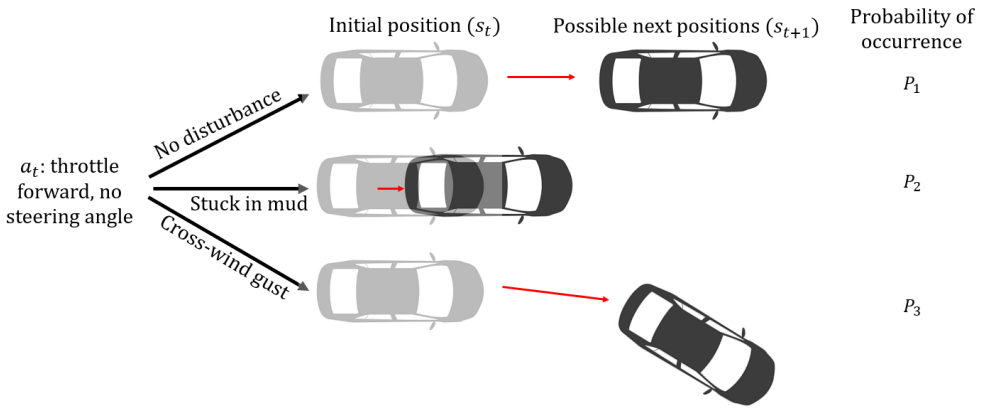


Figure 3.1: Diagram illustrating the concept of a stochastic MDP with a car driving on a dirt road.

cost value r depending on how favourable the state is. For example, the car on the dirt road may be tasked with reaching given GPS coordinates. When the desired location is reached, and is "observed" by the car's GPS, the RL agent may be rewarded with a positive value reward, for example $+1$, indicating that the state is favourable. On the other hand, if the car deviates off its desired trajectory, it may be penalized with a negative value reward, for example -1, indicating that the state is undesirable. The *return* for a MDP is defined as the sum of rewards from the current time t until the very end of the learning episode (the horizon). It is therefore a metric for evaluating how *good* a control policy is, by assigning it a numerical value. For an infinite-horizon process, the return is given by:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots \tag{3.2}$$

where G_t is the return for the current state, r is the reward which is computed at each time step, and γ is a discount factor which allows weighting of future rewards and can also force G_t to converge on a finite value, even when given an infinite-horizon task. Typically, $0 \leq \gamma \leq 1$, with $\gamma = 0$ indicating that the agent is only concerned with the immediate reward r_t, and $\gamma = 1$ indicating that long-term rewards are weighted equally and are therefore as valuable to the agent as immediate rewards.

Since many MDPs are stochastic in practice, especially for physical systems, a RL agent might follow the exact same policy several times from the same initial state, and get a different return G_t. Again taking Figure 3.1 as an example, the three scenarios of "no disturbance", "stuck in mud" and "cross-wind gust" could represent three separate attempts to test the same control policy, in this case, to always full throttle

forward with no steering angle. Each of the three attempts results in three different states, and therefore three different rewards at time $t+1$, resulting in different returns for this same control policy. Using only a single return G_t as a metric for evaluating a control policy is therefore not ideal.

Policies are typically evaluated instead using the *state-value function*. The latter is defined as the average value of the return for the current state s_t, obtained by always following policy π until the horizon. Like the return, it is a function of the initial state and is always associated to a given policy. Unlike the return, it is not subjected to probabilities and circumstance since an average is taken. The state-value function for a specified policy is given by:

$$V^{\pi}(s_t) = \mathbb{E}[G_t \,|\, s = s_t] = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots \,|\, s = s_t] \tag{3.3}$$

That is, the value of the current control policy, given that the agent begins in state $s = s_t$, is equal to the statistical expected value $\mathbb{E}$ of its return, i.e., the return averaged over all possible states and their likelihood of occurring.

RL agents update their control policies π in order to select actions which maximize the state-value function, for every possible state in the state-space. The policy which achieves this, denoted π^*, is called the optimal control policy, and is effectively the "best way" of achieving the goal specified by the reward function r. In order to find the optimal policy, the agent must be made aware of any actions that result in higher state-value functions, i.e. it must explore the possibilities. One way to achieve this is by using the *action-value function*. Like the state-value function, it is the average reward for beginning in state s_t, associated to a specific control policy. However, it is also a function of the initial action a_t, taken while in state s_t:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}[G_t \,|\, s = s_t, a = a_t] = \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \,|\, s = s_t, a = a_t] \tag{3.4}$$

As an example, consider a quadcopter whose state-space is a set of discrete locations on a planar grid, as illustrated in Figure 3.2. Rewards associated to each location on the grid are shown in bold at the center of the grid square. These might correspond to points of interest on a map which should be photographed by the quadcopter. The goal in this case would be to find the trajectory which visits all points of interest. Assuming the current control policy to be "always move right", four different trajectories are shown, where the quadcopter begins in the center of the grid (state s_5), first takes one of the possible actions a, and follows the current policy thereafter.

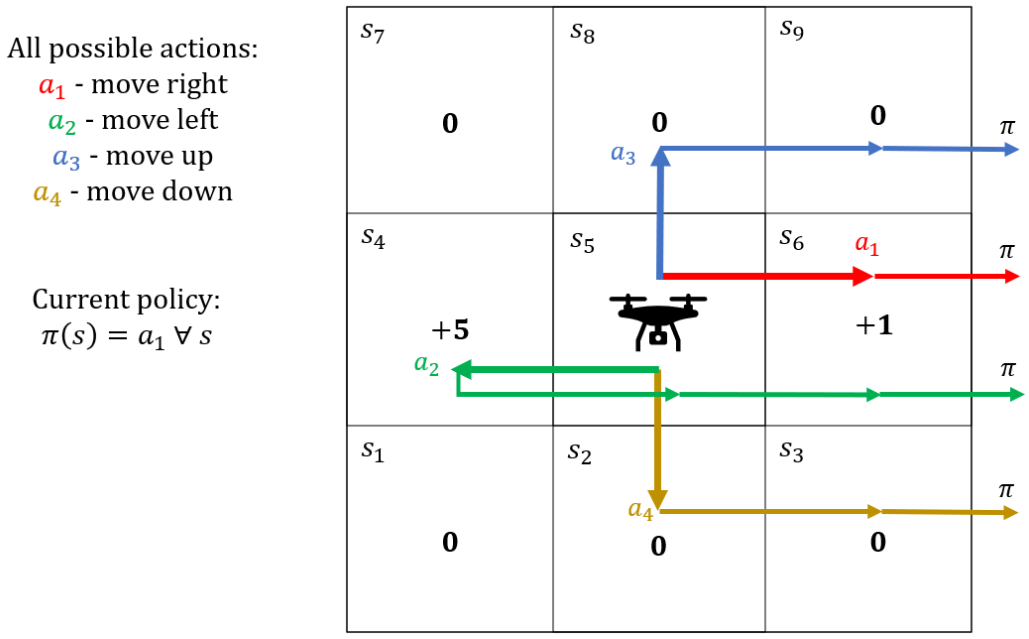


Figure 3.2: Illustration of the concept of action-value functions using a quadcopter on a grid.

It is assumed the learning episode ends when the quadcopter exits the grid. At a glance, it becomes evident that the action of moving left in state s_5, and then following the policy (the green trajectory), yields a higher value than any other. Formally, $Q^\pi(s_5, a_2) \geq Q^\pi(s_5, a) \; \forall \; a$. The action-value function must be obtained for every possible start state s, and every possible first action a in order to make informed decisions as to which action is optimal in every state.

Control policies can then be improved by applying the *policy improvement theorem*, which states that any action with an action-value which is higher than the current state-value must be a more favourable action than that dictated by the current policy. That is, if:

$$Q^\pi(s, a) > V^\pi(s) \quad \text{for some start state } s \tag{3.5}$$

let $\pi'(s)$ be the new, updated policy such that:

$$\pi'(s) = a \quad \text{for the same state } s, \text{ and the favourable action } a \tag{3.6}$$

then

$$V^{\pi'}(s) > V^\pi(s) \tag{3.7}$$

And the policy π' is an improvement upon π. Using the quadcopter example, the current policy for the starting state is $\pi(s_5) = a_1$, the action of moving right. However,

$Q^\pi(s_5, a_2) > V^\pi(s_5)$. The latter relationship becomes apparent when comparing the green trajectory with a total value of +6, and the red trajectory with a total value of +1 (assuming γ to be 1). An improved policy is therefore $\pi'(s_5) = a_2$.

In the case of model-free RL, the state-value and action-value functions for each state and action are initially unknown to the agent, and must be learned from interaction with the environment. This "environment" can be a simulator, a controlled laboratory setting or an outdoor area, for example. The particular algorithms by which an agent learns value functions and improves upon policies will be discussed in future sections. These algorithms are used to "train" the RL agent so it may eventually learn the optimal control policy.

3.3 Value function approximation

The concept of policy improvement detailed above is relatively straightforward to implement if numerical quantities can be directly obtained for the value functions. As mentioned previously, this would need to be done for each and every state in the state-space s, and each and every action in the action-space a in order to make informed decisions. This may seem simple to achieve in the case of a grid with numbered locations as in the quadcopter example of Figure 3.2. However, the problem becomes daunting in the case of real physical applications, where the state-space is often a continuum of values provided by (sometimes analog) sensors. Suppose the state of the quadcopter from Figure 3.2 is no longer given by a grid square number, but by continuous coordinates in the (x, y) plane. One can discretize the (x, y) plane into grid squares and assign each a number, but issues arise concerning the resolution of the grid, and the sheer amount of value function estimates needed to be stored in the system's memory: several for each location. Alternatively, one can use value function approximation (VFA).

The basic principle behind VFA is to express the value function as a parameterized, computable function of the state, rather than discretely storing a separate value for each state. That is:

$$\hat{V}^\pi = f(s, \mathbf{w_V}) \tag{3.8}$$

$$\hat{Q}^\pi = f(s, a, \mathbf{w_V}) \tag{3.9}$$

where the hat operator indicates an approximation of the true value, and $\mathbf{w_V}$ is a

parameter vector which operates on the state s and can be tuned to more accurately approximate the true value. The parameter vector can take many possible forms, for example neural network connection weights, split points in a decision tree, mathematical functions, etc. The state or action value can then be determined by computing the result of the approximate function $\hat{V}^\pi$ or $\hat{Q}^\pi$, for any state that the RL agent finds itself in. Concrete examples of value function approximation are presented later in this section. Note that, for the sake of simplicity, this section deals with approximating the state-value function V^π, but all theory and equations may also be applied to Q^π. Using VFA allows Q^π to be approximated for a single given action, but over the continuum of states in the state-space.

The problem now becomes one of tuning the parameter vector in order to ensure that the approximated value function approaches the true value function, for every state in the state-space. The problem can be approached as one of curve-fitting or regression analysis, by reducing some error metric, often taken to be the mean squared error in the case of VFA:

$$\text{MSE} = \mathbb{E}_\pi[(V^\pi(s) - \hat{V}^\pi(s, \mathbf{w_V}))^2] \tag{3.10}$$

where $\mathbb{E}$ once again denotes the expected value, i.e. the error averaged over all possible states and their likelihood of occurring under the current policy. Here $V^\pi(s)$ is the true value of policy π, and $\hat{V}^\pi(s, \mathbf{w_V})$ is the value approximated by VFA.

Several regression methods may then be used to update the parameters $\mathbf{w_V}$, with the goal of converging on the global minimum of the mean squared error function. Iterative techniques useful to online RL include recursive least squares and stochastic gradient descent [54]. As new true value samples, $V^\pi(s)$, are provided to the agent, the parameter vector $\mathbf{w_V}$ is updated in order to decrease the error between the true value samples and the current approximation. Again, note that for model-free RL, the true value samples are initially unknown to the agent, and must be learned from interaction and experience.

A simple example of VFA is illustrated in Figure 3.3, where the function approximator is simply a line passing through the origin. In this case, the constant dictating the slope of the line is the only parameter in $\mathbf{w_V}$. In this example, it is assumed that the position of a quadcopter along a single axis comprises the continuous state-space. As the quadcopter interacts with its environment, it learns the true state-value function associated to its control policy, for positions that it has previously visited (the black circles). The RL agent can then use VFA to fit the learned values to its function

approximator, and extrapolate the state-value function for states that it has not yet visited or has no prior knowledge of. It would then still be able to estimate the value function $\hat{V}^{\pi}$, even if it finds itself in an entirely new state.

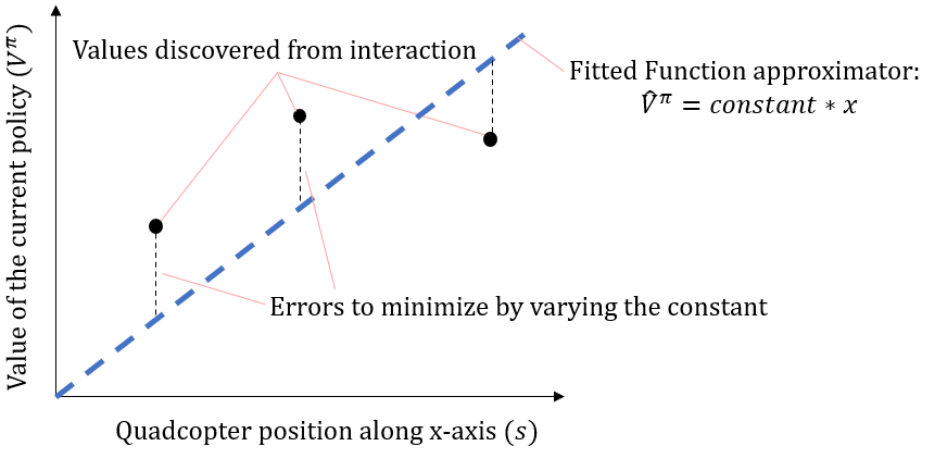


Figure 3.3: Illustration of value function approximation using the simplest linear function.

3.3.1 Linear value function approximation

In order to apply VFA and compute numerical quantities for the value function, each state the RL agent encounters must be represented by a unique, finite set of numerical values. This set of values is often referred to as the state's *features*. Features are effectively a way of "encoding" the characteristics of a state. When considering a dynamical system, the features of a state may be selected rather intuitively, as characteristics fully describing a state are already numerical properties. Feature selection becomes more difficult in applications such as facial recognition, stock market predictions, or backgammon, for example, since the state characteristics which may be important to the RL agent are not always clear, available information may be abundant, or states may be difficult to characterise using numbers (e.g. when dealing with colors, shapes or positions of pieces on a board) [55].

When the numerical characteristics are rather intuitive, and feature selection can be a fairly simple task, linear VFA is often first considered. Linear VFA is computationally efficient and has many convergence guarantees which make its use with model-free RL appealing. In essence, for linear VFA, the parametric function used to approximate the value function becomes a simple dot product:

$$\hat{V}^{\pi}(s, \mathbf{w_V}) = \mathbf{w_V}^T \Phi(s) \tag{3.11}$$

where $\Phi(s)$ is the features vector encoding state s, and $\mathbf{w_V}$, the parameter vector, is now a vector of constants (or "weights") which can be tuned with training of the RL agent. In this particular case, the approximated function $\hat{V}^{\pi}$ effectively becomes a linear combination of the state features. The features vector $\Phi(s)$, if comprising linearly independent quantities, can therefore be said to form a linear basis of the function space to which the approximated value function belongs [56].

For a dynamical system, it may be intuitive to use the numerical characteristics of the state themselves, e.g. position and velocity, as the state features. While this may allow some functions to be approximated, limitations arise in the types of functions which this feature representation can approximate, and in the fact that interactions between characteristics cannot be represented. The concepts of state characteristics, state features and linear bases is illustrated in Figure 3.4 for a simple one degree-of-freedom mass-spring-damper system. Various possibilities are shown for the features vector, with the functions that each is able to approximate.

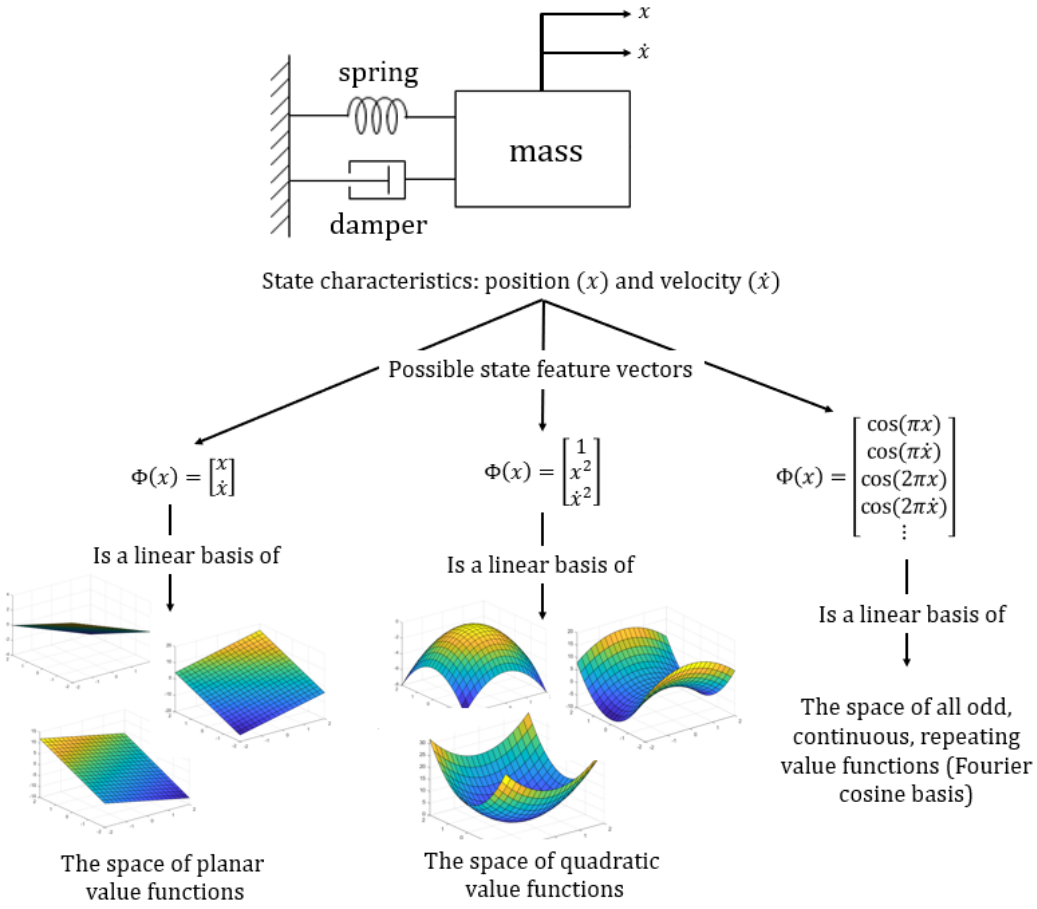


Figure 3.4: Concept of feature vectors and basis functions for linear VFA.

Here the obvious numerical values describing the state are the position x and velocity $\dot{x}$ of the mass. Using these quantities directly as the features vector, as shown on the left branch of Figure 3.4, allows only planar value functions to be approximated. That is, functions of the form $\hat{V}^{\pi} = ax + b\dot{x}$ where a and b are tunable weights. Furthermore, this type of feature representation does not allow interactions between the two variables to be represented. For example, consider the case where the goal of the agent is to steady the mass at $x = 0$. In this case, we would expect the value function to be larger for any state which leads to this goal. If the position is highly negative, we would want the velocity to be highly positive to approach $x = 0$. If the position is highly positive, a highly positive velocity would push the mass further away from the goal, and we would expect the value of this state to be smaller. Simple feature representations do not allow these "conditions" to be encoded. A high positive velocity will yield the same term in the value function, $b\dot{x}$, regardless of the value of the position. As such, using various functions of the state quantities allows different kinds of value functions to be approximated, and allows potential interactions to be represented. As an example, the middle and right branches of Figure 3.4 illustrate the more varied function spaces which can be approximated using quadratic and cosine functions, respectively. It is therefore important to not only select features which uniquely represent a state, but which also form a basis for a large and varied function space, in order to guarantee that the true value function can be accurately estimated to a certain degree of confidence. Knowledge and intuition about the system's behaviour may assist the engineer in selecting an appropriate basis.

Note that a linear function approximation is only as accurate in representing the true value function as its basis allows. Some applications may need extremely accurate approximation, requiring exceedingly large numbers of basis functions. As the dimension of the state-space increases, for example in a problem involving several masses instead of one, the length of the features vector may increase exponentially, subjecting linear VFA to the "curse of dimensionality" [6], [57]. In rare cases, the value function to approximate may also have jump discontinuities, which is difficult to manage with a linear basis [58], [59]. In these cases, non-linear approaches such as artificial neural networks (ANNs) are often used to converge on the parameter vector $\mathbf{w_V}$. While ANNs are able to approximate virtually any value function with a large degree of accuracy, and may even circumvent the issue of selecting relevant features entirely [60], they require far more computation to learn value functions, may be prone to overfitting, and do not possess the same convergence guarantees as linear VFA [6].

3.4 Learning the true value function

As mentioned in previous sections, for a model-free RL agent, the "true" value functions associated to a policy are initially unknown and must be learned from interaction with the environment before any decisions are made as to how to update the control policy. If the agent is not using VFA, the true values of every state must be learned, and if the agent is using VFA, it must be able to sample a "true" value for each state it visits, to fit its function approximator.

There are two main ways in which a RL agent can learn value functions from interaction with its environment. The first, and most obvious, is to run through many episodes of control in their entirety, with the same policy. These could be run on a simulator, or in a physical setting, depending on what the environment is chosen to be. Without changing the control policy, the agent will take the entire series of computed actions until the pre-determined control horizon, recording each state and reward in the process, and afterwards compute the return G_t for each state it has encountered under the current policy. Since the value function is simply an average of the returns, as defined in Equation (3.3), running through enough episodes and averaging returns guarantees that the agent will converge on the true value functions eventually, and then knowing these, be able to make informed decisions. Taking the quadcopter grid of Figure 3.2 as an example, in order to learn the action-value function $Q^\pi(s_5, a_2)$, the quadcopter would have to follow the green trajectory several times, computing the return each time, and average these returns to conclude that $Q^\pi(s_5, a_2)$ is in fact +6. Learning methods which use this approach are named *Monte Carlo* methods, after the popular gambling location, since, when the environment is stochastic, chance plays a major role in the outcome of the learning [61].

While Monte Carlo methods are guaranteed to converge on a true value, they require the agent to run through many episodes of experimentation before it can update its control policy, which may take a very long time. Furthermore, they cannot be used for infinite-horizon processes, since they require an end to the learning episode in order to compute the returns. Examples of infinite-horizon processes include the control of power plant machinery, chemical plant machinery, or traffic lights, to name a few, since these do not have a clear end to the control episode. Temporal-Difference (TD) learning, the second main approach in learning value functions, addresses these issues by gradually learning the true value functions in real-time, as the agent explores the environment. Taking the definition of the value function in Equation (3.3), and

factoring out the discount factor γ yields:

$$\begin{aligned} V^{\pi}(s_t) &= \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots \mid s = s_t] \\ &= \mathbb{E}[r_t + \gamma \left(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \right) \mid s = s_t] \end{aligned} \tag{3.12}$$

Substituting in the definition of the return in Equation (3.2),

$$V^{\pi}(s_t) = \mathbb{E}[r_t + \gamma G_{t+1} \mid s = s_t] \tag{3.13}$$

Or, by the definition of the state-value function,

$$V^{\pi}(s_t) = \mathbb{E}[r_t + \gamma V^{\pi}(s_{t+1})] \tag{3.14}$$

That is, the value of the current state is simply the expected value of its reward added to the discounted value of the next state. This expression is sometimes called the *Bellman Operator*. TD learning begins with an initial "guess" as to the value of every state, and then attempts to leverage the above result in order to quickly converge on an accurate estimate of the true value. To learn in real-time, the true measured reward r_t is used as an unbiased sample of its expected value. That is, at time t, it is assumed that:

$$V^{\pi}(s_t) = r_t + \gamma V^{\pi}(s_{t+1}) \tag{3.15}$$

and the expected value is dropped. The expression on the right side of the equality is called the TD target, and it is what the TD algorithm perceives as being the "true value" of state s_t. The temporal-difference error is then defined by rearranging the above expression:

$$\delta_t = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t) \tag{3.16}$$

That is, the TD error is given by the TD target (the "true value" of the current state) subtracted by the agent's estimate of the value of the current state. The computed TD error is then used to update the estimate of the value to bring it closer to the "true value":

$$V^{\pi}(s_t) \longleftarrow V^{\pi}(s_t) + \varepsilon \, \delta_t \tag{3.17}$$

Where ε is a small step size for the TD update. As more states are visited several times, the TD error eventually converges to zero, indicating that the agent's best "guess" for the value functions will approach the true values and Equation (3.14) will hold. Again, note that while the above equations apply to the state-value function $V^{\pi}(s)$, they may also be applied in the exact same way to the action-value function

$Q^\pi(s,a)$. As an example, again consider the green trajectory in the quadcopter grid of Figure 3.2, which has been redrawn in Figure 3.5 for the sake of clarity.

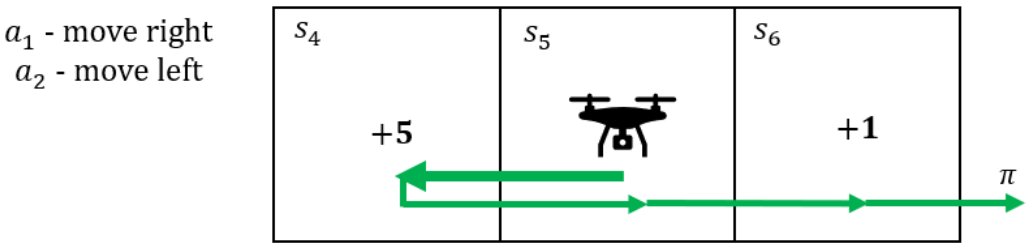


Figure 3.5: Illustration of the green trajectory from the quadcopter grid example of Figure 3.2.

The trajectory can be defined as a series of state-action-reward tuples. Beginning in state s_5, the trajectory is given by the following tuples:

$$\left((s_5, a_2, 0), (s_4, a_1, +5), (s_5, a_1, 0), (s_6, a_1, +1), \text{end}\right) \tag{3.18}$$

where a_1 is the action of moving right by one grid square, and a_2 is the action of moving left by one grid square. Assuming an initial guess of 0 for all action-value functions, a step size ε of 1 and a discount factor γ of 1, the first TD update for the action-value function of the first tuple is given by Equation (3.17):

$$Q^\pi(s_5, a_2) \longleftarrow Q^\pi(s_5, a_2) + \delta_t \tag{3.19}$$

where δ_t is given by Equation (3.16):

$$\begin{aligned} \delta_t &= r_t + Q^\pi(s_{t+1}, a_{t+1}) - Q^\pi(s_t, a_t) \\ &= r_t + Q^\pi(s_4, a_1) - Q^\pi(s_5, a_2) \\ &= 0 + 0 - 0 \\ &= 0 \quad \text{for the current estimates of } Q^\pi \end{aligned} \tag{3.20}$$

And the estimate of $Q^\pi(s_5, a_2)$ remains unchanged for this first time step. Proceeding the same way for the next time step,

$$\begin{aligned} \delta_{t+1} &= r_{t+1} + Q^\pi(s_{t+2}, a_{t+2}) - Q^\pi(s_{t+1}, a_{t+1}) \\ &= r_{t+1} + Q^\pi(s_5, a_1) - Q^\pi(s_4, a_1) \\ &= 5 + 0 + 0 \\ &= 5 \end{aligned} \tag{3.21}$$

And the new estimate of $Q^\pi(s_4, a_1)$ is:

$$\begin{aligned} Q^\pi(s_4, a_1) &\longleftarrow Q^\pi(s_4, a_1) + \delta_{t+1} \\ Q^\pi(s_4, a_1) &\longleftarrow 5 \end{aligned} \tag{3.22}$$

After only two time steps in the first episode, a value function has already been stepped towards its true value.

Note that TD learning depends heavily on the system possessing the Markov property, since, when assigning value estimates, it operates on the assumption that the value of the current state s_t relies heavily on the value of the next state s_{t+1}, and the latter is solely a function of the action taken to reach this next state. Monte Carlo methods do not strictly require that the system be Markov since they do not leverage any assumptions and simply compute the result. Furthermore, while Monte Carlo methods update their estimates of the true value function based on computed averages of a true, measured return, it becomes obvious from the above example that the "true value" target for TD methods is in fact another estimate (that of the value of the next state, which is initially also unknown). This process of updating estimates based on other estimates is often called "bootstrapping". Bootstrapping can lead to biased estimations, and the estimates do not converge as reliably to the true value in the way Monte Carlo methods do. This is especially true if the value functions are approximated using VFA. For the case of VFA, the agent must consult the value function approximator in order to generate the estimates $V^\pi(s_t)$ and $V^\pi(s_{t+1})$ before computing the TD error and updating the values. Fortunately, convergence to a near-optimal solution can still be guaranteed. These guarantees, and the way in which VFA and TD learning interact will be explored further in Chapter 5.

The algorithm detailed above is often called TD(0) or one-step TD since only a single sample of the reward is used to compute the TD error. It is however possible to break down the Bellman Operator further by factoring out the discount factor γ on a further time step. For example, for a two-step TD update, the TD target becomes $V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 V^\pi(s_{t+2})$. In this case, the TD algorithm has to wait until two rewards have been measured before making the update. As even more steps are taken before updating, bootstrapping is lessened, and the TD target begins to approach the true Monte Carlo return. These algorithms, called n-step TD algorithms, attempt to address the bias in the estimations which is inherent to TD learning.

3.5 Action-value methods

Regardless of which approach is used to obtain a value function estimate, the RL agent must then use the learned value functions to make decisions as to how to improve its control policy. The algorithms used to converge on the optimal control policy, or "train" the RL agent, mostly fall into two broad categories: action-value methods and policy-gradient methods.

Action-value methods explicitly make use of the action-value function Q^π, as defined in Equation (3.4). These methods require that a value of Q^π be available for every possible action that the RL agent could take, in every possible state. The policy is then improved according to the policy improvement theorem in Equation (3.5). If using value function approximation, action-value methods still require that a value of Q^π be computable for every possible action. For example, if the quadcopter of Figure 3.2 had to navigate a plane of continuous (x, y) coordinates, it would have to compute four values of Q^π by VFA, one for each of its possible actions, at every time step of learning. It would then select the action corresponding to the highest Q^π as the best action.

Unfortunately, the nature of such algorithms require that the action-space be discrete. That is, only a small, finite number of control actions are possible, since large or continuous action-spaces would require a large number (or an infinite number) of Q^π functions to be computed. This essentially limits the power of action-value methods since they are unable to exploit the full range of control signals inherent to dynamical systems. Consider the mass-spring-damper system of Figure 3.4: its goal of reaching a neutral state could be achieved much more efficiently if the agent could apply any force to the mass given a continuum of values, rather than if it was limited to discrete force values.

Furthermore, action-value methods run into difficulties if the control policy is required to be stochastic: Stochastic control policies are essentially probability distributions, which allow the agent to select between many possible actions if it finds itself in a given state. Stochastic control policies are extremely useful in RL, since they allow the agent to "explore" the effect of taking different actions from the same start state [6]. Some action-value methods called "ϵ-greedy" algorithms attempt to incorporate stochastic control policies by forcing the RL agent to select the action with the highest Q^π value only most of the time [6]. At each time step, the RL agent being trained with

an ϵ-greedy algorithm may instead select a random action with a small probability ϵ, with the purpose of exploring the value of that action. For example, if the quadcopter from figure 3.2 is being trained with an ϵ-greedy action-value method with $\epsilon = 0.05$, it will select the action with the highest Q^π (the optimal action) 95% of the time, and 5% of the time, it will select a random action between its four possible actions, with equal probability, in order to explore the rewards associated to these different actions. The drawback of ϵ-greedy algorithms is that they never truly converge to an optimal policy, because the ϵ probability will always force the RL agent to take a random action with a small probability, even when the control policy has converged. The ϵ probability can also not be "tuned" for specific states, and the agent has the same probability of selecting a random action regardless of which state it is in. Being able to tune the probabilities of selecting certain actions given certain states could be extremely useful in cases where random chance plays a role in the system dynamics. These limitations are what policy-gradient methods attempt to address.

3.6 Policy-gradient methods

Policy-gradient methods do not update the policy by looking up every possible action-value function and selecting the highest one. Instead, the control policy becomes a "parameterized policy". In an approach extremely similar to VFA, the control action a is computed from a parameterized function of the state s. Mathematically,

$$a_t^\pi = f(s_t, \mathbf{w}_a) \tag{3.23}$$

Or, the action taken at time step t is a function of the current state and of another parameter vector $\mathbf{w}_a$.

Recall that for linear VFA, the value parameter vector $\mathbf{w_V}$ is updated to minimize an error metric between samples of the true value function and the approximated value function. This is achieved by some form of linear regression. For policy-gradient, the goal of the parameterized policy is not to approximate an existing function by fitting, but rather to generate the "best" control action from a continuum of actions, for any state the agent may encounter. As such, policy gradient approaches will instead update the action parameter vector $\mathbf{w}_a$ with the goal of maximizing a performance metric, rather than minimizing an error metric. As stated previously, the most obvious performance metric for a control policy is its state-value function, and therefore, many policy gradient approaches step their parameter vectors in the direction of an

increase in the value function (in the direction of the positive gradient of the value function, with respect to $\mathbf{w}_a$). A simple example is illustrated in Figure 3.6, using the same planar quadcopter scenario as in Figure 3.3.

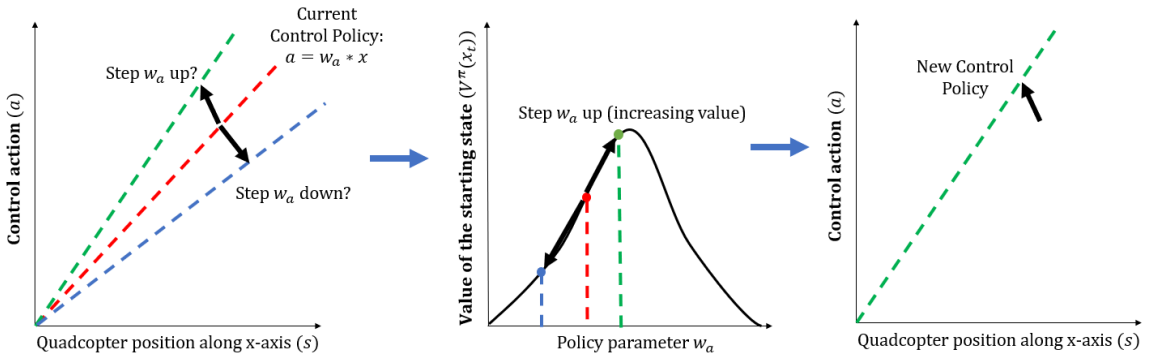


Figure 3.6: Illustration of the concept of policy improvement using value gradients.

In this example, the control policy is parameterized by a straight line going through the origin, with the slope of the line w_a being its only parameter. This could represent the agent requesting a pitching torque of magnitude a as a linear function of the quadcopter's position x, perhaps with the goal of maintaining a hover around $x = 0$. At every learning step, the constant w_a can either be increased (the green line), or decreased (the blue line) to generate a new control policy. The policy gradient algorithm "chooses" to update its policy by increasing the constant, since it has found that an increase in w_a results in an increase of the state-value function for the current position x.

A new issue now arises: Obtaining the gradient of the value function with respect to the policy parameters is not straightforward, since the value function depends on rewards obtained in future states, which in turn are functions of the policy, which is updated and changed in some unknown way at every learning step. Thankfully, the Policy Gradient Theorem (PGT) demonstrates that the gradient of the value function can be computed from the gradient of the policy itself, which is much easier to obtain [6], [62]. By the PGT, the gradient of the value function for the current policy parameterized by w_a, assuming the start state is $s = s_t$, is given by:

$$\nabla V^\pi(s_t) = \mathbb{E}_\pi \left(V^\pi(s_t) \, \nabla \ln \pi \left(a_t | s_t \right) \right) \tag{3.24}$$

Here $\mathbb{E}$ again denotes the expected value taken over all subsequent states and possible actions, based on their probabilities of occurring under the current policy. The gradient of the value function, like the value function itself, depends on which states are

encountered, which actions are chosen from the policy and their likelihood of occurring. However, the PGT demonstrates that only the current policy, and the policy's gradient with respect to its parameters, are required. That is, it is not necessary to know how the future visited states change when the policy changes. Note that Equation (3.24) applies for stochastic policies, that is, policies where more than one action can be selected, given a state. For deterministic policies, where the policy always computes the same action for a given state, the PGT is written slightly differently. An example of a deterministic policy is the one from the quadcopter example of Figure 3.6, where the control action is given by the equality $a = \mathrm{w}_a x$. The full derivations of both the stochastic PGT and deterministic PGT can be found in [62] and [63], respectively.

Policy-gradient algorithms offer many advantages over action-value methods. Besides allowing continuous action spaces, they can also be used to generate stochastic policies with varying probabilities depending on the state, which is useful for stochastic environments. The policy parameterization can also be tailored to the preferences of the engineer, yielding specific behaviours. For example, a quadratic equation rather than a line would force the quadcopter of Figure 3.6 to generate higher torques further away from the goal, and make its behaviour more aggressive. The only restriction placed on the policy is that it must be differentiable with respect to its parameters.

4. Modelling

A simplified model of the system to be controlled is developed in the present section, for the purposes of selecting an appropriate model-free algorithm and simulating the effect of different learning parameters and approximation methods on the general behaviour of the control system and dynamics. The model can also be used to train the adaptive controller to reach a first stable, converged solution which will later be useful to the experiment. The latter concept is discussed further in Chapter 5. The slung load transportation will be conducted using a quadcopter. While the control algorithm could be implemented on any rotorcraft, a quadcopter is relatively simple to model and control, and inexpensive to build, and will provide a good basis for an experimental testbed. A simple coupled model of a quadcopter and slung payload is first derived, after which the full model equations are reduced to yield planar equations.

4.1 Plant Model

The following section details the mathematical modelling of an "X-Flyer" configuration of quadcopter carrying a payload suspended by a cable. "X-Flyer" is often used to denote quadcopters for which the pitching and rolling axes of the vehicle body frame do not align with the motor arms, and are instead located exactly between the motor arms. This allows pitching and rolling moments to be generated by all four propellers, rather than only two.

4.1.1 Reference Frames and Assumptions

Three reference frames are first defined in order to properly describe the masses in the system. The first is the inertial frame {E}, described by the coordinate axes *x, y, z*. The second is the body frame {B}, which is fixed to the center of mass of the quadcopter, and described by the coordinate axes x_b, y_b, z_b. The third is the payload-fixed frame {P} which is fixed to the center of the payload, and described by the coordinate axes x_p, y_p, z_p.

While the goal is to obtain a model for simulation, model-free training on the physical system should enable it to overcome model uncertainties. Several assumptions are

therefore made in order to significantly simplify the mathematical model of the plant, and therefore make the simulator more efficient and easier to implement:

- The quadcopter's mass is concentrated at its center;
- The thrust force/moment pair generated by each of the four propellers act at equal distances from the quadcopter's center of mass;
- The quadcopter is rigid and symmetrical;
- The payload behaves as a point mass;
- The cable by which the payload is attached is rigid and massless;
- The cable's attachment point on the quadcopter body is coincident with the quadcopter's center of mass;
- Aerodynamic drag on the payload resulting from propeller downwash is neglected.

It follows from the rigid and massless cable assumption that this model will not be able to capture complex cable dynamics. These may include cable resonance behaviours or complex vibration patterns which are inherent to flexible systems. If any unmodelled behaviours do in fact have a significant impact on the state of the slung payload, it is expected that the model-free control algorithm derived in the next chapter will adjust its output to account for these model uncertainties, as it operates on the physical system.

A free-body diagram showing the geometry and reference frames of the system is illustrated in Figure 4.1.

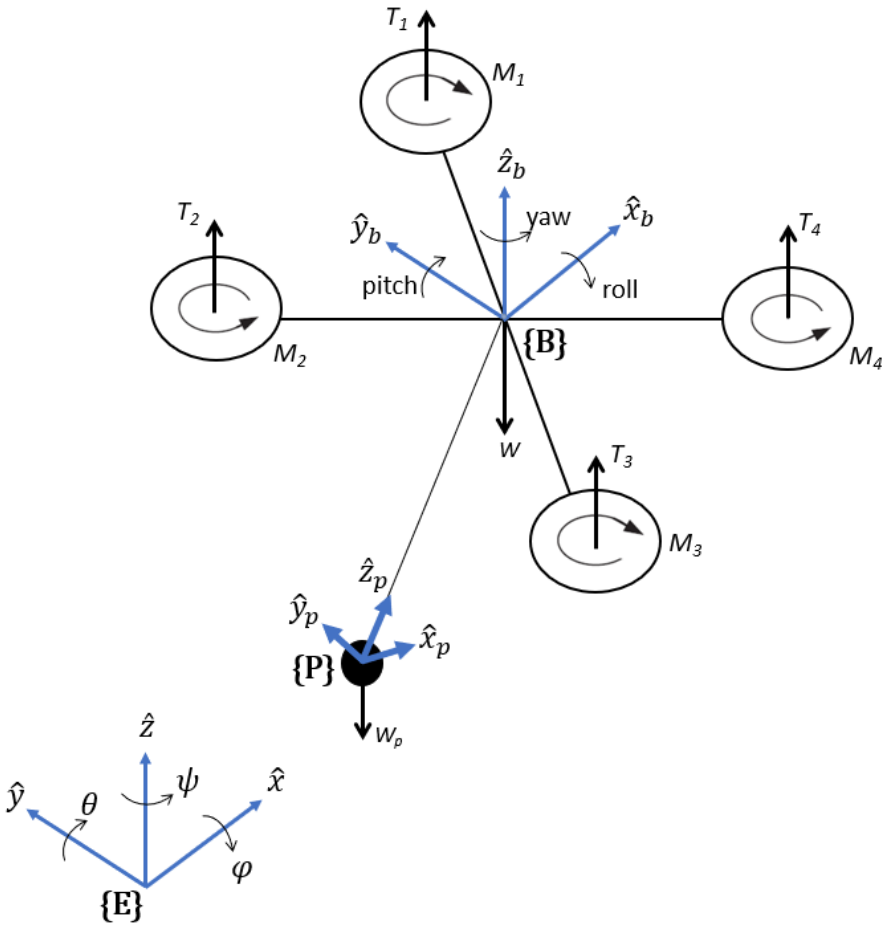


Figure 4.1: Free-body diagram of the quadcopter with slung payload.

4.1.2 State Variables

The variables x, y and z are used to describe the position of the quadcopter's center of mass in the inertial frame. The variables ϕ, θ and ψ are used to describe the attitude of the body frame in relation to the inertial frame, using the aerospace rotation sequence described in Appendix A.1.

The load state was chosen to be modelled using two angles in spherical coordinates, since it would yield the most compact form of the equations of motion. However, as is explained in section 2.2.4, this choice of representation yields singularities at the sphere poles. Furthermore, when using conventional ISO coordinates, the singularities occur at the equilibrium points. This is an especially difficult problem for payload stabilization simulations, where the goal is to bring the slung load to its stable equilibrium. To this effect, a set of unconventional spherical coordinates is used to locate the slung payload. These are illustrated in Figure 4.2. Here the payload-fixed frame

has been redrawn to be coincident to the attachment point, simply to illustrate its rotations in a clearer way.

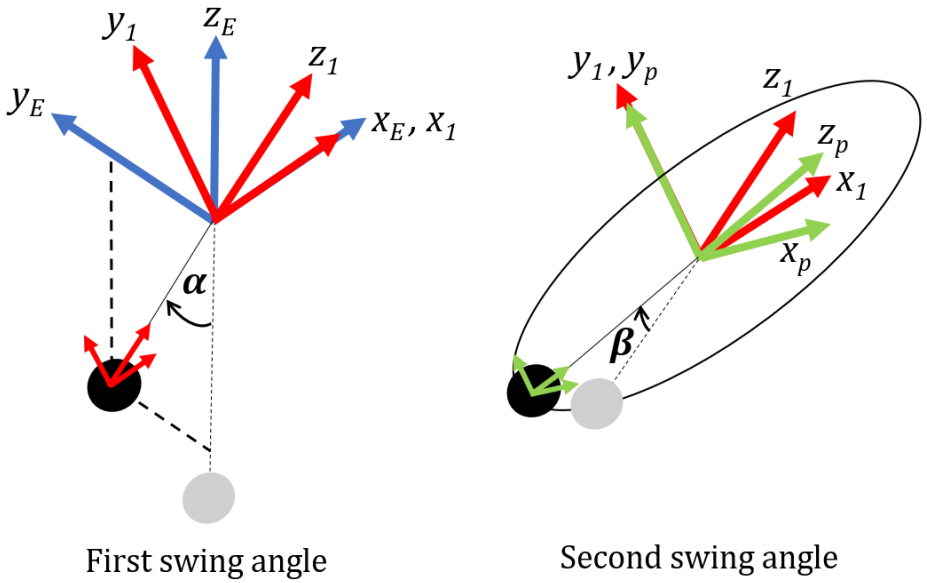


Figure 4.2: Diagram showing the payload frame rotated through angles α and β.

The load begins in its stable equilibrium, that is, directly beneath the center of mass of the quadcopter. In this position, the payload-fixed frame is in the same orientation as the inertial frame $\{E\}$. The first angle α represents the load's swing angle in the inertial y-z plane (the "polar" angle). As the load swings in this direction, the payload-fixed frame undergoes a right-handed rotation about the x-axis by the same angle α, yielding intermediate frame 1, shown in red. The second angle β is then used to represent the load swing angle in the intermediate x-z plane (the "azimuthal" angle). As the load swings in this plane, intermediate frame 1 undergoes a right-handed rotation about its y-axis by the same angle β, yielding the final orientation: the payload-frame $\{P\}$ shown in green. Note that defining the spherical coordinate system in this way does not eliminate the singularity in the equations of motion, but shifts it so that it occurs far away from the equilibrium point, at a load swing angle of 90 degrees with respect to the inertial y-z plane (i.e. $\beta = -90°$, $\beta = 90°$). This was deemed sufficient for the present application, since, in experiments on payload swing angle control, the payload angle has rarely been seen to exceed 65 degrees [12].

Since the payload frame orientation is described by two successive rotations, it becomes straightforward to derive a rotation matrix for converting from the payload-fixed coordinates to the inertial coordinates (the starting reference frame). The approach used is identical to that used to derive the rotation matrix of the quadcopter

with the aerospace rotation sequence, which is detailed in Appendix A.1. The payload rotation sequence is illustrated using planar diagrams in Figure 4.3 for the purpose of deriving a rotation matrix.

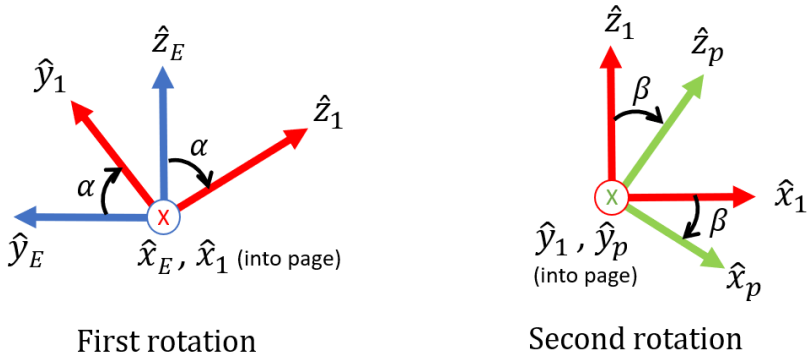


Figure 4.3: Representation of the payload rotation sequence.

Beginning with the load frame coincident to the inertial frame, the load frame first undergoes its right-handed rotation about x, as shown on the left side of Figure 4.3. The rotation matrix for this first rotation is given by:

$$ {}^{E}_{1}R(\alpha) = \begin{bmatrix} {}^{E}\hat{x}_1 & {}^{E}\hat{y}_1 & {}^{E}\hat{z}_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\alpha & -s_\alpha \\ 0 & s_\alpha & c_\alpha \end{bmatrix} \tag{4.1} $$

That is, the matrix is obtained by concatenating the unit vectors of frame 1, and expressing them using the unit directions of the inertial frame, as denoted by the left superscript E. Here s_i and c_i are again shorthand notation for $\sin(i)$ and $\cos(i)$, respectively. The next step involves a right-handed rotation around the intermediate y axis, as illustrated on the right side of Figure 4.3. Its rotation matrix is given by:

$$ {}^{1}_{P}R(\beta) = \begin{bmatrix} {}^{1}\hat{x}_p & {}^{1}\hat{y}_p & {}^{1}\hat{z}_p \end{bmatrix} = \begin{bmatrix} c_\beta & 0 & s_\beta \\ 0 & 1 & 0 \\ -s_\beta & 0 & c_\beta \end{bmatrix} \tag{4.2} $$

Post-multiplying the rotation matrices in turn yields the matrix for converting from

payload-frame coordinates to inertial frame coordinates:

$$
\begin{aligned}
{}^{E}_{P}R(\alpha,\beta) = {}^{E}_{1}R(\alpha)\,{}^{1}_{P}R(\beta) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\alpha & -s_\alpha \\ 0 & s_\alpha & c_\alpha \end{bmatrix} \begin{bmatrix} c_\beta & 0 & s_\beta \\ 0 & 1 & 0 \\ -s_\beta & 0 & c_\beta \end{bmatrix} \\
&= \begin{bmatrix} c_\beta & 0 & s_\beta \\ s_\alpha s_\beta & c_\alpha & -s_\alpha c_\beta \\ -c_\alpha s_\beta & s_\alpha & c_\alpha c_\beta \end{bmatrix}
\end{aligned} \tag{4.3}
$$

Defining α and β as rotations taken relative to the inertial frame reduces the complexity of the equations of motion, since the load state does not rely on the quadcopter's attitude. Furthermore, for load stabilization, it is the angles with respect to these earth-fixed axes which must be minimized (the absolute swing angles), which makes this definition practical and intuitive as well. Typically, the frame from which the angles are measured is chosen to be that of the vehicle (frame $\{B\}$). The latter approach mimics what an onboard sensor would realistically measure, and what a controller would use to compute its control signal. However, in this case, the frame chosen for the model is irrelevant, since the control system does not use any aspect of the model equations themselves to generate a control signal. The purpose of the derived model is merely to study the effect of different model-free approaches and parameters on the general behaviour of the system. It is therefore trivial to select the representation which will yield the most compact form of the equations of motion, and which make the most sense intuitively for load stabilization, such that it can be easily implemented in a simulator.

The quadcopter-slung-load system therefore has eight degrees of freedom and its vector of generalized coordinates is given by:

$$
\mathbf{q} = \begin{bmatrix} x & y & z & \phi & \theta & \psi & \alpha & \beta \end{bmatrix}^T \tag{4.4}
$$

4.1.3 Euler-Lagrange Model

The equations of motion of the system are obtained using Lagrangian mechanics, which were deemed more appropriate due to the system's multi-bodied nature. The quadcopter's translational and rotational states are first described using their own

vectors, for convenience in derivations:

$$\xi_{\mathbf{B}} = \begin{bmatrix} x & y & z \end{bmatrix}^T \tag{4.5}$$

$$\eta = \begin{bmatrix} \phi & \theta & \psi \end{bmatrix}^T \tag{4.6}$$

The Lagrangian of the system is then defined as the sum of kinetic energies minus the sum of potential energies, and is given by:

$$L = KE - PE \tag{4.7}$$

$$L = \frac{1}{2} m \dot{\boldsymbol{\xi}}_{\boldsymbol{B}}^{\ T} \dot{\boldsymbol{\xi}}_{\boldsymbol{B}} + \frac{1}{2} m_p \dot{\boldsymbol{\xi}}_{\boldsymbol{P}}^{\ T} \dot{\boldsymbol{\xi}}_{\boldsymbol{P}} + \frac{1}{2} \boldsymbol{\Omega}_{\boldsymbol{B}}^T I \, \boldsymbol{\Omega}_{\boldsymbol{B}} - mgz - m_p g z_p \tag{4.8}$$

The first term of Equation (4.8) is the translational kinetic energy of the quadcopter, m being its mass and $\boldsymbol{\xi_B}$ being its linear position vector in inertial frame coordinates. The second term is the translational kinetic energy of the payload, m_p being its mass and $\boldsymbol{\xi_P}$ being its linear position vector in inertial frame coordinates. The third term is the rotational kinetic energy of the quadcopter, I being its 3×3 inertia tensor and $\boldsymbol{\Omega_B}$ being its angular velocity vector about its center of mass. The payload has no rotational kinetic energy since it is assumed to be a point mass. Finally, the last two terms of the Lagrangian represent the gravitational potentials of the quadcopter and payload. Each element of the Lagrangian must now be defined as functions of the generalized coordinates or of known geometric quantities, before the equations of motion can be derived.

The payload position can be expressed as a vector sum of the quadcopter's position and of the payload's position with respect to the quadcopter. The payload frame's z axis is always oriented towards the attachment point, regardless of the load angles. Therefore, in payload-frame coordinates, the vector of the load position with respect to the quadcopter is simply given by:

$$^{\boldsymbol{P}}\boldsymbol{\xi}_{\boldsymbol{P/B}} = \begin{bmatrix} 0 \\ 0 \\ -\lambda \end{bmatrix} \tag{4.9}$$

where λ is the length of the load cable. The load-frame rotation matrix can then be

used to express this vector in inertial frame coordinates:

$$^{\boldsymbol{E}}\boldsymbol{\xi}_{\boldsymbol{P/B}} = {}^{E}_{P}R(\alpha,\beta)\,{}^{\boldsymbol{P}}\boldsymbol{\xi}_{\boldsymbol{P/B}} = \begin{bmatrix} c_\beta & 0 & s_\beta \\ s_\alpha s_\beta & c_\alpha & -s_\alpha c_\beta \\ -c_\alpha s_\beta & s_\alpha & c_\alpha c_\beta \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -\lambda \end{bmatrix} = \begin{bmatrix} -\lambda s_\beta \\ \lambda s_\alpha c_\beta \\ -\lambda c_\alpha c_\beta \end{bmatrix} \tag{4.10}$$

The payload position in the inertial frame is therefore given by:

$$\boldsymbol{\xi_P} = \boldsymbol{\xi_B} + \boldsymbol{\xi_{P/B}} = \begin{bmatrix} x + x_{P/B} \\ y + y_{P/B} \\ z + z_{P/B} \end{bmatrix} = \begin{bmatrix} x - \lambda s_\beta \\ y + \lambda s_\alpha c_\beta \\ z - \lambda c_\alpha c_\beta \end{bmatrix} \tag{4.11}$$

The vector $\boldsymbol{\Omega_B}$ was defined as the quadcopter's angular velocity about its own center of mass, i.e. about each of its body frame axes. It must however be expressed using generalized coordinates. Therefore, body-frame angular velocities must be defined using Euler angles in order for the Lagrangian to be usable. The angular velocities, often called the pitch, roll and yaw rates of the quadcopter, are related to the Euler angle rates by Euler's kinematic equations for rotating rigid bodies:

$$\boldsymbol{\Omega_B} = P\dot{\boldsymbol{\eta}} \tag{4.12}$$

$$\begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\sin\theta \\ 0 & \cos\phi & \cos\theta\sin\phi \\ 0 & -\sin\phi & \cos\theta\cos\phi \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{4.13}$$

where ω_x, ω_y and ω_z are the roll, pitch and yaw rates of the quadcopter, respectively. The full derivation of Equation (4.13) can be found in Appendix A.2.

The quadcopter is assumed to be symmetrical about each of its body axes. Therefore, its inertia tensor can be reduced to the diagonal matrix:

$$I = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix} \tag{4.14}$$

Where I_x is the quadcopter's mass moment of inertia about the x_b axis, I_y is that about the y_b axis and I_z is that about the z_b axis. Substituting Equation (4.12) into the rotational kinetic energy term and rearranging according to the properties

of matrices yields:

$$\begin{aligned}\frac{1}{2}\boldsymbol{\Omega_B}^T I\,\boldsymbol{\Omega_B} &= \frac{1}{2}\left(P\dot{\boldsymbol{\eta}}\right)^T\ I\ \left(P\dot{\boldsymbol{\eta}}\right)\\ &= \frac{1}{2}\dot{\boldsymbol{\eta}}^T P^T\, I\, P\dot{\boldsymbol{\eta}}\\ &= \frac{1}{2}\dot{\boldsymbol{\eta}}^T J\dot{\boldsymbol{\eta}}\end{aligned} \tag{4.15}$$

Here J acts as a time-dependent inertia matrix expressed in terms of the generalized coordinates, and is given by the symmetric matrix:

$$J = P^T\, I\, P = \begin{bmatrix} J_{11} & 0 & J_{13} \\ 0 & J_{22} & J_{23} \\ J_{13} & J_{23} & J_{33} \end{bmatrix} \tag{4.16}$$

The full expressions of J_{ij} and can be found in Appendix A.3.

Finally, the payload's vertical position in inertial frame coordinates, z_p, is taken from the third element of its position vector:

$$z_p = z - \lambda c_\alpha c_\beta \tag{4.17}$$

Now that all quantities have been expressed using generalized coordinates, it becomes convenient to separate the Lagrangian into two equations: a rotational Lagrangian and a translational Lagrangian. This is possible since the terms in rotation are fully independent of those in translation, and consequently the equations of motion resulting from each are uncoupled. The Lagrangians of the system are therefore:

$$L_r = \frac{1}{2}\dot{\boldsymbol{\eta}}^T J\dot{\boldsymbol{\eta}} \tag{4.18}$$

$$L_t = \frac{1}{2}m\,\dot{\boldsymbol{\xi}}_{\boldsymbol{B}}^{\ T}\dot{\boldsymbol{\xi}}_{\boldsymbol{B}} + \frac{1}{2}m_p\,\dot{\boldsymbol{\xi}}_{\boldsymbol{P}}^{\ T}\dot{\boldsymbol{\xi}}_{\boldsymbol{P}} - mgz - m_p g z_p \tag{4.19}$$

The Euler-Lagrange constraint can then be applied to each variable to obtain all eight equations of motion:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = Q_i \tag{4.20}$$

where q_i represents each of the generalized coordinates, and Q_i is a generalized force (for linear variables) or moment (for angular variables) applied in the selected variable direction.

Substituting Equations (4.11) and (4.17) into Equation (4.19) and applying the Euler-Lagrange constraint to coordinates x, y and z yields Equations (4.21), (4.22) and

(4.23). These can be solved for the position of the quadcopter's center of mass in inertial frame coordinates:

$$(m+m_p)\ddot{x} + m_p\lambda\left(\dot{\beta}^2 s_\beta - \ddot{\beta}c_\beta\right) = F_x \tag{4.21}$$

$$(m+m_p)\ddot{y} + m_p\lambda\left(\ddot{\alpha}c_\alpha c_\beta - \ddot{\beta}s_\alpha s_\beta - 2\dot{\alpha}\dot{\beta}c_\alpha s_\beta - \left(\dot{\alpha}^2+\dot{\beta}^2\right)s_\alpha c_\beta\right) = F_y \tag{4.22}$$

$$\begin{aligned}(m+m_p)\ddot{z} + m_p\lambda\Big(&\ddot{\alpha}s_\alpha c_\beta + \ddot{\beta}c_\alpha s_\beta - 2\dot{\alpha}\dot{\beta}s_\alpha s_\beta \\ &+ \left(\dot{\alpha}^2+\dot{\beta}^2\right)c_\alpha c_\beta\Big) + (m+m_p)\,g = F_z\end{aligned} \tag{4.23}$$

Here F_x, F_y and F_z are generalized forces applied to the quadcopter center of mass in the x, y and z directions, respectively.

Substituting Equation (4.16) into equation (4.18) and applying the Euler-Lagrange constraint to coordinates ϕ, θ and ψ yields Equations (4.24), (4.25) and (4.26). These can be simultaneously solved for the Euler angles:

$$J_{11}\ddot{\phi} + J_{13}\ddot{\psi} + C_\phi = \tau_\phi \tag{4.24}$$

$$J_{22}\ddot{\theta} + J_{23}\ddot{\psi} + C_\theta = \tau_\theta \tag{4.25}$$

$$J_{13}\ddot{\phi} + J_{23}\ddot{\theta} + J_{33}\ddot{\psi} + C_\psi = \tau_\psi \tag{4.26}$$

Here the terms in J_{ij} are the inertial terms, and C_i are the Coriolis-Centripetal terms. The latter arise as a result of describing the rotational dynamics using the unit directions of a non-inertial frame of reference (the body frame), and are functions of ϕ, θ, ψ and of their first derivatives. The full expressions of the Coriolis-Centripetal terms are rather cumbersome and can be found in Appendix A.4. Here, τ_ϕ, τ_θ and τ_ψ are the generalized torques applied in the ϕ, θ and ψ angular directions, respectively.

Finally, applying Equation (4.20) to coordinates α and β, using the Lagrangian in Equation (4.19), yields equations (4.27) and (4.28) which can be solved for the payload angles:

$$m_p\lambda\left(\lambda\ddot{\alpha}c_\beta^2 - 2\lambda\dot{\alpha}\dot{\beta}s_\beta c_\beta + \ddot{y}c_\alpha c_\beta + \ddot{z}s_\alpha c_\beta + g s_\alpha c_\beta\right) = \tau_\alpha \tag{4.27}$$

$$m_p\lambda\left(\lambda\ddot{\beta} - \ddot{x}c_\beta - \ddot{y}s_\alpha s_\beta + \ddot{z}c_\alpha s_\beta + \lambda\dot{\alpha}^2 s_\beta c_\beta + g c_\alpha c_\beta\right) = \tau_\beta \tag{4.28}$$

Here τ_α and τ_β are generalized torques applied in the α and β directions, respectively. However, since the load is swinging freely, no torques are directly applied to the α and β directions, and one can write $\tau_\alpha = \tau_\beta = 0$.

The external forces and moments applied to each coordinate (F_i, τ_i) must now be described as functions of known or controllable parameters in order to develop a control scheme. These are detailed in the following sections.

4.1.4 Thrust Forces and Moments

The propeller thrust forces acting in the x, y and z directions will be considered first. Referring to the free-body diagram in Figure 4.1 above, the total magnitude of the thrust force, T, is given by:

$$T = T_1 + T_2 + T_3 + T_4 \tag{4.29}$$

This force is assumed to always acts parallel to the propeller axes, i.e. in the z_b direction. It is therefore expressed using body frame coordinates as:

$$^{B}\boldsymbol{T} = \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} \tag{4.30}$$

This vector must now be expressed using inertial frame coordinates for consistency with the equations of motion in x, y and z. Using the rotation matrix in Equation (2.1),

$$^{E}\boldsymbol{T} = {}^{E}_{B}R \; {}^{B}\boldsymbol{T} \tag{4.31}$$

$$^{E}\boldsymbol{T} = \begin{bmatrix} c_\theta c_\psi & s_\phi s_\theta c_\psi - c_\phi s_\psi & s_\phi s_\psi + c_\phi s_\theta c_\psi \\ c_\theta s_\psi & c_\phi c_\psi + s_\phi s_\theta s_\psi & c_\phi s_\theta s_\psi - s_\phi c_\psi \\ -s_\theta & s_\phi c_\theta & c_\phi c_\theta \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} \tag{4.32}$$

Computing the result of the matrix multiplication yields expressions for the generalized forces in x, y and z:

$$^{E}\boldsymbol{T} = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = \begin{bmatrix} T\left(s_\phi s_\psi + c_\phi s_\theta c_\psi\right) \\ T\left(c_\phi s_\theta s_\psi - s_\phi c_\psi\right) \\ T\left(c_\phi c_\theta\right) \end{bmatrix} \tag{4.33}$$

The moments generated by the propellers, acting about the x_b, y_b and z_b axes will be considered next. Again referring to the free-body diagram in Figure 4.1, the moment generated by the propellers about the x_b axis (the "rolling moment") is given by:

$$\tau_\phi = \left(T_1 + T_2 - T_3 - T_4\right) l \tag{4.34}$$

where l is the normal distance from each force's point of application to the x_b-z_b plane. Note that τ_ϕ, as defined above, is already expressed in the body-frame coordinate x_b and no transformation is necessary in order to make it consistent with the ϕ equation of motion. Proceeding similarly for the pitching moment about y_b:

$$\tau_\theta = (T_2 + T_3 - T_1 - T_4)\, l \tag{4.35}$$

where l is the normal distance between the points of application and the y_b-z_b plane, and is equal in magnitude to that given above for τ_ϕ. The yawing moment about z_b is given by the sum of all propeller reaction moments:

$$\tau_\psi = M_2 + M_4 - M_1 - M_3 \tag{4.36}$$

The derived expressions for T, τ_ϕ, τ_θ and τ_ψ become useful when designing a control algorithm on an experimental platform. For the sake of simplicity in simulations, these quantities will continue to be referred to as T, τ_ϕ, τ_θ and τ_ψ. In an actual quadcopter controller, each quantity can be independently controlled by solving a system of four equations that always has a unique solution if the rotor inertias are assumed to be small [64].

4.1.5 Reduction to the planar case

The model can further be reduced to a planar one, in order to obtain a first basic starting point for the design of a model-free control algorithm. Since the purpose of this project is slung load stabilization, the algorithm's performance in stabilizing the load in a single plane can be tested first for feasibility and viability. Furthermore, since a quadcopter is symmetrical about its x_b-z_b plane as well as its y_b-z_b plane, and therefore its dynamics are identical in both planes, a planar model can be used to train model-free algorithms that work simultaneously: one in each plane. The planar model herein will be derived for the y-z plane. Therefore, all generalized coordinates which are undefined in this plane must be set to 0. That is:

$$\begin{aligned} x = \dot{x} = \ddot{x} = 0 \\ \theta = \dot{\theta} = \ddot{\theta} = 0 \\ \psi = \dot{\psi} = \ddot{\psi} = 0 \\ \beta = \dot{\beta} = \ddot{\beta} = 0 \end{aligned} \tag{4.37}$$

The system of equations of motion is therefore reduced to four equations in y, z, ϕ and α. Taking the dynamics for y in Equation (4.22) and substituting in the assumptions in Equation (4.37), the y equation of motion becomes:

$$(m + m_p)\ddot{y} + m_p\lambda\left(\ddot{\alpha}c_\alpha - \dot{\alpha}^2 s_\alpha\right) = F_y \tag{4.38}$$

Next, taking the dynamics for z in Equation (4.23) and substituting in the assumptions in Equation (4.37) yields the second planar equation of motion for z:

$$(m + m_p)\ddot{z} + m_p\lambda\left(\ddot{\alpha}s_\alpha + \dot{\alpha}^2 c_\alpha\right) + (m + m_p)\, g = F_z \tag{4.39}$$

Similarly for Equation (4.24) in ϕ:

$$I_x\ddot{\phi} = \tau_\phi \tag{4.40}$$

And finally, for Equation (4.27) in α:

$$m_p\lambda\left(\lambda\ddot{\alpha} + \ddot{y}c_\alpha + \ddot{z}s_\alpha + gs_\alpha\right) = 0 \tag{4.41}$$

Substituting assumptions (4.37) into the force expressions of Equation (4.33), then substituting the result into the above planar equations, and rearranging, yields the final form of the equations of motion which will be used to simulate the system:

$$\ddot{y} = \left(-Ts_\phi - m_p\lambda\left(\ddot{\alpha}c_\alpha - \dot{\alpha}^2 s_\alpha\right)\right) / (m + m_p) \tag{4.42}$$

$$\ddot{z} = \left(Tc_\phi - m_p\lambda\left(\ddot{\alpha}s_\alpha + \dot{\alpha}^2 c_\alpha\right) - (m + m_p)\, g\right) / (m + m_p) \tag{4.43}$$

$$\ddot{\phi} = \tau_\phi / I_x \tag{4.44}$$

$$\ddot{\alpha} = \left(-\ddot{y}c_\alpha - \ddot{z}s_\alpha - gs_\alpha\right) / \lambda \tag{4.45}$$

5. Model-Free Control Algorithm

A simple model has successfully been derived for the purpose of simulating the general dynamics of the quadcopter-slung-payload system. A control scheme will now be developed in order to stabilize the slung payload when it is subjected to a disturbance. Developing a robust, adaptable, model-free approach would ideally allow controllers which use it to be easily retrained on different rotorcraft carrying loads which may greatly vary in size and shape, without the need for manual re-tuning of controller parameters to account for the characteristics of the new aircraft or new load [65].

As discussed in section 2.1.2, controllers based in reinforcement learning constitute a class of model-free controllers which are garnering much attention, due to the significant rise in computational power of modern computers. The latter fact is true even for computers such as hobby-grade microcontrollers for autonomous robots. RL is an attractive solution, since the burden on the engineer is reduced to selecting appropriate approximation methods and learning parameters, and essentially letting the RL agent devise its own optimal solution to the potentially complex task at hand.

The dynamics surrounding the stabilization of the slung payload are fairly complex, as can be seen from even the simplified model equations derived in Chapter 4 (Equations (4.21) - (4.28)). Furthermore, the optimal approach to achieving the goal using hardware and software which is readily available is not immediately obvious, making model-free RL an appropriate choice. The controller designed in this section is based in policy-gradient RL, and attempts to address the shortcomings of those detailed in Chapter 2. First, in order to ensure that the controller is versatile, as mentioned previously, and to ensure it truly arrives at the best possible solution, no assumptions will be made regarding linearity, and as little restrictions as possible will be placed on the shape of the value function and on the shape of the control function. This will allow the reward function to be selected logically rather than according to a predetermined framework (such as the quadratic functions used in the literature), and will also allow the vehicle characteristics to be changed as needed. Second, the controller will use quantities that would realistically come from sensors and actuators located on board the quadcopter, removing the need for motion capture equipment and heavy or costly actuators such as winches [66], and allowing it to operate outdoors. Third, the proposed algorithm will have no prior knowledge of model dynamics, adapting

its functions from arbitrary initializations, as it learns. Fourth, the algorithm will be made as computationally-efficient as possible in order to remove heavy dependence on simulated results, and allow the RL agent to adapt its functions in real-time, on the physical system. Finally, the proposed control scheme will attempt to address the issue of the discrete action space: the algorithm based in policy-gradient methods will allow the agent to use the full range of its continuous action space, in a manner applicable to a wide variety of different applications.

In the following sections, the value function approximation method is first described, after which the control function updates, based in policy-gradient RL, are detailed. The full algorithm, with all its parameters, is then outlined, and simulation results are presented.

5.1 Choice of VFA Method

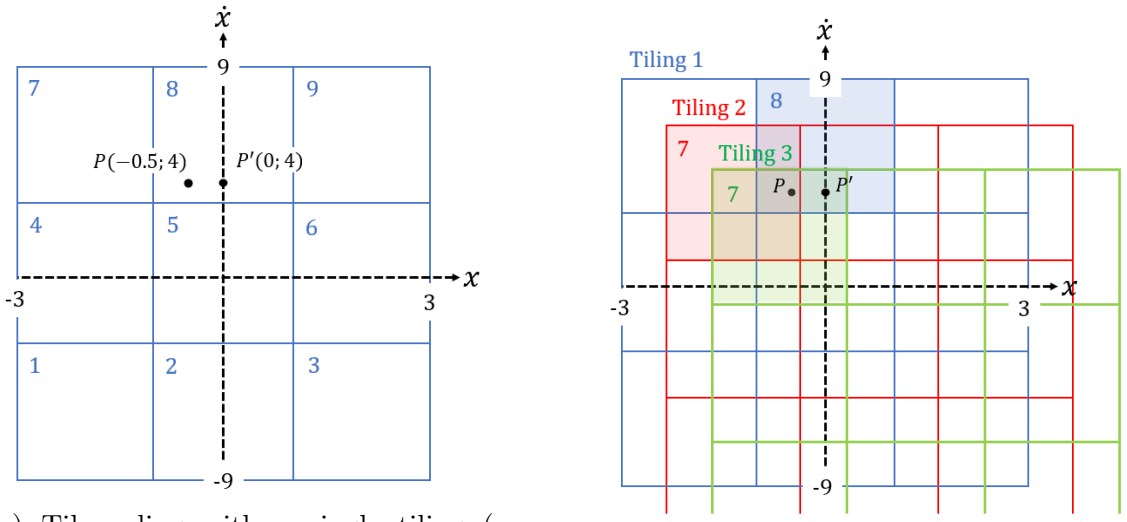
The first aspect of the algorithm that must be dealt with is the dynamic system's continuous state-space, specifically the continuous quantities of the load angle α and load-angle velocity $\dot{\alpha}$ for the planar case, since these are most useful to quantify the performance of the RL agent in stabilizing the load. Storing separate value functions, each for a discrete set of states $(\alpha, \dot{\alpha})$, may not be useful nor practical. While pure discretization of the state-space into large "bins" has been used in the past with physical systems [6], methods based in value function approximation (VFA) are usually favoured. Due to its convergence guarantees, ease of implementation and computational efficiency, linear VFA is a natural first choice for the approximation method.

As discussed briefly in section 3.3.1, the first step in defining a linear function approximator is the choice of linear basis functions, alternatively known as state features. The basis functions must reflect each state in the state-space using a unique vector of numerical values. An issue now arises in trading crude and computationally efficient linear basis representations, such as simple quadratic bases, for accurate and computationally cumbersome ones, such as Chebychev polynomials. While a large amount of orthogonal basis functions is able to represent a much wider variety of functions, they may not be necessary, and may slow down the learning process. An ingenious solution comes in the form of a linear basis representation known as tile-coding. Tile-coded representations are hailed for their ability to generate unique features for multi-dimensional state-spaces, to accurately represent a wide variety of

(sometimes discontinuous) functions, and to be extremely computationally efficient [6]. The latter aspect is crucial in an application where computation time must be minimized and hardware is limited in its resources and power.

5.1.1 Tile-coded approximation

Tile-coding is, in its most basic form, similar to discretizing the state-space. However, unlike a pure discretization, tile-coding is able to generalize its function approximation results to wide regions of the state space, while still uniquely differentiating between states that differ slightly. This is achieved by layering many, slightly-offset discretizations over the state-space. This concept is illustrated in Figure 5.1. Consid-



(a) Tile-coding with a single tiling (pure state-space discretization)

(b) Tile-coding with three tilings

Figure 5.1: Illustration of the concept of a tile-coded approximator.

ering the mass-spring damper of Figure 3.4, whose state-space consists of its position x and velocity $\dot{x}$, one may want to compute value functions for positions between -3m and 3m, and velocities between -9m/s and 9m/s, for example. The simplest approach would be to discretize this state space into bins, or "tiles", and number them, as mentioned previously. This is illustrated in Figure 5.1a. In this case, the point P corresponding to a position of -0.5 m and a velocity of 4m/s would land in tile number 8, and be assigned the single stored value associated to all points which fall within tile 8. Alternatively, one can use a slightly more complex, tile-coded representation. In Figure 5.1b, three grids, or "tilings" have been laid over the state-space at slight offsets. The same point P can now be said to belong to tile 8 of tiling 1, tile 7 of tiling 2 and tile 7 of tiling 3 (the tiles shaded in blue, red and green). The value,

or "weight" associated to each of these three tiles can then contribute to the total value of point P. If the next point is $P'(0;4)$, for the pure discretization on the left, it would still fall only in tile 8 and no difference would be noted in the value of this new state relative to the previous state. However, with three tilings, the new point now falls in tile 8 of tiling 2, rather than tile 7, which is a different feature representation, and therefore has a different value. On the other hand, the new point still falls in the same tiles of tilings 1 and 3, and its value therefore shares some similarities with that of the previous one. This allows the agent to generalize what is known about the value of the previous state to the value of the new one.

If one is using a tile-coded representation for linear VFA, the features vector $\mathbf{\Phi(s)}$ becomes what is known as a "binary" features vector. Each tile in each tiling is assigned an entry in the features vector. In the example from Figure 5.1b, the features vector would be $9 \times 3 = 27$ entries long, since there are three tilings, and nine tiles in each. The first nine entries would be assigned to the first tiling, the next nine to the second, and so on, as is typical with this type of representation. To obtain the features vector for a specific point in the state-space, each tile in which the point is located must have its entry set to one, while all other entries are zero, hence the term "binary". The tiles whose indices are set to one are often called the "active tiles". For point $P(-0.5;4)$ above, the features vector $\mathbf{\Phi(s)}$ would be all zeros, except for entries number 8, 16 and 25, corresponding to tiles 8, 7 and 7 of each tiling. This indicates that these tiles are "active" for that particular state. From Equation (3.11), the value function approximation for this state becomes:

$$\hat{V}^{\pi}(s = P, \mathbf{w_V}) = \mathbf{w_V}^T \mathbf{\Phi}(s = P) \tag{5.1}$$

Let $\mathbf{w_{V,\,active}}$ be a subset of $\mathbf{w_V}$ containing only the indices of the active tiles. Then the features vector $\mathbf{\Phi(s)}$ is effectively the indicator vector of $\mathbf{w_{V,\,active}}$, defined as:

$$\Phi_i = \begin{cases} 1 \text{ if } \mathrm{w}_{Vi} \in \mathbf{w_{V,\,active}} \\ 0 \text{ if } \mathrm{w}_{Vi} \notin \mathbf{w_{V,\,active}} \end{cases} \tag{5.2}$$

Here w_{Vi} are the individual weights stored within the weight vector $\mathbf{w_V}$. Computing the dot-product for this particular state therefore yields:

$$\begin{aligned} \hat{V}^{\pi}(s = P, \mathbf{w_V}) &= \mathrm{w}_{V8} \cdot 1 + \mathrm{w}_{V16} \cdot 1 + \mathrm{w}_{V25} \cdot 1 \\ &= \mathrm{w}_{V8} + \mathrm{w}_{V16} + \mathrm{w}_{V25} \\ &= \sum \mathbf{w_{V,\,active}} \end{aligned} \tag{5.3}$$

Here $\sum \mathbf{w_{V,\,active}}$ is used as shorthand notation for the operation of summing all elements in the vector $\mathbf{w_{V,\,active}}$. The value function for a given state is therefore simply the sum of the weights stored at the indices of its active tiles. From a computational perspective, this reduces computational cost drastically, in that only as many values as there are tilings have to be looked up and added, rather than computing a full vector of expressions and performing the full dot product.

Tile-coding is able to approximate a wide array of functions, whether they be nonlinear or discontinuous. However, the degree of accuracy relies heavily on the number of tilings, on the size of each tile, and even on the shape of the tiles, which dictates how generalization occurs in different dimensions. As more tilings are added, states which are very close together in the state-space will have different feature representations, and the discretization will approach a continuum of states, allowing the agent to distinguish between them uniquely. If tiles are larger, broader generalizations to large regions of the state-space can be made. In this manner, tile-coding attempts to bridge the gap between pure discretization, which is more computationally efficient and can generalize its result to large subsets of the state-space, and linear basis functions, which are unique to each state in the continuous space and allow for more accurate approximations. Figures 5.2 and 5.3 illustrate the use of tile-coding to approximate a nonlinear function as well as a discontinuous function, in the regions of the state space corresponding to $0 \leq x \leq 2\pi$, $0 \leq \dot{x} \leq 2\pi$.

For the nonlinear function, large tiles are an advantage, since the function varies smoothly and generalization to broad areas of the state-space allows states in similar locations to share similar values. However, using many tilings is best in order to achieve an accurate approximation. For the discontinuous function, smaller tiles are a better option, since the function does not vary smoothly, and generalizing too broadly forces all states in the general vicinity of the step to share similar values in the approximation, when they do not in reality. However, fewer tilings are necessary, since the function does not vary between states which are extremely close to each other in the state space, and having fewer tilings means less memory is required to store the weight vector. Selecting the appropriate tile size and tiling number therefore becomes a matter of intuition or trial-and-error, and studies have been conducted into automating the process of selecting tile shapes and numbers of tilings in order to yield an optimal approximation [58], [67].

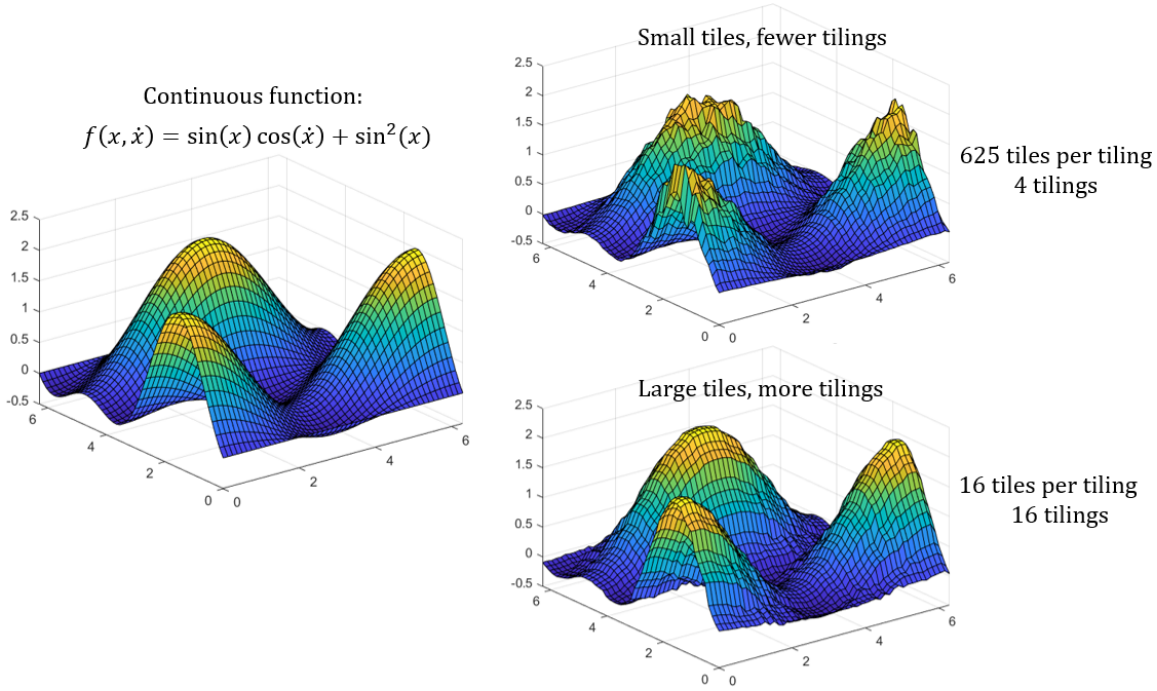


Figure 5.2: Tile-coded approximation of a nonlinear function.

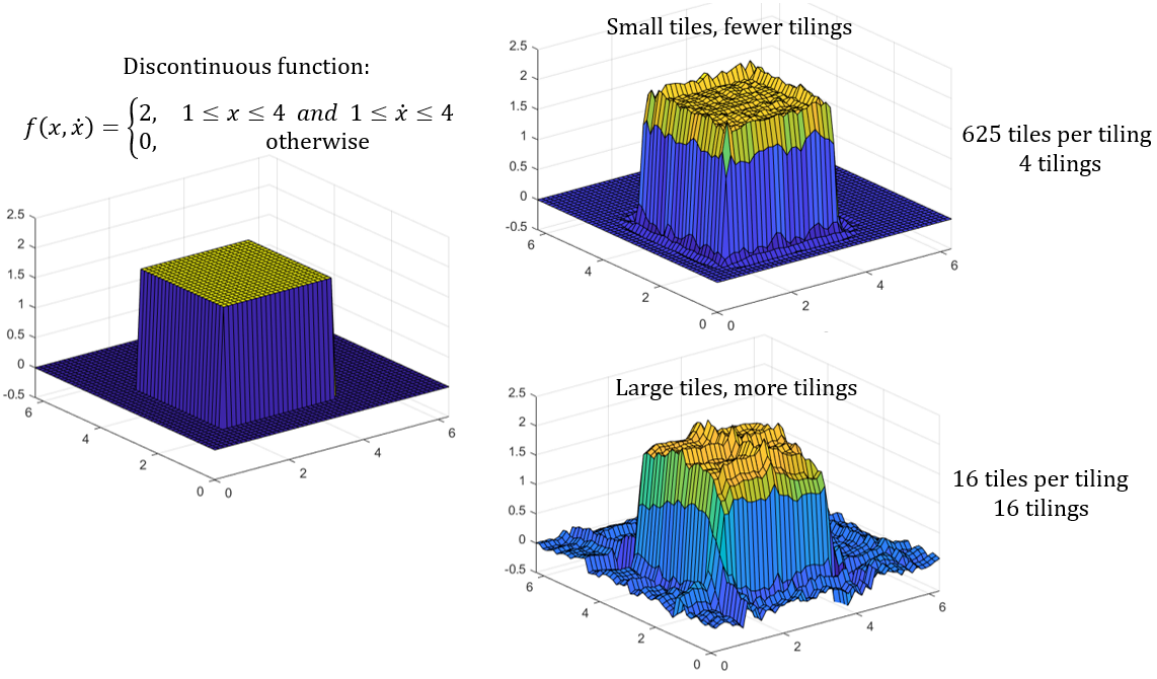


Figure 5.3: Tile-coded approximation of a discontinuous function.

Typically, it is better to limit the number of tiles and tilings as much as possible, all the while ensuring that the accuracy of the estimation is not sacrificed. This ensures that the learning process is not too slow, since fewer weights in the weight vector have

to be learned, and avoids the so-called "curse of dimensionality" which is notorious to tile-coding. As more tiles are added, and as the dimension of the state-space increases, the size of the weight vector grows exponentially and may start to occupy far too much space in memory, requiring computational techniques like hashing in order to access its entries efficiently [67]–[69].

5.1.2 Updating the value function estimates

Now that a feature representation has been selected, the technique by which the agent will update the weight vector and converge on the true value function must be chosen. As explained in section 2.1.2, the problem essentially reduces to one of curve-fitting. While many curve-fitting approaches exist, the chosen approach should allow the agent to adapt its estimate in real-time, as new data points become available. Stochastic gradient descent (SGD) is a commonly-used approach to solving this problem, due to its ease of implementation and ability to adapt in real-time, and will be selected for this application.

The SGD method, like many other curve-fitting approaches, attempts to reduce the mean-squared-error between the approximated value function and its true value, as defined in Equation (3.10). However, it does not wait to compute the mean, and instead uses a single sample of the error to perform its update. The error metric in Equation (3.10), for a single data point, becomes:

$$Error = \left(V^{\pi}(s) - \hat{V}^{\pi}(s, \mathbf{w_V})\right)^2 \tag{5.4}$$

Where V^{π} is the true value of state s and $\hat{V}^{\pi}$ is that estimated by the tile-coded approximator. Substituting in the expression for linear VFA:

$$Error = \left(V^{\pi}(s) - \mathbf{w_V}^T \mathbf{\Phi}(s)\right)^2 \tag{5.5}$$

A useful update of the parameter vector $\mathbf{w_V}$ is one which reduces this error. Taking the first derivative of the error with respect to $\mathbf{w_V}$ (the gradient) will indicate how a change in the weight vector affects the error function. Using the properties of matrix differentiation, the gradient of the error is given by:

$$\nabla \mathbf{Error} = -2\left(V^{\pi}(s) - \mathbf{w_V}^T \mathbf{\Phi}(s)\right)\mathbf{\Phi}(s) \tag{5.6}$$

For a SGD update, the weight vector is stepped opposite this gradient, i.e. it is

updated in the direction of decreasing error. The update is therefore given by:

$$\begin{aligned} \mathbf{w_V} &\longleftarrow \mathbf{w_V} - \varepsilon_V \nabla \mathbf{Error} \\ \mathbf{w_V} &\longleftarrow \mathbf{w_V} + \varepsilon_V \left(V^\pi(s) - \mathbf{w_V}^T \mathbf{\Phi}(s) \right) \mathbf{\Phi}(s) \end{aligned} \tag{5.7}$$

Here ε_V is a small step size for the update, and has absorbed the 2 in the expression of $\nabla\mathbf{Error}$. Recall that for the tile-coded approximator, the features vector $\mathbf{\Phi}(s)$ is zero for all the entries corresponding to non-active tiles, and one for all active tiles, meaning that only the weights located at the indices of the active tiles are updated at each step, and all others remain unchanged. The update specific to a tile-coded approximator therefore reduces to:

$$\mathbf{w_{V,\,active}} \longleftarrow \mathbf{w_{V,\,active}} + \varepsilon_V \left(V^\pi(s) - \mathbf{w_V}^T \mathbf{\Phi}(s) \right) \tag{5.8}$$

And, as demonstrated in Equation (5.3), the dot product can be rewritten as:

$$\mathbf{w_{V,\,active}} \longleftarrow \mathbf{w_{V,\,active}} + \varepsilon_V \left(V^\pi(s) - \sum \mathbf{w_{V,\,active}} \right) \tag{5.9}$$

For tile-coding, the step size ε_V is often selected to be proportional to the inverse of the number of tilings, i.e. $\varepsilon_V \propto 1/n$, where n is the number of tilings [6], [69]. This ensures that the total value, which is the sum of all active weights, will always be updated by the same amount even if the number of tilings changes. ε_V should also be sufficiently small to ensure that the updates gradually converges on the true value as the states are revisited, and that they do not step too drastically and overshoot the minimum in the error function.

5.1.3 Choosing the true value target

The next step is to select which metric will be considered as the true value V^π on which to base the SGD update. While a Monte Carlo approach with linear VFA guarantees SGD's convergence to the optimum, it is often much slower to converge, and it cannot be executed in real-time, requiring the agent to wait for the episode of learning to be complete before calculating the returns and using them as the "true value" [6]. For these reasons, temporal-difference (TD) learning will be used instead. While n-step TD methods typically perform better, they are more complex, requiring more computation time and more memory storage [6]. As such, a TD(0) algorithm will be used for the first iteration of this controller. Using the expression for the sampled Bellman Operator in Equation 3.15, the true value target for the

SGD update becomes:

$$V^{\pi}(s) = r_t + \gamma V^{\pi}(s_{t+1}) \tag{5.10}$$

Here the value $V^{\pi}(s_{t+1})$ is, in fact, another estimate, as TD updates rely heavily on bootstrapping. Since a linear VFA method is used, the agent must consult the approximator to estimate a value for the next state. Again, taking the result of Equation (5.3) and substituting in the tile-coded approximator for the value of the next state yields:

$$V^{\pi}(s) = r_t + \gamma \sum \mathbf{w_{V,\,active,\,t+1}} \tag{5.11}$$

Finally, substituting the TD target in Equation (5.11) into the weight vector update of Equation (5.9) yields the final update rule for the value function which will be used by the RL agent:

$$\mathbf{w_{V,\,active,\,t}} \longleftarrow \mathbf{w_{V,\,active,\,t}} + \varepsilon_V \left(r_t + \gamma \sum \mathbf{w_{V,\,active,\,t+1}} - \sum \mathbf{w_{V,\,active,\,t}} \right) \tag{5.12}$$

Here the quantity in brackets is effectively the TD error, which should reduce to zero as more states are visited, as more true rewards are recorded, and as the weight vector is updated.

If the target for the weight update is unbiased, and if the learning rate ε_V is properly chosen, the VFA method with SGD is guaranteed to converge on a local optimum. Furthermore, since a linear VFA approach is used, the error function only possesses one optimum: the global optimum [6]. That is, after a sufficient amount of learning, the value function approximator will reflect the true value function as well as the basis functions allow, and the mean squared error will be at its global minimum. However, when using TD learning, the target is not unbiased, since it also relies on the function approximator. Fortunately, it can be demonstrated that TD methods with linear VFA converge to a point *near* the optimum called the TD fixed-point. How near depends on the basis functions' ability to reflect the true value function. Convergence proofs can be found in [6]. Therefore, while one cannot expect the approach to compute the true value function exactly, the tile-coded approximator, being a precise method under the right conditions, should enable the agent to approximate it within a reasonable degree of accuracy. This will allow the RL agent to devise a near-optimal solution to the task, as will be demonstrated in section 5.4.3.

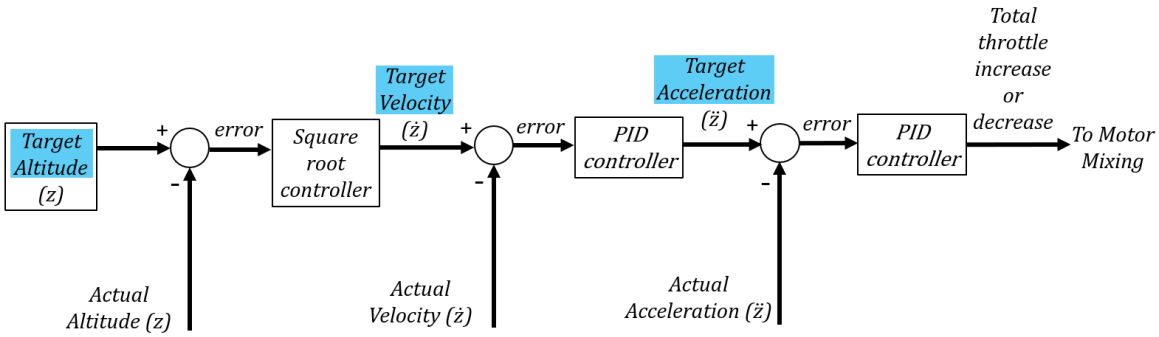
5.2 Choice of Control Policy

Now that the VFA method has been defined, the agent must be able to use the learned value function to arrive at an optimal control policy. The first step in defining the control policy is selecting the appropriate action space, which may be limited to what is realistically feasible or convenient. Then, a control function must be selected to parameterize the policy as a function of the state, and make use of the continuous action-space. These steps are explained in detail in the following sections.

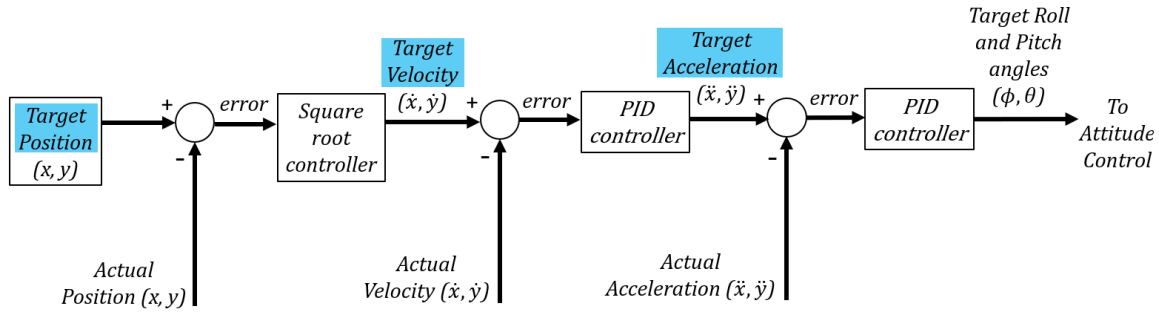
5.2.1 The ArduPilot framework

In order for the controller developed herein to be easily implemented and potentially be widely used and accessible, it must be able to work within the framework of an existing and common aerial vehicle autopilot. ArduPilot is one such autopilot program which is open-source, highly tested, versatile and has been implemented in over a million vehicles worldwide [70]. The ArduPilot project is currently partnered with dozens of UAV and robotics companies around the world, and its firmware can be run on many different types of controller boards, to control many different types of vehicles. For these reasons, it will be used to handle the lower-level functions of the quadcopter's control scheme, and the RL agent will have to learn to work within its framework. An overview of the ArduPilot control scheme for a rotorcraft vehicle using autonomous control is illustrated in Figure 5.4. The control of the quadcopter states is heavily cascaded, especially when it comes to the (x, y) position. Cascade control refers to the fact that the output of one control loop constitutes the input to the next. For the (x, y) position, the control loop is "upstream" from the velocity control, the acceleration control, the attitude control, the body rotation rate control and finally the throttle proportion control. For cascade control to work properly, the dynamics of the inner loops must be much faster than those of the outer loops [74]. With the appropriate PID gains, the response to an attitude command is typically much faster than the response to a position, velocity or acceleration command for a multirotor.

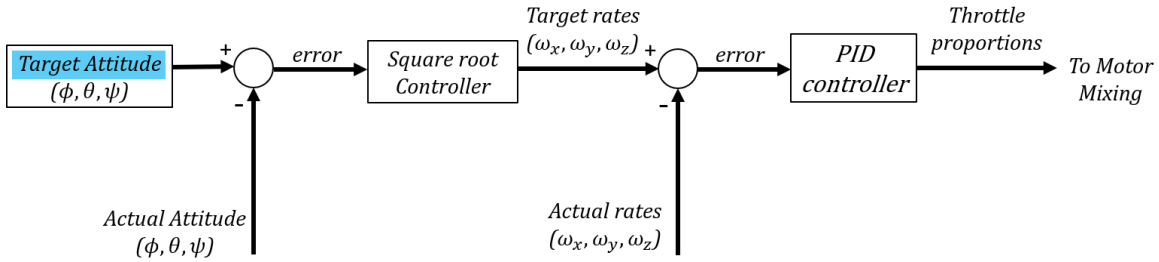
The current ArduPilot functionality allows an external application to directly manipulate all the "target" quantities highlighted in blue. The body frame rotation rates ω_x, ω_y, ω_z, as of yet, cannot be directly set [75]. An external application is any piece of software which runs on a companion computer (located on board the vehicle or not), and interacts with the autopilot firmware. In the context of this project,



(a) Control loop for the altitude of the rotorcraft [71]



(b) Control loop for the (x, y) position of the rotorcraft [72]



(c) Control loop for the attitude of the rotorcraft [73]

Figure 5.4: Control scheme for a rotorcraft UAV running ArduPilot firmware.

this would be the RL agent. Using an external application to control the quadcopter allows the RL agent to command the vehicle regardless of the version of ArduPilot installed and regardless of the type of rotorcraft, and removes the need for customizing and rebuilding the ArduPilot source code. However, it should be noted that this approach is not without its drawbacks: One is limited to the range of vehicles which ArduPilot currently supports, and to the controllable quantities highlighted in the block diagrams of Figure 5.4. It follows from the latter that low-level control inputs such as thrust forces or even propeller speeds cannot be directly manipulated. This limits the efficiency and update frequency of the RL agent, since its control over the dynamics of the vehicle is at the mercy of other controllers further downstream in the control hierarchy.

According to the simplified equations of motion for the load angles in Equations (4.27) and (4.28), controlling the accelerations $\ddot{x}$, $\ddot{y}$ and $\ddot{z}$ should allow the RL agent to indirectly affect the dynamics of the load angles. This approach was found to be popular in the literature. However, manipulating the z dynamics may not be wise, as sudden changes in altitude could cause a crash if the aircraft is low to the ground. This leaves $\ddot{x}$ and $\ddot{y}$. As is made obvious by the cascaded control scheme in Figure 5.4b, the latter accelerations can be controlled by commanding a target attitude, since they are "upstream" from the attitude control of Figure 5.4c. The accelerations' reliance on attitude is verified by the force expressions F_x and F_y in Equation (4.33).

Commanding the attitude is advantageous, since it is closer to the innermost control loop, and therefore has much faster response dynamics, allowing the RL agent to update its commanded control action at a faster rate. For the planar case, the RL agent simply has to command a roll angle ϕ. The RL agent's action space will therefore consist of the quadcopter's full range of possible roll angles. The overall control logic for the RL agent is illustrated in Figure 5.5. That is, the RL agent commands a roll angle ϕ which will affect acceleration in the y axis, which will in turn affect the load angle α. The specific way in which the roll angle affects α can remain unknown until it is learned by the agent through experience. That is, the RL agent will simply select a control action, apply it, and observe only its downstream effect on the load-angle state in order to generate an optimal control policy.

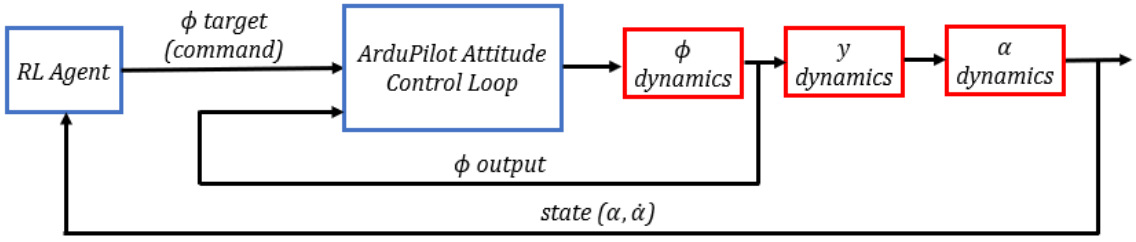


Figure 5.5: Control logic used by the model-free RL agent.

5.2.2 Choice of parameterized policy

To use a policy-gradient method, and therefore take advantage of the full continuous action space, the control policy must be defined as a parameterized function of the state, similarly to the value function. Ideally, the policy should be stochastic, allowing the agent to account for randomness in the environment and system dynamics,

allowing it to explore the effect of different actions in the same state, and allowing the policy-gradient theorem in Equation (3.24) to be directly applied to facilitate the learning. Therefore, π should define a probability distribution which itself is a function of the state [6].

A common approach is to define the control policy as a standard distribution, such as a Gaussian distribution, with mean μ and standard deviation σ being parameterized functions of the state. Formally,

$$\begin{aligned} \mu &= f(s, \mathbf{w}_\mu) \\ \sigma &= f(s, \mathbf{w}_\sigma) \end{aligned} \tag{5.13}$$

Here $\mathbf{w}_\mu$ and $\mathbf{w}_\sigma$ are used to represent two distinct portions of the action parameter vector $\mathbf{w}_a$: $\mathbf{w}_\mu$ is the weight vector storing the mean, and $\mathbf{w}_\sigma$ is the weight vector storing the standard deviation. In this manner, for this particular case with the Gaussian distribution, the action parameter vector is in fact two vectors. Note that f is only used here to indicate that both the mean and standard deviation are *functions* of the state and of their respective weight vector. The two functions are not necessarily identical (in fact, they are usually not identical).

For each new state the agent finds itself in, it will compute the mean and standard deviation associated to that state, construct the Gaussian distribution, and sample a control action from this distribution to be applied. Using a Gaussian distribution for the action selection will inevitably yield some control actions which are far away from the mean, depending on the magnitude of the standard deviation. This will allow the RL agent to explore its environment in a more efficient way as it is learning, trying out sometimes wildly random actions and observing their effect. If the environment is also stochastic, the agent may find it is advantageous to keep a larger standard deviation for certain states, and simply choose to do something a little more random to account for the environmental stochasticity. In this case, the agent is implicitly assuming that the randomness in the environment is also Gaussian. On the other hand, if the system is more deterministic, the policy gradient algorithm should adapt and shrink the standard deviation to a value near zero once the policy has converged on the optimum, meaning that the same action will be selected each time the state is revisited.

The next logical step is therefore to choose the functions which will compute the mean and standard deviation. In order to simplify the logic of the algorithm, a second linear

function of the state features will be used to compute the mean:

$$\mu = {\mathbf{w}_\mu}^T \mathbf{\Phi}(s) \tag{5.14}$$

Here $\mathbf{\Phi}(s)$ is again the binary features vector obtained by tile-coding. While a third linear function can be used to compute the standard deviation, the Gaussian distribution requires the latter to always be positive, which cannot be guaranteed with a simple dot product. As such, the standard deviation is often taken to be Euler's number e raised to the power of the linear function [6], in order to ensure it remains positive. That is:

$$\sigma = e^{{\mathbf{w}_\sigma}^T \mathbf{\Phi}(s)} \tag{5.15}$$

Again, using the tile-coded approximator to generate the functions for μ and σ will allow virtually any functions to be represented, and will not restrict the algorithm to a single class of functions for the optimal control policy. Using the shorthand sum notation of Equation (5.3), the mean and standard deviation approximated by tile-coding can be expressed as:

$$\begin{aligned} \mu &= \sum \mathbf{w}_{\mu,\mathbf{active}} \\ \sigma &= e^{\sum \mathbf{w}_{\sigma,\mathbf{active}}} \end{aligned} \tag{5.16}$$

5.2.3 Updating the control policy towards optimality

The control weight vectors $\mathbf{w}_\mu$ and $\mathbf{w}_\sigma$ must gradually be updated in order to eventually yield the optimal control policy π^*. In the case of the chosen Gaussian policy, at the end of learning, each state in the state space should have its corresponding Gaussian distribution of actions, with an optimal mean μ^* and optimal standard deviation σ^* which guarantees that the agent will take an optimal action in this state in order to reach the goal. However, unlike the value function update, samples of this "optimal" policy do not already exist, and the problem cannot be approached as one of curve-fitting. It is instead approached as a maximization problem.

As explained briefly in section 2.1.2, policy-gradient approaches step their control policies in the direction of an increase in the value function, in order to reach the maximum. Fortunately, the same stochastic gradient approach used to converge on the value function parameters $\mathbf{w_V}$ can be used again. Since the agent is stepping the policy towards a maximum value, rather than towards a minimum error, the approach is called stochastic gradient ascent (SGA). Formally, in the case of the Gaussian policy

described above:

$$\begin{aligned} \mathbf{w}_\mu &\longleftarrow \mathbf{w}_\mu + \varepsilon_\mu \nabla_{\mathbf{w}_\mu} V^\pi(s) \\ \mathbf{w}_\sigma &\longleftarrow \mathbf{w}_\sigma + \varepsilon_\sigma \nabla_{\mathbf{w}_\sigma} V^\pi(s) \end{aligned} \tag{5.17}$$

Here ε_μ and ε_σ are small step sizes for the SGA updates of $\mathbf{w}_\mu$ and $\mathbf{w}_\sigma$, respectively. The subscripts $\mathbf{w}_\mu$ and $\mathbf{w}_\sigma$ accompanying the gradient indicate that the gradient of the value function is taken with respect to these values for their respective updates. In this manner, both the mean and standard deviation functions are updated towards increasing the value function.

Applying the PGT in Equation (3.24) to each of the value gradients yields:

$$\begin{aligned} \nabla_{\mathbf{w}_\mu} V^\pi(s) &= \mathbb{E}_\pi \left(V^\pi(s) \nabla_{\mathbf{w}_\mu} \ln \pi(a|s) \right) \\ \nabla_{\mathbf{w}_\sigma} V^\pi(s) &= \mathbb{E}_\pi \left(V^\pi(s) \nabla_{\mathbf{w}_\sigma} \ln \pi(a|s) \right) \end{aligned} \tag{5.18}$$

SGA, like SGD, performs real-time updates at every step of learning, and therefore it cannot wait to compute the mean. The expected value can therefore be dropped when performing the update, and a single datapoint can be used as a sample. An expression for the gradient of the logarithm of the policy must now be derived for both μ and σ. The Gaussian probability distribution is given by the following function [76]:

$$\pi(a|s) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(a-\mu)^2/(2\sigma^2)} \tag{5.19}$$

Applying the natural logarithm and rearranging according to the properties of logarithms yields:

$$\ln \pi(a|s) = -\ln(\sqrt{2\pi}) - \ln(\sigma) - \frac{(a-\mu)^2}{2\sigma^2} \tag{5.20}$$

Substituting in the expression for μ in Equation (5.14) and taking the first derivative of $\ln \pi$ with respect to $\mathbf{w}_\mu$ yields:

$$\begin{aligned} \nabla_{\mathbf{w}_\mu} \ln \pi(a|s) &= \nabla_{\mathbf{w}_\mu} \left(-\ln(\sqrt{2\pi}) - \ln(\sigma) - \frac{(a - {\mathbf{w}_\mu}^T \mathbf{\Phi}(s))^2}{2\sigma^2} \right) \\ &= \nabla_{\mathbf{w}_\mu} \left(-\frac{(a - {\mathbf{w}_\mu}^T \mathbf{\Phi}(s))^2}{2\sigma^2} \right) \\ &= -\frac{1}{2\sigma^2} \left(2 \left(a - {\mathbf{w}_\mu}^T \mathbf{\Phi}(s) \right) (-\mathbf{\Phi}(s)) \right) \\ &= \frac{1}{\sigma^2} \left(a - {\mathbf{w}_\mu}^T \mathbf{\Phi}(s) \right) \mathbf{\Phi}(s) \end{aligned} \tag{5.21}$$

And by the definition of μ in Equation (5.14):

$$\nabla_{\mathbf{w}_\mu} \ln \pi(a|s) = \frac{(a-\mu)}{\sigma^2}\mathbf{\Phi}(s) \tag{5.22}$$

Substituting the expression for σ in Equation (5.15) into Equation (5.20) above, and taking the first derivative with respect to $\mathbf{w}_\sigma$ yields:

$$\begin{aligned}
\nabla_{\mathbf{w}_\sigma} \ln \pi(a|s) &= \nabla_{\mathbf{w}_\sigma} \left(-\ln(\sqrt{2\pi}) - \ln(e^{\mathbf{w}_\sigma{}^T \mathbf{\Phi}(s)}) - \frac{(a-\mu)^2}{2(e^{\mathbf{w}_\sigma{}^T \mathbf{\Phi}(s)})^2} \right) \\
&= \nabla_{\mathbf{w}_\sigma} \left(-\mathbf{w}_\sigma{}^T \mathbf{\Phi}(s) - \frac{(a-\mu)^2}{2} e^{-2\mathbf{w}_\sigma{}^T \mathbf{\Phi}(s)} \right) \\
&= -\mathbf{\Phi}(s) - \frac{(a-\mu)^2}{2}(-2\mathbf{\Phi}(s)) e^{-2\mathbf{w}_\sigma{}^T \mathbf{\Phi}(s)} \\
&= \left((a-\mu)^2 e^{-2\mathbf{w}_\sigma{}^T \mathbf{\Phi}(s)} - 1 \right) \mathbf{\Phi}(s)
\end{aligned} \tag{5.23}$$

And, by the definition of σ in Equation (5.15):

$$\nabla_{\mathbf{w}_\sigma} \ln \pi(a|s) = \left(\frac{(a-\mu)^2}{\sigma^2} - 1 \right) \mathbf{\Phi}(s) \tag{5.24}$$

For a policy following a Gaussian distribution, the update rule therefore becomes:

$$\begin{aligned}
\mathbf{w}_\mu &\longleftarrow \mathbf{w}_\mu + \varepsilon_\mu \, V^\pi(s) \frac{(a-\mu)}{\sigma^2} \mathbf{\Phi}(s) \\
\mathbf{w}_\sigma &\longleftarrow \mathbf{w}_\sigma + \varepsilon_\sigma \, V^\pi(s) \left(\frac{(a-\mu)^2}{\sigma^2} - 1 \right) \mathbf{\Phi}(s)
\end{aligned} \tag{5.25}$$

The natural next step is to substitute the "true" value function $V^\pi(s)$ for the TD target, since this is what the agent is using as its "true value" for VFA as well. However, in practice, the true value is often instead substituted for a true value estimate to which is subtracted an arbitrary "baseline" value. It can be demonstrated mathematically that subtracting this baseline value does not actually contribute to the value gradient, and therefore no equations are violated, but that it does significantly reduce the variance in the SGA samples and therefore speeds up the learning. Mathematical proofs can be found in [6]. In the case of a one-step TD target, this would mean that the true value would be substituted for $(r_t + \gamma V^\pi(s_{t+1})) - b$, where b is the arbitrary baseline value. The choice of baseline value is up to the engineer, but when using VFA, it is typically taken to be the value function estimate of the current state. That

is, $b = \hat{V}^{\pi}(s_t)$. The update rule with VFA therefore becomes:

$$\begin{aligned}
\mathbf{w}_{\mu} &\longleftarrow \mathbf{w}_{\mu} + \varepsilon_{\mu} \left(r_t + \gamma \hat{V}^{\pi}(s_{t+1}) - \hat{V}^{\pi}(s_t) \right) \frac{(a-\mu)}{\sigma^2} \mathbf{\Phi}(s) \\
\mathbf{w}_{\sigma} &\longleftarrow \mathbf{w}_{\sigma} + \varepsilon_{\sigma} \left(r_t + \gamma \hat{V}^{\pi}(s_{t+1}) - \hat{V}^{\pi}(s_t) \right) \left(\frac{(a-\mu)^2}{\sigma^2} - 1 \right) \mathbf{\Phi}(s)
\end{aligned} \tag{5.26}$$

Where the approximations $\hat{V}^{\pi}$ are made using tile-coding. By using the TD target as truth and the value of the current state as the baseline, the update is effectively made using the TD error. That is,

$$\begin{aligned}
\mathbf{w}_{\mu} &\longleftarrow \mathbf{w}_{\mu} + \varepsilon_{\mu}\, \delta_t \, \frac{(a-\mu)}{\sigma^2} \mathbf{\Phi}(s) \\
\mathbf{w}_{\sigma} &\longleftarrow \mathbf{w}_{\sigma} + \varepsilon_{\sigma}\, \delta_t \, \left(\frac{(a-\mu)^2}{\sigma^2} - 1 \right) \mathbf{\Phi}(s)
\end{aligned} \tag{5.27}$$

Again, due to the nature of the binary features vector $\mathbf{\Phi}(\mathbf{s})$, only the weight entries located at the indices of the active tiles will be updated, while the others remain unchanged. One can therefore write:

$$\begin{aligned}
\mathbf{w}_{\mu,\mathbf{active}} &\longleftarrow \mathbf{w}_{\mu,\mathbf{active}} + \varepsilon_{\mu}\, \delta_t \, \frac{(a-\mu)}{\sigma^2} \\
\mathbf{w}_{\sigma,\mathbf{active}} &\longleftarrow \mathbf{w}_{\sigma,\mathbf{active}} + \varepsilon_{\sigma}\, \delta_t \, \left(\frac{(a-\mu)^2}{\sigma^2} - 1 \right)
\end{aligned} \tag{5.28}$$

and Equation (5.28) above will be used directly as the update rule for the control policy. The full algorithm, making use of all previously defined aspects of the value function approximation and policy-gradient updates, is outlined in the next section.

5.3 The full control algorithm

The full control algorithm belongs to a class of policy-gradient algorithms called *actor-critic* methods. An actor-critic algorithm has two function estimators which are attempting to converge on an optimum simultaneously. The first is the policy estimator (the "actor"), which chooses an action and updates the control policy based on feedback from the value function. The second is the value function estimator (the "critic") which updates the value function approximation each time the policy is changed, and is consulted in order to know in which direction to update the action weights. At each step of learning, the policy is updated by the actor, and the critic must constantly be "catching up" to the new policy in order to estimate its value function. Eventually, both estimators converge on their optimums (or near their

optimums in the case of TD-learning) to produce the optimal control policy π^*, with the highest value functions across the full state-space.

5.3.1 Final RL requirements

A few final parameters and functions must be properly defined before implementing the algorithm. Since no real restrictions are placed on the system, on its value function or on its control policy, most of these aspects are entirely up to the engineer. The ones outlined herein describe only one possible approach to solving the problem, and are chosen based on intuition and hypotheses of what may work best to speed up the convergence and yield a controller that can be realistically implemented.

Useful state-space

The goal of the RL agent is to bring the load to its stable equilibrium in the event that a wind gust or other violent disturbance causes it to swing. For the planar controller, the agent's state-space will consist solely of the load angle α and load angle velocity $\dot{\alpha}$. It is hypothesized that only these quantities are necessary to include, since only these require associated values in order for the agent to quantify having reached the goal of stabilizing the slung load, and it order for it to select relevant control actions. The way in which the control input affects the other state variables is not relevant at this time. Including only the angle characteristics reduces the dimension of the state-space to only two, yielding smaller weight vectors and therefore requiring less space in memory and less time to learn the optimal control policy.

Initial state of the episode

The learning episode begins with the load perturbed by a random angle α between $-\pi/2$ rad and $\pi/2$ rad ($-90°$ and $90°$), and with an initial load velocity of zero. This mimics the behaviour of a load which is at the height of its swing after being hit by a wind gust. While a random initial velocity can also be set, it is difficult to define a range of initial velocities which is realistic and which correspond to realistic initial angles. Setting only a random angle simplifies the learning algorithm and constrains the load dynamics to states which are realistic, yet still part of a large angle range. Furthermore, it is hypothesized that, as the agent explores the consequences of taking certain actions, it will visit many states where the load velocity is not zero, and be able to generalize its control policy to these cases. It may therefore not even be necessary to set an initial velocity.

Learning horizon and learning step

Once the initial state is set, the agent will be allowed to attempt to stabilize the load for a set amount of time, after which the learning episode will be terminated regardless of whether the load has been stabilized or not, and a new episode will be started. As such, the process has a finite horizon. A finite horizon process was deemed more appropriate, since it would allow a large variety of different random start states to be chosen and learned from. A finite horizon process for the learning also suits the application: the goal of the mission itself, for example, may be to deliver a package in an efficient way, and payload stabilization should only occur if the load is disturbed by a large gust. Once the load is disturbed, the stabilization algorithm can bring it back to its equilibrium, and subsequently, the mission can resume. In this manner, the stabilization process is episodic, having a clear beginning ("load disturbed by a large amount"), and a clear end ("load stable").

As the agent learns, it takes small steps towards the optimal weight vectors with each stochastic gradient update, and these are dictated by the step sizes ε. While the stochastic gradient step sizes can be chosen to be small constant values, convergence can be achieved more smoothly if the action step sizes ε_μ and ε_σ instead slowly decrease as the learning episode goes on [6]. This ensures that the steps toward the optimal control policy become smaller as the latter is approached, and decreases the likelihood of overshooting it. It also ensures that the value function update (the critic) can adequately "keep up" with the policy change (the actor) since the actor is forced to make smaller changes to the policy as time goes on. The action step sizes will therefore be multiplied by a discounter which will gradually decrease as the learning episode goes on.

Reward function

Rewards are measured using the load angle and its velocity. At each step, a negative reward which is proportional to the load angle and load velocity will be assigned. The higher the load angle and velocity, the more negative the reward, and therefore the more unfavourable the state. It is hypothesized that constructing the reward function in this manner will force the load towards its stable equilibrium, since this goal state has a reward of zero ($\alpha = 0$, $\dot{\alpha} = 0$). Furthermore, in order to ensure that the load strictly remains between the limits of $-90°$ and $90°$, a highly negative reward will be assigned to the agent if the limit is exceeded, at which point the episode will be terminated. It is hypothesized that assigning a large punishment to these states will

quickly drive the agent to avoid the regions of the state-space which exceed the angle limits.

Useful action-space

The selected control action, the roll angle command ϕ, must also be constrained to realistic values that are useful and achievable by the vehicle. As such, once the control action a is selected by the agent, a limit of $\pm 20^\circ$ is placed on it to generate the command ϕ_{cmd}. This limit was chosen since it was deemed to be large enough to allow the agent to choose from a wide range of actions and therefore stabilize the load quickly, but not so large that it would make the quadcopter manoeuvre too aggressively. The limit can be made larger or smaller, and it is simply a matter of preference and application. While the agent will have no "knowledge" of this saturation point, the latter will indirectly affect the dynamics of the load angle, and, again, the agent should be able to learn the effect of its action selection (the input) on the dynamics (the output) without being made aware of the specifics.

The full implemented algorithm using tile-coding and TD(0) with the Gaussian policy is shown in Algorithm 1 for the planar case. MATLAB-like pseudocode is used to describe the steps. The training begins by setting up the grid which covers the state-space and which will be used to generate active tile indices for each visited state. The weight vectors which store the learned information must also be set. Each learning episode begins by resetting the action step-size discounter back to one, ensuring that the agent can learn smoothly for each new start state. If applying the algorithm in simulation, a starting angle α is then selected at random between the allowable limits by sampling from a uniform distribution. All other quadcopter states are then initialized: the altitude is set to an arbitrary hover altitude, and all other states are set to zero. These states simulate a quadcopter which is initially in hover and whose payload has just been disturbed through a random angle. If training is being conducted on the physical system, the initial states are simply read from the vehicle sensors. The tile indices of the initial state ($\alpha = \alpha_0, \dot{\alpha} = 0$) are then obtained in order to later retrieve the tile-coded approximations of the value and action. The MATLAB code for computing the indices of the active tiles can be found in Appendix B.1. The value of the initial state is then approximated and stored by computing the sum in Equation (5.3).

At each step of the learning episode, the agent must first choose an action to perform. The mean and standard deviation of the Gaussian distribution are computed in much

Algorithm 1: One-step actor-critic using tile-coding with a Gaussian policy.

Initialize the tile-coder grid;
Initialize the weight vectors $\mathbf{w_V}$, $\mathbf{w}_\mu$, $\mathbf{w}_\sigma$;
for *i=1:num_training_episodes* **do**
  $I \leftarrow 1$; % Initialize the discounter
  $\alpha_t \sim \mathcal{U}(-\frac{\pi}{2}, \frac{\pi}{2})$; % Sample α from a uniform distribution/measure
  $z_t \leftarrow$ hover altitude ; % Initialize/measure other states
  $(y, \dot{y}, \dot{z}, \phi, \dot{\phi}, \dot{\alpha})_t \leftarrow 0$;
  $tiles_t \leftarrow$ indices of the currently active grid tiles ;
  $\hat{V}_t \leftarrow \sum \mathbf{w_{V,active,t}}$;
  for *j=1:max_steps_per_episode* **do**
    % Choose an action and compute the policy gradients
    $\mu \leftarrow \sum \mathbf{w}_{\mu,\mathbf{active,t}}$;
    $\sigma \leftarrow \exp(\sum \mathbf{w}_{\sigma,\mathbf{active,t}})$;
    $a \sim \mathcal{N}(\mu, \sigma)$;
    $\nabla\mu \leftarrow (a - \mu)/\sigma^2$;
    $\nabla\sigma \leftarrow ((a - \mu)^2/\sigma^2) - 1$;
    % Take the action and observe the result
    $\phi_{cmd} \leftarrow \min(\phi_{limit}, \max(-\phi_{limit}, a))$; % Saturate ϕ
    $(y, \dot{y}, z, \dot{z}, \phi, \dot{\phi}, \alpha, \dot{\alpha})_{t+1} \leftarrow$ Dynamics after RL_time_step seconds ;
    if $|\alpha_{t+1}| \geq \alpha_{max}$ **then**
      $r_t \leftarrow$ highly negative reward ; % Termination condition
      $\mathbf{w_{V,active,t}} \leftarrow \mathbf{w_{V,active,t}} + \varepsilon_V \left(r_t - \hat{V}_t\right)$;
      $\mathbf{w}_{\mu,\mathbf{active,t}} \leftarrow \mathbf{w}_{\mu,\mathbf{active,t}} + \varepsilon_\mu I \left(r_t - \hat{V}_t\right) \nabla\mu$;
      $\mathbf{w}_{\sigma,\mathbf{active,t}} \leftarrow \mathbf{w}_{\sigma,\mathbf{active,t}} + \varepsilon_\sigma I \left(r_t - \hat{V}_t\right) \nabla\sigma$;
      break
    else
      $tiles_{t+1} \leftarrow$ indices of the new active grid tiles ;
      $\hat{V}_{t+1} \leftarrow \sum \mathbf{w_{V,active,t+1}}$;
      $r_t \leftarrow$ reward $\propto -(\alpha, \dot{\alpha})_{t+1}$;
      $\mathbf{w_{V,active,t}} \leftarrow \mathbf{w_{V,active,t}} + \varepsilon_V \left(r_t + \gamma\hat{V}_{t+1} - \hat{V}_t\right)$;
      $\mathbf{w}_{\mu,\mathbf{active,t}} \leftarrow \mathbf{w}_{\mu,\mathbf{active,t}} + \varepsilon_\mu I \left(r_t + \gamma\hat{V}_{t+1} - \hat{V}_t\right) \nabla\mu$;
      $\mathbf{w}_{\sigma,\mathbf{active,t}} \leftarrow \mathbf{w}_{\sigma,\mathbf{active,t}} + \varepsilon_\sigma I \left(r_t + \gamma\hat{V}_{t+1} - \hat{V}_t\right) \nabla\sigma$;
      $I \leftarrow \gamma I$;
      $tiles_t \leftarrow tiles_{t+1}$;
      $\hat{V}_t \leftarrow \hat{V}_{t+1}$;
  end
end

the same way as the value function, using Equation (5.16). A control action is then selected by sampling from a Gaussian distribution which has the computed mean μ and standard deviation σ. The quantities relevant to the policy gradients of Equation (5.28), here denoted by a shorthand notation as $\nabla\mu$ and $\nabla\sigma$, are then computed and stored for the later actor update. Next, the agent must apply the chosen action and observe the result of taking that action. The control action is first limited to the allowable range of roll angle commands ($\pm 20°$) and is then sent to the vehicle autopilot. After the chosen learning time interval has passed, the new quadcopter states are observed, and the load angle state is assigned a reward. If the allowable load angle range is exceeded (here $\pm 90°$), the agent is assigned a highly negative reward, the actor and critic weights are updated assuming no next state ($\hat{V}_{t+1} = 0$), and the episode is terminated. If the angle is not exceeded, the indices of the active tiles of the new state are obtained, after which the value of the new state is computed. The agent is assigned a reward proportional to the new load angle state, and the actor and critic weights are updated according to their update rules in Equations (5.9) and (5.28). In preparation for the next step of the episode, the discounter I is reduced by multiplying it by the discount factor γ, and the active tiles and value function of the "next state" are stored as those of the current state. The episode loops continuously until the maximum allowable number of learning steps has been reached, or until the allowable load angle range has been exceeded, at which time a new learning episode begins. The algorithm loops over as many learning episodes as are necessary to guarantee the convergence of each of the weight vectors on their optimal values, for as many states (tiles) as possible.

5.4 Simulations

Algorithm 1 was first applied in simulation in order to tune the various learning parameters and optimize the learning process for the vehicle model. First training the agent with a model presents several advantages: Firstly, potentially dangerous control actions can be explored without causing damage to the vehicle testbed or its operators. Secondly, a converged solution can be reached in little time since the dynamics are calculated by a computer rather than observed in real-time. Note that this converged solution can even be used as an initialization point for the training of other rotorcraft-payload systems. These may have a different rotorcraft type or different payload characteristics, for example. The RL agent of a "new" system may be able to glean some information about the dynamics and task from the already

trained quadcopter model, and not be forced to begin the training process from an arbitrary initialization.

For the purpose of the proof of concept, once an acceptable solution has been reached using the quadcopter model, training can then proceed on a quadcopter testbed. Despite the fact that the model is highly simplified, as explained in Chapter 4, it is expected that its dynamics reflect the true vehicle dynamics to an extent that will allow the physical testbed to continue training with reduced risk, and with initial acceptable results where load stabilization is concerned. The selected parameters and assumptions made for the simulation are detailed in the following sections.

5.4.1 Vehicle Dynamics

For the simulated training, the vehicle dynamics update performed on line 17 of Algorithm 1 must be done by solving the system of planar model equations derived in Chapter 4 (Equations (4.42) - (4.45)). An expression must first be obtained for the total thrust force T, since the latter is expected to vary significantly: As the roll angle of the vehicle is varied at the request of the RL agent, so will the total thrust in order to maintain the vertical component required to keep the vehicle in flight. Even a simple simulation should account for this variation, since the total thrust also affects the dynamics in y and consequently in α.

In practice, the total thrust force is controlled by the altitude loop shown in Figure 5.4a on page 56, however, it is difficult to determine exactly how the total thrust varies with the altitude error. Accurate mathematical expressions for all the blocks in Figure 5.4a can be difficult to obtain without using system identification techniques, and in any case, there is no simple way to directly measure the thrust while the vehicle is in flight in order to obtain these expressions. For the sake of simplicity in simulation, and applicability of the simulator to a wide array of vehicles and loads, it is therefore preferable to look for a simple way of approximating T: one that makes use of the model equations and leaves complex dynamics and optimization to training on the vehicle testbed.

The quadcopter should be flying at a constant altitude throughout the load stabilization, for reasons of safety and convenience for the mission. It is therefore assumed that the ArduPilot control loop, after proper tuning, will be able to force the quadcopter's altitude to stay relatively constant. One can therefore assume that the sum of forces on the vehicle in the inertial z direction, and therefore the acceleration in

the inertial z direction, will be held near zero for the duration of the stabilization. While this assumption seems a little reductive in practice, it is hypothesized that it will allow for proper RL parameter selection and tuning, and enable the agent to reach an acceptable solution before proceeding with training on the vehicle testbed, where the more complex z dynamics can be accounted for. The force T was therefore approximated in simulation by taking the equation of motion in Equation (4.43) and setting it to zero, yielding:

$$T = \left(m_p \lambda \left(\ddot{\alpha} s_\alpha + \dot{\alpha}^2 c_\alpha\right) + \left(m + m_p\right) g\right) / c_\phi \tag{5.29}$$

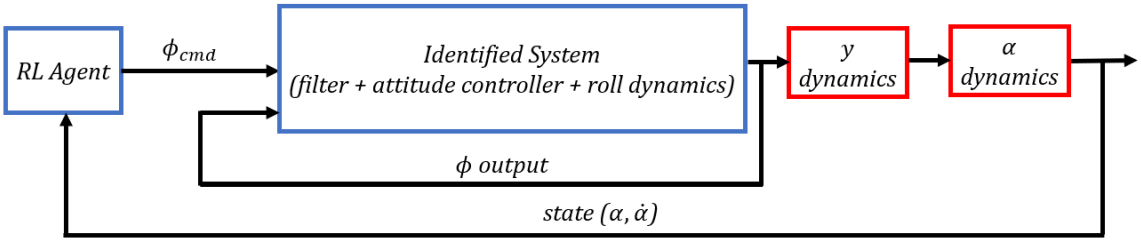
The states $\ddot{\alpha}$, $\dot{\alpha}$, α and ϕ are already computed at each simulation time step by solving the system of model equations (Equations (4.42) - (4.45)). It then becomes simple to use them directly to compute the total thrust force T. Experimental data will be used to confirm whether the constant altitude assumption is valid, and whether it significantly affects the experimental results in any way.

Since the roll angle dynamics will be controlled entirely by the autopilot, and therefore the roll moment values are largely unknown, the planar model for ϕ in Equation (4.44) is also not entirely useful to the simulation. It is expected that the roll dynamics will follow step-response dynamics, since the angle commanded by the RL agent is essentially "constant" for the duration of the RL time step. Two approaches were considered to model the step-response dynamics. The first is to ignore the actual dynamics and assume that the roll output follows the command exactly. This would model the ArduPilot attitude controller as a "perfectly-tracking" controller, which is not realistic but may be sufficient. The second approach is to obtain a mathematical model for the true ϕ dynamics of the particular vehicle which will be used. It was hypothesized that assuming perfect tracking might be too simplistic, and that the delay and errors introduced by the true dynamics may have an effect on the output which is too significant to ignore, even when trying to reach only an acceptable solution. Therefore, to understand how the ϕ output varies with the commanded input when using ArduPilot, system identification was conducted on the vehicle testbed.

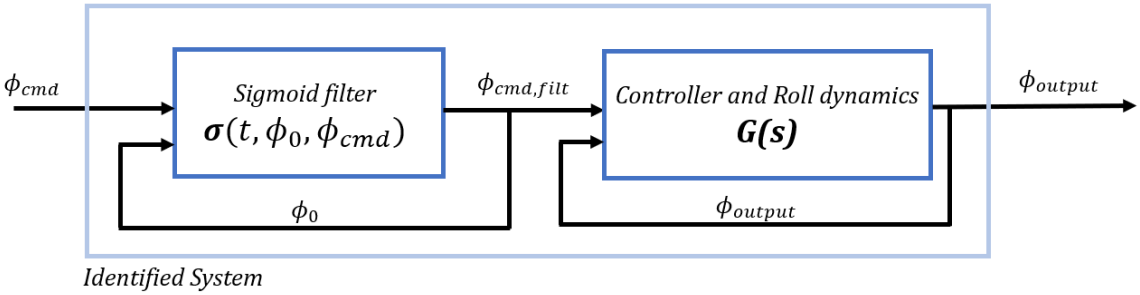
The full system identification process, experiment, results and validation on the experimental testbed are described in detail in [77]. The dynamics were identified by sending a series of roll commands at varying frequencies to the vehicle autopilot and recording the roll output. The roll commands varied between $\pm 5°$ and $\pm 20°$, and the frequency of the commands was varied between 2Hz and 10Hz. The input-output relationship was then obtained using system identification approaches. The identified

expressions were validated by generating a series of random roll commands, plotting a roll output predicted by the model, and comparing the prediction against the true roll behaviour of the vehicle subjected to the same commands.

It was found that, when subjected to a step command, the autopilot first filters the command to generate a smoothed function, after which it is passed to the attitude PID control loop of Figure 5.4c. The block diagram of the full identified roll dynamics, and how they fit into the overall control logic of in Figure 5.5, is illustrated in Figure 5.6.



(a) RL control logic with identified system used for simulation.



(b) Structure of ArduPilot's attitude control obtained by system identification.

Figure 5.6: Block diagrams of the roll dynamics obtained by system identification.

The first block of Figure 5.6b corresponds to the command filtering expression and is written in the time-domain. The second block is an s-domain transfer function describing the relationship between the filtered input and the vehicle's roll output. After inspecting the shape of the command, it was decided to model the filter as a sigmoid function, here denoted by a bold σ. The standard sigmoid function was therefore used as the "black-box" model of the command filter. It is given by the following time-domain expression:

$$\phi_{cmd,filt} = \frac{A}{1 + e^{-B(t-C)}} + D \tag{5.30}$$

where A is the amplitude of the S-curve, B dictates the slope of the incline, C is a

shift along the time axis, and D is a shift along the $\phi_{cmd,filt}$ axis. Using the curve-fitting methodology detailed in [77], the parameters were found to be given by the following expressions:

$$\begin{aligned} A &= |\phi_{cmd} - \phi_0| \\ B &= \frac{30.67\,\mathrm{sgn}(\phi_{cmd} - \phi_0)}{\sqrt{A}} \\ C &= 0.1325\sqrt{A} \\ D &= \min(\phi_0, \phi_{cmd}) \end{aligned} \tag{5.31}$$

Here ϕ_{cmd} is again used to denote the command requested by the RL agent at every step of the algorithm (the step setpoint after saturation), while ϕ_0 is used to denote the value of the filtered roll angle command at the precise moment a new setpoint is requested (i.e. the last recorded value of $\phi_{cmd,filt}$). The lumped control and roll dynamics were modelled as a second-order transfer function given by the following s-domain expression:

$$G(s) = \frac{\phi(s)}{\phi_{cmd,filt}(s)} = \frac{861.1473}{s^2 + 14.7963s + 861.1473} \tag{5.32}$$

Implementation

During simulated training, data points for the dynamics were computed at a fixed frequency of 100 Hz (the simulation frequency). That is, each time the RL agent applies a control action to the system, the states of the vehicle are computed and updated at intervals of 0.01 seconds until the end of the RL time step, at which point a new action is applied. The optimal value for the RL time step was found to be 0.1 s, as is explained in section 5.4.2. The simulation frequency of 100 Hz was chosen since it was deemed large enough to compute smooth, accurate dynamics without significantly slowing down the simulation by computing too many datapoints.

At each simulation step, the total thrust force in Equation (5.29) is computed first, using the values of $\ddot{\alpha}$, $\dot{\alpha}$, α and ϕ from the previous time step. All equations of motion are then solved simultaneously. For ϕ, once the command setpoint is obtained from the RL agent, a filtered command is generated for the full duration of the RL time step by sampling Equation (5.30) at the simulation frequency. The transfer function for ϕ in Equation (5.32), once discretized, is then used to compute the ϕ dynamics for the full RL time step. This process is illustrated in Figure 5.7 for an RL time step of approximately 0.5 seconds. The "x" markers are used to show what the sampled function would look like for a frequency of 100 Hz. For y, z, and α, a fourth-order

Runge-Kutta solver is used to solve their planar model equations, and their final values are stored at the end of the RL time step.

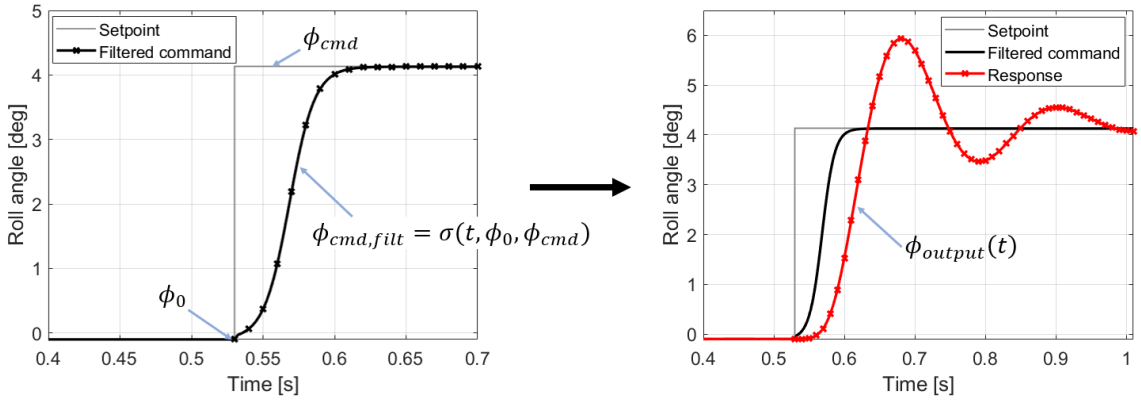


Figure 5.7: 2-step computation of the ϕ dynamics for the duration of the RL time step.

5.4.2 Simulation Parameters

The list of parameters selected to train the RL agent are shown in Table 5.1.

Table 5.1: Simulation parameters used to train the RL agent.

	Parameter	Value	Unit
Model Parameters	g	9.81	m/s^2
	m	0.9	kg
	m_p	0.04	kg
	λ	1	m
Simulation Parameters	Learning frequency	10	Hz
	Learning horizon	200	steps
	γ	0.99	-
Grid Parameters (Tile-coding)	Tiles per dimension	8	tiles
	State-space α limits	± 1.6	rad
	State-space $\dot{\alpha}$ limits	± 7	rad/s
	Number of tilings	8	tilings
Policy Gradient Parameters	ε_V	$1/(5n)$	-
	ε_μ	$1/(1.75E6n)$	-
	ε_σ	$1/(1E7n)$	-

The mass of the vehicle, mass of the payload and length of the cable were chosen so as to match the characteristics of the vehicle testbed. Design and construction of the testbed is detailed in Chapter 6.

A learning frequency of 10 Hz was selected for the training algorithm, meaning that the RL agent calculates a new control action every 0.1 seconds. Increasing the frequency beyond 10 Hz was not found to be entirely useful, since the dynamics of the roll angle are too slow to respond to commands which update at much faster rates. Frequencies which are too low should also be avoided, since they make the response of the RL agent rather sluggish as new states are measured. The learning horizon was chosen to be 200 learning steps. For a frequency of 10 Hz, this means that the agent has 20 seconds to stabilize the load, at which point the episode is terminated. The horizon is chosen somewhat arbitrarily, but 200 steps was deemed long enough to allow the agent to compute long-term rewards while also keeping the simulation computationally efficient. The discount factor γ was chosen to be 0.99 to ensure that long-term rewards would be valued almost as much as immediate rewards, which is crucial to this application, since the goal is long-term (an eventually stable load). Setting the discount factor to be less than one ensures that the value function estimates do not grow to unmanageable values.

For the tile-coded grid which covers the useful state-space, eight tiles span each dimension for the main tiling, partitioning the state-space into 64 tiles. One tile in each dimension had to be added in order for the subsequent tilings to still cover the full state space, even when offset from the main tiling. The range of useful load angle values was limited to ± 1.6 rad, since the episode would be terminated anyway if the value ever exceeds $\pi/2$ or 1.57 rad. The limits for the load angle velocity of ± 7 rad/s were selected by trial-and-error, where the learning algorithm was allowed to run for several episodes, increasing the limits each time a point was found to exceed the allowable state-space. Studies conducted on tile-coding as an estimation method suggest selecting a number of tilings that is an integer power of two greater than or equal to $4 \times dims$, where $dims$ is the number of dimensions in the state-space [78]. As such, eight tilings ($n = 2^3 = 4 \times 2$ dimensions) were laid over the state space at varying offsets from each other. It is also suggested to offset each tiling using multiples of odd integer vectors in order to help with uniform generalization [78]. That is, the second tiling should be offset from the main one by 1/n tiles in the first dimension, and 3/n in the second. The third tiling should be offset from the main one by 2/n tiles in the first dimension, and 6/n in the second, and so on.

The learning step sizes ε were chosen to be proportional to the inverse of the number of tilings as mentioned in section 5.1. The step size is relatively large for the value function approximation, while those which approximate the control action are several

orders of magnitude smaller. This ensures that the critic can quickly approximate changes in the value function by taking larger steps, before the policy is changed by the actor. The actor is limited to small changes in the action function, which limit the policy gradients to reasonable values in order to approach the optimum slowly. The reward function for each time step is given by:

$$r = \begin{cases} -150 & \text{if } |\alpha| \geq \pi/2 \\ -0.9|\alpha| - 0.1|\dot{\alpha}| & \text{otherwise} \end{cases} \tag{5.33}$$

The value of -150 was chosen heuristically: it is large enough to generate a large gradient which quickly shifts the control policy away from actions resulting in load angles over 90°, yet small enough that the gradient does not cause the control policy to overshoot the optimum and diverge. The values of 0.9 and 0.1 were also chosen heuristically: Less importance was placed on minimizing the angular velocity, since it was thought that the agent should be allowed to leverage large load angle velocities in order to bring the angle quickly down to zero.

5.4.3 Training

Algorithm 1 was implemented in MATLAB and simulated training was conducted on a laptop computer. The first successful training attempt was achieved with the assumption that the ArduPilot attitude controller had perfect tracking (i.e. neglecting the roll angle dynamics). Perfect tracking was initially assumed in order to speed up the simulation and allow tuning and selection of learning parameters to proceed quickly and efficiently.

Once adequate learning parameters were identified, the agent was trained on the simulator, again neglecting the roll dynamics, for 10000 episodes to ensure a first successful convergence of the weight vectors and to study the basic learning characteristics of the algorithm. For these passes of training, the value and mean weight vectors $\mathbf{w_V}$ and $\mathbf{w}_\mu$ were initialized to zero, while the elements of the standard deviation weight vector $\mathbf{w}_\sigma$ were initialized to -1. The initial value and mean weights were chosen arbitrarily, while the initial standard deviation weights were chosen to yield a small initial standard deviation for the action selection. A small standard deviation should allow the agent to explore slightly different actions in the beginning, while also allowing it to approach a deterministic policy ($\sigma \approx 0$) more quickly. It is not expected that the learned standard deviation will grow by the end of training, since

the simulation does not include random chance or noise in the dynamics which would cause the agent to select random actions in certain states.

Since the effect of the training itself is largely random, due to the random initial load angles, learning should continue for a large number of episodes. In this manner, by the law of large numbers, variance should be reduced and the weight vectors should have converged for a sufficient number of points in the state-space. Note that the learning step ε_μ can be increased to accelerate the learning, resulting in more efficient load stabilization after the episodes are complete, but caution is advised since too large of an increase can cause the solution to diverge. Smaller step sizes, while slowing down the learning, avoid this risk.

The initial 10000-episode training described above was repeated ten times using the learning step sizes in Table 5.1 in order to draw the learning curve for the algorithm and illustrate the initial stages of the convergence. The curve is shown in Figure 5.8 and was drawn by computing the total sum of rewards per episode, and averaging the results over the ten passes of training. The element of randomness present during training causes the curve to appear noisy, which is why the filtered curve, computed using a moving average filter with a window of 200 datapoints, was superimposed onto the learning curve in order to illustrate the trend.

For the first 2000 episodes, the agent is observed to rapidly adjust its control policy to quickly increase the amount of reward collected. This behaviour is expected since the algorithm updates the action weights in proportion to the temporal-difference error, which, in the beginning, is relatively high since the agent's initial estimate of the value function is chosen arbitrarily and is highly inaccurate. As the agent learns the value function and the temporal-difference error starts to decrease, the action weight updates become less aggressive and the slope of the curve becomes less steep. At this point, the learning is characterized by small changes in the action weights, and corresponding small updates of the value function. After 10000 episodes, it can be seen that the sum of rewards is still on a slight upward trend, indicating that the solution has perhaps not yet converged precisely to the optimum for the "perfectly tracking" model.

Simulated results after 10000 episodes of training, for one of the trained RL agents, are shown in Figure 5.9. The results of 5 separate load stabilization episodes are shown, where the initial load angle is selected at random. The start of a new episode is indicated by the red, vertical lines.

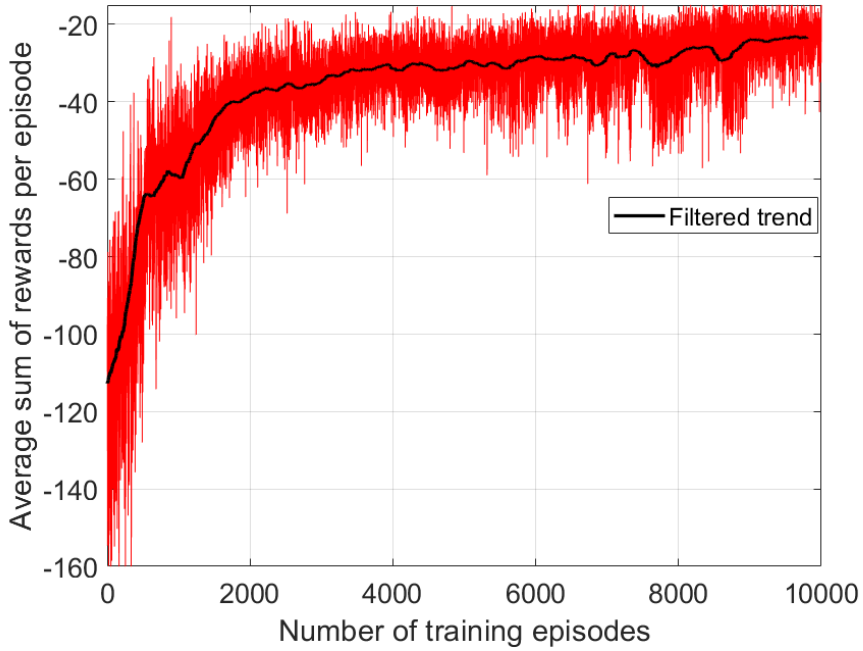


Figure 5.8: Learning curve for the actor-critic algorithm applied to a quadcopter carrying a slung payload.

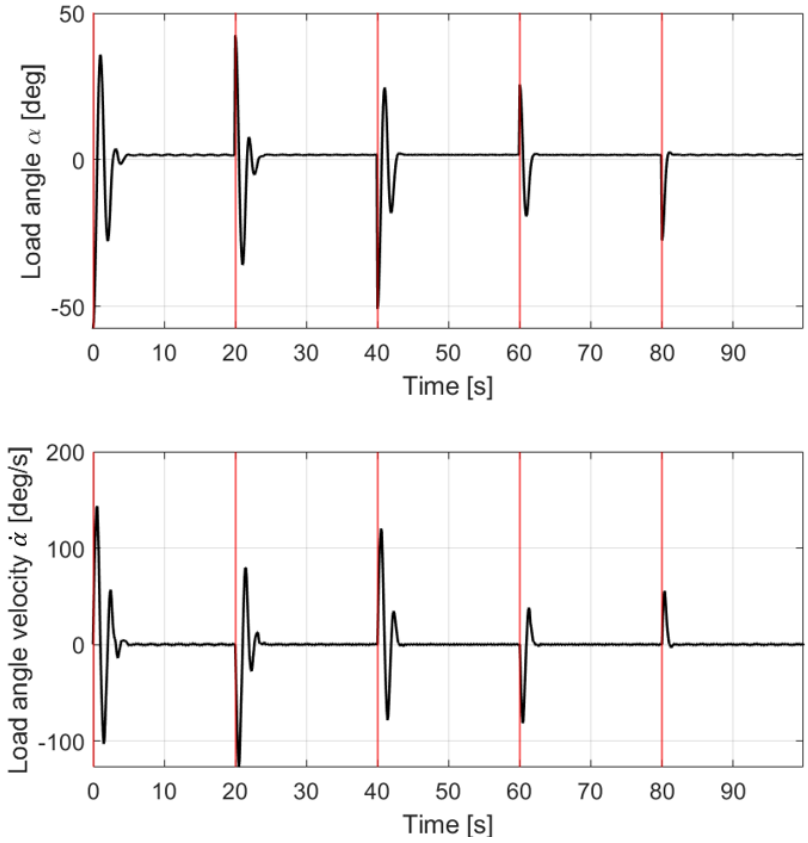


Figure 5.9: Results of the load stabilization simulation after 10000 episodes of training with perfect roll angle tracking.

From the preliminary results, it is clear that the reward function was appropriately chosen, and negatively rewarding the magnitude of the load-angle and load-angle velocity has driven the RL agent to a control policy which stabilizes the slung-payload efficiently. The load angle and load-angle velocity are driven down to zero within a few seconds, despite the agent begin given 20 seconds to accomplish the task.

The weight vectors obtained at the end of the 10000-episode training were then transferred to a different simulator which accounted for the dynamics of the roll angle. Training proceeded on this simulator in order to further improve the weights to account for the delayed roll angle dynamics. This type of sequential training can be favourable, since the agent is not required to learn from the initializations for the more realistic simulator. It can instead leverage its previous knowledge to reach an optimal solution more efficiently. The learning step size ε_μ was increased to $1/(1.5E6n)$ for this second phase of training, to accelerate the learning. As mentioned previously, the control policy changes in proportion to the temporal-difference error. While a system with slightly different dynamics is expected to have different value functions, it is not expected that the error between the already learned function and the new function will be large enough to risk divergence if the step size is increased.

It was initially hypothesized that assuming perfect roll angle tracking would be too simplistic, and that the delay in following the command would have a significant impact on the converged solution. To verify the hypothesis, the control functions for μ, the selected roll angle mean, were compared before and after the first 5000 episodes of training with the realistic roll angle dynamics. It was found that the overall shape of the control function did not change in a meaningful way. The agent was trained for an additional 5000 episodes to ensure that this observation was not due to a lack of exploration with the new dynamics. The results of the training are shown in Figure 5.10 for one of the training passes.

When considering the topography of the function as shown by the color scale, all three functions have the same general shape, with only the scaling differing between them. This indicates that the peaks and valleys become more pronounced as training goes on, but that the agent chooses not to alter its general learned behaviour, even after 10000 additional episodes of training. It is difficult to state whether the different scaling is due to the agent accounting for the delay, and therefore requesting larger roll angles, or whether this is simply the result of continuing training with a larger step size for an agent which has not yet reached the optimum, as was evidenced by the upward

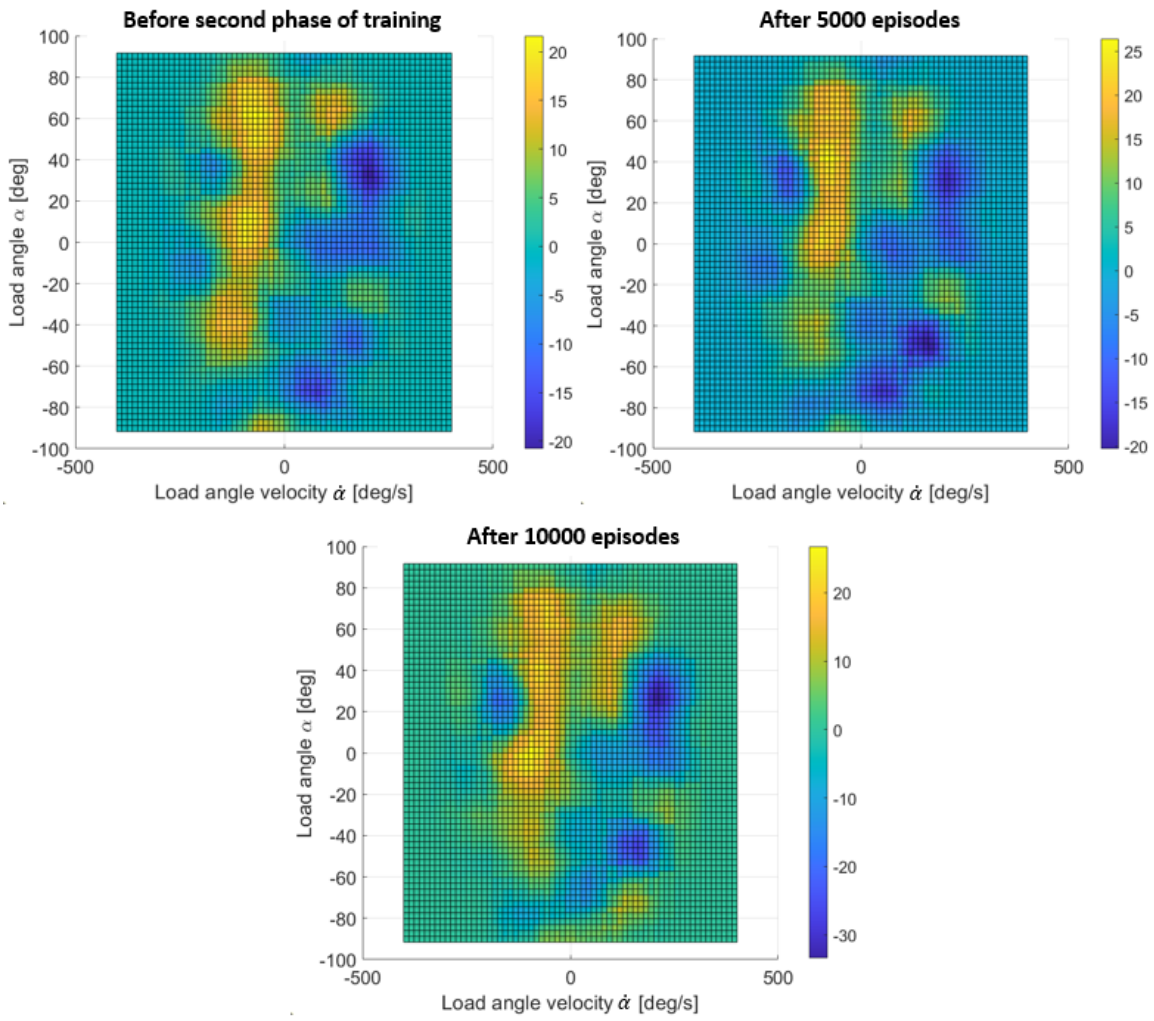


Figure 5.10: Control action (roll angle) mean μ in degrees at various stages of training on the delayed roll angle dynamics.

slope at the end of the graph of Figure 5.8. If the dynamics had a significant effect in changing the system behaviour, it is likely that the temporal difference error would be much larger, and would drive the agent to seek out a different, better solution more quickly, which is not what is observed in practice. Further rigorous testing of the learning characteristics of this algorithm should be conducted to determine whether training with the realistic roll dynamics is in fact necessary, and possible approaches to testing are discussed in Chapter 8. The learned standard deviation was not observed to change significantly, oscillating around $\sigma = 10^{-4}$ over the full state space at the end of learning. This indicates that the agent has not found it useful to "randomize" its actions, and that the latter will not deviate much from the mean during action selection. What is interesting to note about the control function of Figure 5.10 is that the selected action mean appears to mostly rely on the load-

angle velocity. The peaks and valleys are sharper when moving across the range of $\dot{\alpha}$, suggesting that the function gradients are larger along this coordinate, while the changes are a lot less pronounced along the α direction.

The purpose of training on the simulator was simply to obtain a first acceptable solution that would be used as a basis for training on the physical system, which was achieved. The experimental setup is described in detail in the next chapter.

6. Experimental Setup

6.1 Quadcopter Testbed

The quadcopter used to test the RL algorithm was built from off-the-shelf components and is shown in Figure 6.1.

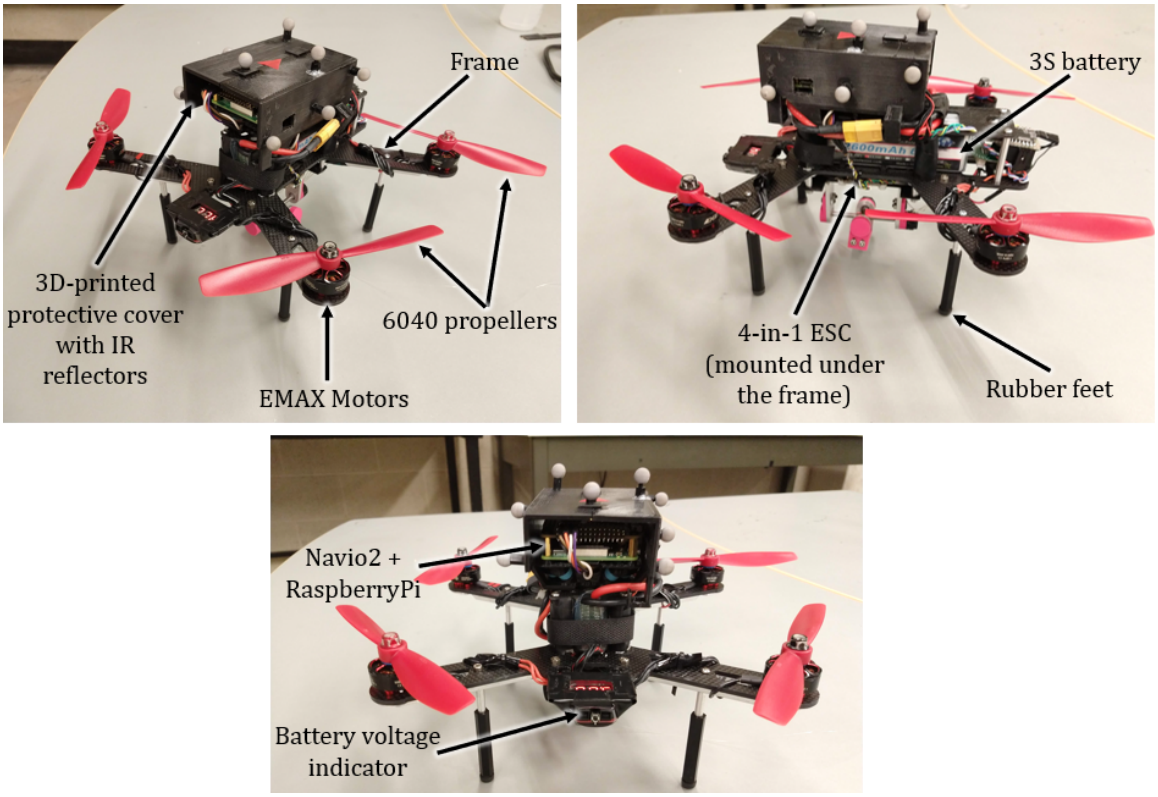


Figure 6.1: Quadcopter testbed with main components labelled.

A small carbon fiber frame with an aluminum base was chosen as the quadcopter's X-flyer frame, and a custom protective cover was 3D printed in order to shield the flight control stack from debris and protect it in case of a crash. The flight control stack is mounted on a base fitted with four vibration dampeners, in order to reduce the effect of motor vibrations on the sensor measurements. Four extended legs with rubber stoppers were added to provide space for the payload attachment. A voltage buzzer was connected to the battery's balance charge connector in order to notify the pilot when the battery is depleted, and a power module was used to distribute power

between the motors and control electronics. A detailed list of the remaining electrical and mechanical components located onboard can be found in table 6.1.

Table 6.1: List of electrical and mechanical components of the quadcopter.

Component	Type selected
Motors	EMAX RSII 2206 1900Kv BLDC motor [79]
ESC	DYS Aria F45A 4-in-1 ESC [80]
Battery	HRB 3S 2600mAh 60C LiPo battery [81]
Propellers	Dalprop 6040 2-bladed bullnose propellers [82]
Flight controller	Navio2 with RaspberryPi 3B+ [83], [84]

6.1.1 Propulsion System Testing

Details concerning the components of the vehicle propulsion system listed in Table 6.1, as well as the reasons for their selection, can be found in Appendix C.1. The propulsion system was mounted on an RCBenchmark test stand in order to obtain the relationship between motor thrust and its control input from the flight controller, the pulse-width-modulation (PWM) signal. After calibrating the electronic-speed-controller (ESC), the PWM signal was slowly incremented from $1000\mu s$ (minimum throttle) to $2000\mu s$ (maximum throttle), and thrust, voltage, current and RPM values were recorded for each of the four motors. The thrust curve, obtained by averaging the measurements of all motors, is shown in Figure 6.2, where the PWM value is graphed as a percentage of its duty cycle ($1000\mu s$ being 0% and $2000\mu s$ being 100%). Only thrust values over 15% PWM are shown, since not enough thrust is produced at values below 15% to be reliably measured.

Each of the four motors produced curves which were extremely similar, as evidenced by the small error bars in Figure 6.2, which are used to represent the standard deviation for the four motor measurements. The relationship between PWM and thrust was also found to be fairly linear, as can be seen from the linear trend line. The latter fact becomes important for the initial setup of the vehicle autopilot, specifically for the selection of the motor thrust exponent. The process of setting the exponent is described in detail in Appendix C.2. At 100% throttle, the thrust output of one motor is around 5.4 N, meaning the maximum total thrust force produced by the propulsion system is approximately 21.6 N. For vertical-takeoff-and-landing vehicles such as rotorcraft, the max thrust to weight ratio is typically well over 1.2, and a general rule of thumb is that the vehicle must be able to output a thrust that is twice

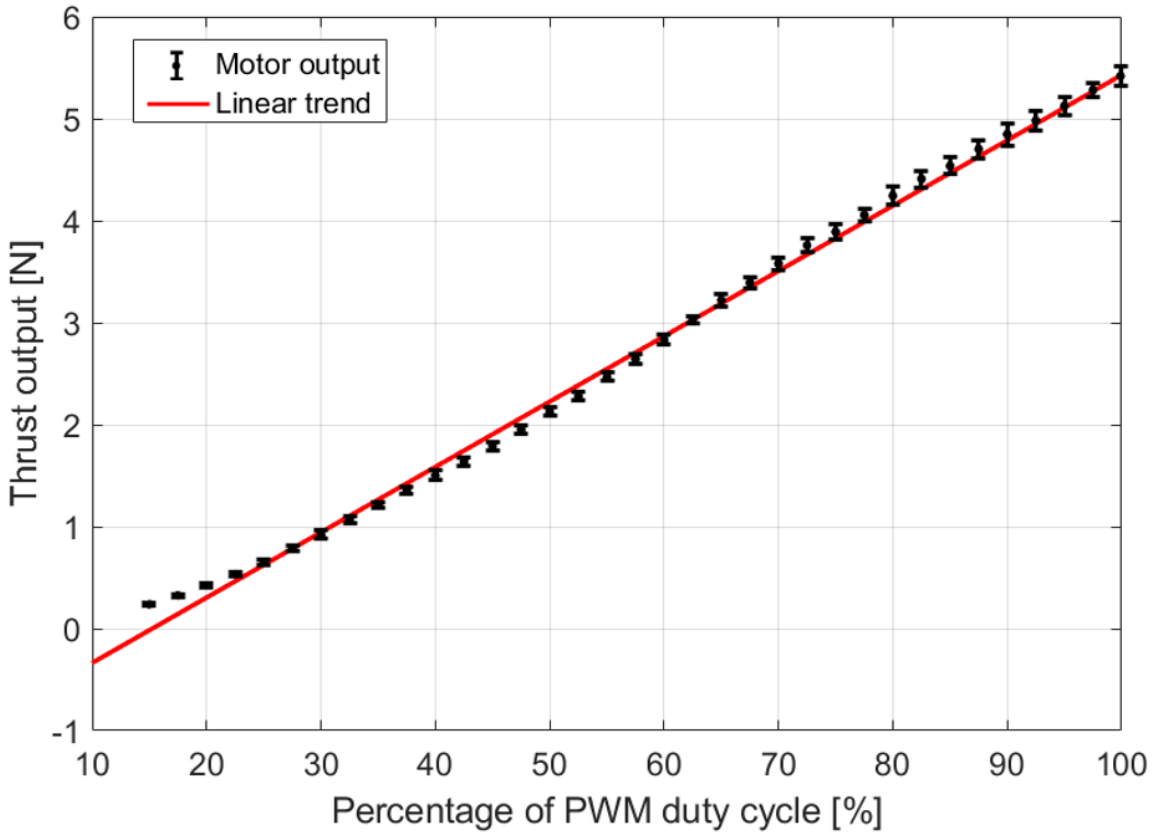


Figure 6.2: Thrust curve of one of the motors for the chosen propulsion system.

its weight [1]. This means the total weight of the vehicle should ideally not exceed 10.3 N, which is around 1 kg of mass. The total current draw at max throttle was found to be around 17 A per motor, which is well under the maximum 45 A per motor of the ESC, as well as the maximum 156 A total of the battery, and therefore all components should function properly with minimal heating.

6.1.2 ArduPilot Setup and first flight

A Navio2 board connected to a Raspberry Pi 3B+ were used as the vehicle autopilot. All the sensors required for navigation (inertial measurement unit, 3-axis gyroscope, barometer, compass) as well as the PWM output pins which control motor speed are located on board the Navio2, while the Raspberry Pi handles all the computations necessary to running the ArduPilot software. The Raspberry Pi can also directly function as the companion computer to the autopilot, allowing it to run external applications such as the RL algorithm. The setup of the autopilot was done by following the standard ArduPilot setup approach for a copter, which is outlined in the documentation [85].

A ground control station consisting of a laptop running QGroundControl ground station software [86] was used to verify vehicle telemetry, to perform standard vehicle setup, and to take manual control of the vehicle in case of emergency during testing. The version of ArduPilot used to pilot the vehicle is ArduCopter 4.3, configured for an X-flyer quadcopter vehicle. All software uploads and communications with the vehicle were conducted over the laboratory's Wi-Fi network, using Raspberry Pi's built-in Wi-Fi functionality. Note that when operating outdoors, the communications network would have to be revisited to allow for mobile field operations, and potentially replaced by a protocol that is more robust and has more range (low radio frequency, for example). ArduPilot, like many other UAV autopilots communicates using the MAVLink protocol. This protocol is a fast and efficient way of sending data packets over the preferred communications link [87]. As such, all commands to and responses from the autopilot must be properly packaged following the MAVLink convention. QGroundControl automatically packages its commands following this convention.

The accelerometers, gyroscopes and barometer were calibrated through QGround-Control, using ArduPilot's calibration functions. Since testing is being conducted indoors, the vehicle does not have access to satellite navigation during flight (GPS). Furthermore, due to the magnetic interference from computers, laboratory equipment, machinery and HVAC systems which is inherent to indoor operations, the compass could not be reliably used to obtain the heading of the vehicle, and an Optitrack motion capture systems was used to obtain a heading instead.

One of the main advantages of the current controller design is its ability to operate using only on-board sensing. Relying on a motion capture system would seemingly negate this advantage. It must be made clear that the load stabilization algorithm uses no aspect of the motion capture data, as the only data it requires to function properly is the load angle and load-angle velocity, which are obtained by the onboard sensor described in the next section, and the attitude, which is obtained by the fusion of the accelerometer and gyroscope data. The motion capture system was used during the experiment for two reasons: the first is to imitate a compass reading unaffected by the magnetic interference of an indoor laboratory setting. This ensured that the vehicle faced the same direction for the duration of testing, making the data easier to process. The second is to make sure the experiment proceeds safely. As the available testing area is limited, it was deemed safer to implement a failsafe which is triggered each time the vehicle approaches the limits of the testing area during training. The motion capture system therefore kept track of the vehicle location with

respect to the fence, and triggered the failsafe each time the position became unsafe. Infrared reflectors were secured to the vehicle's protective cover for tracking. These can be seen in Figure 6.1. Once the position and heading are recorded by the system, they are sent to the autopilot in a MAVLink packet over Wi-Fi. The full setup and operation of the Optitrack system and its communications links are explained in detail in [77]. Detailed instructions on how the autopilot was set up and configured to accept autonomous commands are provided in Appendix C.2.

After initial setup, the vehicle was ready for first flight. With the default PID gains provided by ArduCopter, the vehicle was found to oscillate about the roll axis on takeoff and on commanded hover. This would suggest that some of the gains were too high for this small vehicle, and therefore a large amount of overshoot was introduced in the roll response. As such, the roll rate proportional gain P was reduced until oscillations were no longer observed. Once the vehicle could hover reliably, ArduPilot's built-in PID tuning program was executed, in order to optimize the PID gains of the attitude control loop shown in Figure 5.4c for this vehicle. The process by which the gains are tuned is heuristic: small impulse rate commands are injected into the system and the response is measured. The gains are then slowly adjusted up or down until desired response characteristics are met according to ArduPilot's optimization criteria (fast response time, minimal overshoot, etc.) [88]. First, the roll rate response was tuned, then the pitch rate, and finally the yaw. The final values of the PID gains are shown in table 6.2. The stabilization P gain is tuned last and serves only to scale the magnitude of the error before it is sent to the PID rate controller. Its value is increased from one simply to accelerate the response.

Table 6.2: Tuned PID gains of the quadcopter testbed.

	Roll rate (ω_x)	Pitch rate (ω_y)	Yaw rate (ω_z)
Stabilization P gain	14.44	6.00	8.84
Rate P gain	0.0610	0.0726	0.3614
Rate I gain	0.0610	0.0726	0.03614
Rate D gain	0.001687	0.002275	0

Once the PID gains were tuned, the setup of the quadcopter testbed was complete, and it could then be used to perform the system identification briefly detailed in section 5.4.1, as well as run autonomous missions such as the load stabilization algorithm. Autonomous missions useful to testing (take-off, land, waypoint tracking, load stabilization by reinforcement learning) were programmed in Python using the Py-

mavlink library [89] which facilitates communication between a companion computer and an autopilot using the MAVLink protocol.

6.2 Payload sensor design

An on-board load-angle sensor was designed in order to measure the load-angle states α and $\dot{\alpha}$ which are relevant to the load stabilization algorithm. As mentioned in Chapter 2, one could use a vision-based system, but these tend to have high latency if the image processing software runs onboard the vehicle. Since the computations required for the RL algorithm are extremely time-sensitive and must be as efficient as possible, a design using two rotary optical encoders was opted for instead.

6.2.1 Mechanical Design

Two main types of high-precision digital encoders are available off-the-shelf: optical and magnetic. Magnetic encoders rely on magnetic field detection in order to make accurate angle estimates. However, it is a well-documented fact that electric propulsion systems such as those on-board small UAVs can cause magnetic field perturbation [90]–[93], which may affect the readings of a magnetic encoder. Furthermore, optical encoders are known to have higher measurement resolution and accuracy which leads to more reliable sensor readings [94]. For mainly these reasons, the design developed herein uses optical encoders. An isometric view of the CAD of the device is shown in Figure 6.3.

The design is based on that of a universal joint, with one side fixed to the bottom of the vehicle, and one side free to rotate about two axes. The load angles are measured by the two optical encoders located on the sides of each rotation axis. The angles are measured in the exact same rotation sequence as for the mathematical model, as shown in Figure 4.2 on page 36. When the moving portion of the design rotates about the x direction, the first encoder records the angle α_{enc} of the payload, by measuring the motion of the shaft which is aligned with the x-axis. As the payload attachment portion rotates about the resulting y direction, the second encoder records the angle β_{enc} of the payload, by measuring the motion of the payload attachment with respect to the second shaft. The sequence is illustrated in Figure 6.4 as measured by the device.

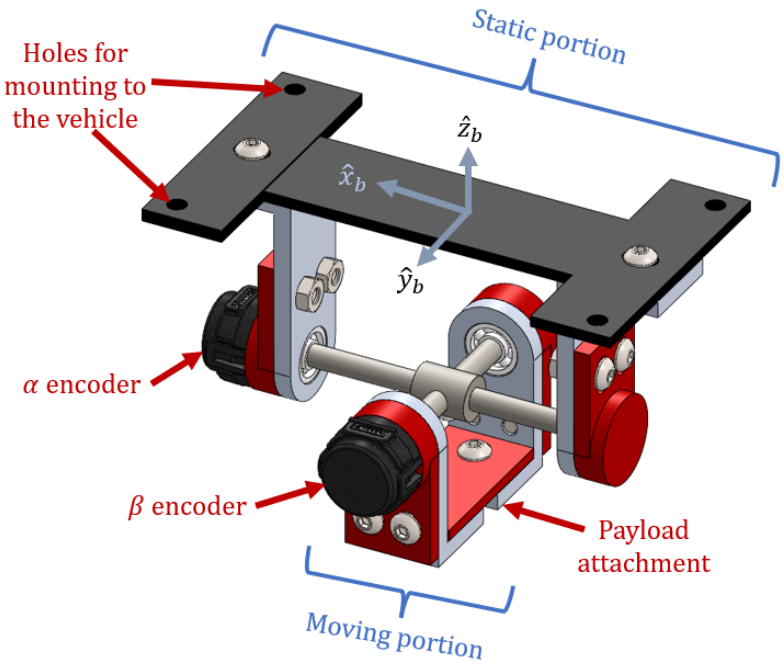


Figure 6.3: Isometric view of the load angle sensing device in its resting position ($\alpha_{enc} = 0$, $\beta_{enc} = 0$)

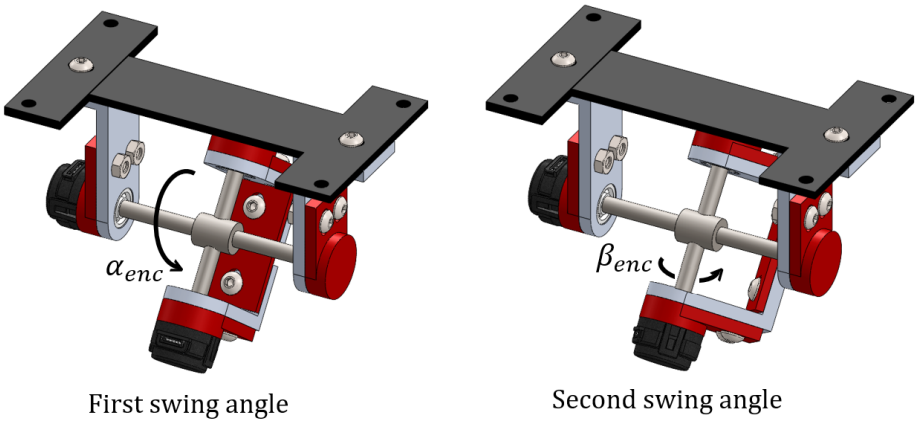


Figure 6.4: Measuring sequence for the load angle sensing device.

A small hole in the bottom of the payload attachment portion allows the load cable to pass through and be tied to the device. The full mechanical design details for the device, as well as its assembly instructions, are presented in Appendix C.3.

The device was manufactured and assembled, and was found to weigh around 50g and measure approximately 9cm in length, 6cm in width and 6cm in height, making it extremely compact and lightweight. The range of load angles which the device can measure, for α_{enc}, is the full 180 degree range. However, the range of the β_{enc} angle is

limited to around 120 degrees due to the payload attachment portion colliding with the larger shaft when the angle exceeds this range.

6.2.2 Integration

Optical encoders estimate angular displacement by recording discrete pulses of high voltage caused by a light receptor capturing the light from an emitter. By counting voltage pulses, one is able to determine by how many degrees the shaft has turned with respect to the start position. If an encoder comprises two emitter-receptors, one slightly offset from the other, the direction of rotation can also be determined. Encoders with two pulse measuring devices are called quadrature encoders. The E16 Micro optical encoder is a quadrature encoder with 4096 slots, or pulses per revolution (PPR). The two signals it records from its two emitter-receptors are referred to as channel A and channel B. A diagram of the output of a typical quadrature encoder is shown in Figure 6.5.

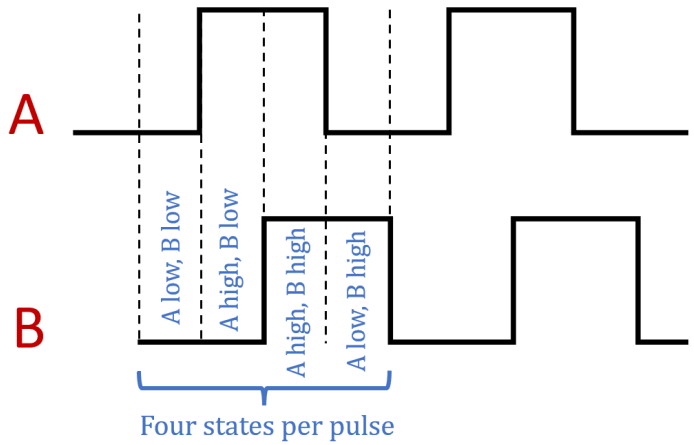


Figure 6.5: Pulse diagram of a quadrature encoder.

Channels A and B are usually offset by a quarter of an electrical cycle. When channel A leads (goes high before channel B), as is shown in the figure, the disk is rotating clockwise. When channel B leads, the rotation is counter-clockwise. Quadrature encoders can increase their measurement resolution beyond that of the number of slots, since there are effectively four unique states which arise for each pulse cycle (i.e. per slot). These are labelled in Figure 6.5. The E16 encoder can therefore output $4 \times 4096 = 16384$ states in total, or counts per revolution (CPR). Its angle measurement resolution is therefore around 0.02 degrees.

While it is possible for many microcontrollers to directly read the voltage pulses from channels A and B, and write a program to convert them to an angle, the Raspberry Pi has most of its input pins blocked by the Navio2 autopilot. However, the Navio2 provides access to some of the Raspberry Pi's serial communications pins, for reading data from external devices. The encoder pulses must therefore be converted to a serial communications protocol in order to be read by the companion computer, converted to an angle, and used by the RL algorithm. While there are a few ways to accomplish this, using a quadrature counter was deemed the simplest and most straightforward.

A quadrature counter is an integrated circuit which records the counts of a quadrature encoder and stores the value for access by a controller. As the shaft rotates clockwise, the stored number of counts increases, and as it rotates counter-clockwise, the stored number decreases. The quadrature counter which is readily available off-the-shelf is the LS7366R integrated circuit [95], which directly interfaces with channels A and B of the encoder, stores the number of counts since a reset was requested, and allows access to its data registers using the serial peripheral interface (SPI) communications protocol. Unfortunately, this is the only communications protocol supported by quadrature counters which are most commonly found off-the-shelf, and if a different protocol is required, it must be converted from SPI. This happens to be the case for the current autopilot/companion computer setup, in that, while the Raspberry Pi does have SPI functionality, only the pins corresponding to the inter-integrated-circuit (I2C) protocol are available through the Navio2.

The I2C/SPI bridge commonly available off-the-shelf is the SC18IS602B integrated circuit [96], which acts as a "translator" between an I2C controller and an SPI peripheral. A diagram of the full circuit used to communicate between the encoders and the Raspberry Pi is illustrated in Figure 6.6. For the SC18IS602B and LS7366R, click-boards containing the integrated circuits were used for ease of assembly and prototyping [97], [98]. These were soldered to a prototyping printed circuit board to facilitate the connection to the common ground and voltage line. Photos of the assembly and board stack are shown in Figure 6.7.

Upon startup, the quadrature counter must first be configured to count four states per pulse. Its communications frequency and bit order can also be configured. This is done by sending configuration commands to the quadrature counter, from the Raspberry Pi over the I2C bus. Before flying, the counts must also be reset to zero when the load is at rest, and therefore when its angles are $\alpha_{enc} = 0$ and $\beta_{enc} = 0$. This

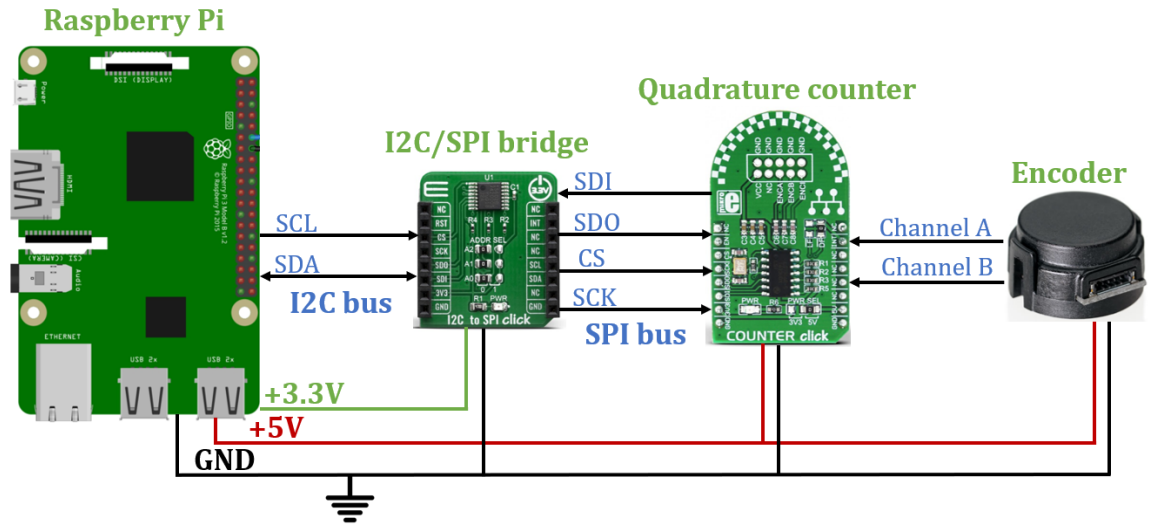
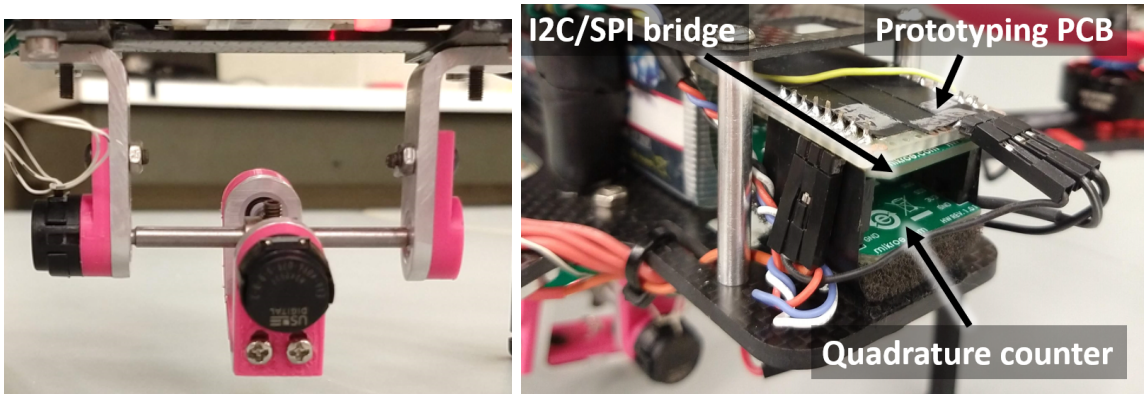


Figure 6.6: Diagram illustrating the connections between the Raspberry Pi, I2C/SPI bridge, quadrature counter and encoder.



(a) Load-angle sensing assembly.

(b) PCB stack.

Figure 6.7: Full assembly of the load-angle sensing device and communications boards.

is again done by sending a reset command over the I2C bus. Reading the number of counts is achieved in much the same way: by requesting a count number over the I2C bus, and reading the reply sent from the I2C/SPI bridge. Counts are converted to angles by dividing 360 degrees by the encoder's CPR.

With all new components mounted on the vehicle, the latter was found to weigh around 900g, which is slightly under the ideal limit of 1kg. A load of up to 100g can therefore be safely used to test the algorithm. As such, a load of 40g suspended by a cable of 1m in length was arbitrarily chosen to train the simulator and to test the algorithm experimentally. A 40g lead fishing sinker served as the payload, while the cable used was a 1m length of fishing line. The weight of the fishing line was found

to be negligible and was therefore not considered in the payload weight.

6.3 Testing the Payload Sensor Design

To determine whether the encoder sensor could reliably measure the load angle and load angle velocity, its measurements were compared against those obtained using the motion capture system. The Optitrack system running with Motive triangulation software has a tracking accuracy of around 0.2mm [99], and therefore its estimate of the load angle was taken as truth. Tests were first conducted for a single axis (the α axis), to determine whether the new sensor could function properly for the planar RL algorithm derived herein. Equivalent mathematical expressions for the load angles must be derived in order to compare identical quantities as measured by the Optitrack system and by the load-angle sensing device.

6.3.1 Encoder load-angle conversion

The load angle α, i.e. the state relevant to the RL algorithm, is the absolute load angle and is defined relative to a set of fixed axes (the inertial axes), as explained in Chapter 4. However, the load angles measured by the sensing device can only be measured with respect to the body-fixed axes, since the device is secured to the vehicle. A conversion is therefore necessary, to transform the angles measured by the device to the absolute load angles (those taken relative to the inertial frame). As explained in the previous section, the device measures the angles in the exact same rotation sequence as for the model. The only difference is the initial frame of reference from which they are measured. The rotation matrix derived in Chapter 4 for the payload can therefore be used again. That is:

$$ {}^{B}_{P}R(\alpha_{enc}, \beta_{enc}) = \begin{bmatrix} c_{\beta_{enc}} & 0 & s_{\beta_{enc}} \\ s_{\alpha_{enc}} s_{\beta_{enc}} & c_{\alpha_{enc}} & -s_{\alpha_{enc}} c_{\beta_{enc}} \\ -c_{\alpha_{enc}} s_{\beta_{enc}} & s_{\alpha_{enc}} & c_{\alpha_{enc}} c_{\beta_{enc}} \end{bmatrix} \tag{6.1} $$

describes the rotation of the payload frame with respect to the vehicle body frame, using the angles measured by the encoders α_{enc} and β_{enc}. The payload position with respect to the vehicle center of mass, expressed in the body-reference frame, is

therefore given by:

$$
\begin{aligned}
{}^{\boldsymbol{B}}\boldsymbol{\xi}_{\boldsymbol{P/B}} &= {}^{B}_{P}R(\alpha_{enc}, \beta_{enc})\ {}^{\boldsymbol{P}}\boldsymbol{\xi}_{\boldsymbol{P/B}} \\
&= \begin{bmatrix} c_{\beta_{enc}} & 0 & s_{\beta_{enc}} \\ s_{\alpha_{enc}} s_{\beta_{enc}} & c_{\alpha_{enc}} & -s_{\alpha_{enc}} c_{\beta_{enc}} \\ -c_{\alpha_{enc}} s_{\beta_{enc}} & s_{\alpha_{enc}} & c_{\alpha_{enc}} c_{\beta_{enc}} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -\lambda \end{bmatrix} \\
&= \begin{bmatrix} -\lambda s_{\beta_{enc}} \\ \lambda s_{\alpha_{enc}} c_{\beta_{enc}} \\ -\lambda c_{\alpha_{enc}} c_{\beta_{enc}} \end{bmatrix}
\end{aligned} \tag{6.2}
$$

The expression can be transformed to the inertial reference frame by pre-multiplying the vehicle rotation matrix defined in equation (2.1):

$$
\begin{aligned}
{}^{\boldsymbol{E}}\boldsymbol{\xi}_{\boldsymbol{P/B}} &= {}^{E}_{B}R\ {}^{\boldsymbol{B}}\boldsymbol{\xi}_{\boldsymbol{P/B}} \\
&= \begin{bmatrix} c_\theta c_\psi & -c_\phi s_\psi + s_\phi s_\theta c_\psi & s_\phi s_\psi + c_\phi s_\theta c_\psi \\ c_\theta s_\psi & c_\phi c_\psi + s_\phi s_\theta s_\psi & -s_\phi c_\psi + c_\phi s_\theta s_\psi \\ -s_\theta & s_\phi c_\theta & c_\phi c_\theta \end{bmatrix} \begin{bmatrix} -\lambda s_{\beta_{enc}} \\ \lambda s_{\alpha_{enc}} c_{\beta_{enc}} \\ -\lambda c_{\alpha_{enc}} c_{\beta_{enc}} \end{bmatrix} \\
&= \lambda \begin{bmatrix} -s_{\beta_{enc}} c_\theta c_\psi + s_{\alpha_{enc}} c_{\beta_{enc}} (-c_\phi s_\psi + s_\phi s_\theta c_\psi) - c_{\alpha_{enc}} c_{\beta_{enc}} (s_\phi s_\psi + c_\phi s_\theta c_\psi) \\ -s_{\beta_{enc}} c_\theta s_\psi + s_{\alpha_{enc}} c_{\beta_{enc}} (c_\phi c_\psi + s_\phi s_\theta s_\psi) - c_{\alpha_{enc}} c_{\beta_{enc}} (-s_\phi c_\psi + c_\phi s_\theta s_\psi) \\ s_{\beta_{enc}} s_\theta + s_{\alpha_{enc}} c_{\beta_{enc}} s_\phi c_\theta - c_{\alpha_{enc}} c_{\beta_{enc}} c_\phi c_\theta \end{bmatrix}
\end{aligned} \tag{6.3}
$$

Taking the model expression in Equation (4.10), comparing it with the above expression and dividing both sides by λ yields:

$$
\begin{bmatrix} -s_\beta \\ s_\alpha c_\beta \\ -c_\alpha c_\beta \end{bmatrix} = \begin{bmatrix} -s_{\beta_{enc}} c_\theta c_\psi + c_{\beta_{enc}} (s_{\alpha_{enc}} (s_\phi s_\theta c_\psi - c_\phi s_\psi) - c_{\alpha_{enc}} (s_\phi s_\psi + c_\phi s_\theta c_\psi)) \\ -s_{\beta_{enc}} c_\theta s_\psi + c_{\beta_{enc}} (s_{\alpha_{enc}} (c_\phi c_\psi + s_\phi s_\theta s_\psi) - c_{\alpha_{enc}} (c_\phi s_\theta s_\psi - s_\phi c_\psi)) \\ s_{\beta_{enc}} s_\theta + s_{\alpha_{enc}} c_{\beta_{enc}} s_\phi c_\theta - c_{\alpha_{enc}} c_{\beta_{enc}} c_\phi c_\theta \end{bmatrix} \tag{6.4}
$$

Taking the second element and dividing by the third element on both sides yields:

$$
-\tan\alpha = \frac{-s_{\beta_{enc}} c_\theta s_\psi + c_{\beta_{enc}} (s_{\alpha_{enc}} (c_\phi c_\psi + s_\phi s_\theta s_\psi) - c_{\alpha_{enc}} (c_\phi s_\theta s_\psi - s_\phi c_\psi))}{s_{\beta_{enc}} s_\theta + s_{\alpha_{enc}} c_{\beta_{enc}} s_\phi c_\theta - c_{\alpha_{enc}} c_{\beta_{enc}} c_\phi c_\theta} \tag{6.5}
$$

And therefore,

$$
\alpha = \arctan\left(\frac{s_{\beta_{enc}} c_\theta s_\psi - c_{\beta_{enc}} (s_{\alpha_{enc}} (c_\phi c_\psi + s_\phi s_\theta s_\psi) - c_{\alpha_{enc}} (c_\phi s_\theta s_\psi - s_\phi c_\psi))}{s_{\beta_{enc}} s_\theta + s_{\alpha_{enc}} c_{\beta_{enc}} s_\phi c_\theta - c_{\alpha_{enc}} c_{\beta_{enc}} c_\phi c_\theta} \right) \tag{6.6}
$$

And, from the first elements of the payload position expression:

$$
-s_\beta = -s_{\beta_{enc}} c_\theta c_\psi + c_{\beta_{enc}} (s_{\alpha_{enc}} (s_\phi s_\theta c_\psi - c_\phi s_\psi) - c_{\alpha_{enc}} (s_\phi s_\psi + c_\phi s_\theta c_\psi)) \tag{6.7}
$$

And therefore,

$$\beta = \arcsin\left(s_{\beta_{enc}} c_\theta c_\psi - c_{\beta_{enc}}(s_{\alpha_{enc}}(s_\phi s_\theta c_\psi - c_\phi s_\psi) - c_{\alpha_{enc}}(s_\phi s_\psi + c_\phi s_\theta c_\psi))\right) \tag{6.8}$$

Equations (6.6) and (6.8) can be used to obtain the load angles with respect to the inertial frame of reference, using the angles measured by the device and using the vehicle attitude which is regularly broadcast by the autopilot. Further simplifications of the expressions can be made if it is assumed that the yaw remains constant at zero degrees for the duration of the load stabilization. This assumption simply supposes that the inertial frame is yawed by the same amount as the vehicle frame at the start of the load stabilization experiment, and that the autopilot is able to hold the yaw constant. The behaviour of the control system is not affected, and computation is accelerated by removing the need to request and account for the yaw angle. Accounting for the constant yaw assumption, the expressions become:

$$\alpha = \arctan\left(\frac{-c_{\beta_{enc}}(s_{\alpha_{enc}} c_\phi + c_{\alpha_{enc}} s_\phi)}{s_{\beta_{enc}} s_\theta + s_{\alpha_{enc}} c_{\beta_{enc}} s_\phi c_\theta - c_{\alpha_{enc}} c_{\beta_{enc}} c_\phi c_\theta}\right) \tag{6.9}$$

$$\beta = \arcsin\left(s_{\beta_{enc}} c_\theta - c_{\beta_{enc}}(s_{\alpha_{enc}} s_\phi s_\theta - c_{\alpha_{enc}} c_\phi s_\theta)\right) \tag{6.10}$$

After applying the planar dynamics assumption ($\beta_{enc} = 0$ and $\theta = 0$), the above expression in α was used to compute the load angle in the y-z plane during testing of the sensing device and of the planar stabilization algorithm derived in Chapter 5.

6.3.2 Optitrack load-angle conversion

Since the vehicle is already being tracked for safety purposes, its position in the inertial frame can be readily obtained by the Optitrack system and broadcast to the companion computer storing the experimental data. However, model assumptions place the payload attachment point as coincident to the vehicle center of mass, not to its geometric center. As such, for the sake of comparing the load angle sensing device to an accurate model, the rigid body center calculated by the Optitrack system was moved to coincide with the vehicle center of mass. The location of the center of mass was determined by weighing and measuring each component of the vehicle, and modelling them as accurately as possible using SolidWorks. The center of mass of the full assembly was estimated using SolidWorks' mass properties feature. This approach assumes each component of the assembly has a uniform mass distribution, which was deemed sufficient for the purpose of computing a more accurate estimate of the inertial load angle.

The payload (the fishing sinker) was inserted into a slightly larger block of foam, and infrared reflectors were affixed to the outside of the block to obtain its geometric center. From Equation (4.11) on page 40:

$$^{\boldsymbol{E}}\boldsymbol{\xi}_{\boldsymbol{P/B}} = {}^{\boldsymbol{E}}\boldsymbol{\xi}_{\boldsymbol{P}} - {}^{\boldsymbol{E}}\boldsymbol{\xi}_{\boldsymbol{B}} \tag{6.11}$$

That is, the payload position with respect to the vehicle center of mass, in inertial frame coordinates, is obtained by subtracting the vehicle position from the payload position. Substituting in the model expression in Equation (4.10) yields:

$$\lambda \begin{bmatrix} -s_\beta \\ s_\alpha c_\beta \\ -c_\alpha c_\beta \end{bmatrix} = \begin{bmatrix} x_p - x \\ y_p - y \\ z_p - z \end{bmatrix} \tag{6.12}$$

Dividing the first element by the second on both sides yields:

$$-\tan\alpha = \frac{y_p - y}{z_p - z} \tag{6.13}$$

The above expression can be solved for the first load angle:

$$\alpha = \arctan\left(\frac{y - y_p}{z_p - z}\right) \tag{6.14}$$

And, from the first element on both sides:

$$-\lambda s_\beta = x_p - x \tag{6.15}$$

The above expression can be solved for the second load angle:

$$\beta = \arcsin\left(\frac{x - x_p}{\lambda}\right) \tag{6.16}$$

All necessary quantities can be directly obtained from the tracking system. The above expression for α was used to compute the load angle in the y-z plane during testing.

6.3.3 Test results

Two sets of tests were first conducted to assess the performance of the load-angle sensing device. For the first, the vehicle was instructed to hover in place at an altitude of 2m while the load was disturbed using a long rod and left to swing freely. This was done to assess its accuracy for large load angles. For the second, the vehicle was instructed to follow a series of waypoints, stopping and starting abruptly in order

to disturb the load, mimicking measurements obtained during a "normal" flight with irregular load swing. The load angles were sampled at a frequency of 10 Hz, that is, the same frequency as the RL loop. Results are shown in Figure 6.8 for three load disturbance tests in hover.

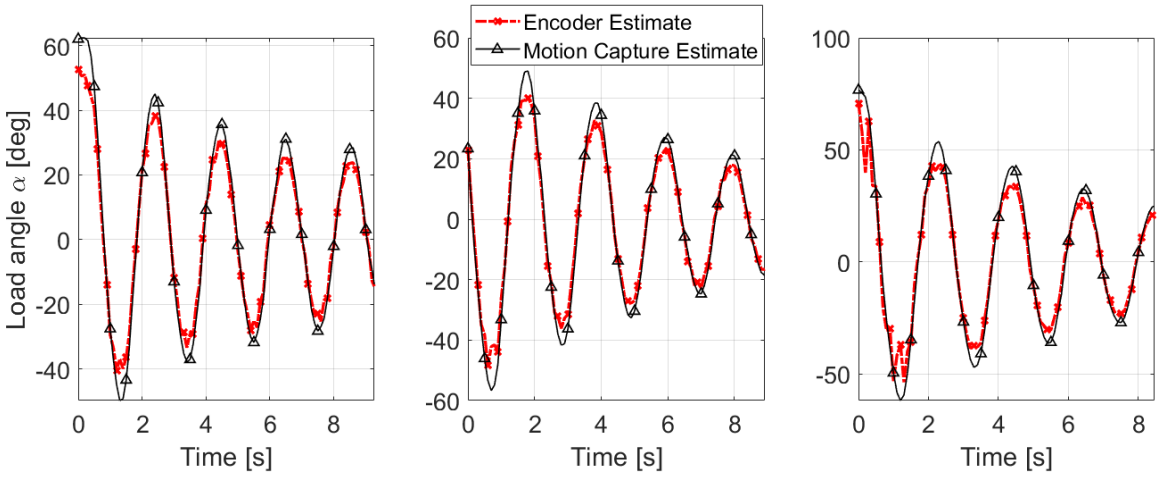


Figure 6.8: Comparison of the load angle measurements in hover with load disturbance.

Overall, the device was found to follow the motion capture data with relatively small discrepancies. The mean average errors (MAEs) of the three shown hover experiments were found to be around 2.96, 2.60 and 3.39 degrees respectively, increasing in proportion to the maximum load angle reached. These MAEs correspond to less than 5% of their maximum swing amplitudes. While the MAE was generally found to vary slightly with the quality of the initial encoder calibration, it was rarely found to exceed 5% of the maximum amplitude for the hover tests. However, the difference between the measurements becomes most obvious at the height of the load swing: by observing the peaks in Figure 6.8, it is apparent that the sensing device underestimates the magnitude of the load angle at the height of the swing, contributing greatly to the MAEs. Furthermore, the measurement of the angle at these peaks becomes slightly noisy, as is made most apparent during the third hover test, at around 1 second.

In order to illustrate the discrepancy in a clearer way, the absolute error between the two measurements was plotted and is shown in Figure 6.9. The locations of the "peaks" are identified on the graph by placing markers where the load angular velocity changes sign, as measured by the motion capture system. While the absolute

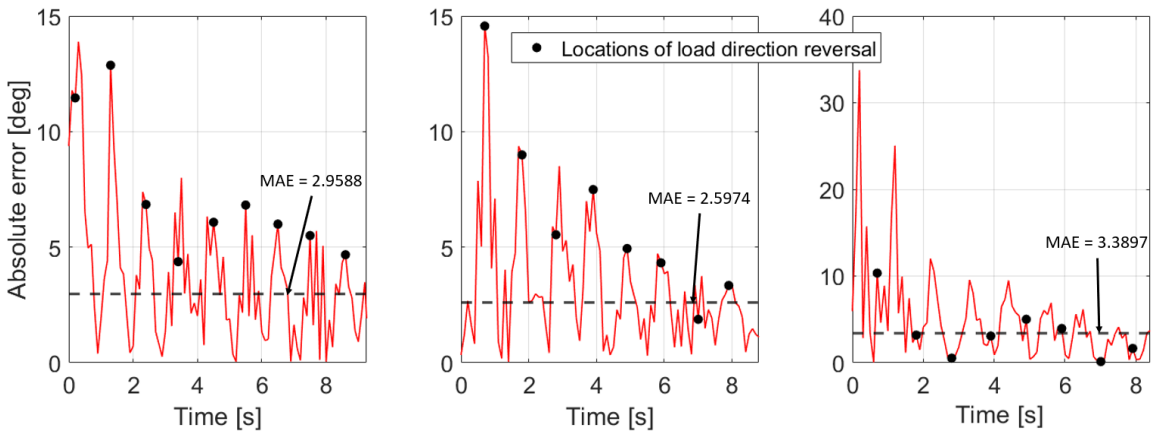


Figure 6.9: Absolute error between the angle measurements in hover with load disturbance.

error can be seen to generally stay well below the MAE when the load is in mid-swing, it is observed to peak sharply at most locations which coincide with the load direction reversal (i.e. the height of the swing), regularly exceeding 20% of the swing amplitude. The error also seems to grow exponentially with the swing amplitude.

There are two main reasons for a discrepancy in the measurements. The first is that the accuracy of the sensing device is contingent on the load cable being taut or rigid, and on the payload attachment portion aligning with the cable. At the height of the swing, when velocity is zero, the component of payload weight which acts along the cable direction may not be large enough to produce the tension required to pull the payload attachment along and measure an accurate angle. In this case, the moving portion of the sensing device would swing under its own weight, and measure an angle which is smaller than the true angle, which is observed in practice. Another possible cause for a discrepancy is that the sensing device is not located directly on the center of mass of the vehicle, which is what is assumed by the motion capture measurement, but rather directly beneath it. In hover, the difference in the measured angles should therefore grow as the load angle increases. This would typically result in the motion capture estimate being smaller in magnitude than the encoder estimate, which does not appear to be the case here. It is therefore unlikely that the center of mass assumption affects the measurements significantly.

Results for the second type of test, the waypoint tracking, are shown in Figure 6.10. Similarly to the hover tests, the waypoint tracking test appeared to produce similar data for the encoder and for the motion capture system. The waypoint tests demonstrate that the sensor can accurately measure the angle in regular flight conditions,

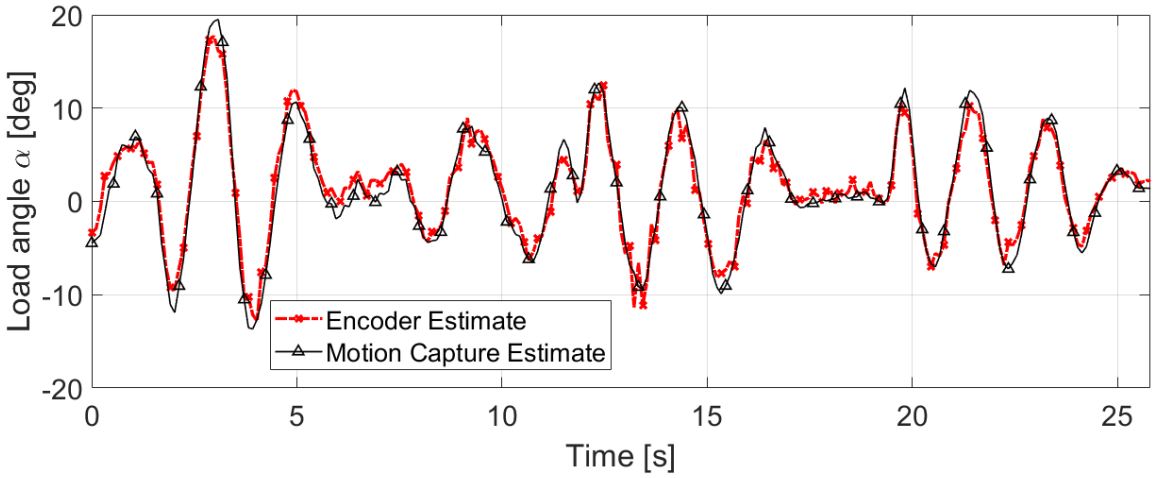
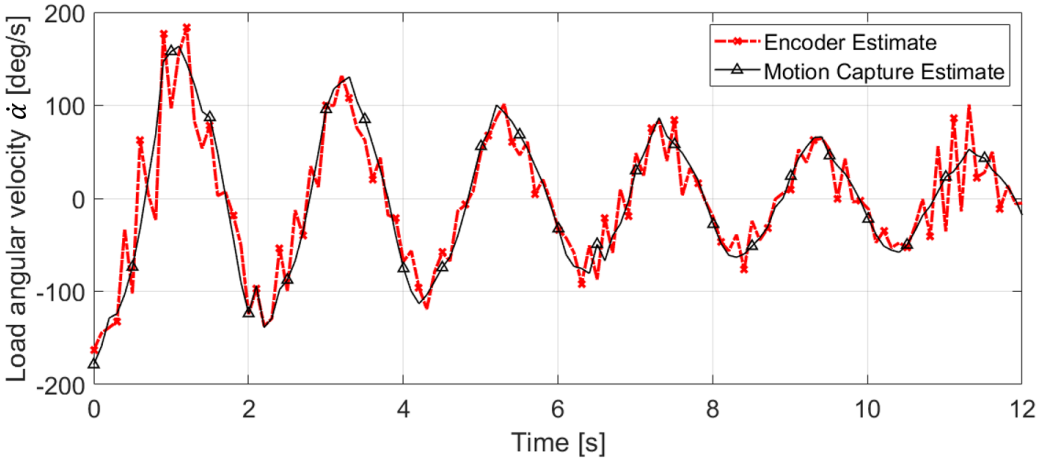


Figure 6.10: Comparison of the load angle measurements during waypoint tracking.

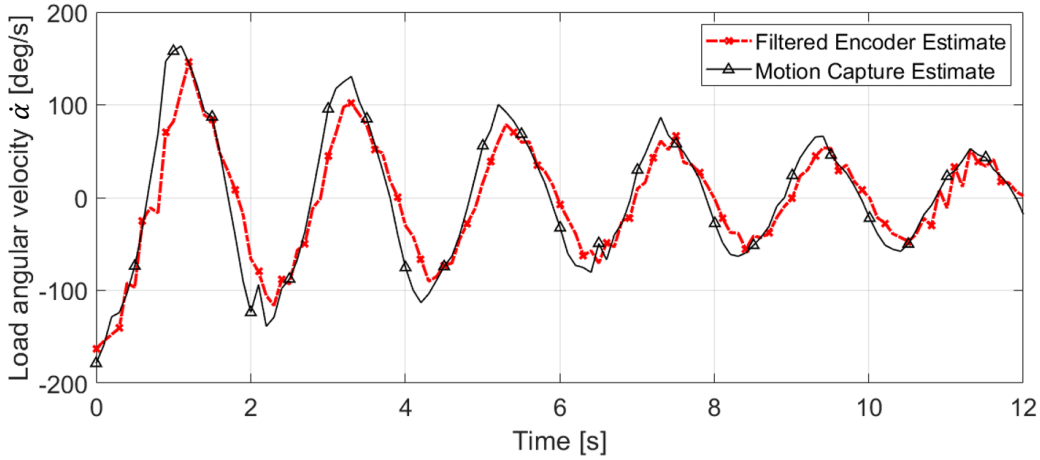
with irregular swing, albeit for smaller load angles. The MAE for these types of tests was rarely found to exceed 2 degrees, and the absolute error was rarely found to exceed 5 degrees.

Load angle velocity was obtained by dividing two consecutive load angle measurements by the recorded time step. The encoder estimate was found to be rather noisy when compared to the motion capture estimate. To yield more accurate results, the encoder velocity measurement was filtered using a simple exponential filter with a coefficient of 0.45. The exponential filter was selected for its ease of implementation and ability to filter in real-time, and the coefficient was tuned by trial-and-error until the filtered data was smoothed without introducing a large delay or error, across all hover tests. A comparison of the measured load angular velocities is shown in Figure 6.11 for one of the hover tests, with and without filtering of the encoder estimate.

From Figure 6.11a, it can be seen that the data is plagued by sharp oscillations across the full range of angular velocities measured during the test. The oscillations become especially prominent around 11 seconds, when the amplitude of the load-angle velocity is smallest. It is likely that the taut cable condition was not exactly met in this portion of the test, and the encoder recorded slightly varying angles as the weight of the moving portion caused the shaft to turn. Small variations, when converted to a velocity by dividing by a small time step, translate to sharp peaks. From Figure 6.11b, however, it can be seen that the peaks observed in the unfiltered signal have been smoothed considerably, and that the filtered velocity trails the motion capture estimate by about 0.1 seconds, due to the delay introduced by filtering. It is expected



(a) No filtering.



(b) With exponential filtering.

Figure 6.11: Comparison of the load angular velocity measurements for one of the hover tests.

that this small delay will not have much effect on the measurements as they are relevant to the RL algorithm, since the dynamics of the load angle are relatively slow compared to the RL time step. The RL experiment should validate this hypothesis.

Overall, the device was deemed to have a satisfactory performance when considering the results from the two sets of tests, on the condition that a filter is applied to the angular velocity measurement. The device was therefore used for the onboard computation of the load angle and load angle velocity for the stabilization experiment. The results of the latter experiment should provide insight into whether the sensing is accurate enough to stabilize the load efficiently barring any model uncertainties, and to facilitate the adaptation of the algorithm to the physical system. These are presented in the next chapter.

7. Results and Discussion

The following chapter describes and discusses the experimental results obtained during the testing of the reinforcement learning algorithm developed in Chapter 5, when applied to the experimental testbed.

7.1 Load Stabilization using the Sensing Device

Algorithm 1 was implemented in Python for the purpose of training on the physical system. A set of converged weight vectors from the simulations, as described in Chapter 5, was uploaded to the Raspberry Pi and used as the "initialization" of the load stabilization algorithm for the experimental testbed. Each RL computation step was found to take less than 10 ms while running on the Raspberry Pi, indicating that the RL frequency of 10 Hz (0.1 seconds) is slow enough to accommodate the computation in real-time. The training episodes were structured in a similar way to the hover tests described in the previous chapter: The vehicle was instructed to hover in place at an altitude of 2m, and the load was disturbed with a long rod, simulating a wind gust or unforeseen strike to the slung payload. The load stabilization algorithm was then "activated" by the operator while the load was in mid-swing.

After observing the vehicle behave very erratically during the first stabilization test, the angle sensing used by the algorithm was switched to the motion capture estimate, and the test was run again, comparing the data from both sensors. Results for one of the tests are shown in Figure 7.1. While both measurements appear to agree in the beginning of the experiment, as was observed for the hover tests, the measurements begin to diverge considerably when the load stabilization algorithm takes over at around 1.7 seconds. While the motion capture estimate stays relatively smooth, the encoder estimate becomes jittery and noisy. This is likely again due to the taut cable condition. When the vehicle manoeuvres aggressively and rolls abruptly to stabilize the load, its acceleration in the y-axis undergoes large fluctuations. This can be seen from Figure 7.2, where the vehicle acceleration in the y direction was recorded for another of the load stabilization tests. Before the load stabilization control takes over, the acceleration registered by the inertial measurement unit (IMU) is relatively low, and can be attributed simply to the vibration generated by the

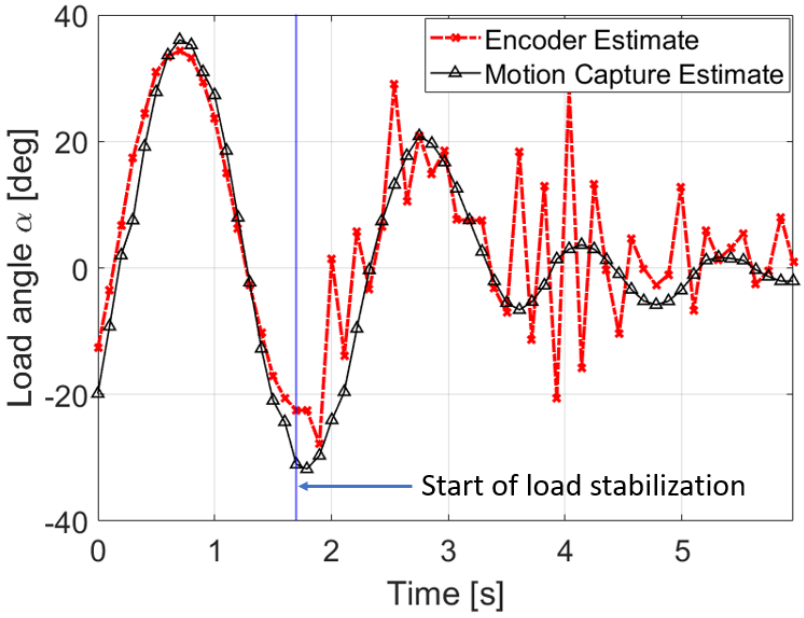


Figure 7.1: Comparison of the load angle measurements during load stabilization.

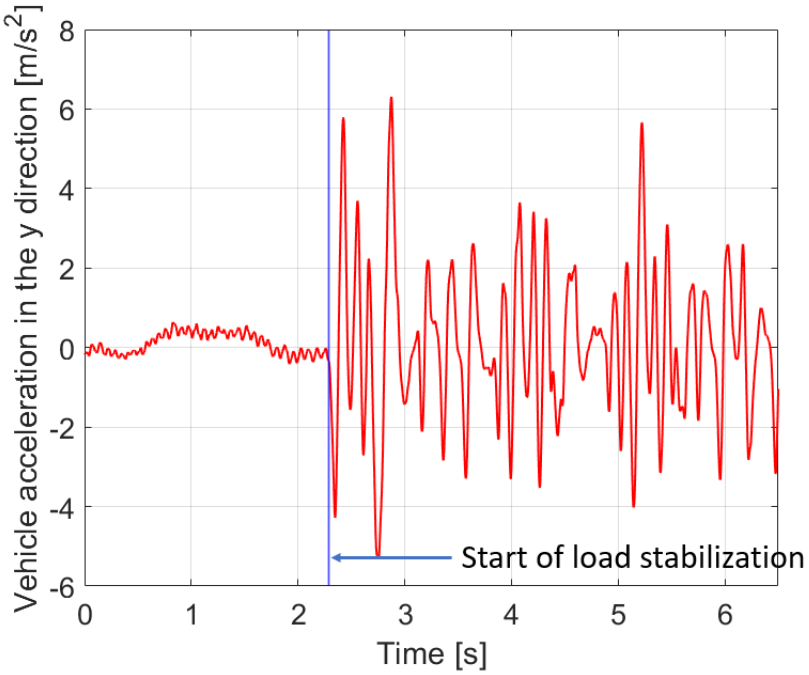
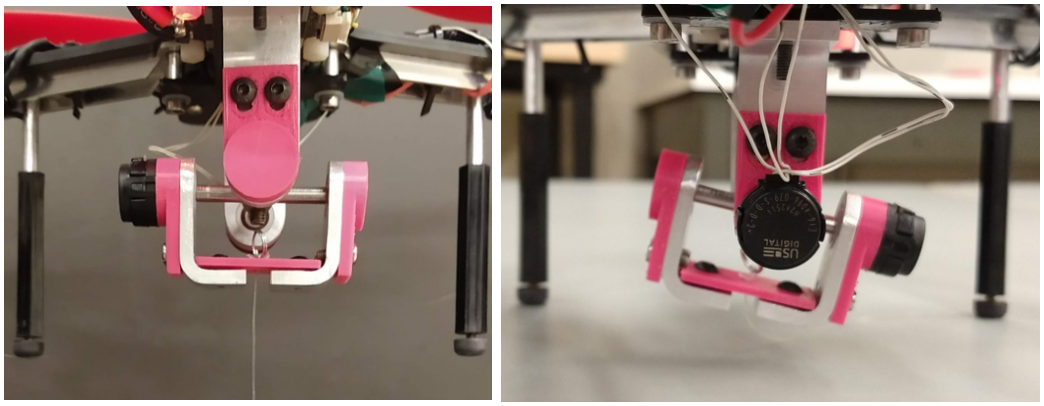


Figure 7.2: Vehicle acceleration in the y direction during load stabilization.

propulsion system. However, when stabilization begins, the aggressive manoeuvring of the vehicle generates large variations in its acceleration. As such, jerk is transmitted to the cable, which causes its tension to change abruptly, perhaps occasionally decreasing to zero. When there is little or no tension to direct it, the moving portion of the device tends to swing freely under its own weight, as was briefly mentioned previously. The problem is likely made worse by the fact that the moving portion of the sensing device is unbalanced.

As illustrated in Figure 7.3a, when tension in the cable is present, the moving portion of the device aligns with the cable direction, since the tension is distributed on each side and contributes to moment equilibrium about the rotation axis. However, when there is no tension, the weight distribution of the moving portion of the device causes a moment imbalance. Looking at Figure 7.3b, it becomes clear that the center of mass of the moving portion is located beneath the rotation axis, and slightly shifted towards the encoder, which manifests as a slight tilt at equilibrium. It is likely that this imbalance is exaggerating the sharp oscillations in the angle measurement each time tension in the cable decreases, by attempting to bring the moving portion back to its unloaded equilibrium position. During the waypoint experiments, the roll commands tended to be relatively smooth compared to those of the stabilization experiment, and so the cable was likely kept taut for most of the test, which is why this behaviour was rarely observed.



(a) Device in tension. (b) Device unloaded.

Figure 7.3: Photos of the load-angle sensing device in tension and at no load.

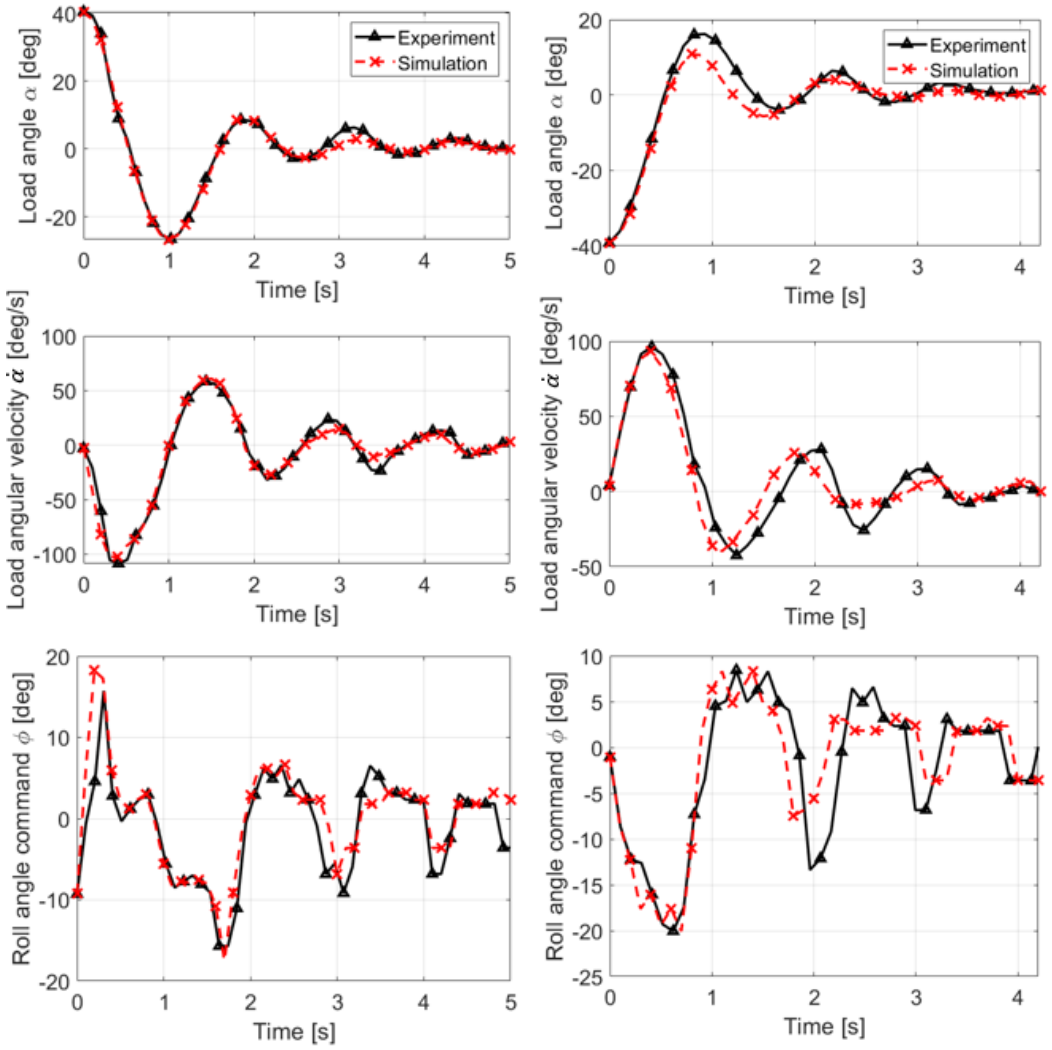
While the device was found to produce reliable measurements during smooth vehicle manoeuvres, its measurements become far too inaccurate and unreliable during aggressive manoeuvring to be used for control.

As with any controller, the RL agent's knowledge of its state is entirely dependent on sensor accuracy, and its control action can therefore only be computed in reaction to this "observed" state. Furthermore, observing the evolution of the state is crucial to the RL agent's learning process: if a different "next state" is expected than the one observed, a temporal-difference error is generated, and the RL agent adjusts its control policy accordingly. It is therefore essential that the sensor reporting the state have only a small amount of noise and bias. Small sensor errors would generate small temporal-difference errors with some amount of variance, and it is expected that the RL agent would eventually adjust its control policy in reaction to slightly biased or noisy measurements. These would constitute something like a model discrepancy to be accounted for through training by the RL agent. However, sensor errors of the magnitude shown in Figure 7.1 are too large to even allow the agent to continue its training safely and without incurring significant errors in its updates to its control policy. Due to this unfortunate shortcoming, without reworking the design, the load-angle sensor could not be used for the load stabilization experiment. In order to determine whether the stabilization algorithm could still work on the experimental testbed given reliable sensing, it was decided to use the motion capture estimate of the load angle and load angle velocity to test the RL algorithm. These tests are detailed in the following section.

7.2 Load stabilization algorithm using Optitrack

Various experiments were conducted to assess the performance of the algorithm, given reliable sensing. First, stabilization control was activated when the motion capture system registered a load angle velocity near zero, in order to mimic the training that the agent received in simulation ($\dot{\alpha}_0 = 0$). Initial load angles approximately ranged between $\pm 10^\circ$ and $\pm 40^\circ$. Results for two of the experiments on the larger swing angles, one with a positive initial angle and one with a negative initial angle, are shown in Figure 7.4. The experimental data was compared to the simulation prediction (using the delayed roll angle dynamics), to observe the effect of model discrepancies on the stabilization behaviour.

Overall, when the initial angular velocity was near zero, the controller was able to reliably stabilize the slung payload in a few seconds for angles ranging between $\pm 10^\circ$ and $\pm 40^\circ$, in a manner similar to the simulation. However, model discrepancies can still be observed, especially in Figure 7.4b for the large, negative initial load angle.



(a) Positive initial angle.

(b) Negative initial angle.

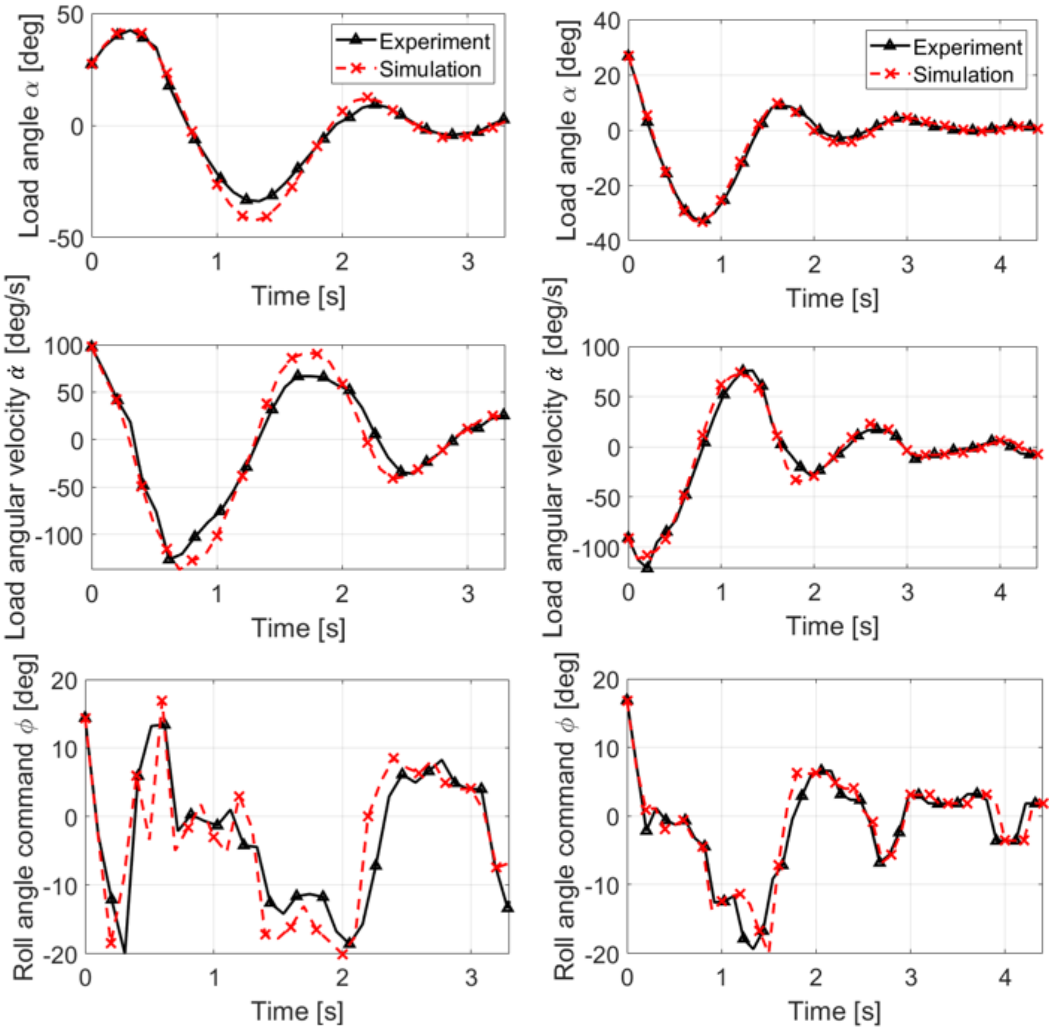
Figure 7.4: Experimental results of the load stabilization for $\alpha \approx \pm 40°$ and $\dot{\alpha} \approx 0$.

In both cases, the simulation predicts that the load will be stabilized more quickly and efficiently than it is in reality, notably when magnitude of the load-angle velocity stays below $50°/s$. This is to be expected since the majority of the training was conducted on the simulator, and therefore the final weight vectors approach optimality for a system which behaves like the model. It is likely that the simulated controller leverages the rigid cable assumption in order to stabilize the payload more efficiently. In reality, a force can only be transmitted in tension through the cable, while in the simulator, the "cable" can be used to transmit tension and compression, and manipulate the dynamics of the payload in this manner. In the experiment, the controller is still observed to react appropriately to the "new" encountered dynamics, leveraging

knowledge it has acquired from the simulation. For example, around 2 seconds into the negative load angle stabilization, the agent commands a larger roll angle than what is predicted it will need, in response to the larger load angle velocity of the payload. Overall, the differences observed between the experiment and the prediction are minimal, and with enough training, the controller should be able to learn the new dynamics of the system and optimize its control policy accordingly. Sufficient training would translate to a temporal-difference error which is shown to consistently be below an acceptable threshold, indicating that the solution has converged.

For the next tests, stabilization control was activated arbitrarily by the operator, in order to assess the performance of the algorithm in stabilizing loads with non-zero velocities. Initial load angles approximately ranged between $\pm 5^\circ$ to $\pm 50^\circ$, while initial load-angle velocities approximately ranged between $\pm 35^\circ/s$ and $\pm 125^\circ/s$. Results for two of the experiments are shown in Figure 7.5, in order to demonstrate the behaviour with both positive and negative load angle velocities. Again, the experimental data was compared to the simulation prediction.

In both the simulated and experimental case, the controller was observed to stabilize the slung payload with a non-zero initial load-angle velocity, suggesting that the initial hypothesis regarding the training was correct, and the agent is still be able to generalize its knowledge to non-zero velocities despite the fact that it was trained for initial load angle velocities of zero only. Interestingly, when looking at the load angle and load-angle velocity plots of Figure 7.5a, it appears that the load is stabilized slightly more efficiently in practice than in simulation. During this particular experiment, the load-angle velocity registered very large values ($\approx -125^\circ/s$ to $\approx 100^\circ/s$). It is possible that the unusually high velocities caused an increased amount of wind drag to act on the foam block payload, which contributed to it stabilizing more quickly than for the simulation. The effect does not appear to be as pronounced for the experiment of Figure 7.5b, where the velocities reached were not as high. It is also possible that the contribution of wind drag is the reason why the differences between the simulation and experiment were not as pronounced for the $\dot{\alpha}_0 \approx 0$ experiments of Figure 7.4, when the amplitude of the velocity was above $50^\circ/s$. While the weight vectors are optimized for the model, and therefore stabilization should be more efficient on the model, the contribution of wind drag, which increases exponentially with velocity, helps stabilization. The contribution of wind drag is yet another model discrepancy which the agent must optimize for in future training, since aerodynamics were not considered for the model.



(a) Positive initial angle.

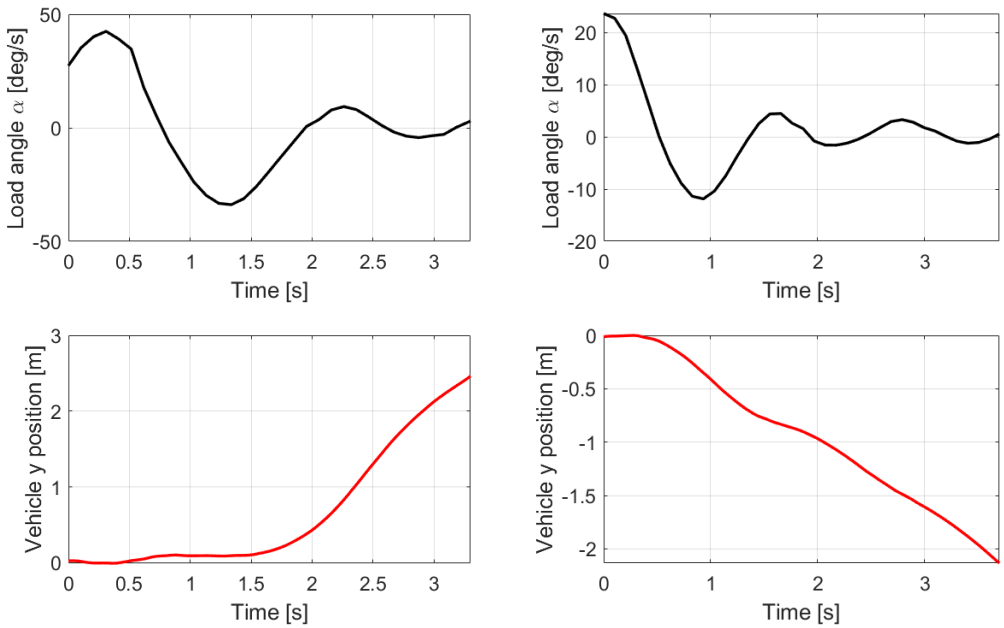
(b) Negative initial angle.

Figure 7.5: Experimental results of the load stabilization for non-zero load-angle velocities.

It is interesting to note that, across most experiments, once the initial stabilization has occurred, the series of control actions commanded by the agent becomes extremely similar despite the fact that the stabilization was started from very different load-angle states. This can be seen in Figure 7.4, where the sequence of commanded roll angles becomes very similar after two seconds for the positive initial angle, and after one second for the negative initial angle. The control sequence also resembles that of Figure 7.5b once the experiment is around the 1.7s mark. The control policy therefore appears to be bringing the load to a familiar point in the state-space where it is quasi-stable, and then always applying a similar series of actions from then on in order to stabilize it completely.

By consulting the vehicle flight logs, it was found that the PWM signal output by the autopilot rarely exceeded 1700 μs (70% throttle) across all tests, indicating that the motor thrust exponent, which was set for accurate throttle characteristics below 70%, was selected appropriately. Details regarding the exponent selection can be found in Appendix C.2. Furthermore, during the experiments, the vehicle was not observed to drift significantly in the z direction, with its altitude always staying within a few centimeters of the hover altitude of 2m, while its fluctuations in acceleration were mostly attributed to the vibration of the propulsion system. Since the experimental data for stabilization follows the simulation rather closely, it can be said that the hypothesis for the expression of the total thrust force T was correct, and assuming a constant z acceleration was sufficient to achieve a first RL solution which would be stable and allow the physical system to achieve the goal of slung payload stabilization.

However, during the experiments, the vehicle was found to exceed the allowable testing area ($y = \pm 3$ m) a few times, requiring the controller to stop and return to the center of the area for safety. The position of the vehicle in the y direction is plotted in Figure 7.6 for two of the stabilization experiments in order to illustrate how the vehicle moves in the y-z plane during load stabilization.



(a) Positive y drift.

(b) Negative y drift.

Figure 7.6: Motion of the vehicle along the y direction during load stabilization.

Since the agent is commanding a roll angle and the latter directly affects the y dy-

namics of the vehicle, the quadcopter is observed to drift fairly significantly along the y axis. Exceeding the testing area happened often for experiments where the load had a large, positive angular velocity near the beginning of stabilization, since these areas of the state-space corresponded to aggressive changes in the roll angle command. This result illustrates an important flaw in the design of this control algorithm: since the useful state-space consists only of the load-angle state, the agent is not made aware of its position y as it moves to stabilize the slung payload. It therefore uses as much space as is necessary to achieve the goal as quickly and efficiently as possible. While this results in rapid load stabilization and smaller weight vectors, it is not ideal for applications where space might be limited. Possible improvements to remedy this shortcoming are discussed in the concluding remarks.

8. Conclusions and Future Work

Overall, the control algorithm, with minimal training on the experimental testbed, was found to stabilize the slung payload consistently for initial load angles of less than 50° in magnitude across a range of initial load-angle velocities of $\pm 125°/s$. The proposed method was found to be a general success, and can also be said to have made numerous contributions to the application of slung payload stabilization using model-free RL. These are outlined at the end of this chapter.

However, the solution presented herein is still only a proof of concept. Several improvements can be made to the algorithm and testbed, and additional studies should be conducted before using the proposed method for parcel delivery. Recommendations for future work on the simulation, load-angle sensing device and experiment are presented in the next section.

8.1 Future Work

8.1.1 Simulations

The training of the RL agent on the simulator could be accelerated and improved by incorporating specific metrics for a successful stabilization, and successful convergence of the weight vectors. Currently, each training episode lasts 20 seconds, regardless of whether the agent has accomplished the goal or not. This adds unnecessary computations since the simulator computes the dynamics for the full 20-second span, sometimes when the load has in fact already been stabilized, which is not relevant to the learning. The episodes could be instead terminated either after 20 seconds, or when the magnitude of both load states has decreased to within a certain threshold. The number of episodes (10000 or 5000) is also currently arbitrary: it is sufficiently large to assume adequate performance at the end of learning, but there is currently no mathematical metric which dictates when an "acceptable" solution has been reached. A condition could be placed on the temporal-difference error which would terminate the learning when the error is sufficiently small. However, due to the random nature of the training, the error would need to be verified across the entirety of the state-space, in order to ensure that all states have been visited a sufficient number of times. This would ensure adequate convergence and would avoid possible erratic

or dangerous behaviour for states which have not been visited in simulated training. While making these changes to the simulator would not affect the approach nor the final solution, they would enable the operator to train a new vehicle more quickly and efficiently.

Further studies could also be conducted to determine whether incorporating the roll angle dynamics in the training is in fact necessary. In the case of the particular vehicle used herein, it seems as though incorporating these was not necessary, as the control function obtained assuming perfect roll angle tracking did not appear to adapt to the realistic dynamics. Testing could be conducted using a vehicle with slower roll angle dynamics, in order to determine whether the dynamics of the quadcopter used herein were simply too fast to produce a meaningful difference between perfect and imperfect tracking, or whether the dynamics of the roll angle are in fact always irrelevant to the optimal solution. If the roll dynamics are found to always be irrelevant to the solution, the algorithm becomes easier to implement, and more versatile, since it no longer relies on prior system identification.

Finally, studies could be conducted to look into the role of the standard deviation in the control function. If it is found to be largely unnecessary, the amount of parameters which need to be stored onboard the vehicle could be reduced. The algorithm used herein would only need to be slightly modified to make use of the deterministic policy gradient theorem (PGT). While the classic actor-critic algorithm relies on a stochastic policy for exploration and to account for random noise in the environment, the incorporation of a standard deviation does not appear entirely useful in this case. It may be useful to the simulated training, since it allows different actions to be explored, yet no random noise is considered in the simplified model and the standard deviation simply decays to a very small value. In practice, the assumption of Gaussian random noise in the environment might be too simplistic to produce a useful difference between a control function which is deterministic (i.e. with $\sigma = 0$) and one which is stochastic. Testing in an outdoor setting with actual wind gusts may provide insight into whether a stochastic policy is necessary.

8.1.2 Load-angle sensing device

The design of the load-angle sensing device proved to be advantageous in a few ways: it is very small and lightweight, consumes low amounts of power and is simple to manufacture. However, based on the results of the experiments, it is clear that the

design must be reworked if it is to be useful for practical applications: The algorithm cannot be used in an outdoor setting unless sensing is fully on-board. The main problem with the design was identified as a weight imbalance in the moving portion, which generated a righting moment each time the cable tension decreased. In order to remedy this issue, future iterations of the design should have the weight balanced, such that the center of mass of the moving portion becomes coincident with the rotation axis. In this manner, if cable tension is removed, moments about the rotation axis would remain balanced and the device would not try to right itself, eliminating most of the oscillations in the sensing of the load angle. Additionally, during assembly, it was found that the two sides of the static portion had to be perfectly aligned in order to ensure that the bearings would remain straight and allow rotation with minimal friction. In the future, to facilitate the assembly, the full static portion could be machined out of a single piece of aluminum U-channel, which would ensure alignment of the bearing holes.

Since it was demonstrated that the load-angle velocity is crucial to the selection of the control action, particular care should be used in selecting a filter for the velocity as measured by the encoder. Experiments could be conducted into optimizing the parameter of the exponential filter described herein, or into comparing the performance of different types of filters for the present application.

8.1.3 Experiments

While the experiments demonstrated that the algorithm works in practice for a certain range of load angles and angular velocities, additional experiments could be conducted to determine whether it is still useful for the full range which the RL agent was trained for. For safety purposes, since the way in which the load was disturbed required a physical strike with a rod, the load angle only reached magnitudes of about $50°$. In the future, the load could be disturbed using actual wind gusts, generated by large fans for example, which would allow for larger and perhaps more realistic disturbances to be introduced into the system. Furthermore, training could continue for several more episodes on the experimental testbed, to study the learning characteristics of the algorithm when applied to the physical system. While this would require a considerable amount of time, since the operator is required to wait in real-time for the load stabilization to complete, it would provide insight into how the vehicle would adapt long-term, while on parcel delivery missions.

Changes could be made to the slung payload, in order to determine how adaptable

and versatile the algorithm is when dealing with payloads which vary in size and shape. The size of the load could be increased in order to determine to what extent wind drag affects the results and the learning, while its weight could be modified in order to determine whether the learned function can still apply directly for different parcels without the need for retraining on the model.

One important drawback was identified during testing: The algorithm cannot be implemented for applications where space is limited, since the vehicle was found to require a considerable amount of space while manoeuvring to stabilize the load. In the future, if space is limited, the position of the vehicle y could be added to the dimensions of the state-space, and a negative reward could be assigned if the agent exceeds the allowable area. In this manner, the RL agent would be able to assign a value to its position and limit its movements to an appropriate radius. Alternatively, the limits on the commanded angle could be decreased, in order to ensure that the vehicle manoeuvres less aggressively.

Finally, the algorithm should be ported to the x-z plane, in order to determine whether two RL agents working in tandem are able to stabilize both load angles α and β. Once proof of concept is achieved and a reliable on-board sensor is designed, testing can proceed in an outdoor setting to ensure reliable load stabilization for parcel delivery.

8.2 Contributions

Despite the drawbacks identified in the design of the load-angle sensor and in the vehicle drift, the method developed herein has made several important contributions to rotorcraft slung payload stabilization using model-free RL. While the load-angle sensing is not conducted fully on-board as of yet, the approach still:

1. is able to stabilize the slung payload when the latter is subjected to a large disturbance, using only the motions of the rotorcraft vehicle;

2. is implemented directly in Python, using any available version of ArduCopter autopilot software, making it simple, straightforward and convenient to use;

3. is computationally-efficient enough to run on a standard Raspberry Pi micro-computer and to enable control adaptation in real-time, on a physical system;

4. is independent of linear dynamics, vehicle characteristics and reward function forms, making it versatile and easy to tailor to the needs of the operator.

8.3 Final Comments on Combining Approaches

While an entirely model-based approach was deemed inappropriate for the reasons outlined in Chapter 1, an entirely model-free approach, such as the one presented herein, has an important drawback. Most notably, the amount of training episodes required to converge on an optimal solution can get very large, especially for large numbers of state-space dimensions, and large numbers of linearly independent basis functions, since more function weights have to be learned. While starting with no guiding assumptions can be useful for complex problems where the biases of the engineer should not influence the solution, oftentimes, guiding assumptions may accelerate the adaptation of model-free approaches. It is therefore possible to leverage the potential of both model-based and model-free approaches, in order to achieve fast, efficient and adaptable control. A few methods for accelerating the learning are therefore proposed.

One way to incorporate model-based assumptions is to reduce the amount of basis functions in the features vector used to approximate the value function: One can use a small set of basis functions that is known to be adequate for the estimation of the value function. As an example, if the value function is definitively known to be a quadratic, through the selection of an appropriate reward function, and through prior knowledge of the system dynamics, a smaller set of basis functions can be selected: one that is a known basis for the space of quadratic functions. Using a simplified estimation of the value function in this manner may be adequate, since the value function is effectively a guide towards achieving an optimal control function.

Another way to accelerate the learning would be to initialize the control policy to a vector which is already believed to produce an adequate solution, rather than to an arbitrary vector, and allow the RL agent to make small adjustments to this solution as it is trained. This vector could be obtained by analyzing a model. For the particular approach presented herein, one could also modify the shape of the tiles which estimate the control function by tile-coding, using prior knowledge of the control function's characteristics. As an example, since it was observed that the control function for the mean μ had small gradients along the load-angle direction of the state space (and therefore the control function varies little along this direction), tiles could be made much larger in this dimension. This would reduce the number of weights to estimate, and would allow for smooth generalization of the function along the load-angle axis.

One could also use a different approach entirely, in order to reap the benefits of both

model-based and model-free approaches: The model-free RL algorithm could instead be used to learn a more accurate model of the system and its environment, and the model can then be used to compute a control law by some model-based design approach. This would ensure robustness, stability and adaptability in the control design.

References

[1] J. Gundlach, *Designing unmanned aircraft systems: a comprehensive approach*. American Institute of Aeronautics and Astronautics, 2012.

[2] P. Kotaru, G. Wu, and K. Sreenath, "Differential-flatness and control of quadrotor (s) with a payload suspended through flexible cable (s)," in *2018 Indian Control Conference (ICC)*, IEEE, 2018, pp. 352–357.

[3] D. K. Villa, A. S. Brandão, and M. Sarcinelli-Filho, "Load transportation using quadrotors: A survey of experimental results," in *2018 International conference on unmanned aircraft systems (ICUAS)*, IEEE, 2018, pp. 84–93.

[4] T. Aldhizer, A. Morock, K. Hughes, *et al.*, "Suspended load swing stabilization," in *2020 IEEE Integrated STEM Education Conference (ISEC)*, IEEE, 2020, pp. 1–6.

[5] Z. Hou and S. Jin, *Model free adaptive control: Theory and Applications*. CRC press Boca Raton, FL, 2013.

[6] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[7] Transport Canada, *Canadian aviation regulations*, 2021. [Online]. Available: `https://laws-lois.justice.gc.ca/eng/regulations/SOR-96-433/FullText.html`.

[8] I. Palunko and R. Fierro, "Adaptive control of a quadrotor with dynamic changes in the center of gravity," *IFAC Proceedings Volumes*, vol. 18, Aug. 2011. DOI: `10.3182/20110828-6-IT-1002.02564`.

[9] T. Oktay and C. Sultan, "Modeling and control of a helicopter slung-load system," *Aerospace Science and Technology*, vol. 29, pp. 206–222, Aug. 2013. DOI: `10.1016/j.ast.2013.03.005`.

[10] Y. Feng, C. A. Rabbath, S. Rakheja, and C.-Y. Su, "Adaptive controller design for generic quadrotor aircraft platform subject to slung load," in *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, IEEE, 2015, pp. 1135–1139.

[11] B.-Y. Lee, H.-I. Lee, D.-W. Yoo, *et al.*, "Study on payload stabilization method

with the slung-load transportation system using a quad-rotor," in *2015 European Control Conference (ECC)*, IEEE, 2015, pp. 2097–2102.

[12] S. Tang, V. Wüest, and V. Kumar, "Aggressive flight with suspended payloads using vision-based control," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 1152–1159, 2018.

[13] K. Sreenath, T. Lee, and V. Kumar, "Geometric control and differential flatness of a quadrotor uav with a cable-suspended load," in *52nd IEEE Conference on Decision and Control*, IEEE, 2013, pp. 2269–2274.

[14] F. A. Goodarzi, D. Lee, and T. Lee, "Geometric stabilization of a quadrotor uav with a payload connected by flexible cable," in *2014 American Control Conference*, IEEE, 2014, pp. 4925–4930.

[15] Y. Ren, K. Li, and H. Ye, "Modeling and anti-swing control for a helicopter slung-load system," *Applied Mathematics and Computation*, vol. 372, p. 124 990, 2020.

[16] Y. Fang, B. Ma, P. Wang, and X. Zhang, "A motion planning-based adaptive control method for an underactuated crane system," *IEEE Transactions on Control Systems Technology*, vol. 20, no. 1, pp. 241–248, 2012. DOI: 10.1109/TCST.2011.2107910.

[17] R. Ortega, A. J. van der Schaft, I. Mareels, and B. Maschke, "Energy shaping control revisited," in *Advances in the control of nonlinear systems*, Springer, 2007, pp. 277–307.

[18] I. Palunko, P. Cruz, and R. Fierro, "Agile load transportation: Safe and efficient load manipulation with aerial robots," *IEEE robotics & automation magazine*, vol. 19, no. 3, pp. 69–79, 2012.

[19] C. De Crousaz, F. Farshidian, and J. Buchli, "Aggressive optimal control for agile flight with a slung load," in *IROS 2014 Workshop on machine learning in planning and control of robot motion*, 2014, p. 7.

[20] E. Todorov and W. Li, "A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems," in *Proceedings of the 2005, American Control Conference, 2005.*, IEEE, 2005, pp. 300–306.

[21] A. Faust, I. Palunko, P. Cruz, R. Fierro, and L. Tapia, "Learning swing-free trajectories for uavs with a suspended load," in *2013 IEEE International Conference on Robotics and Automation*, IEEE, 2013, pp. 4902–4909.

[22] S. Brown, *Machine learning, explained*, Apr. 2021. [Online]. Available: https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained.

[23] J. H. Lee, J. Shin, and M. J. Realff, "Machine learning: Overview of the recent progresses and implications for the process systems engineering field," *Computers & Chemical Engineering*, vol. 114, pp. 111–121, 2018.

[24] A. Faust, I. Palunko, P. Cruz, R. Fierro, and L. Tapia, "Automated aerial suspended cargo delivery through reinforcement learning," *Artificial Intelligence*, vol. 247, pp. 381–398, 2017.

[25] X. Li, J. Zhang, and J. Han, "Trajectory planning of load transportation with multi-quadrotors based on reinforcement learning algorithm," *Aerospace Science and Technology*, vol. 116, p. 106 887, 2021.

[26] I. Palunko, A. Faust, P. Cruz, L. Tapia, and R. Fierro, "A reinforcement learning approach towards autonomous suspended load manipulation using aerial robots," in *2013 IEEE international conference on robotics and automation*, IEEE, 2013, pp. 4896–4901.

[27] A. Faust, N. Malone, and L. Tapia, "Planning preference-balancing motions with stochastic disturbances," in *Machine Learning in Planning and Control of Robot Motion Workshop at IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2014.

[28] R. A. Barron-Gomez, L. E. Ramos-Velasco, E. S. Espinoza-Quesada, H. Xu, and L. R. G. Carrillo, "Model-free adaptive controller for autonomous aerial transportation of suspended loads with unknown characteristics," in *Unmanned Systems Technology XIX*, SPIE, vol. 10195, 2017, pp. 356–373.

[29] 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, 2021.

[30] J. J. Craig, *Introduction to Robotics, Mechanics and Control*, 3rd ed. Upper Saddle River, NJ, USA: Pearson Prentice Hall, 2005.

[31] D. J. Diston, "Platform kinematics," in *Computational Modelling and Simulation of Aircraft and the Environment Volume 1: Platform Kinematics and Synthetic Environment*. Chichester, West Sussex, UK: John Wiley & Sons, 2009, ch. 2, pp. 25–62.

[32] M. D. Ardema, *Newton-Euler Dynamics*, 1st ed. New York, NY, USA: Springer Science, 2005.

[33] A. W. Pila, *Introduction to Lagrangian Dynamics*, 1st ed. Cham, Switzerland: Springer Nature Switzerland, 2020.

[34] L. R. Garcia Carrillo, A. Dzul, R. Lozano, and C. Pégard, *Quad Rotorcraft Control - Vision-Based Hovering and Navigation*. Jan. 2012.

[35] G. D. Padfield, "Modelling helicopter flight dynamics: Building a simulation model," in *Helicopter Flight Dynamics*. Hoboken, NJ, USA: John Wiley & Sons, Ltd, 2018, ch. 3, pp. 73–162. DOI: https://doi.org/10.1002/9781119401087.ch3.

[36] G. Hoffmann, H. Huang, S. Waslander, and C. Tomlin, "Quadrotor helicopter flight dynamics and control: Theory and experiment," in *AIAA guidance, navigation and control conference and exhibit*, 2007, p. 6461.

[37] D. Fusato, G. Guglieri, and R. Celi, "Flight dynamics of an articulated rotor helicopter with an external slung load," *Journal of the American Helicopter Society*, vol. 46, no. 1, pp. 3–13, 2001.

[38] Y. Feng, C. A. Rabbath, and C.-Y. Su, "Modeling of a micro uav with slung payload," in *Handbook of Unmanned Aerial Vehicles*, K. P. Valavanis and G. J. Vachtsevanos, Eds., Dordrecht, Netherlands: Springer Publishing Company, 2014, ch. 50, pp. 1257–1272.

[39] T. Lee, M. Leok, and N. H. McClamroch, "Lagrangian mechanics and variational integrators on two-spheres," *International Journal for Numerical Methods in Engineering*, vol. 79, no. 9, pp. 1147–1174, 2009.

[40] H. R. Dullin, "Semi-global symplectic invariants of the spherical pendulum," *Journal of Differential Equations*, vol. 254, no. 7, pp. 2942–2963, 2013.

[41] T. Ikeda, Y. Harata, and A. Takeeda, "Nonlinear responses of spherical pendulum vibration absorbers in towerlike 2dof structures," *Nonlinear Dynamics*, vol. 88, no. 4, pp. 2915–2932, 2017.

[42] F. A. Goodarzi, D. Lee, and T. Lee, "Geometric control of a quadrotor uav transporting a payload connected via flexible cable," *International Journal of Control, Automation and Systems*, vol. 13, no. 6, pp. 1486–1498, 2015.

[43] S. Tang and V. Kumar, "Mixed integer quadratic program trajectory generation for a quadrotor with a cable-suspended payload," in *2015 IEEE interna-*

tional conference on robotics and automation (ICRA), IEEE, 2015, pp. 2216–2222.

[44] P. Foehn, D. Falanga, N. Kuppuswamy, R. Tedrake, and D. Scaramuzza, "Fast trajectory optimization for agile quadrotor maneuvers with a cable-suspended payload," pp. 1–10, 2017.

[45] S. Tang, K. Sreenath, and V. Kumar, "Multi-robot trajectory generation for an aerial payload transport system," in *Robotics Research*, Springer, 2020, pp. 1055–1071.

[46] K. Sreenath and V. Kumar, "Dynamics, control and planning for cooperative manipulation of payloads suspended by cables from multiple quadrotor robots," *rn*, vol. 1, no. r2, r3, 2013.

[47] M. E. Guerrero, D. Mercado, R. Lozano, and C. García, "Passivity based control for a quadrotor uav transporting a cable-suspended payload with minimum swing," in *2015 54th IEEE Conference on Decision and Control (CDC)*, IEEE, 2015, pp. 6718–6723.

[48] M. Gassner, T. Cieslewski, and D. Scaramuzza, "Dynamic collaboration without communication: Vision-based cable-suspended load transport with two quadrotors," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 5196–5202.

[49] Y. Ren, J. Liu, H. Chen, and Y. Liu, "Vision-encoder-based payload state estimation for autonomous mav with a suspended payload," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, pp. 9632–9638.

[50] T. Krajník, M. Nitsche, J. Faigl, *et al.*, "A practical multirobot localization system," *Journal of Intelligent & Robotic Systems*, vol. 76, pp. 539–562, 2014.

[51] *I2c bus specifications and user manual*, UM10204, Rev. 7, NXP Semiconductors, Oct. 2021.

[52] P. Prajapati, S. Parekh, and V. Vashista, "On-board cable attitude measurement and controller for outdoor aerial transportation," *Robotica*, pp. 1–15, 2021.

[53] S. D. Whitehead and L.-J. Lin, "Reinforcement learning of non-markov decision processes," *Artificial Intelligence*, vol. 73, no. 1, pp. 271–306, 1995, Computational Research on Interaction and Agency, Part 2, ISSN: 0004-3702.

DOI: https://doi.org/10.1016/0004-3702(94)00012-P. [Online]. Available: https://www.sciencedirect.com/science/article/pii/00043702 9400012P.

[54] S. Slobodyan, A. Bogomolova, and D. Kolyuzhnov, "Stochastic gradient versus recursive least squares learning," *CERGE-EI Working Paper*, no. 309, pp. 1–21, 2006.

[55] A. L. Blum and P. Langley, "Selection of relevant features and examples in machine learning," *Artificial intelligence*, vol. 97, no. 1-2, pp. 245–271, 1997.

[56] E. Kreyszig, *Advanced engineering mathematics, 5th ed.* John Wiley & Sons, Inc., 1983.

[57] V. Kurková and M. Sanguineti, "Comparison of worst case errors in linear and neural network approximation," *IEEE Transactions on Information Theory*, vol. 48, no. 1, pp. 264–275, 2002.

[58] A. A. Sherstov and P. Stone, "Function approximation via tile coding: Automating parameter choice," in *International Symposium on Abstraction, Reformulation, and Approximation*, Springer, 2005, pp. 194–205.

[59] F. Aràndiga, R. Donat, L. Romani, and M. Rossini, "On the reconstruction of discontinuous functions using multiquadric rbf–weno local interpolation techniques," *Mathematics and Computers in Simulation*, vol. 176, pp. 4–24, 2020.

[60] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[61] IBM, *What is monte carlo simulation?* [Online]. Available: https://www.ibm.com/topics/monte-carlo-simulation.

[62] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in neural information processing systems*, vol. 12, 1999.

[63] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International conference on machine learning*, Pmlr, 2014, pp. 387–395.

[64] E. Altuğ, J. P. Ostrowski, and C. J. Taylor, "Control of a quadrotor helicopter using dual camera visual feedback," *The International Journal of Robotics Research*, vol. 24, no. 5, pp. 329–341, 2005.

[65] R. A. Barron-Gomez, L. E. Ramos-Velasco, E. S. Espinoza-Quesada, H. Xu, and L. R. G. Carrillo, "Model-free adaptive controller for autonomous aerial transportation of suspended loads with unknown characteristics," in *Unmanned Systems Technology XIX*, SPIE, vol. 10195, 2017, pp. 356–373.

[66] F. B. Abdallah, N. Azouz, L. Beji, and A. Abichou, "Modeling and stabilization of a cable-driven parallel platform suspended by an airship," in *2017 11th International Workshop on Robot Motion and Control (RoMoCo)*, IEEE, 2017, pp. 53–58.

[67] S. Whiteson *et al.*, *Adaptive representations for reinforcement learning*. Springer, 2010, vol. 291.

[68] M. Bellemare, J. Veness, and M. Bowling, "Sketch-based linear value function approximation," *Advances in Neural Information Processing Systems*, vol. 25, 2012.

[69] J. B. Travnik and P. M. Pilarski, "Representing high-dimensional data to intelligent prostheses and other wearable assistive robots: A first comparison of tile coding and selective kanerva coding," in *2017 International Conference on Rehabilitation Robotics (ICORR)*, IEEE, 2017, pp. 1443–1450.

[70] ArduPilot, *Ardupilot*. [Online]. Available: `https://ardupilot.org/`.

[71] ——, *Altitude hold mode*. [Online]. Available: `https://ardupilot.org/copter/docs/altholdmode.html`.

[72] ——, *Copter position control and navigation*. [Online]. Available: `https://ardupilot.org/dev/docs/code-overview-copter-poscontrol-and-navigation.html`.

[73] ——, *Copter attitude control*. [Online]. Available: `https://ardupilot.org/dev/docs/apmcopter-programming-attitude-control-2.html`.

[74] A. C. Dimian, C. S. Bildea, and A. A. Kiss, *Integrated design and simulation of chemical processes*. Elsevier, 2014.

[75] ArduPilot, *Copter commands in guided mode*. [Online]. Available: `https://ardupilot.org/dev/docs/copter-commands-in-guided-mode.html#copter-commands-in-guided-mode-set-attitude-target`.

[76] R. Balan and G. Lamothe, *Expect the unexpected: a first course in biostatistics*. World Scientific, 2011.

[77] N. Al Saud, "Waypoint control of a quadrotor carrying a slung payload using reinforcement learning," unpublished, M.S. thesis, University of Ottawa, 2024.

[78] W. T. Miller and F. H. Glanz, "Unh_cmac version 2.1 the university of new hampshire implementation of the cerebellar model arithmetic computer-cmac august 31, 1994 (last modified july, 1996)," 1994.

[79] EMAX, *Emax rsii-2206-1900kv brushless motor for fpv racing*. [Online]. Available: https://hobbyking.com/en_us/emax-rsii-2206-1900kv-ccw-thread.html?___store=en_us.

[80] DYS, *Dys aria f45a 4in1 blheli$_{32}$ 45a esc*. [Online]. Available: https://hobbyking.com/en_us/dys-aria-f45a-4in1-blheli-32bit-45a-esc-3-6s.html.

[81] HRB, *Hrb 2pcs 3s 11.1v 2600mah 60c xt60 lipo battery*. [Online]. Available: https://www.amazon.ca/gp/product/B088R6D91L/ref=ewc_pr_img_1?smid=AEXGS89RF8KCG&amp&psc=1.

[82] DAL, *Dalprop 6040 props multirotor propellers*. [Online]. Available: https://www.fpvfaster.com.au/products/dal-prop-6040-multirotor-propellers-2-paris?variant=39493851849.

[83] Emlid, *Navio2*. [Online]. Available: https://www.pishop.ca/product/navio2/.

[84] R. Pi, *Raspberry pi 3 model b+*. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/.

[85] ArduPilot, *First time setup*. [Online]. Available: https://ardupilot.org/copter/docs/initial-setup.html.

[86] QGroundControl, *Qgc - qgroundcontrol - drone control*, Feb. 2020. [Online]. Available: http://qgroundcontrol.com/.

[87] DroneCode, *Introduction to the mavlink protocol*. [Online]. Available: https://mavlink.io/en/.

[88] ArduPilot, *AutoTune*. [Online]. Available: https://ardupilot.org/copter/docs/autotune.html.

[89] DroneCode, *Pymvalink (python-mavgen)*. [Online]. Available: https://mavlink.io/en/mavgen_python/.

[90] M. Euston, P. Coote, R. Mahony, J. Kim, and T. Hamel, "A complementary filter for attitude estimation of a fixed-wing uav," in *2008 IEEE/RSJ inter-*

national conference on intelligent robots and systems, IEEE, 2008, pp. 340–345.

[91] S. Sabikan and S. W. Nawawi, "Open-source project (osps) platform for outdoor quadcopter," *Journal of Advanced Research Design*, vol. 24, pp. 13–27, 2016.

[92] L Tuck, C. Samson, J. Laliberté, M Wells, and F Bélanger, "Magnetic interference testing method for an electric fixed-wing unmanned aircraft system (uas)," *Journal of Unmanned Vehicle Systems*, vol. 6, no. 3, pp. 177–194, 2018.

[93] M. Wells, "Attenuating magnetic interference in a uav system," Ph.D. dissertation, Carleton University, 2008, ch. 3.

[94] J. Ireland, "Magnetic vs optical engines: What you should know when choosing the right encoder engine for your application," Dynapar, Tech. Rep., 2023.

[95] *32-bit quadrature counter with serial interface*, LS7366R, LSI Computer Systems, Inc., 2023.

[96] *I2c-bus to spi bridge*, SC18IS602B, Rev. 6, NXP Semiconductors, Oct. 2017.

[97] MikroElektronika, *I2c to spi click*. [Online]. Available: `https://www.mikroe.com/i2c-to-spi-click`.

[98] ——, *Counter click*. [Online]. Available: `https://www.mikroe.com/counter-click`.

[99] OptiTrack, *Motive - optical motion capture software*. [Online]. Available: `https://optitrack.com/software/motive/`.

[100] ArduPilot, *Motor thrust scaling*. [Online]. Available: `https://ardupilot.org/copter/docs/motor-thrust-scaling.html#motor-thrust-scaling`.

[101] *E16 micro optical kit encoder*, E16-4096-079-S-D-D-2, US Digital, Aug. 2023. [Online]. Available: `https://www.usdigital.com/products/encoders/incremental/kit/e16/?q=E16-4096-079-S-D-D-2`.

[102] B. Canada, *Plastic bearing pom623 glass balls 3x10x4*. [Online]. Available: `https://www.bearingscanada.com/Plastic-Bearing-POM-623-Glass-Balls-3x10x4-p/pom623.htm`.

Appendices

A. Full Mathematical Derivations

A.1 Euler Angles and the Aerospace Rotation Sequence

The attitude of an object in three-dimensional space can be described by three successive rotations of its body frame about its own axes, assuming it is initially in the same orientation as the inertial reference frame. By convention for aeronautical applications, the successive rotations describing an aircraft's attitude are known as the aerospace rotation sequence. The sequence is illustrated in Figure A.1, where E denotes the inertial frame and B denotes the body frame. Beginning with both reference frames coincident, the moving frame first undergoes a right-handed rotation about its z-axis by an angle ψ (the "yaw" angle), resulting in intermediate frame 1. Then, the latter undergoes a right-handed rotation about its y-axis by an angle θ (the "pitch" angle), yielding intermediate frame 2. Finally, the latter undergoes a right-handed rotation about its x-axis by an angle ϕ (the "roll" angle), yielding the final body frame orientation [31]. Different combinations of the angles ϕ, θ and ψ, usually referred to as the *Euler angles*, can yield any orientation of any frame in three-dimensional space.

The mapping of any vector from one orientation to another in three-dimensional space is performed using 3×3 rotation matrices. These are constructed by expressing the unit vectors of one frame of reference using the unit directions of the other frame, and writing each result as a column of the rotation matrix [30]. Referring to the first

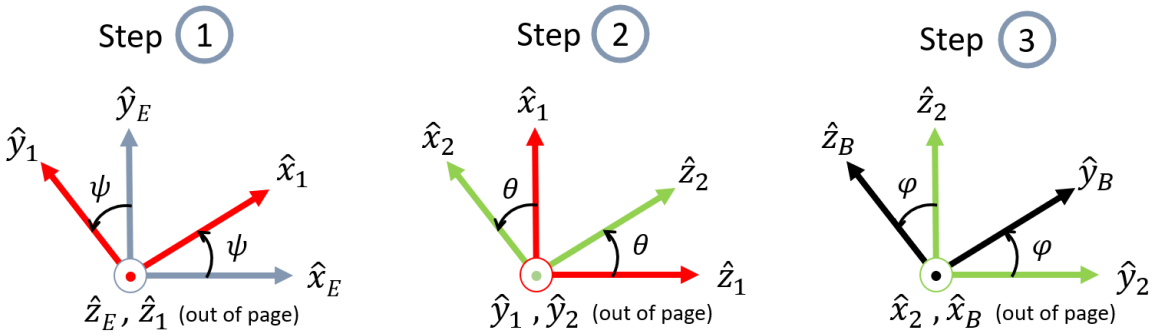


Figure A.1: Visual representation of the aerospace rotation sequence.

step in Figure A.1, the rotation matrix for converting from frame 1 to frame E is constructed as follows:

$$ {}^{E}_{1}R(\psi) = \begin{bmatrix} {}^{E}\hat{x}_1 & {}^{E}\hat{y}_1 & {}^{E}\hat{z}_1 \end{bmatrix} = \begin{bmatrix} c_\psi & -s_\psi & 0 \\ s_\psi & c_\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.1} $$

where ${}^{E}\hat{x}_1$ indicates that the vector $\hat{x}_1$ is expressed using the unit directions of frame E, and so on, and c_i and s_i are shorthand notation for $cos(i)$ and $sin(i)$ respectively. Proceeding the same way for steps 2 and 3,

$$ {}^{1}_{2}R(\theta) = \begin{bmatrix} {}^{1}\hat{x}_2 & {}^{1}\hat{y}_2 & {}^{1}\hat{z}_2 \end{bmatrix} = \begin{bmatrix} c_\theta & 0 & s_\theta \\ 0 & 1 & 0 \\ -s_\theta & 0 & c_\theta \end{bmatrix} \tag{A.2} $$

$$ {}^{2}_{B}R(\phi) = \begin{bmatrix} {}^{2}\hat{x}_B & {}^{2}\hat{y}_B & {}^{2}\hat{z}_B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi & c_\phi \end{bmatrix} \tag{A.3} $$

The rotation matrix for converting from the body frame directly to the inertial frame can be obtained by simply post-multiplying each rotation matrix in turn [30]:

$$ {}^{E}_{B}R(\psi,\theta,\phi) = {}^{E}_{1}R(\psi)\,{}^{1}_{2}R(\theta)\,{}^{2}_{B}R(\phi) = \begin{bmatrix} c_\psi & -s_\psi & 0 \\ s_\psi & c_\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_\theta & 0 & s_\theta \\ 0 & 1 & 0 \\ -s_\theta & 0 & c_\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi & c_\phi \end{bmatrix} \tag{A.4} $$

$$ {}^{E}_{B}R = \begin{bmatrix} c_\theta c_\psi & -c_\phi s_\psi + s_\phi s_\theta c_\psi & s_\phi s_\psi + c_\phi s_\theta c_\psi \\ c_\theta s_\psi & c_\phi c_\psi + s_\phi s_\theta s_\psi & -s_\phi c_\psi + c_\phi s_\theta s_\psi \\ -s_\theta & s_\phi c_\theta & c_\phi c_\theta \end{bmatrix} \tag{A.5} $$

Any vector expressed using the unit directions of the body frame can therefore be expressed using those of the inertial frame, simply by pre-multiplying the body-frame vector by the above matrix.

A.2 Euler's Kinematic Equations

In order to derive an expression for the angular velocity vector of the quadcopter using Euler angles, it is useful to first break it down into successive rotations using

the intermediate frames obtained in Appendix A.1:

$$\boldsymbol{\Omega}_{\mathbf{B/E}} = \boldsymbol{\Omega}_{\mathbf{B/2}} + \boldsymbol{\Omega}_{\mathbf{2/1}} + \boldsymbol{\Omega}_{\mathbf{1/E}} \tag{A.6}$$

That is, the rotational velocity of the quadcopter body B, relative to the stationary inertial frame E, is a vector addition of rotational velocity vectors: that of B relative to intermediate frame 2, added to that of frame 2 relative to intermediate frame 1, and finally added to that of frame 1 relative to the inertial frame E. Each of these velocity vectors can now be expressed using the Euler angles and their derivatives.

The first Euler rotation in the sequence generates intermediate frame 1 relative to the inertial frame E, and simply involves a rotation in yaw. Therefore,

$$^{\mathbf{1}}\boldsymbol{\Omega}_{\mathbf{1/E}} = \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} \tag{A.7}$$

Note that the left superscript 1 again indicates that this vector is expressed using the unit directions of frame 1. Similarly, the second Euler rotation generates frame 2 relative to frame 1 by a rotation in pitch:

$$^{\mathbf{2}}\boldsymbol{\Omega}_{\mathbf{2/1}} = \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} \tag{A.8}$$

. This vector is expressed using the unit directions of frame 2. Finally, the last rotation generates the body frame relative to frame 2 by a rotation in roll:

$$^{\mathbf{B}}\boldsymbol{\Omega}_{\mathbf{B/2}} = \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} . \tag{A.9}$$

This vector is expressed using the unit directions of frame B. In order to add the vectors, they must all be expressed using the same unit directions. It is most useful to convert them all to the body-frame directions, since the original angular velocity vector $\boldsymbol{\Omega}_{\mathbf{B}}$ expresses rotation about the body frame axes. Using the rotation matrices derived in Appendix A.1, one can convert the above vectors to body frame coordinates

by pre-multiplying the vector by the appropriate matrix:

$$\mathbf{^B\Omega_{1/E}} = {}^B_2R(\phi)\ {}^2_1R(\theta)\ \mathbf{^1\Omega_{1/E}} \tag{A.10}$$

$$\mathbf{^B\Omega_{2/1}} = {}^B_2R(\phi)\ \mathbf{^2\Omega_{2/1}} \tag{A.11}$$

And $\mathbf{^B\Omega_{B/2}}$ is already expressed in body coordinates. Here the rotation matrices have been transposed, since the opposite transformations of those shown in Equations (A.2) and (A.3) are required. Substituting Equations (A.10) and (A.11) into Equation (A.6), and substituting in the transposed rotation matrices yields the full equation of the angular velocity vector in body coordinates:

$$\mathbf{\Omega_{B/E}} = \begin{bmatrix}\dot{\phi}\\0\\0\end{bmatrix} + \begin{bmatrix}1&0&0\\0&c_\phi&s_\phi\\0&-s_\phi&c_\phi\end{bmatrix}\begin{bmatrix}0\\\dot{\theta}\\0\end{bmatrix} + \begin{bmatrix}1&0&0\\0&c_\phi&s_\phi\\0&-s_\phi&c_\phi\end{bmatrix}\begin{bmatrix}c_\theta&0&-s_\theta\\0&1&0\\s_\theta&0&c_\theta\end{bmatrix}\begin{bmatrix}0\\0\\\dot{\psi}\end{bmatrix} \tag{A.12}$$

Computing the result yields:

$$\mathbf{\Omega_{B/E}} = \begin{bmatrix}\dot{\phi} - \dot{\psi}s_\theta\\\dot{\theta}c_\phi + \dot{\psi}s_\phi c_\theta\\-\dot{\theta}s_\phi + \dot{\psi}c_\phi c_\theta\end{bmatrix} \tag{A.13}$$

And, isolating the Euler Angle rates as a vector:

$$\mathbf{\Omega_{B/E}} = \begin{bmatrix}1&0&-s_\theta\\0&c_\phi&s_\phi c_\theta\\0&-s_\phi&c_\phi c_\theta\end{bmatrix}\begin{bmatrix}\dot{\phi}\\\dot{\theta}\\\dot{\psi}\end{bmatrix} \tag{A.14}$$

A.3 Rotational Inertia Matrix

The symmetric matrix J behaves as an inertia matrix for the quadcopter, when rotation about the body-frame axes is expressed using Euler angle rates. It is given by:

$$J = P^T I P$$

$$J = \begin{bmatrix}1&0&0\\0&\cos\phi&-\sin\phi\\-\sin\theta&\cos\theta\sin\phi&\cos\theta\cos\phi\end{bmatrix}\begin{bmatrix}I_x&0&0\\0&I_y&0\\0&0&I_z\end{bmatrix}\begin{bmatrix}1&0&-\sin\theta\\0&\cos\phi&\cos\theta\sin\phi\\0&-\sin\phi&\cos\theta\cos\phi\end{bmatrix} \tag{A.15}$$

Performing the matrix operations yields:

$$J = \begin{bmatrix} I_x & 0 & -I_x s_\theta \\ 0 & (I_y - I_z)\, c_\phi^2 + I_z & (I_y - I_z)\, c_\theta s_\phi c_\phi \\ -I_x s_\theta & (I_y - I_z)\, c_\theta s_\phi c_\phi & \left((I_z - I_y)\, c_\phi^2 + I_y - I_x\right) c_\theta^2 + I_x \end{bmatrix} \tag{A.16}$$

Again, s_i and c_i are shorthand notation for $\sin i$ and $\cos i$, respectively. Hence,

$$J_{11} = I_x \tag{A.17}$$

$$J_{12} = J_{21} = 0 \tag{A.18}$$

$$J_{13} = J_{31} = -I_x s_\theta \tag{A.19}$$

$$J_{22} = (I_y - I_z)\, c_\phi^2 + I_z \tag{A.20}$$

$$J_{23} = J_{32} = (I_y - I_z)\, c_\theta s_\phi c_\phi \tag{A.21}$$

$$J_{33} = \left((I_z - I_y)\, c_\phi^2 + I_y - I_x\right) c_\theta^2 + I_x \tag{A.22}$$

A.4 Euler Angle Equations of Motion

The expressions for the Euler Angle equations of motion are:

$$J_{11}\ddot{\phi} + J_{13}\ddot{\psi} + C_\phi = \tau_\phi \tag{A.23}$$

$$J_{22}\ddot{\theta} + J_{23}\ddot{\psi} + C_\theta = \tau_\theta \tag{A.24}$$

$$J_{13}\ddot{\phi} + J_{23}\ddot{\theta} + J_{33}\ddot{\psi} + C_\psi = \tau_\psi \tag{A.25}$$

where the J_{ij} terms are as defined in section A.3, and the coriolis-centripetal terms are:

$$C_\phi = (I_y - I_z)\, s_\phi c_\phi \dot{\theta}^2 - \left(2\,(I_y - I_z)\, c_\phi^2 + I_x - I_y + I_z\right) c_\theta \dot{\theta}\dot{\psi} - (I_y - I_z)\, c_\phi^2 c_\phi s_\phi \dot{\psi}^2 \tag{A.26}$$

$$C_\theta = \left(2\,(I_y - I_z)\, c_\phi^2 + I_x - I_y + I_z\right) c_\theta \dot{\phi}\dot{\psi} - 2\,(I_y - I_z)\, s_\phi c_\phi \dot{\phi}\dot{\theta} - \left((I_y - I_z)\, c_\phi^2 + I_x - I_y\right) s_\theta c_\theta \dot{\psi}^2 \tag{A.27}$$

$$C_\psi = \left(2\,(I_y - I_z)\, c_\phi^2 - I_x - I_y + I_z\right) c_\theta \dot{\phi}\dot{\theta} + 2\,(I_y - I_z)\, c_\theta^2 s_\phi c_\phi \dot{\phi}\dot{\psi} - (I_y - I_z)\, s_\phi c_\phi s_\theta \dot{\theta}^2 + 2\left((I_y - I_z)\, c_\phi^2 + I_x - I_y\right) s_\theta c_\theta \dot{\theta}\dot{\psi} \tag{A.28}$$

B. MATLAB Codes

B.1 Active tile indices for tile-coding

```
% Definition of the object class TileCoder_Grid
% The instantiated object is a layering of grids based on state-space
% limits, dimensions, desired tiles per dimension and desired number of tilings

% Adapted from Kris de Asis' GitHub code by Eleni Sabourin
% Original python code: https://github.com/MeepMoop/tilecoding

classdef TileCoder_Grid

    properties
       % Class properties
        tiling_offsets      % Stores the odd offset vector [1, 3, 5, ...] * fraction of tile
        limits              % Stores the limits of the state-space
        norm_dims           % Stores the number of tiles per state-space unit
        tiling_first_ind    % Stores the index of the first tile in each tiling
        hash_vec            % Stores the multiplier for converting tile coordinates to tile index
        n_tiles             % Stores the total number of tiles (weight vector length)
        coords              % Stores the coordinates of the state-space point
    end

    methods
        function obj = TileCoder_Grid(dims, tiles_per_dim, limits, n_tilings)
        % This function is executed each time a new TileCoder_Grid object is instantiated

            % Add another tile in each dimension to make sure the state space is fully covered
            tiling_dims = tiles_per_dim + 1;

            % Initialize the offset vector and tiling offset arrays
            offsets = (0:dims-1)';
            obj.tiling_offsets = zeros(n_tilings,dims);

            % Generate the offset vector (odd integers)
            offsets = 2.*offsets + 1;

            % Generate the offsets for each tiling (as fractions of a tile)
            obj.tiling_offsets(:,1) = (0:n_tilings-1);

            for j=1:dims
                obj.tiling_offsets(:,j) = offsets(j)*obj.tiling_offsets(:,1);
            end

            obj.tiling_offsets = mod(obj.tiling_offsets/n_tilings,1);


            % Assign the limits of the state space to the object
            obj.limits = limits;
```

```
            % Get number of tiles covering each "unit" of the state space in each dimension
            obj.norm_dims = tiles_per_dim ./ (obj.limits(:,2) - obj.limits(:,1));

            % Get the index of the first tile in each tiling
            obj.tiling_first_ind = prod(tiling_dims) * (0:n_tilings-1) + 1;

            % Initialize the hash vector (multiplier vector)
            obj.hash_vec = zeros(dims,1);
            obj.hash_vec(1) = 1;

            % Determine the multipliers to convert tile coordinates to tile index
            for i=2:dims
                obj.hash_vec(i) = prod(tiling_dims(1:i-1));
             end

             % Get the total number of tiles (to define the length of the weight vector)
             obj.n_tiles = n_tilings*prod(tiling_dims);

             % Initialize the array to store the point coordinates
             obj.coords = zeros(n_tilings,dims);

        end

        function active_tiles = encode(obj, n_tilings, P, active_tiles)
        % This function retrieves the indices of the active tiles each time a new
        % point in the state-space is obtained

            % Compute the "location" of the point in the base/main tiling
            base_coords = (P - obj.limits(:,1)) .* obj.norm_dims;

            % Compute the "location" of the point in each subsequent tiling
            for i=1:n_tilings
                obj.coords(i,:) =  obj.tiling_offsets(i,:) + base_coords';
            end

            % Round down to obtain tile coordinates in each dimension
            obj.coords = floor(obj.coords);

            % Convert tile coordinates to tile indices
            for i=1:n_tilings
                active_tiles(i) = obj.coords(i,:)*obj.hash_vec;
            end

            % Convert the obtained indices to total index (concatenating the vectors)
            active_tiles = active_tiles + obj.tiling_first_ind';
        end
    end

end
```

C. Experimental Testbed Details

C.1 Details of the vehicle propulsion system

The vehicle propulsion system consists of four 1900Kv brushless DC (BLDC) motors on which 6040 propellers are mounted, a 45A 4-in-1 electronic speed controller (ESC) and a 3S, 2600mAh lithium-polymer (LiPo) battery. The Kv rating of the motor, also sometimes called the voltage constant, is essentially its rpm-to-voltage ratio at no load. A high constant of 1900 was selected for this application since higher Kv ratings are typically suited to smaller builds of aircraft with smaller propellers, and can output high angular velocity at smaller torques [1]. The propellers were selected to be six inches in diameter ("60"), since this was the largest diameter of propeller that the frame and motors could accommodate, and hence thrust would be maximized for this build [1]. A four inch nominal pitch ("40") was also found to be rather aggressive and able to provide a large amount of thrust, and is readily available off-the-shelf. The battery was chosen to be a three-cell (3S) LiPo battery mainly due to its small size and weight and large energy and power density [1]. The battery capacity of 2600mAh was found to be one of the largest available considering the small stature of the battery, and would therefore allow for longer flight times. A discharge capacity of 60C ($60 \times 2600mA = 156A$ of total current discharged) was deemed more than sufficient to supply current to all four motors, even at maximum throttle, without significantly heating the battery. Finally, the electronic speed controller (ESC) was chosen for its large discharge capacity of 45A per motor, and for the fact that it had four controllers in one board, making the design more compact and more lightweight.

C.2 ArduPilot Initial Setup

Once the sensors had undergone initial calibration, the vehicle's response to attitude commands was adjusted so that it would be as sharp as possible. This was accomplished by setting the attitude time constant parameter (ATC_INPUT_TC) to zero through QGroundControl. Since this parameter controls the slope and delay of the filtered command, as seen on the left side of Figure 5.7 on page 72, setting it to its minimum value ensures that the delay is minimized. The autopilot was also instructed

to use external navigation (the motion capture data) for its heading and local positioning. This was achieved by setting the parameters which dictate the sensor sources for the extended Kalman filter (*EK3_SRC*), by turning off the compass, and by configuring the GPS type parameter (*GPS_TYPE*) to receive its position data through MAVLink messages. Finally, in order to improve the stability and performance of the vehicle during flight, motor thrust characteristics were set by modifying the motor thrust exponent parameter (*MOT_THST_EXPO*). While flying, the autopilot refers to the shape of the thrust curve in order to command specific PWM values and output a desired thrust. Modifying the motor thrust exponent allows the user to "match" the curve assumed by the autopilot to the true curve measured on the thrust stand. If the thrust curve of the propulsion system is more or less linear, as was observed in Figure 6.2 on page 82 for the current setup, the exponent should be set to a small value, usually between 0 and 0.2 [100]. As such, the motor thrust exponent was increased by increments of 0.05 between the values of 0 and 0.3, and the thrust curve estimation, as provided by the ArduPilot documentation, was compared visually to the true thrust curve at every increment. The mean absolute error (MAE) between the two curves was also obtained for each new value of the thrust exponent, to serve as an objective metric. Motor thrust characteristics are shown in Figure C.1.

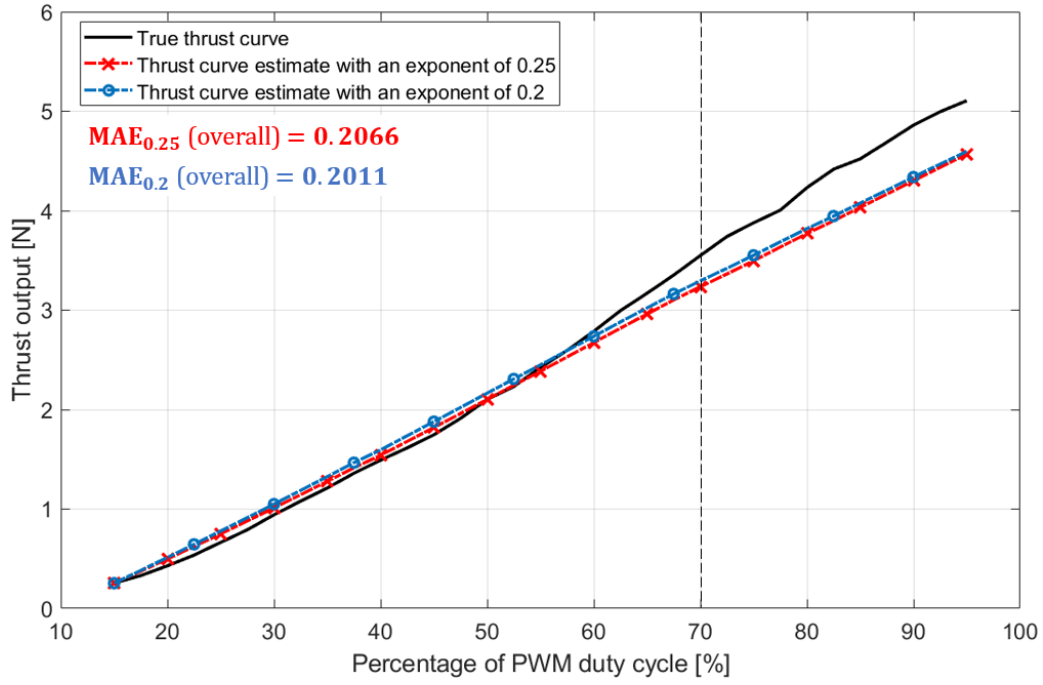


Figure C.1: Comparison of the true thrust curve and the curves estimated by the motor thrust exponent.

The smallest MAE was found to be 0.2011 over the full range of values, for a motor thrust exponent of 0.2. However, an exponent of 0.25 was deemed more appropriate for this application. As can be seen from Figure C.1, while the 0.2 curve has the lowest MAE overall (0.2011), the 0.25 curve is very slightly closer to the true thrust curve at PWM values below 70%. Its MAE in this range is 0.0888, while the 0.2 curve has an MAE of 0.1002. Since the vehicle will be hovering for most of the duration of the mission, the total thrust output should very rarely exceed 70%. Even when rolled by 20 degrees and manoeuvring aggressively, the thrust output at 70% should be enough to keep a 1kg vehicle in flight.

C.3 Details of the Load-Angle Sensor Design

A cross-section of the device, accompanied by a part list in Table C.1, is shown in Figure C.2 to clarify its assembly and operation. Only a cross-section of the static portion is shown, since the moving portion is designed in exactly the same way. Coarse M3 fasteners are used unless otherwise specified.

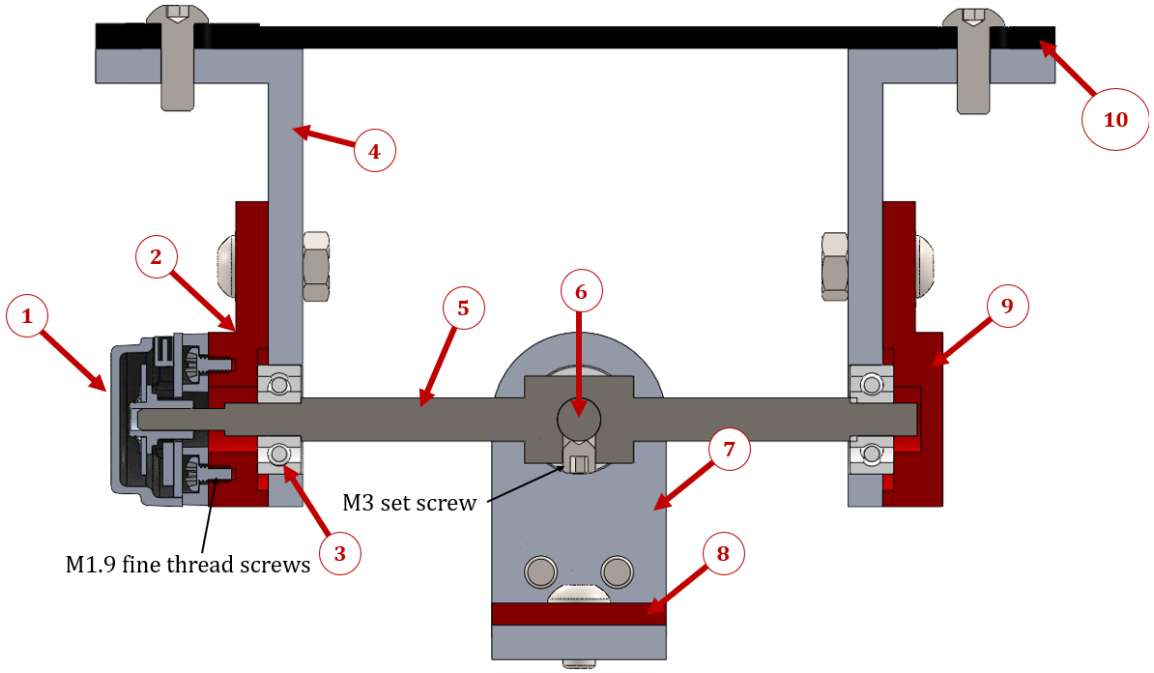


Figure C.2: Cross-section of the load angle sensing device with parts numbered.

First, the cross piece is assembled using two steel stepped shafts which are fastened together by passing one through the other, and securing them with a set screw. This design was chosen for ease of manufacturing, requiring only a lathe and drill, and this

Table C.1: List of parts used to construct the load angle sensing device.

Number	Name	Amount	Material	Manufacturing process
1	E16 Micro optical encoder	2	-	Off-the-shelf (E16-4096-079) [101]
2	Encoder side cap	2	PLA	3D printed
3	Ball bearing	4	Acetal/glass balls	Off-the-shelf (POM623) [102]
4	Large bracket	2	6061 Al alloy	Machined from 1.75" angle
5	Large stepped shaft	1	AISI 1030 steel	Machined from 1" rod
6	Small stepped shaft	1	AISI 1030 steel	Machined from 1" rod
7	Small bracket	2	6061 Al alloy	Machined from 1.75" angle
8	Load attachment plate	1	PLA	3D printed
9	Shaft side cap	2	PLA	3D printed
10	Vehicle mounting plate	1	PLA	3D printed

material was selected since it is inexpensive and better suited to machining small, precise parts. Next, the ball bearings are secured onto the shafts by press-fitting, until they rest against the shaft shoulders. Plastic ball bearings with glass balls were selected due to their low weight and extremely low friction, which will allow the assembly to move easily and follow the angle of the load cable precisely. Next, the moving portion of the assembly is put together, first by securing the plastic caps onto the two small aluminium brackets with screws, and next by sliding the resulting pieces over the bearings, and securing them together using the load attachment plate. The static portion of the design is assembled in much the same way, using the larger aluminium brackets. Aluminium was chosen for the brackets since it is lightweight and since the parts are easy to machine out of standard aluminium angle, which is readily available off-the-shelf. The plastic parts are 3D-printed using PLA plastic, due to their more complex geometry. The encoders can be secured last following the assembly approach outlined by the manufacturer [101]. Once the device is fully assembled, it can be secured to the bottom of the vehicle with screws, using the four holes in the mounting plate. The mounting plate can be easily redesigned depending on the available space or mounting options.