Modelling and Control of an Aerial Manipulator

Paul Lassen (s182864)

Master of Science in Engineering

2021

DTU Electrical Engineering
Department of Electrical Engineering

Modelling and Control of an Aerial Manipulator

Report written by:
Paul Lassen (s182864)

Advisor(s):
Matteo Fumagalli, Associate Professor at the Electrical Engineering Department of DTU

DTU Electrical Engineering
Technical University of Denmark
2800 Kgs. Lyngby
Denmark

elektro@elektro.dtu.dk

Project period: 25 February 2021- 23 April 2021

ECTS: 35

Education: M.Sc.

Field: Electrical Engineering

Class: Public

Edition: 1. edition

Remarks: This report is submitted as partial fulfillment of the requirements for graduation in the above education at the Technical University of Denmark.

Abstract

Motivated by surface contact applications, such as contact inspection, drilling, and polishing, the active area of research in aerial manipulators promises to expand the task space of UAVs. One thing that has not been studied is what can be done with the simplest manipulators using standard flight controllers. This thesis fills this gap by considering two simple manipulator configurations, controlled by an industry standard flight controller and UAV platform.

A dynamic model of static contact is developed to evaluate the limits of force application for the different manipulator configurations. A simulation model and experimentation with a physical UAV are used to verify the analysis of the dynamic model.

The maximum achievable contact force is found for each manipulator configuration. The effects of several UAV design parameters are presented and suggestions are made for maximizing the realisable contact force.

Acknowledgements

I wish to thank Matteo Fumagalli for sending me on this journey and advising me. I am grateful to the help and support I received from Kristian Weber Larsen both in constructing the UAV and in managing my project. Additionally I thank David Wuthier for lending his expertise and skills to my aid. I wish to extend my gratitude to my dear friends Brynjar Sævarsson, Rishav Bose and Nichlas Max Bjørndal who lent me several hours of their time in proof reading this document.

I am indebted to my brother, Philip, who provided me with new view points with which to tackle my research.

Lastly I would like to give a special thanks to my parents, who provided me with a sanctuary where I could find the peace of mind to complete this thesis in a tumultuous time; my mother, Sharmila, who provided a bottomless supply of fresh meals and moral support and my father, Søren Lassen, whose patience and guidance are likely the only reason I was able to finish this thesis.

Contents

CHAPTER 1

Introduction

Over the last 20 years, Unmanned Aerial Vehicles (UAVs) have captured the interest of robotics researchers and society at large, showing promise in a broad range of application areas, including film, last-mile delivery, survey and mapping, surveillance, and inspection. The typical payloads for UAVs are sensor packages which take advantage of the mobility and range afforded by the UAV. More recent research has begun to investigate their potential as flying manipulators, capable of physically interacting with the surrounding environment. Unlocking this capability would enable UAVs to take on a much broader range of applications, including tasks which currently pose a risk of injury to the people performing them. The real world applications motivating this thesis are surface contact tasks, including contact inspection, drilling, and polishing, involving walls and wall-like surfaces.

UAVs come in a variety of form factors, but the most accessible of them is the multirotor. Multirotors are also simpler to construct and fly than other UAV platforms, making them a popular choice. The ability to hover and move freely in 3 dimensions allows the multirotor to navigate complex confined environments providing it access to difficult and potentially dangerous spaces. This ability to hover serves to make multirotors the ideal platform for investigating physical interaction. Even among multirotors, there are variety of configurations, including tilt rotors, non-coplanar and coplanar configurations. The most common among them, by a wide margin, are coplanar multirotors.

While their aerial nature provides them with maneuverability, it also poses unique challenges to the task of physical interaction. Without a rigid surface to support them, UAVs must not only generate the desired contact forces, but also counteract the resultant forces and torques from the environment. All while remaining in flight.

To tackle the task of physical interaction, there are three main dimensions of research. First, there are varied form factors for UAVs, as mentioned above. Second, there are different manipulator arm designs, from full three joint robot arms to more specialized designs. Finally, several control schemes have been developed to deal with the challenges of physical interaction.

As the price of UAVs continues to fall, multirotors are becoming more and more available to hobbyists and other consumers and a relevant question then becomes: what are the capabilities of these consumer and hobbyist grade UAVs?

This thesis seeks to address this question. The goal of this project is to evaluate the capabilities and limitations of a UAV in contact with a surface using a consumer grade flight

controller. To this end, a coplanar hexarotor outfitted with the popular, open-source Pixhawk 4 flight controller and the ubiquitous Raspberry Pi 3 is modelled, simulated, and tested to evaluate the potential of such a platform when equipped with a simple manipulator.

The objectives of this project are to

- Develop a dynamic model of a UAV and manipulator in free flight and in contact with a surface
- Simulate the aerial manipulator in free flight and in contact to test control strategies
- Conduct experiments to verify the results of the modelling and simulation
- Evaluate the capabilities of standard UAV configurations
- Make suggestions for designing aerial manipulators for standard UAV configurations

Fewer experiments were conducted than planned due to the global COVID-19 pandemic. This led to limited sporadic access to physical facilities for conducting experiments. Most notably, experiments with the 1-DOF manipulator were not carried out.

The structure of this thesis follows the process of development of this project. Analysis of the problem space and a literature survey are done in chapter 2. In chapter 3, the dynamic model of a UAV in free flight is developed. Here the controllers used in chapter 4 are also developed. In chapter 4, the dynamic model for free flight is extended to consider interaction with other bodies. Finally Chapter 5 presents the conclusions of this thesis.

CHAPTER 2

Background

The field of aerial manipulators presents many opportunities for real world applications, surveyed by Kumar [13], and has motivated a large body of research. As multirotors have exploded in popularity and accessibility over the last decade, research into aerial manipulation has followed.

Research in this field covers an extensive solution space. UAV morphology is varied, and manipulators come in a multitude of shapes, employing many different control schemes. The research literature was recently surveyed by Ding et al [5] and Ruggiero et al [21]. The research and development falls into three overlapping areas: UAV design, manipulator design, and controller design.

The variety of UAV morphologies found in the literature seek to overcome the underactuated nature of the coplanar multirotor. These solutions involve tilting the propellers to provide the extra degree of actuation. Non-coplanar fixedly-tilted propeller configurations use static placement of six [22, 24] or eight [17, 25] propellers for tasks such as industrial contact inspection. Actuated tilting propellers are utilised in [1], [3], and in [10] which has been brought to market. More radical yet, a UAV designed to perch on contact surfaces for door-opening is explored in [26].

Manipulators vary wildly by task. Simple manipulators with one or no degrees of freedom are explored in [8] for contact inspection and [28, 29] for surface cleaning, where an aerial manipulator erases a white board. A 3-DOF arm is used to open drawers in [11]. An aerial manipulator with two 3-DOF arms turns valves in [16]. A 2-DOF manipulator is used for contact inspection in [24].

These rotorcraft use many different control schemes. Some examples: PID controllers are used for contact inspection of pipe welds in industrial plants in [24], LQR controllers are designed for sustained contact in [28], and non-linear controllers are built in [9] for structure inspection by contact from the underside.

The explosion in popularity of multirotors amongst hobbyists and consumers has been driven by access to affordable and robust multirotor and flight controller solutions. Chief amongst the flight controller solutions is the PX4 flight controller architecture [18] and the Pixhawk implementation [2]. Its integration with the Robot Operating System (ROS), a popular software framework for robotics [19], has led to widespread adoption by the hobby flight community and industry. Its communication protocol, MAVLINK [12], facilitates communication with the PX4 flight controller.

An under-explored topic in the field of aerial manipulation is the capabilities of consumer grade UAVs with simple manipulators utilising off-the-shelf flight controllers. This thesis seeks to answer exactly that.

CHAPTER 3

Free Flight

This chapter develops a dynamic model, a simulation model and a physical model of a UAV in free flight, laying the groundwork for an exploration of contact in chapter 4.

In this chapter a dynamic model describing the behaviour of the UAV in flight will be developed from first principles using the Newton-Euler equations. A simulation model will be developed to approximate the behaviour of the UAV in flight to increase the iteration speed and explore the limits of the UAV without risking damage in flight tests.

The simulation model is extended with with a cascaded position controller, consisting of three PID controllers, an attitude controller, an altitude controller and a position controller. The altitude and position controllers are also implemented on the physical UAV, which already has an attitude controller built into its flight controller.

Figure 3.1: The physical UAV in flight.

The construction of the physical UAV is described along with the experimental setup used in flight tests, including a motion capture system and a ground station. The parameters of the physical UAV are introduced here. The steps taken to align the behaviour of the simulated attitude controller with its physical counterpart is described in detail.

Finally, simulations and flight tests are conducted using the same inputs and the results are compared.

3.1 Dynamic Model

The dynamic model developed in this section describes how the UAV's state evolves over time and how actuation of its motors affect this evolution. The model is developed from the structure of the UAV from first principles using the Newton-Euler equations. Additionally, a motor mixer matrix is derived to provide the first step towards controlling the UAV. The derivation in this section follows the approach and notation of [14].

3.1.1 Rigid Body Dynamics

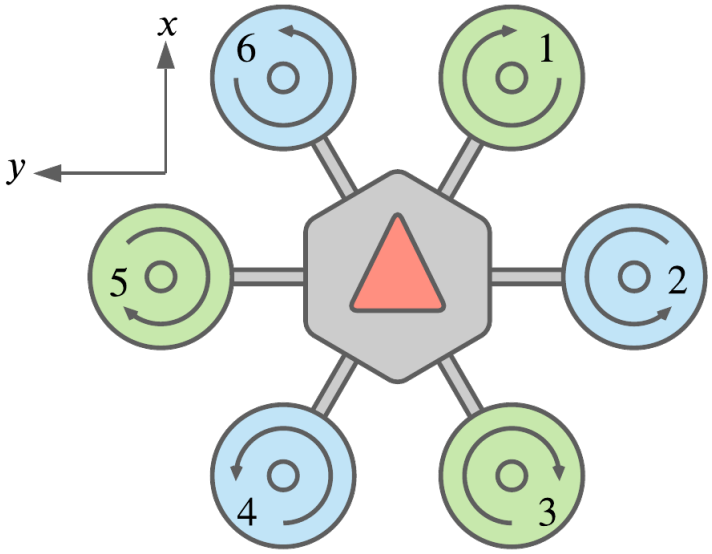


Figure 3.2: Diagram of the Hexarotor X configuration as seen from above. The red arrow points along the x-axis of the body fixed frame, in the direction that the UAV is facing. The z-axis of the body fixed frame points towards the reader.

The UAV is a Hexarotor in the 'X' configuration, consisting of six coplanar rotors arranged in counter rotating pairs. The 'X' denotes the placement of the rotors around the UAV, as shown in figure 3.2.

The absolute position of the UAV in the inertial frame is given by $\boldsymbol{\xi}_B^I$, where I denotes the inertial frame and B denotes the body-fixed frame. In general, the superscript will be used to denote the frame of reference and the subscript will be used to identify the frame of interest. When the frames of reference and interest coincide, the subscript is omitted, e.g., $\mathbf{f}^B$ means the force generated within the body frame as seen from the body frame.

The body-fixed frame B is a coordinate system with its origin located at the center of mass of the UAV. Its x-, y-, and z-axes are fixed to the UAV's body and point forward, left, and up, as seen in figure 3.2.

Each rotor, numbered according to figure 3.2, is centered at a point, $\mathbf{r}_i = \begin{bmatrix} x_i & y_i & z_i \end{bmatrix}^T$ with respect to the hexarotor's center of mass. The i^{th} rotor generates a force $\mathbf{f}_i$ along its axis of rotation and a torque $\boldsymbol{\tau}_i$, given by

$$\mathbf{f}_i = k \begin{bmatrix} 0 \\ 0 \\ \omega_i^2 \end{bmatrix} \tag{3.1}$$

$$\boldsymbol{\tau}_i = \mathbf{r}_i \times \mathbf{f}_i + b \begin{bmatrix} 0 \\ 0 \\ \omega_i^2 \end{bmatrix} + I_M \begin{bmatrix} 0 \\ 0 \\ \dot{\omega}_i \end{bmatrix} = S(\mathbf{r}_i)\mathbf{f}_i + b \begin{bmatrix} 0 \\ 0 \\ \omega_i^2 \end{bmatrix} + I_M \begin{bmatrix} 0 \\ 0 \\ \dot{\omega}_i \end{bmatrix} \tag{3.2}$$

where ω_i is the angular velocity of the rotor, k is the lift constant, b is the drag constant, I_M is the moment of inertia of the rotor, and $S(\cdot)$ maps a vector to its skew symmetric matrix,

$$S(\mathbf{r}_i) = \begin{bmatrix} 0 & -z_i & y_i \\ z_i & 0 & -x_i \\ -y_i & x_i & 0 \end{bmatrix} \tag{3.3}$$

The derivative motor term $I_M\dot{\omega}_i$ is small because I_M is minuscule and is therefore omitted for the rest of the derivation of the system dynamics.

The total wrench (force and torque) generated by the UAV in the body frame is the sum of the forces and torques generated by each rotor

$$\mathbf{w}^B = \begin{bmatrix} \mathbf{f}^B \\ \boldsymbol{\tau}^B \end{bmatrix} = \sum_{i=1}^{6} \begin{bmatrix} \mathbf{f}_i \\ \boldsymbol{\tau}_i \end{bmatrix} \tag{3.4}$$

It is evident from equation 3.1 that the total force generated by the rotors has only one non-zero force component in the z direction. The force generated by the rotors is called the thrust and the z component is labelled T. The total wrench then has 4 non-zero terms, force in the z direction and torque around the $x, y,$ and z axes. These are labelled $\tau_\phi, \tau_\theta,$ and τ_ψ, respectively. Throughout this thesis, the zero terms of the wrench will be omitted. Thus, the total wrench generated by the UAV has 4 dimensions and is of the form

$$\mathbf{w}^B = \begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} \tag{3.5}$$

From these equations we can now derive the linear dynamics in the inertial frame. The linear dynamics are driven by the sum of the forces acting upon the UAV including the force of thrust f^B, the force of gravity f_g, and air resistance. Air resistance is omitted

in the dynamic model, because the goal of this thesis is to apply wrench to objects in static contact scenarios, where the UAV velocity is low. Air resistance generates a force resisting motion when an object moves through a fluid and scales with the velocity of the object and so becomes a relatively small force at low speeds. The acceleration of the UAV in the inertial frame is then

$$m\ddot{\boldsymbol{\xi}}_B^I = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_B^I \mathbf{f}^B \tag{3.6}$$

where m is the mass of the UAV, $\ddot{\boldsymbol{\xi}}_B^I$ is the acceleration of the UAV in the inertial frame, g is the acceleration due to gravity, and R_B^I is the rotation matrix. The rotation matrix used for these calculations uses yaw-pitch-roll convention.

$$R_B^I = R_z(\psi)R_y(\theta)R_x(\phi) \tag{3.7}$$

where $R_z(\psi)$ represents a rotation around the z-axis, followed by $R_y(\theta)$ around the new y-axis, and then $R_x(\phi)$ around the final x-axis. The inverse of this operation, that is, the rotation from the inertial frame to the body frame, is given by

$$R_I^B = (R_B^I)^{-1} = (R_B^I)^T \tag{3.8}$$

by the general properties of rotation matrices.

To derive the rotational dynamics of the body in the inertial frame, it must first be derived in the body frame. The orientation of $\boldsymbol{\xi}_B^I$ is parameterized by $\boldsymbol{\eta} = [\phi, \theta, \psi]^T$. The instantaneous angular velocity in the body fixed frame is denoted $\boldsymbol{\nu} = [p, q, r]^T$. The relationship between the torque and angular velocity and acceleration is given by

$$\mathbf{I}\dot{\boldsymbol{\nu}} + \boldsymbol{\nu} \times (\mathbf{I}\boldsymbol{\nu}) = \boldsymbol{\tau}^B \tag{3.9}$$

where $\mathbf{I}$ is the moment of inertia of the UAV,

$$\mathbf{I} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{3.10}$$

Rearranging equation 3.9 to solve for the angular acceleration,

$$\dot{\boldsymbol{\nu}} = \mathbf{I}^{-1}\left(\boldsymbol{\tau}^B - \boldsymbol{\nu} \times (\mathbf{I}\boldsymbol{\nu})\right) = \mathbf{I}^{-1}\begin{bmatrix} \tau_\phi + (I_{\mathrm{yy}} - I_{\mathrm{zz}})\, q\, r \\ \tau_\theta + (I_{\mathrm{zz}} - I_{\mathrm{xx}})\, p\, r \\ \tau_\psi + (I_{\mathrm{xx}} - I_{\mathrm{yy}})\, p\, q \end{bmatrix} = \begin{bmatrix} \frac{\tau_\phi + (I_{\mathrm{yy}} - I_{\mathrm{zz}})q\, r}{I_{\mathrm{xx}}} \\ \frac{\tau_\theta + (I_{\mathrm{zz}} - I_{\mathrm{xx}})p\, r}{I_{\mathrm{yy}}} \\ \frac{\tau_\psi + (I_{\mathrm{xx}} - I_{\mathrm{yy}})p\, q}{I_{\mathrm{zz}}} \end{bmatrix} \tag{3.11}$$

Equation 3.11 gives the angular acceleration in the body fixed frame. It is convenient, however, to find the angular acceleration in the inertial frame, denoted $\ddot{\boldsymbol{\eta}}$, as $\boldsymbol{\eta}$ is used to describe the orientation.

The transformation from the angular velocity of the UAV with respect to the world frame, $\dot{\boldsymbol{\eta}}$, into the body-fixed frame, $\boldsymbol{\nu}$ can be expressed as a matrix $\mathbf{W}_{\boldsymbol{\eta}}$. See equation 5.46 in [15]. $\mathbf{W}_{\boldsymbol{\eta}}$ and its inverse are

$$\mathbf{W}_{\boldsymbol{\eta}} = \begin{bmatrix} 1 & 0 & -S_\theta \\ 0 & C_\phi & C_\theta S_\phi \\ 0 & -S_\phi & C_\theta C_\phi \end{bmatrix} \tag{3.12}$$

$$\mathbf{W}_{\eta}^{-1} = \begin{bmatrix} 1 & T_\theta S_\phi & T_\theta C_\phi \\ 0 & C_\phi & -S_\phi \\ 0 & S_\phi/C_\theta & C_\phi/C_\theta \end{bmatrix} \tag{3.13}$$

where $\cos(\alpha)$, $\sin(\alpha)$, $\tan(\alpha)$ are written C_α, S_α, T_α, a notation which will be used throughout the rest of this thesis.

Using equation 3.13, the angular velocity in the inertial frame $\dot{\boldsymbol{\eta}}$ can be stated in terms of $\boldsymbol{\nu}$

$$\dot{\boldsymbol{\eta}} = \mathbf{W}_{\boldsymbol{\eta}}^{-1}\boldsymbol{\nu} \tag{3.14}$$

Taking the time derivative of both sides reveals

$$\ddot{\boldsymbol{\eta}} = \frac{d}{dt}(\mathbf{W}_{\boldsymbol{\eta}}^{-1})\boldsymbol{\nu} + \mathbf{W}_{\boldsymbol{\eta}}^{-1}\dot{\boldsymbol{\nu}} \tag{3.15}$$

where

$$\frac{d}{dt}(\mathbf{W}_{\boldsymbol{\eta}}^{-1}) = \begin{bmatrix} 0 & \dot{\phi}C_\phi T_\theta + \dot{\theta}S_\theta/C_\theta^2 & -\dot{\phi}S_\phi T_\theta + \dot{\theta}C_\phi/C_\theta^2 \\ 0 & -\dot{\phi}S_\phi & -\dot{\phi}C_\phi \\ 0 & \dot{\phi}C_\phi/C_\theta + \dot{\theta}S_\phi T_\theta/C_\theta & -\dot{\phi}S_\phi/C_\theta + \dot{\theta}C_\phi T_\theta/C_\theta \end{bmatrix} \tag{3.16}$$

Now given equations 3.4, 3.6, 3.11, 3.14, and 3.15 the motion of the UAV in the inertial frame is fully described.

3.2 Simulation of the Dynamic Model

This section desribes a simulation model of the UAV. It forms a foundation that will be extended with controllers in section 3.3 and manipulators in chapter 4. The purpose of the simulation model is to approximate the behaviour of the UAV in flight scenarios so as to increase the iteration speed and to explore the limits of the UAV without risking damage in flight tests. The simulation model implementation is largely independent of the dynamic model and therefore, will also be used to verify the predictions of the dynamic models in chapter 4.

The simulation model was created in MATLAB and Simulink. Simulink is a block diagramming tool for modelling and simulating dynamic systems [6]. The Simscape Multibody Toolbox was used to provide a simulation environment for 3D mechanical systems [23].

Using the Simscape Multibody Toolbox to develop the simulation confers some advantages over building the simulation in Simulink based on the dynamic model derived in section 3.1. The blocks in the Simscape Multibody Toolbox provide tools to quickly build a model of a body under the influence of the forces and torques acting upon it.

Figure 3.3: The block diagram of the simulated UAV. The UAV is modelled using blocks from the Simscape Multibody Toolbox.

The block diagram of the UAV body is shown in figure 3.3. The simulation is separated into three different frames: the inertial world frame, the body fixed and world aligned local frame, and the body fixed and aligned body frame. The local and body frames are constrained by a translational cartesian joint and a rotational spherical joint, respectively. The block labelled 'Motor Speed to Body Wrench', shown in figure 3.4, takes six motor speeds as input and converts them to forces and torques as described by equation 3.4.

Aside from the 'Motor Speed to Body Wrench' block, the rest of the simulation model dynamics are modelled using Simscape Multibody Toolbox. By using a commercial toolbox, the simulation model provides an independent implementation of the dynamic model.

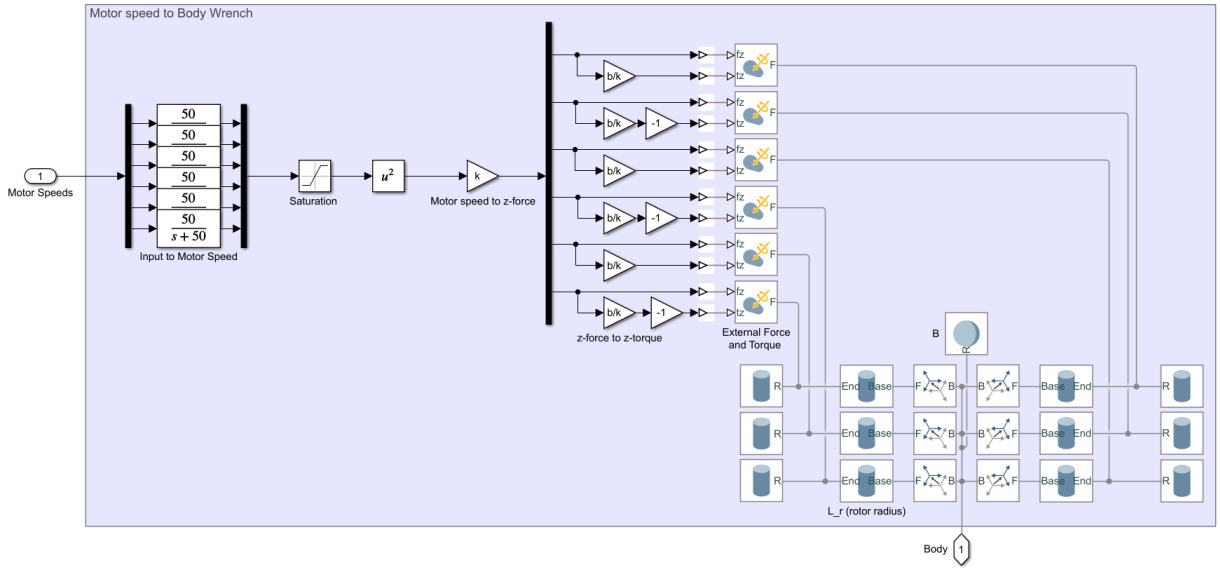


Figure 3.4: The block diagram of 'Motor Speed to Body Wrench'. Note the 1st order transfer functions. They model the dynamics of the brushless DC motors used to drive the propellers, using the BLDC response times found in [27]. The saturation block provides the physical limits on the motor speeds. The forces and torques are applied at points at a distance L_r from the center of mass of the UAV.

Table 3.1 lists the parameters used for the simulation.

Mass	m	2.2 kg
Inertia about x	I_{xx}	0.0265
Inertia about y	I_{yy}	0.0265
Inertia about z	I_{zz}	0.0529
Rotor length	L_r	0.31 m
Lift constant	k	$5.1597 \cdot 10^{-6}$
Drag constant	b	$2.5798 \cdot 10^{-7}$

Table 3.1: Parameters of the simulated UAV.

To simulate the UAV, a vector of six motor speeds is provided as input. The outputs of the simulation are the linear and rotational states of the simulated UAV: position, velocity, acceleration, orientation, angular velocity and angular acceleration. Figure 3.5 shows the motor speed input for each of the rotors for one such simulation. The resulting motion of the simulated UAV is shown in figures 3.6 and 3.7. The parameters used for the simulation are shown in table 3.1.

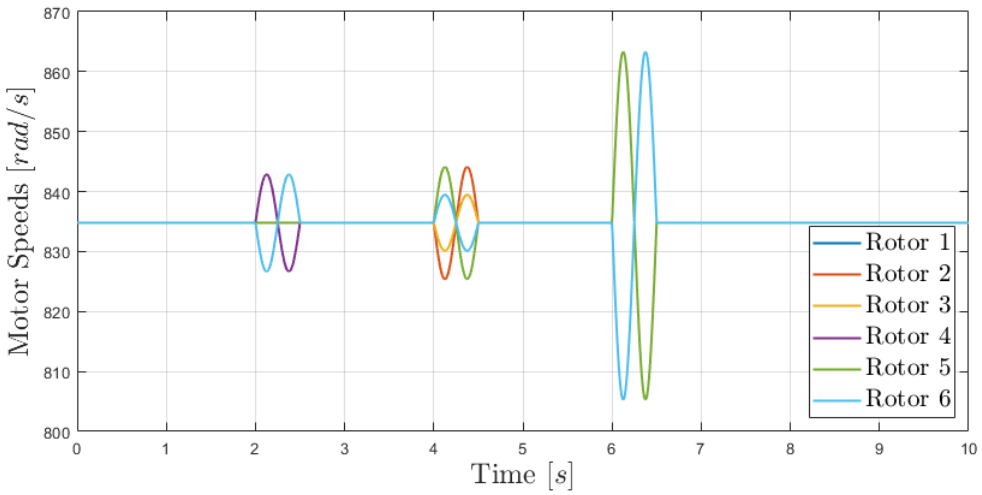


Figure 3.5: Motor speed input to simulated UAV.

The rotational dynamics of the UAV are shown in figure 3.6. The rotational accelerations clearly respond to imbalances in the motor speeds. The simulation outputs the rotational dynamics in the body frame. These outputs are converted to the inertial roll, pitch, and yaw parameterization using equation 3.14.

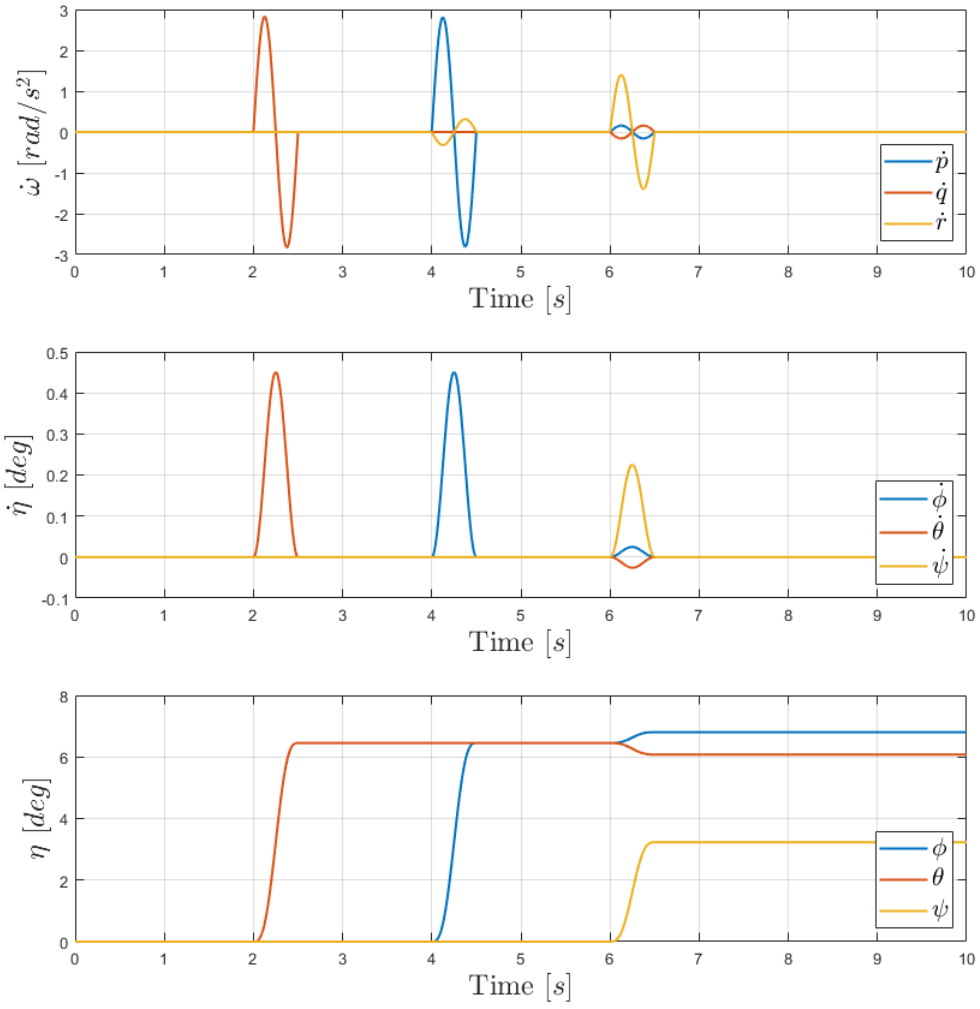


Figure 3.6: Rotational dynamics of the simulated UAV. From top to bottom, rotational acceleration in the body frame, rotational velocity in the inertial frame, and the orientation of the UAV in the inertial frame

The resulting linear dynamics of the simulated UAV are shown in figure 3.7. The acceleration clearly responds to the changes in the pitch and roll angles. As these angles remain steady, the UAV continues to accelerate in both x and y throughout the simulation.

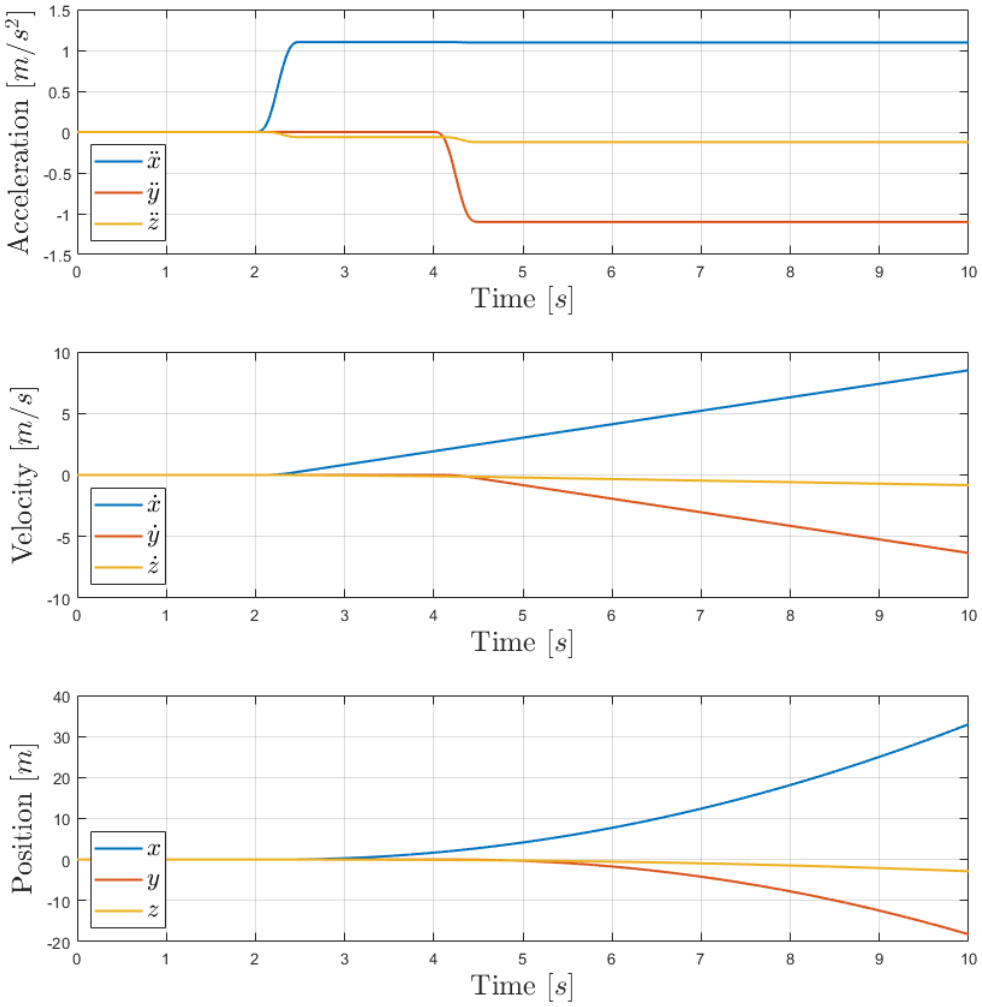


Figure 3.7: Linear dynamics of the simulated UAV. From top to bottom, acceleration, velocity, and position of the simulated UAV in the inertial frame.

3.3 Controllers

The purpose of this section is to develop a position controller for both the physical and the simulated UAV. The position controller is split into two parts. The x and y components of the position controller are considered together and will be referred to as the position controller throughout this thesis. The z component of the position controller is referred to as the altitude controller and considered separately.

The position controller takes a desired x and y position as input and outputs attitude

targets that are fed to the attitude controller. The altitude controller on the other hand, takes a z position as input and outputs the required thrust force, T_{ref}. While the physical UAV has a built-in attitude controller, one also has to be defined for and added to the simulated UAV described in section 3.2. The attitude controller receives attitude targets from the position controller and outputs the torques $\boldsymbol{\tau}_{\text{ref}}$ required to achieve them. With this attitude controller description, it is necessary to create a module that can translate the desired torque to required motor speeds $\boldsymbol{\omega}$. This module is called the motor mixer. Figure 3.8 shows the control architecture for the UAV in the simulation.

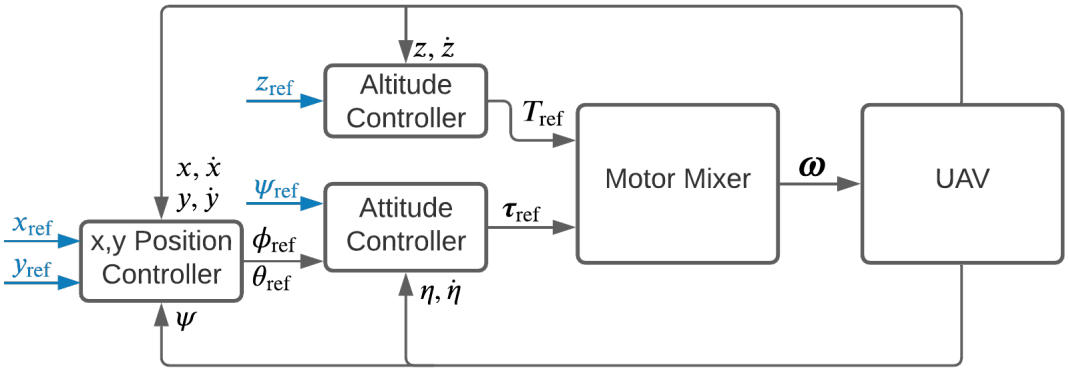


Figure 3.8: Block diagram of position controller architecture. The blue signals represent the inputs to the system. In the simulation, all of the states of the simulated UAV are directly available to the controllers. Details on the physical UAVs system architecture can be found in section 3.4.2.

In this section, the controllers are developed from the inside out. First, the motor mixer module is defined. Then, a PID attitude controller is developed for the simulation to approximate the behaviour of the onboard controller. Next, the position controller is designed in two parts, the altitude controller (the z component) and the x, y-position controller.

The controllers are implemented and tested in the simulation as they are developed.

3.3.1 Motor Mixer

The motor mixer, $\mathbf{M}$ takes the desired wrench as an input and outputs the motor speeds required to achieve the wrench. A hexarotor cannot, however, achieve an arbitrary wrench due to its coplanar rotor configuration. While a hexarotor platform can achieve a torque in all three dimensions, it can only produce a positive force along the z-axis of the body, perpendicular to the rotor plane. This leaves the UAV with 4 degrees of freedom.

An examination of equation 3.4 reveals that the wrench $\mathbf{w}^B$ is a linear combination of the squared motor speeds. Letting $\mathbf{\Omega}$ be the squared motor speeds and $\mathbf{w}_{\text{in}}$ be the desired wrench in 4 degrees of freedom,

$$\mathbf{\Omega} = \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \cdots \\ \omega_6^2 \end{bmatrix}, \qquad \mathbf{w}_{\text{in}} = \begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix}$$

the motor mixer then is the matrix $\mathbf{M}$ such that

$$\mathbf{\Omega} = \mathbf{M}\mathbf{t}_{\text{in}} \tag{3.17}$$

To find $\mathbf{M}$, consider the Jacobian of $\mathbf{w}^B$ from equation 3.4 with respect to $\mathbf{\Omega}$, denoted $J(\mathbf{w}^B, \mathbf{\Omega})$, which gives the relationship between $\mathbf{\Omega}$ and the real generated torque $\mathbf{t}_{\text{out}}$

$$\mathbf{w}_{\text{out}} = J(\mathbf{w}^B, \mathbf{\Omega})\mathbf{\Omega} \tag{3.18}$$

Plugging equation 3.17, into equation 3.18,

$$\mathbf{w}_{\text{out}} = J(\mathbf{w}^B, \mathbf{\Omega})\mathbf{M}\mathbf{t}_{\text{in}} \tag{3.19}$$

To make $\mathbf{w}_{\text{out}} = \mathbf{w}_{\text{in}}$, the motor mixer matrix $\mathbf{M}$ is then simply the pseudoinverse of $J(\mathbf{w}^B, \mathbf{\Omega})$,

$$\mathbf{M} = J(\mathbf{w}^B, \mathbf{\Omega})^+ \tag{3.20}$$

which has the property, $J(\mathbf{w}^B, \mathbf{\Omega})J(\mathbf{w}^B, \mathbf{\Omega})^+ = [\mathbf{1}]$, where $[\mathbf{1}]$ is the identity matrix.

For a hexarotor with a center of mass coincident with the body frame and rotors at a distance L_r from the center, the Jacobian and motor mixer are given by

$$J(\mathbf{w}^B, \mathbf{\Omega}) = \begin{bmatrix} k & k & k & k & k & k \\ -\frac{\sigma}{2} & -\sigma & -\frac{\sigma}{2} & \frac{\sigma}{2} & \sigma & \frac{\sigma}{2} \\ -\frac{\sqrt{3}\sigma}{2} & 0 & \frac{\sqrt{3}\sigma}{2} & \frac{\sqrt{3}\sigma}{2} & 0 & \frac{\sqrt{3}\sigma}{2} \\ b & -b & b & -b & b & -b \end{bmatrix} \tag{3.21}$$

and

$$\mathbf{M} = \begin{bmatrix} \frac{1}{6k} & -\frac{1}{6\sigma} & -\frac{\sqrt{3}}{6\sigma} & \frac{1}{6b} \\ \frac{1}{6k} & -\frac{1}{3\sigma} & 0 & -\frac{1}{6b} \\ \frac{1}{6k} & -\frac{1}{6\sigma} & \frac{\sqrt{3}}{6\sigma} & \frac{1}{6b} \\ \frac{1}{6k} & \frac{1}{6\sigma} & \frac{\sqrt{3}}{6\sigma} & -\frac{1}{6b} \\ \frac{1}{6k} & \frac{1}{3\sigma} & 0 & \frac{1}{6b} \\ \frac{1}{6k} & \frac{1}{6\sigma} & -\frac{\sqrt{3}}{6\sigma} & -\frac{1}{6b} \end{bmatrix} \tag{3.22}$$

where $\sigma = kL_r$.

Adding the motor mixer to the simulation is done by simply adding a gain block with the mixer matrix M from equation 3.22, shown in figure 3.9.

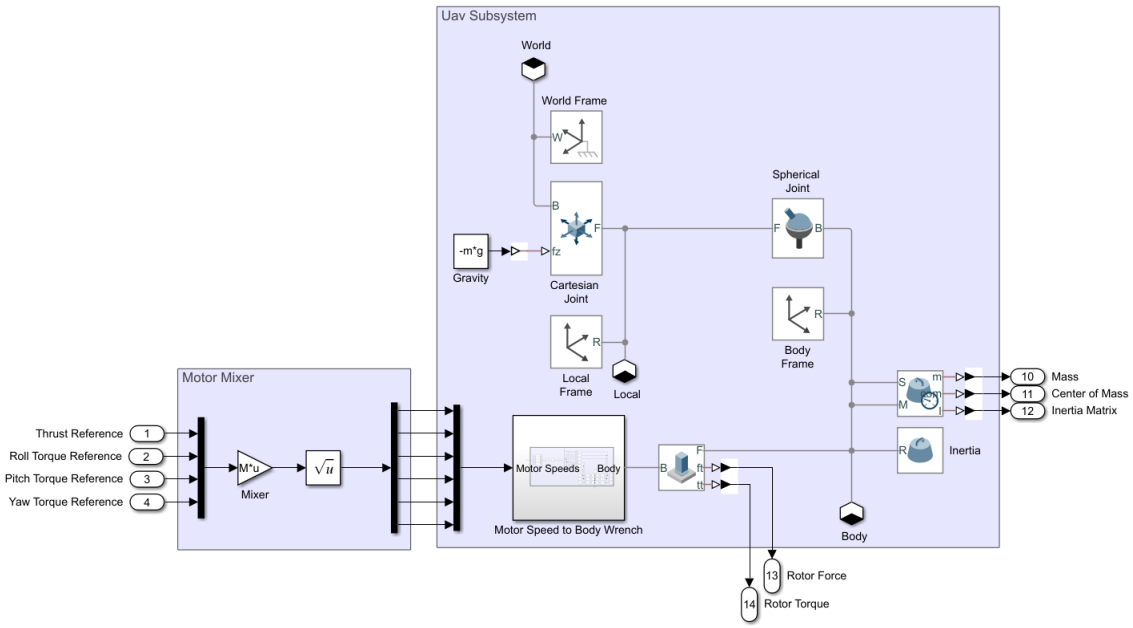


Figure 3.9: Block diagram of motor mixer.

The motor mixer was tested in simulation by feeding in wrench inputs and comparing them to the wrench and torque applied to the body of the UAV. Figure 3.10 shows the results of a simulation with the motor mixer.

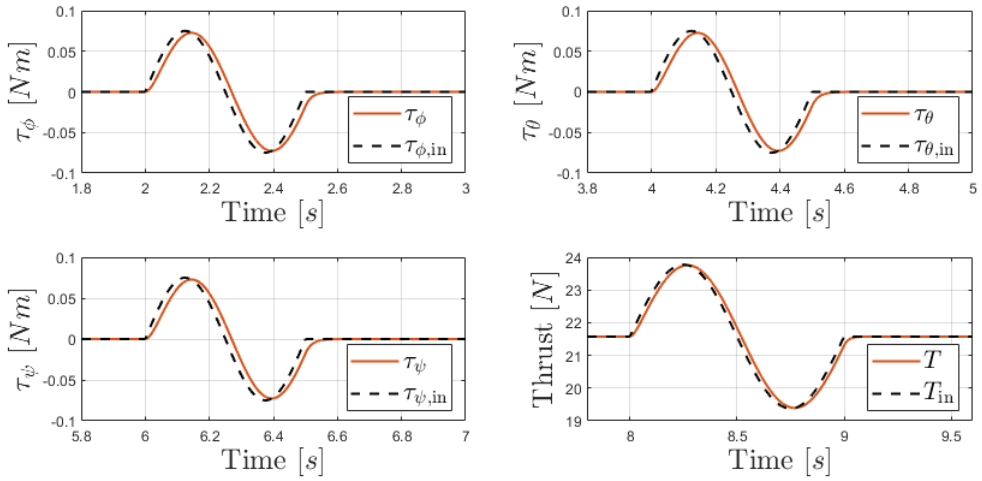


Figure 3.10: Motor mixer in simulation. A sinusoidal input was provided to each of the elements of $\mathbf{w}^B$, dashed black line, and the wrench on the body was measured, in orange. The slight delay in the output is due to the dynamics of the BLDC motors shown in figure 3.4.

3.3.2 PID Attitude Controller

The attitude controller takes roll, pitch, and yaw references as input and outputs the required wrench into the motor mixer designed above.

The attitude controller on board the Pixhawk 4 flight controller is a cascaded controller with an inner and an outer loop. The inner loop is a PID angular rate controller which takes angular velocity references as input from the outer loop, a non-linear, quaternion-based attitude controller detailed in [4].

For the purposes of the simulation, a PID control architecture was used. PID control was chosen for quick setup and ease of manual tuning, and the structure of a PID controller lends itself to the analysis of contact in chapter 4.

A PID controller is based on three error calculations, a proportional error $\mathbf{e}_{p,\boldsymbol{\eta}}$, a derivative error $\mathbf{e}_{d,\boldsymbol{\eta}}$, and an integral error $\mathbf{e}_{i,\boldsymbol{\eta}}$,

$$\mathbf{e}_{p,\boldsymbol{\eta}} = \boldsymbol{\eta}_{ref} - \boldsymbol{\eta} \tag{3.23}$$

$$\mathbf{e}_{d,\boldsymbol{\eta}} = -\dot{\boldsymbol{\eta}} \tag{3.24}$$

$$\mathbf{e}_{i,\boldsymbol{\eta}} = \int_0^t \mathbf{e}_{p,\boldsymbol{\eta}} \tag{3.25}$$

The derivative error $\mathbf{e}_{d,\boldsymbol{\eta}}$ is set to the negative of the angular velocity, rather than the derivative of $\mathbf{e}_{p,\boldsymbol{\eta}}$, to smooth the response to a change in reference. A stepwise change to the reference results in a discontinuity in $\mathbf{e}_{p,\boldsymbol{\eta}}$, and thereby a generates a spike in $\mathbf{e}_{d,\boldsymbol{\eta}}$.

These errors are scaled by their corresponding gains, $\mathbf{K}_{p,\boldsymbol{\eta}}$, $\mathbf{K}_{d,\boldsymbol{\eta}}$, and $\mathbf{K}_{i,\boldsymbol{\eta}}$, which take the form

$$\mathbf{K}_{p,\boldsymbol{\eta}} = \begin{bmatrix} k_{p,\phi} & 0 & 0 \\ 0 & k_{p,\theta} & 0 \\ 0 & 0 & k_{p,\psi} \end{bmatrix} \tag{3.26}$$

These errors and gains combine to create a controller of the form

$$\mathbf{u}_{\boldsymbol{\eta}} = \mathbf{K}_{p,\boldsymbol{\eta}}\mathbf{e}_{p,\boldsymbol{\eta}} + \mathbf{K}_{i,\boldsymbol{\eta}}\mathbf{e}_{i,\boldsymbol{\eta}} + \mathbf{K}_{d,\boldsymbol{\eta}}\mathbf{e}_{d,\boldsymbol{\eta}} \tag{3.27}$$

The output of the controller, $\mathbf{u}_{\boldsymbol{\eta}}$ is then scaled by the inertia of the UAV to provide the torque targets, which are sent to the motor mixer from section 3.3.1.

$$\boldsymbol{\tau}_{\text{ref}} = I^{-1}u_{\boldsymbol{\eta}} \tag{3.28}$$

Figure 3.11 shows the implementation of the controller in simulation. Tuning of this controller is discussed in section 3.4.3, the results of which are shown in figures 3.25 and 3.26.

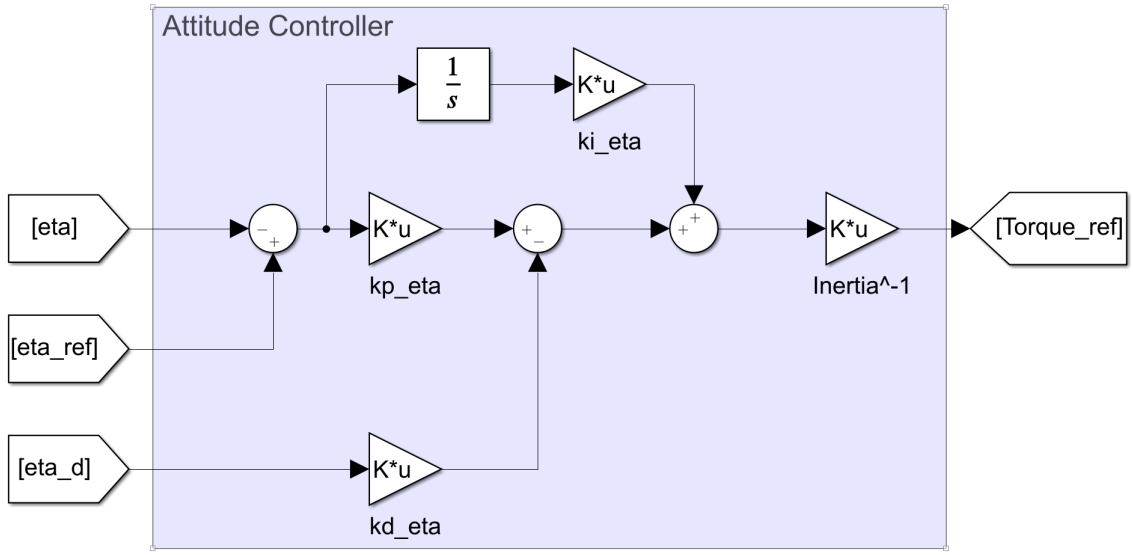


Figure 3.11: The block diagram implementation of the attitude controller in Simulink.

The attitude controller was tested in simulation by feeding attitude references inputs and comparing them to the attitude of the simulated UAV. Figure 3.12 shows the results of a simulation with the attitude controller.

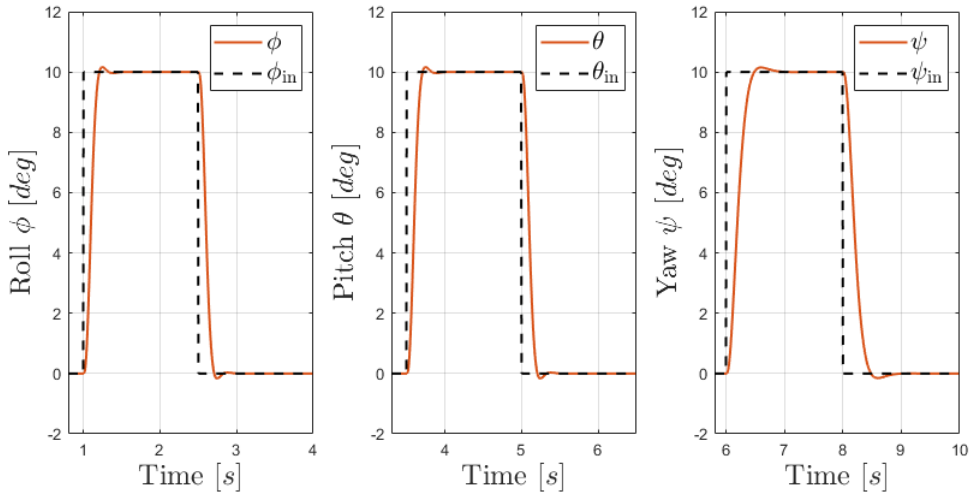


Figure 3.12: Attitude controller in simulation. A step input was provided to each of the elements of the attitude η, dashed black line, and the attitude of the simulated UAV was recorded, in orange. The controller gains used can be found in table 3.2.

	k_p	k_d	k_i
ϕ	0.17	0.017	0
θ	0.17	0.017	0
ψ	0.17	0.034	0

Table 3.2: PID attitude controller gains used in the simulation shown in figure 3.12.

3.3.3 Altitude Controller

The process for the altitude controller is nearly identical. The only differences are that the altitude requires control around a non-zero operating point (gravity) and dynamic scaling to handle rotation of the thrust vector. Handling this operating point is simply a matter of adding an extra term to the controller output to counteract gravity, like so

$$u_z = k_{p,z}e_{p,z} + k_{d,z}e_{d,z} + k_{i,z}e_{i,z} + g \tag{3.29}$$

When the UAV has an attitude of $\mathbf{0}$, this controller functions as expected. When the UAV begins to tilt, however, the thrust vector no longer projects vertically and so the thrust needs to be increased to maintain flight. The thrust in the body frame (from equation 3.4) has only a z component, T. The thrust vector in the inertial frame is then

$$\mathbf{f}^I = R_B^I \mathbf{f}^B = R_B^I \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} = T \begin{bmatrix} S_\phi S_\psi + C_\phi C_\psi S_\theta \\ C_\phi S_\psi S_\theta - C_\psi S_\phi \\ C_\phi C_\theta \end{bmatrix} \tag{3.30}$$

Maintaining a thrust of T in the z direction then requires scaling the input thrust by $C_\phi C_\theta$. This scaling can be done by simply dividing u_z by the cosines of roll and pitch and multiplying by the mass m of the UAV. The thrust reference T_{ref} to be sent to the motor mixer becomes

$$T_{ref} = \frac{m u_z}{C_\phi C_\theta} \tag{3.31}$$

Figure 3.13 shows the implementation of the controller in simulation.

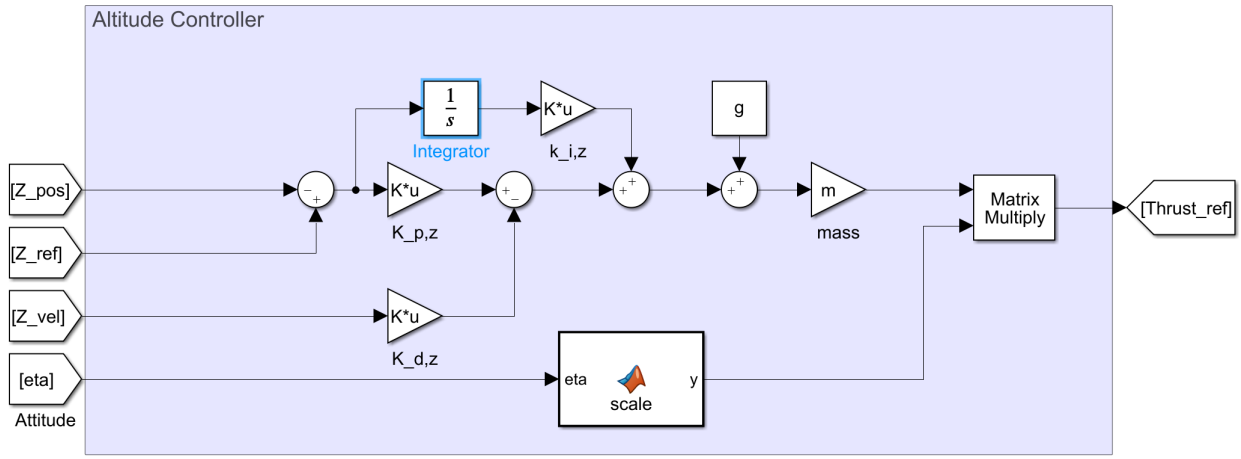


Figure 3.13: The block diagram implementation of the altitude controller in Simulink.

The altitude controller was tested in simulation by feeding z position reference inputs to the altitude controller and comparing them to the z position of the simulated UAV. Figure 3.14 shows the results of a simulation with the altitude controller.

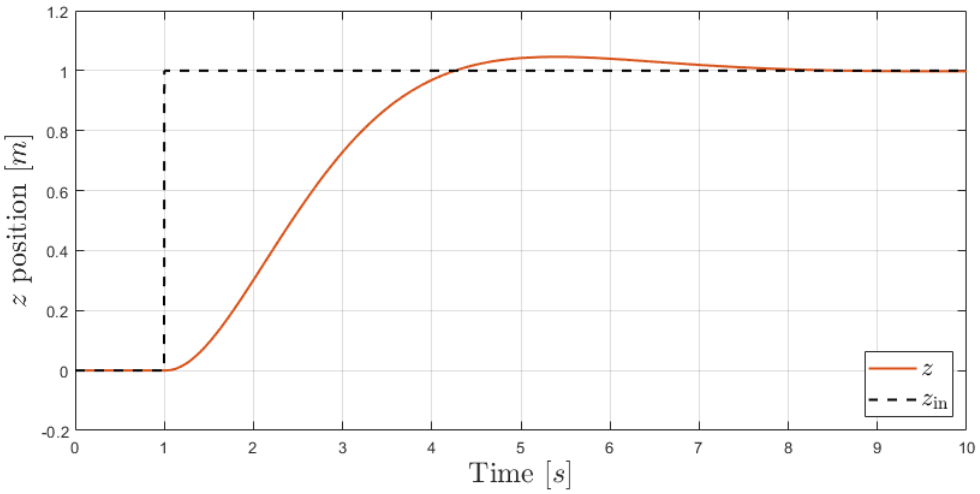


Figure 3.14: Altitude controller in simulation. A step input was provided as the z position reference, dashed black line, and the z position of the simulated UAV was recorded, in orange. The controller gains used can be found in table 3.3.

	k_p	k_d	k_i
z	1	1.4	0

Table 3.3: PID altitude controller gains used in the simulation shown in figure 3.14.

3.3.4 PID Position Controller

The position controller is the outermost control loop required for autonomous flight. It consists of a PID controller which takes an input position in world coordinates and outputs the required attitude reference to move the UAV to the desired position. The position controller is designed in much the same way as the attitude controller in section 3.3.2.

To design the x- and y-controllers, we start by assuming that the UAV is hovering at some arbitrary point in the world coordinates with its axes parallel to the corresponding world axes. In other words, we assume that $\boldsymbol{\eta} = \mathbf{0}$. If the UAV begins to roll such that $\phi > 0$ while θ and ψ are held at 0, the UAV will begin to accelerate along the y-axis in the negative direction. Likewise, if the UAV begins to pitch such that $\theta > 0$ while ϕ and ψ are held at 0, the UAV will begin to accelerate along the x-axis in the positive direction. The implication of these results is that x- and y-controllers can be decoupled from each other and considered separately. The x-controller will then look at the error in the x-direction and output a roll target, while the y-controller will look at the error in the y-direction and output a pitch target.

Starting with the x-controller, using the standard error formulation,

$$e_{p,x} = x_{ref} - x \tag{3.32}$$

$$e_{d,x} = -\dot{x} \tag{3.33}$$

$$e_{i,x} = \int_0^t e_{p,x} \tag{3.34}$$

$$u_x = k_{p,x}e_{p,x} + k_{i,x}e_{i,x} + k_{d,x}e_{d,x} \tag{3.35}$$

where $\theta_{ref} = u_x$ is the pitch component of the attitude target $\boldsymbol{\eta}_{ref}$.

Likewise for the y-controller,

$$e_{p,y} = y_{ref} - y \tag{3.36}$$

$$e_{d,y} = -\dot{y} \tag{3.37}$$

$$e_{i,y} = \int_0^t e_{p,y} \tag{3.38}$$

$$u_y = k_{p,y}e_{p,y} + k_{i,y}e_{i,y} + k_{d,y}e_{d,y} \tag{3.39}$$

where $\phi_{ref} = -u_y$ is the roll component.

The x- and y-controllers designed above function as intended when $\psi = 0$. If $\psi \neq 0$, then the x component of the error no longer maps directly to the required pitch. The same goes for the y component and roll. The solution to this problem is to rotate the

error vector around the z axis by ψ. In other words, taking the error with respect to the direction that the UAV is facing. The new error vectors, denoted e', are then

$$\begin{bmatrix} e'_{p,x} \\ e'_{p,y} \end{bmatrix} = R(\psi) \begin{bmatrix} e_{p,x} \\ e_{p,y} \end{bmatrix} \tag{3.40}$$

$$\begin{bmatrix} e'_{d,x} \\ e'_{d,y} \end{bmatrix} = R(\psi) \begin{bmatrix} e_{d,x} \\ e_{d,y} \end{bmatrix} \tag{3.41}$$

$$\begin{bmatrix} e'_{i,x} \\ e'_{i,y} \end{bmatrix} = \int_0^t \begin{bmatrix} e'_{p,x} \\ e'_{p,y} \end{bmatrix} \tag{3.42}$$

where $R(\psi) = \begin{bmatrix} C_\psi & -S_\psi \\ S_\psi & C_\psi \end{bmatrix}$.

With these new error vectors the x- and y-position controllers become

$$u_x = k_{p,x}e'_{p,x} + k_{i,x}e'_{i,x} + k_{d,x}e'_{d,x} \tag{3.43}$$

$$u_y = k_{p,y}e'_{p,y} + k_{i,y}e'_{i,y} + k_{d,y}e'_{d,y} \tag{3.44}$$

Figure 3.15 shows the implementation of the controller in simulation.

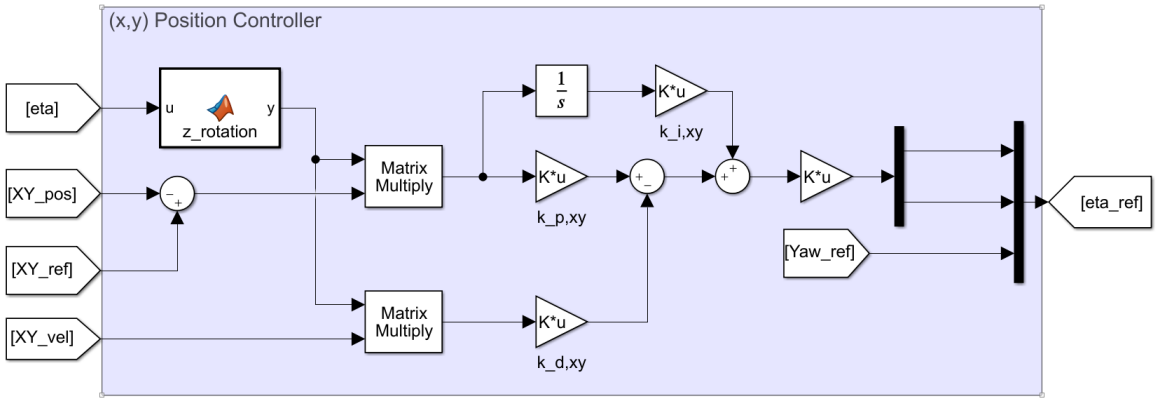


Figure 3.15: The block diagram implementation of the position controller in Simulink.

The position controller was tested in simulation by feeding x and y position reference inputs to the position controller and comparing them to the x and y positions of the simulated UAV. Figure 3.16 shows the results of a simulation with the position controller.

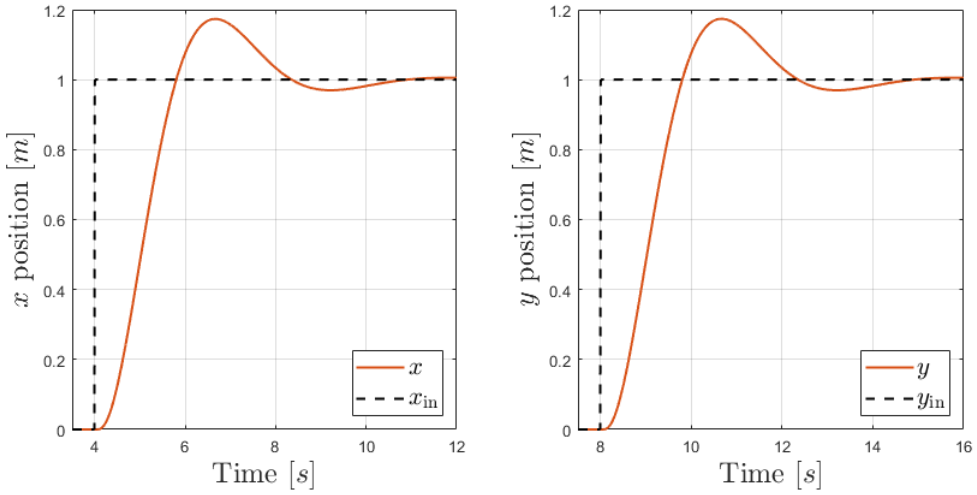


Figure 3.16: Altitude controller in simulation. A step input was provided to each of the inputs of the position controller, dashed black line, and the position of the simulated UAV was recorded, in orange. The controller gains used can be found in table 3.4.

	k_p	k_d	k_i
x	10	8	0
y	10	8	0

Table 3.4: PID position controller gains used in the simulation shown in figure 3.16.

3.4 Test Setup

This section describes the experimental setup used for flight tests. First the construction of the physical UAV is described. Then the architecture of the whole test setup, comprised of a motion capture system, a control computer, and the UAV, is described. Finally, the process for attitude controller alignment is described.

3.4.1 Physical UAV

The physical UAV was a multirotor in the coplanar hexarotor configuration as described in section 3.1. The body of the UAV contained the battery, power management board, the flight controller, and the flight computer. Six rotor arms were attached, each mounted with an electronic speed controller (ESC) and brushless DC (BLDC) motor, as shown in figure 3.17. All of the components used in the construction of the UAV were consumer grade and were purchased from a leading hobby shop, HobbyKing.

Figure 3.17: The physical UAV.

The total weight of the UAV was 2.2kg. A 4-cell LiPo battery contributed a quarter of the weight and powered all of the components onboard UAV, providing a nominal voltage of 14.8V. The flight controller was a Pixhawk 4 and the flight computer was a Raspberry Pi 3b+. The rotor arms were hollow carbon fiber tubes with mounts for the motors.

The BLDC motors were rated at 950kV. When driven by the 4-cell battery, they could achieve a maximum rotation speed of approximately 15,000 RPM. Through testing, the maximum thrust was found to be approximately 80N, or $3.71mg$. These values were used to estimate $k = 5.1597 \cdot 10^{-6}$ and $b = 2.5798 \cdot 10^{-7}$.

3.4.2 System Description

This section provides a system level description of the physical UAV, both the onboard systems and the offboard systems. The UAV received instructions from an offboard control computer and received measurements from an offboard motion capture (MOCAP) system. The system architecture is shown in figure 3.18.

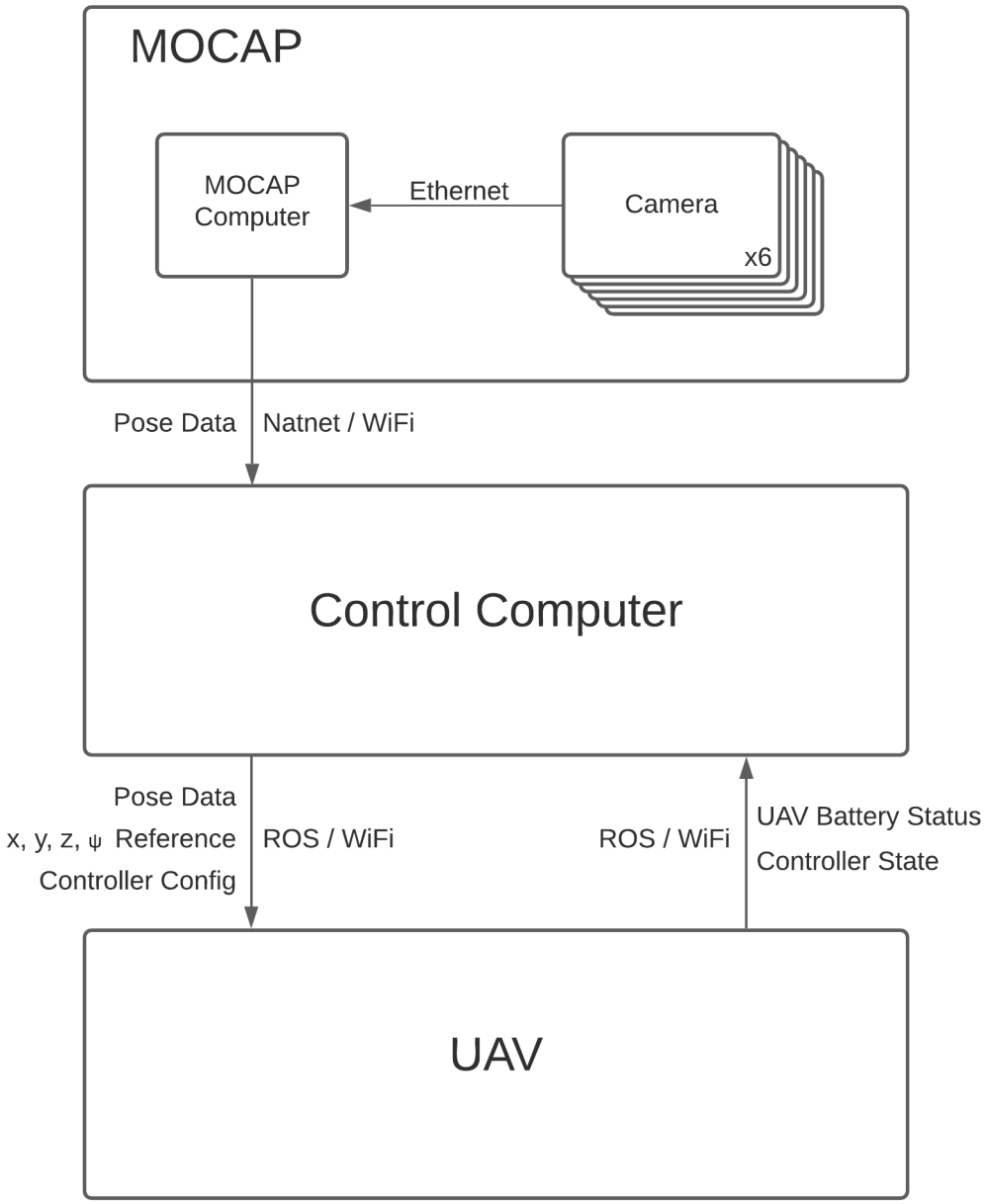


Figure 3.18: System Architecture

A six-camera OptiTrack motion capture system was set up to provide real-time high-resolution pose measurements (position and orientation) to the system, broadcast over WiFi using the Natnet protocol. Six motion capture cameras were connected to a Windows computer running OptiTrack's Motive software. The UAV's pose was tracked using a set of 4 infrared markers mounted on the rotor arms.

The control computer fulfilled the role of a ground station for the UAV. It provided a

command line interface (CLI) for the user to control and monitor the state of the UAV. The control computer interfaced with the UAV over WiFi using ROS.

On board the UAV, the flight computer received instructions from the control computer, interfaced with the flight controller, and returned status updates to the control computer.

3.4.2.1 Control Computer

The control computer functioned as the ground station for the UAV. Figure 3.19 shows the internals. A Linux laptop running Ubuntu 18.04 served as the control computer and hosted several ROS nodes, including the ROS master. These nodes each served a different purpose.

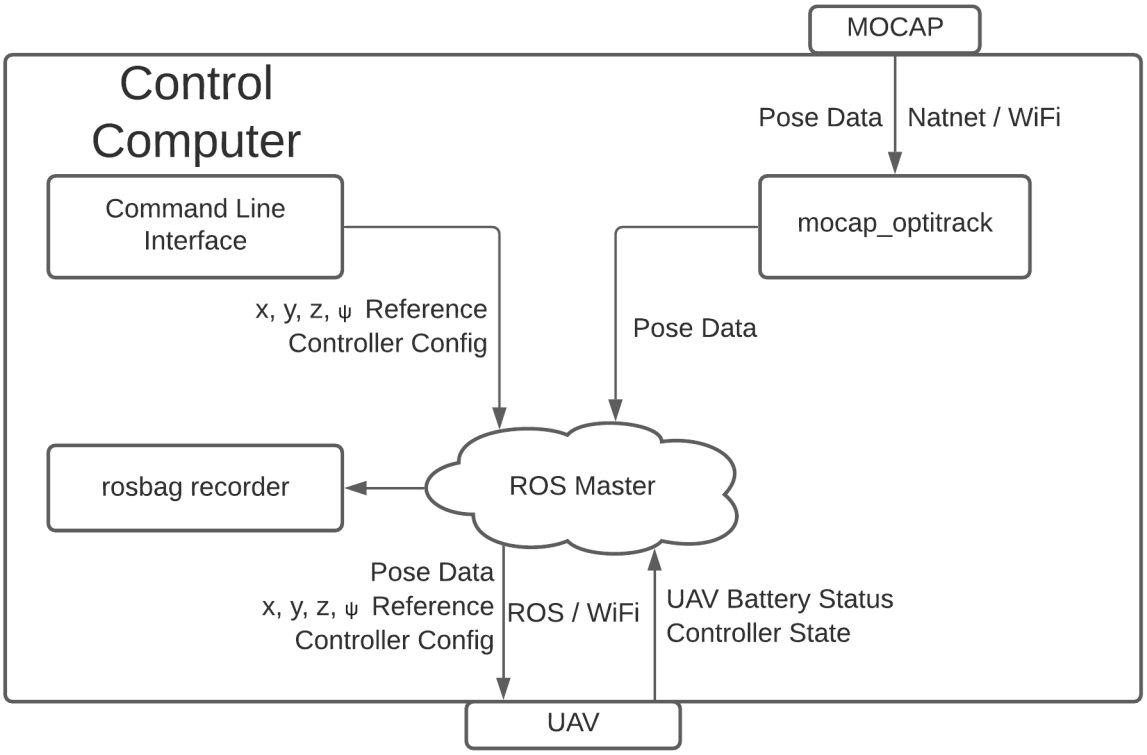


Figure 3.19: System Architecture

A mocap_optitrack node was used to broadcast the MOCAP pose data onto the ROS network, making it accessible to the UAV and to the user through the CLI.

A CLI node was written to provide the user with the tools to configure, control, and monitor the UAV during flight. Through the CLI, the user was able to set the gains of the position controller (discussed further in section 3.3.4). In flight the user was able to send position and yaw references for the UAV to follow. Finally, the CLI provided the

user with real-time status information, including the battery level, position, orientation, and controller state information. The CLI was written in Python using the rospy library. Figure 3.20 shows the CLI. The source code can be found in appendix A.

Finally, the rosbag package was used to record flight data for later analysis.

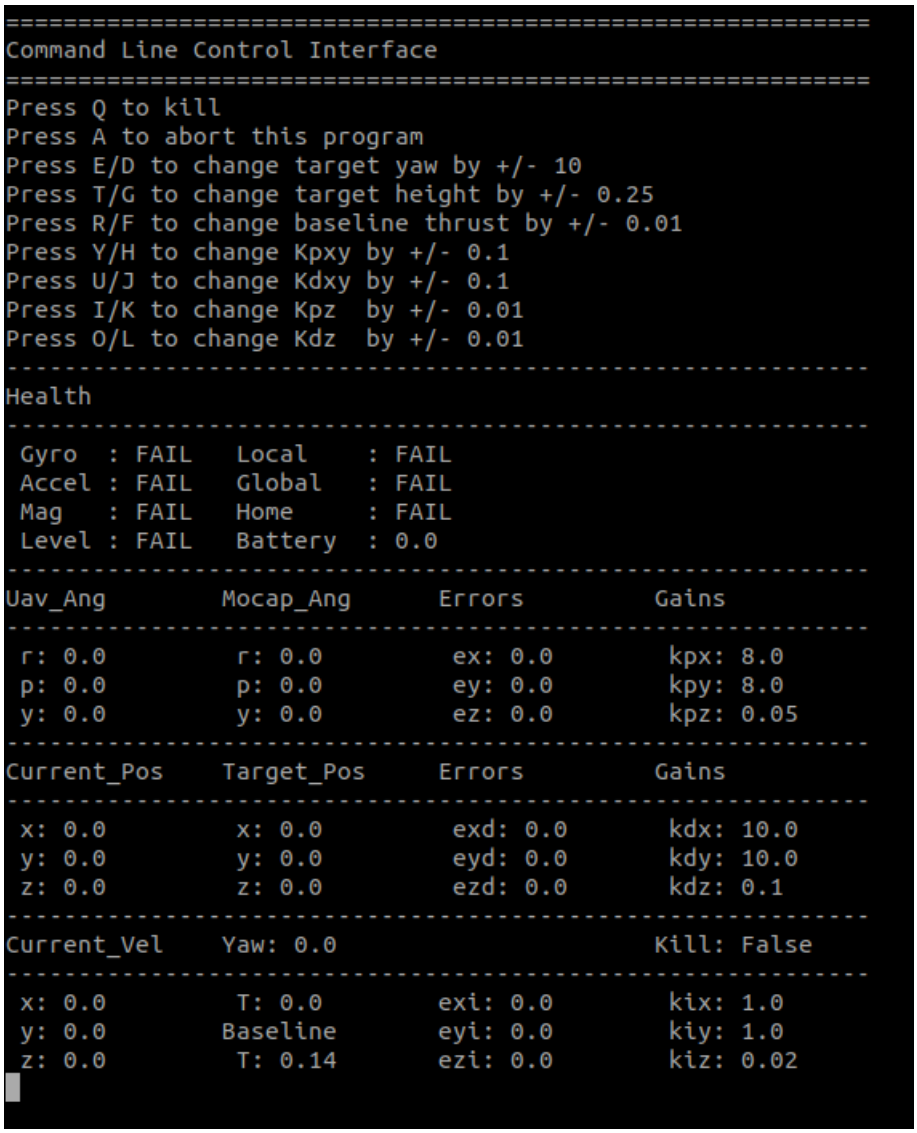


Figure 3.20: An instance of the CLI. In this instance, neither the UAV nor the MOCAP system are connected, so most of the values are zero.

3.4.2.2 UAV

The UAV was built out of several subsystems as shown in figure 3.21.

A Raspberry Pi 3b+ played the role of the flight computer, providing an interface to the offboard systems over WiFi. A position controller node on the flight computer was

written to receive the controller configurations, position and yaw references, and the pose measurements from the ROS network. The position controller node implemented a PID position controller, as described in sections 3.3.3 and 3.3.4. The outputs of the position controller were sent to the flight controller over a USB connection using the MAVlink protocol. The position controller node received battery status updates from the flight controller over MAVlink and forwarded them to the ROS network. The position controller node was written in C++ using the roscpp and MAVSDK libraries to provide the ROS and MAVlink interfaces, respectively. The source code can be found in appendix A.

A Pixhawk 4 was used as the flight controller. The primary function of the flight controller was to act as an attitude controller and interface with the motors through the ESCs. The flight controller received attitude and thrust references from the flight computer over MAVlink and provided the ESCs with speed references, encoded in PWM. These ESCs then drove the motors.

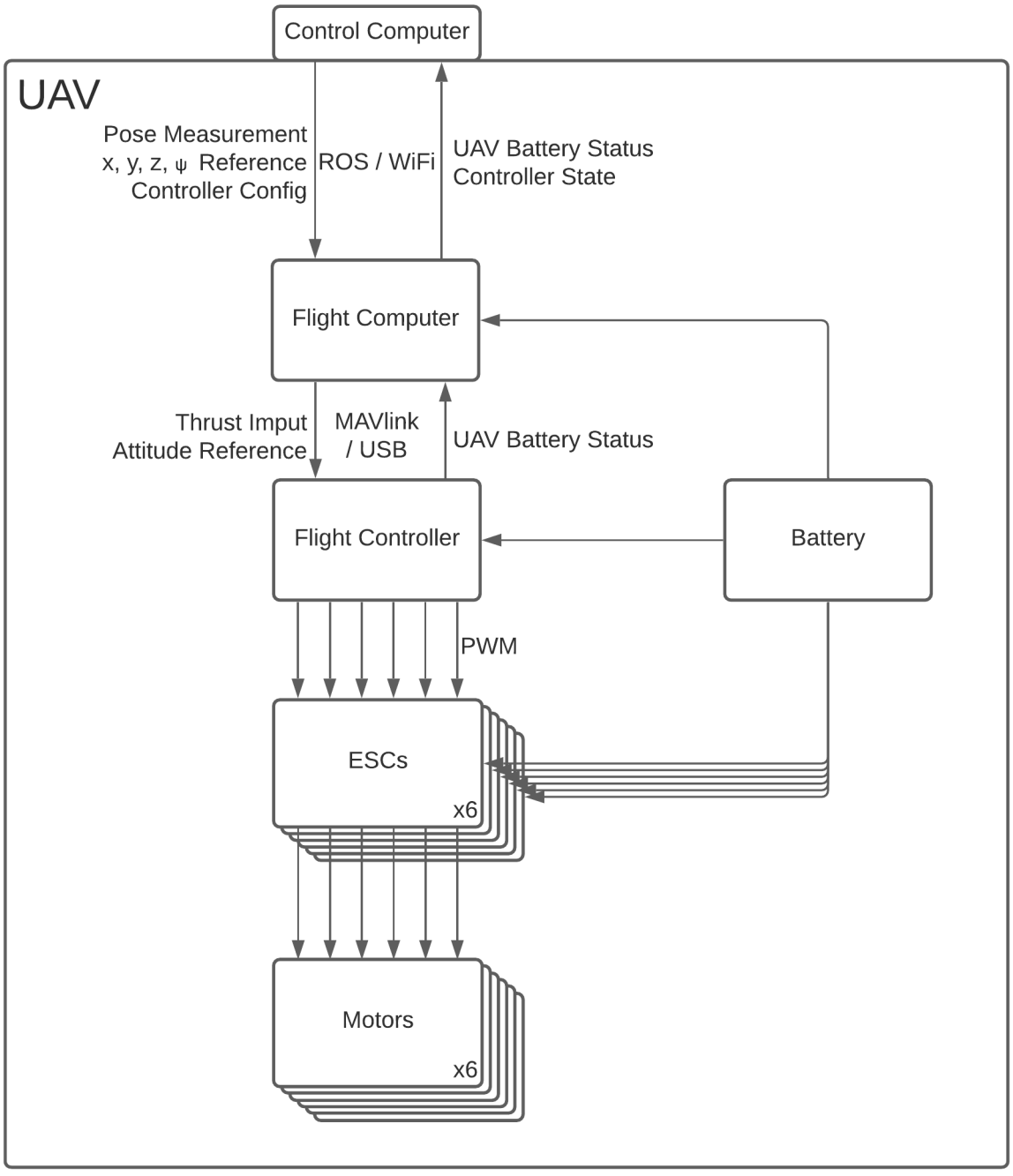


Figure 3.21: System Architecture

Thrust Scaling

During early flight tests, the inputs required to keep the UAV airborne were found to increase over the course of a flight. This was traced to the battery. The UAV is powered by a 4 cell, Li-Po battery, with a nominal voltage of 14.8V. The real voltage, however, ranges from 16.8V at full charge to 12V at full discharge. For each motor, the motor speed ω_i is roughly proportional to the motor voltage V_b,

$$\omega_i \propto V_b$$

and so, recalling equations 3.1 and 3.4, the produced force $\mathbf{f}^{\mathcal{B}}$ is roughly proportional to its square,

$$T \propto V_b^2$$

The motors on the UAV are fed directly from the battery and so the motor speeds are directly coupled to the state of the battery. This means that to provide a constant set of motor speeds or a constant thrust, the input to the motors must increase as the battery voltage decreases.

The thrust input T_{in} to the onboard flight controller is a value between 0 and 1. Through testing, it was determined that this value represents the percentage of max thrust, with 1 representing 100%. To be able to reliably control the altitude, the inputs to the UAV controller needed to be scaled to take the falling battery level into account. First, data was taken from a flight where the UAV hovered in place until the battery was mostly discharged. This was done to collect data over a large portion of the non-linear voltage drop of the Li-Po battery. The input values were then compared to the real acceleration of the UAV, as measured by the OptiTrack motion capture system, to find the input output relationship. The acceleration of the UAV, along its z-axis, is 0 during hovering and the only forces acting upon it are thrust and gravity, leading to

$$T = mg \tag{3.45}$$

where m and g are both constants. Figure 3.22, shows the attempts at scaling, the result of which is

$$T_{\text{in}} = \frac{T}{f_{scale} V_b^2} \tag{3.46}$$

where $f_{scale} = 0.3368$.

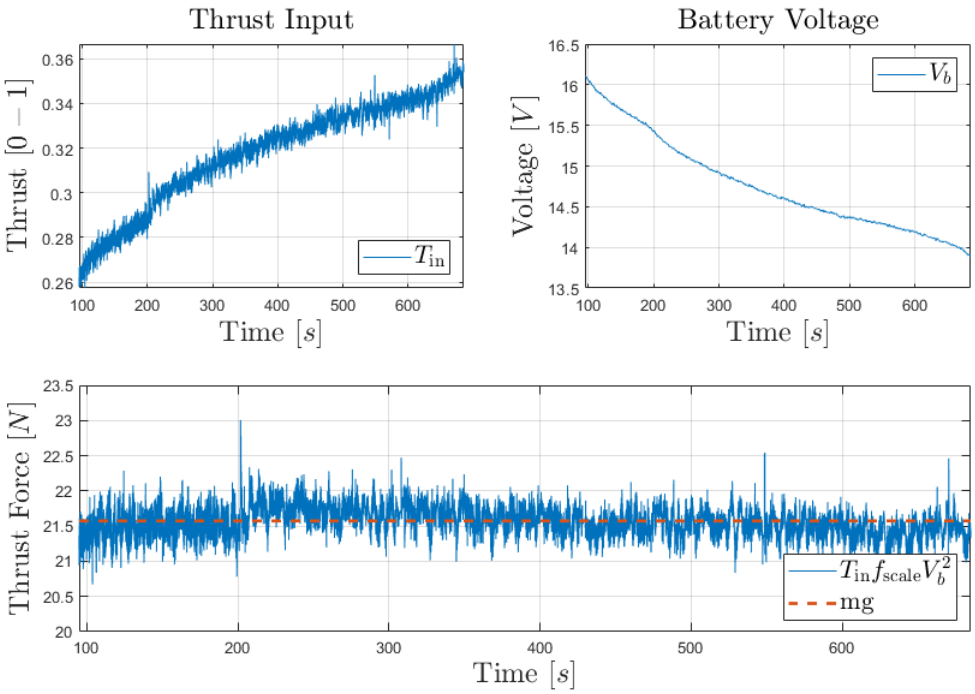


Figure 3.22: Battery test flight results. The upper two plots show the thrust input and battery voltage over time. The bottom plot shows the results of scaling. Note that there was a small increase in the z reference at $t = 200$s.

3.4.3 Attitude Controller Alignment

The purpose of the simulation model is to approximate the behaviour of the physical UAV in flight, so as to increase iteration speed and prevent damage during flight tests. The primary difference between the simulated and physical UAVs is the attitude controller, as discussed in section 3.3.2. For the simulation to be useful, it is necessary to align the response of the simulated attitude controller with one present on the physical UAV. To acquire the required attitude data, infrared tracking dots were attached to the UAV, which was then flown in view of an OptiTrack motion capture system. The UAV was then to be fed step inputs on roll, pitch, yaw and height. This setup, however, posed a problem. The small indoor space used for the motion capture prevented the UAV from taking large step inputs. The solution was to build a test rig, which was then attached to a pole. This test rig, shown in figures 3.23 and 3.24, constrains the motion of the UAV to rotation about its y-axis. The test rig was designed in Onshape and printed on a Creality Ender 3 3D printer.

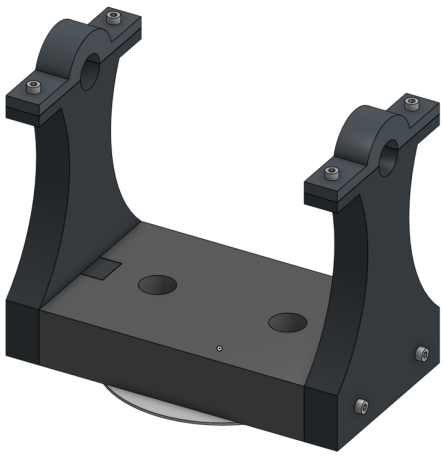


Figure 3.23: A 3D render of the test rig. The arms of the UAV pass through the hole on either side, limiting the UAV to only one degree of freedom, pitching.

3.4.3.1 Pitch and Roll Controllers

In the process of designing the test rig, some assumptions needed to be made. The center of mass was assumed to be located at the center of the UAV body, lying on a plane passing through the center of the rotor arms. Additionally it is assumed that the UAV's pitch and roll dynamics are approximately the same, that is, that they both exhibit the same response to a step input. Designing a 1-DOF test rig for pitch is mechanically simpler, due to a rotor arm pair ($\mathbf{r}_2$ and $\mathbf{r}_5$) aligned along the UAV's y-axis. The results from this rig can then be generalized from pitch to roll, using the assumptions outlined above.

Figure 3.24: The physical UAV mounted on the test rig.

The response to the step input is shown in figure 3.25. With this data to test against, the roll and pitch gains of the attitude controller were hand-tuned to achieve the desired response, also shown in figure 3.25.

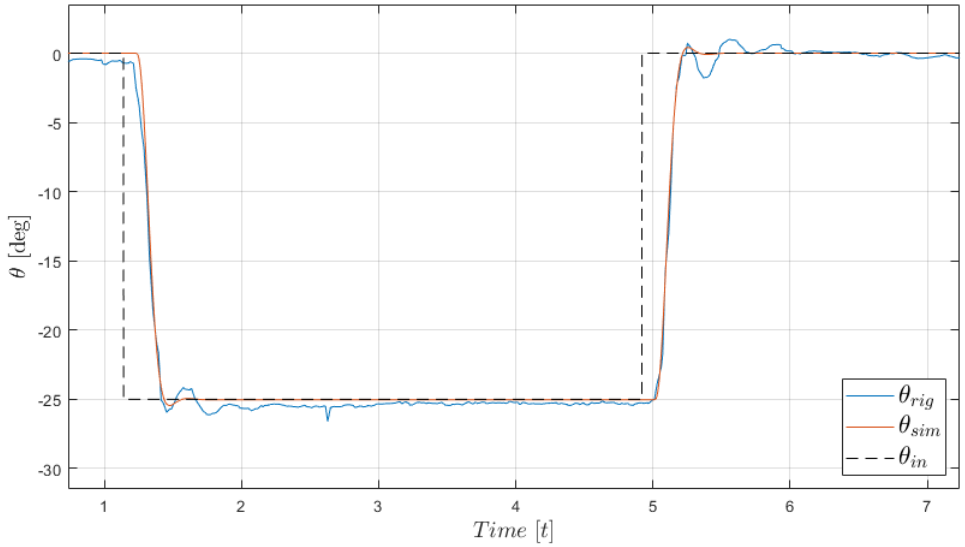


Figure 3.25: Closed Loop response of the pitch attitude controller. The reference pitch θ_{in} is shown by the dashed black line. The measured response from the physical UAV in the test rig θ_{rig} is shown as the blue line. The orange line shows the output of the attitude PID θ_{in} detailed in section 3.3.2. The gains used are shown in table 3.5.

	k_p	k_d	k_i
ϕ	0.17	0.0119	0.0017
θ	0.17	0.0119	0.0017
ψ	0.051	0.0119	0

Table 3.5: Attitude PID controller gains.

3.4.3.2 Yaw Controller

The final step in aligning the attitude controller in simulation with the physical UAV is to tune the yaw component of the PID controller. While the pitch and roll controllers could be tuned together due to the symmetry of the UAV's contruction, the yaw system has different dynamics. This is due to the means by which yaw torque is produced. The yaw torque is produced as a reaction torque, countering the torque required to spin the propeller. This is a less efficient method of producing torque than that of the roll and pitch torques, which are generated by applying forces at a distance from the center of mass, so larger motor speeds changes are required to achieve the same torque.

To collect the data for tuning the attitude PIDs, the UAV was flown in a hover at an altitude of 1.5 meters and given a step input to the yaw controller. With the data collected, the PID was tuned in the same way as for the pitch and roll controllers. The results of the data collection and tuning are shown in figure 3.26. The gains used are shown in table 3.5.

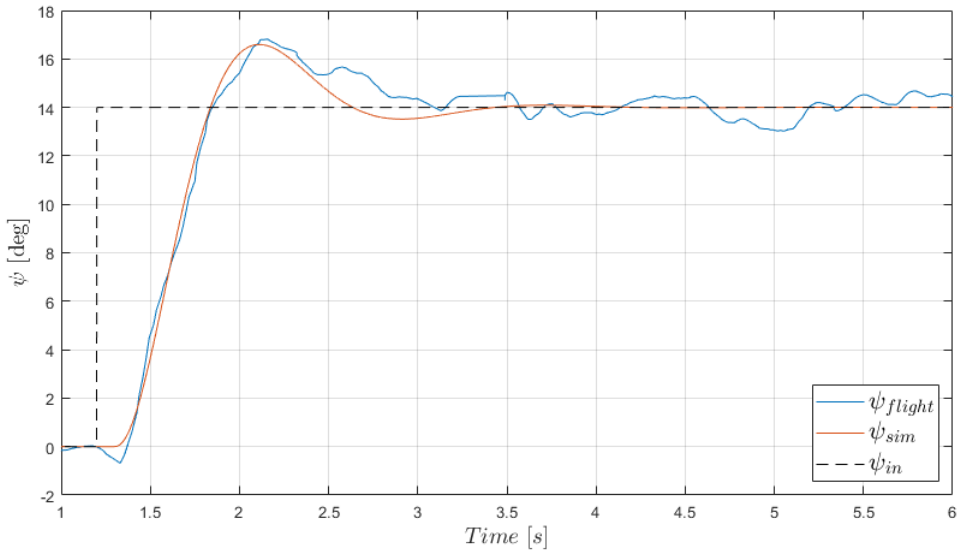


Figure 3.26: Closed loop step response of the yaw controller. The reference yaw ψ_{in} is shown by the black dashed line. The measured response from the physical UAV in flight ψ_{flight} is shown as the blue line. The orange line shows the output ψ_{sim} of the attitude PID in simulation.

With the yaw controller tuned, all of the components of the simulated attitude controller are tuned to the response of the physical UAV.

3.5 Verification

To verify that the tuned attitude controller in simulation properly emulates the embedded controllers on the physical UAV, a test flight was conducted both in the simulation and in experiment. Both flights used the same input signals and position controllers with the same gains, shown in table 3.6. The results of the test flight are shown in figure 3.27.

	k_p	k_d	k_i
x	8	10	1
y	8	10	1
z	0.05	0.1	0.02

Table 3.6: Position PID controller gains for equations 3.29, 3.43, and 3.44.

The plots show that the simulation matches the experiment well enough to predict the behaviour of the UAV to a reasonable degree of precision, thus validating the simulation model. The small differences between the UAV in simulation and the physical UAV in experiment reveal that the latter responds slower. The rise time is slightly slower and the settling time is also slower. The differences are likely due to several factors. One is the differences between the PID attitude controller used in simulation and the actual PX4 onboard attitude controller. Furthermore, there are forces acting on the physical UAV which were not modelled in the simulation, including air resistance and wind. Finally, there is measurement noise in the experiment data.

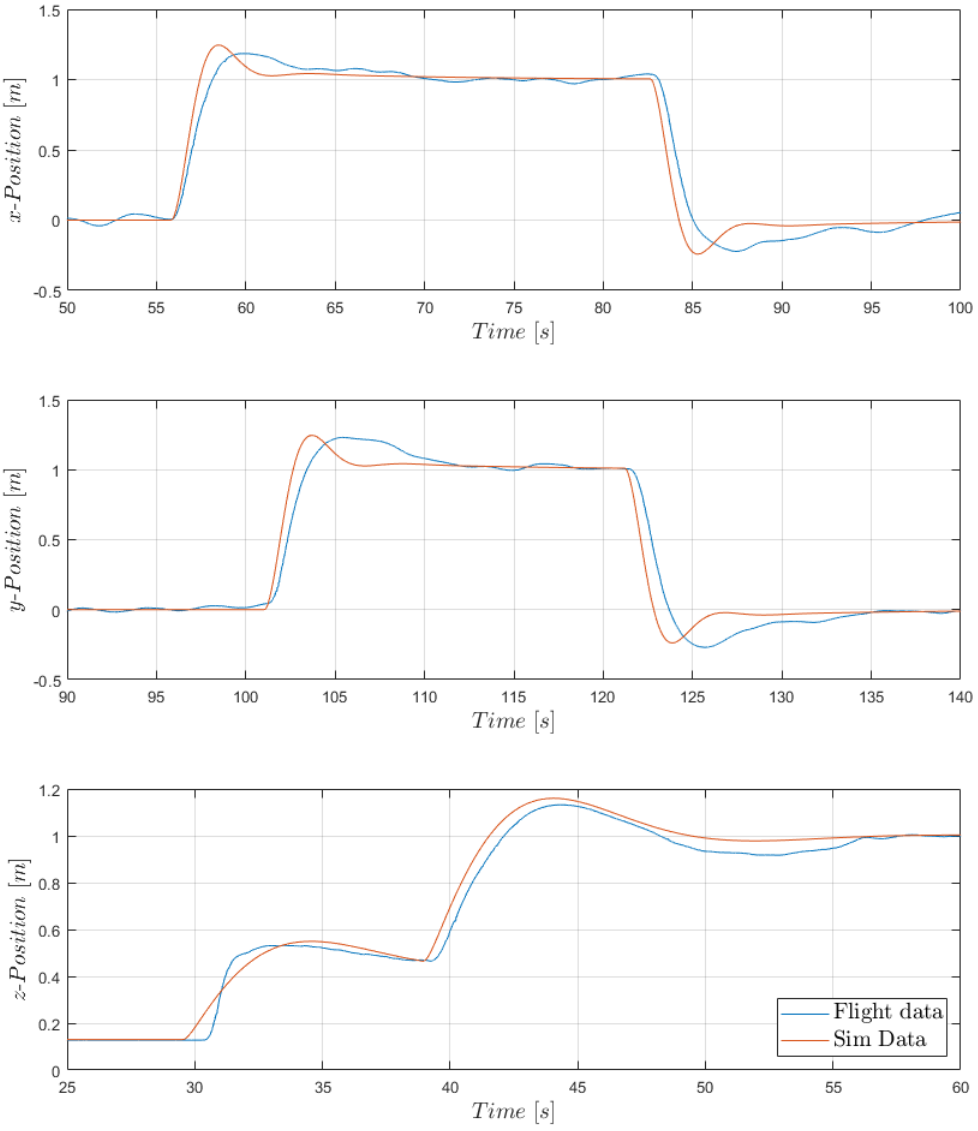


Figure 3.27: Results of a test flight with position controllers in simulation and in experiment. The plots show the position of the UAV along each of the axes of the inertial frame, from top to bottom, x, y, and z. The blue line is the data from the physical UAV and the orange line is the data from the simulation.

3.6 Summary

In this chapter, a mathematical model of a UAV in a coplanar hexarotor configuration was developed from the Newton-Euler formulation of the laws of motion. The model was then used to derive the motor mixer to control the rotor speeds to produce a desired wrench. An equivalent simulation model was developed in MATLAB and Simulink, with the Simscape Multibody Toolbox.

A cascaded control scheme was developed for position control. It consisted of a PID position controller as the outer loop and an attitude controller as the inner loop. The physical UAV used the attitude controller built into the onboard flight controller. The simulation model implemented a PID attitude controller, which was tuned to match that of the physical UAV.

The physical UAV was constructed and the experimental setup, with the MOCAP system and the ground station, was described. A command line interface was developed to control the UAV over a ROS network. Using ROS to receive measurements from the MOCAP system and targets from the CLI, and MAVSDK to send commands to the Pixhawk 4, the PID position controller was implemented on the UAV flight computer which was then used to validate the simulation model.

The challenge posed by the motor voltage dropping as the battery discharged was overcome by performing and analysing flight tests and developing a method to compensate. This compensation was then implemented on the flight computer.

The response of the simulated attitude controller was aligned with the one already present on the physical UAV. This was accomplished in several steps. A test rig was designed, printed, and assembled to overcome the challenge of measuring large attitude step responses in a small indoor space. The simulated attitude controller gains were tuned to match the behavior of the physical UAV.

The results from flight tests and simulations conducted using the same inputs were in correspondence and thus verified the simulation model.

The dynamic model, simulation model, controllers, and test setup developed in this chapter provide a platform which will be used in the next chapter to model and test contact interactions.

CHAPTER 4

Contact

With the model for the UAV in free flight developed in the previous chapter, it is now time to look at how the UAV interacts with the environment. The real world applications motivating this thesis are surface contact tasks, including contact inspection, drilling, and polishing. These tasks all involve interactions with walls or wall-like surfaces. The scope of the interaction in this chapter is limited to static contact, that is, applying wrench to a stationary point on a surface.

In this chapter the dynamic model developed in Chapter 3 will be extended to add a manipulator arm. The primary focus will be a manipulator arm rigidly attached to the center of mass of the UAV. Contact forces are introduced to the dynamic model from which a stable operating region of pitch and thrust for static contact is identified and maximum applicable interaction force is established.

The controllers from chapter 3 will be introduced to the discussion. The contact force will be traced back through the controllers to the inputs to the position controller, resulting in the description of a force controller. The constraints of the cascaded PID control scheme will be evaluated to identify a realisable subregion of the operating region and establish the maximum realisable interaction force. This analysis will be corroborated in simulation.

The simulation model from chapter 3 will be extended with the manipulator and a model for contact and used to simulate a contact scenario. The force controller will also be implemented and tested in simulation. Experiments will be done with a physical UAV and collected data will be compared against simulation data.

A second manipulator configuration consisting of a manipulator arm attached to an actuated joint with one degree of freedom (1-DOF) will also be modelled and simulated.

Throughout this chapter several parameters affecting contact are identified. Together these constitute a set of UAV design parameters which can be optimized for contact application. Physical design parameters include choice of manipulator configuration, UAV mass, choice of end-effector material, and manipulator length. Controller design parameters include the gains on the attitude controller.

4.1 Dynamic Model

In this section the dynamic model is extended to consider contact with a manipulator. Contact forces are introduced and a stable operating region of pitch angles and thrust force is identified. Additionally, the maximum applicable interaction force is identified.

Contact between two bodies is a complex phenomenon with many non-linear and emergent properties. The two primary properties to account for are friction and normal forces, which resist motion parallel to and normal to the contact plane, respectively. Friction forces have two modes, *static friction* which resists acceleration of a stationary object and *kinetic friction* which resists the motion of an object. Normal forces prevent penetration and thus exert force only in one direction, outward.

For the purposes of this chapter it is assumed that the objects with which the UAV interacts have flat faces oriented such that the plane of contact lies parallel to the inertial $y, z-$plane and the normal force f_n is applied to the UAV along the $x-$axis. Put simply, the interaction surface is a wall perpendicular to the x-axis. The manipulator, in its default position, lies along the UAV's $x-$axis and is affixed to the UAV's center of mass. The manipulator consists of a thin massless cylindrical rod of length L_m tipped with a correspondingly small sphere, which is treated as a point.

For this chapter, it is convenient to define a local frame L with an origin located at the center of mass of the UAV and whose axes are aligned with those of the inertial frame. The rotation matrix from B to L is given by

$$R_B^L = R_B^I \tag{4.1}$$

When the manipulator comes into contact with an object in the environment, the normal force is applied at the point of contact, such that the manipulator does not penetrate the object.

Friction forces occur during contact and resist motion parallel to the contact surface, in this case along the y, z-plane. Unlike the normal force, these forces are bounded. The upper bound on the magnitude of friction is linearly dependent on the magnitude of the normal force,

$$||\mathbf{f}_f|| \leq \mu |f_n| \tag{4.2}$$

where μ is the coefficient of friction. For the purposes of the static scenarios explored in this thesis, only static friction will be considered, with $\mu = 0.7$.[1]

Together the friction forces and normal force work to resist motion at the point of contact, that is, they work to bring the net force to zero.

The contact force $\mathbf{f}_c$ is the force applied to the UAV from the contact surface. This is necessarily the opposite of the interaction force applied to the surface by the UAV. As

[1] According to the table in [7], the coefficient of friction for silicon or rubber (the end-effector) and many wall like surfaces is around 0.7.

the normal force acts along the x-axis and the friction force acts along the perpendicular plane, they combine in the inertial frame to create the contact force as follows

$$\mathbf{f}_c^I = \begin{bmatrix} f_n \\ \mathbf{f}_{f,y} \\ \mathbf{f}_{f,z} \end{bmatrix} \tag{4.3}$$

where $\mathbf{f}_{f,y}$ and $\mathbf{f}_{f,z}$ are the y and z components of the friction force and

$$||\mathbf{f}_f|| = \sqrt{\mathbf{f}_{f,y}^2 + \mathbf{f}_{f,z}^2} \tag{4.4}$$

The manipulator extends out from the UAV's center of mass, along the x-axis of the body-fixed frame, with a length L_m. The rigid attachment means that the point of contact occurring at the end of the manipulator is constant in the body frame. The position of the end-effector (the sphere) is given by

$$\boldsymbol{\xi}_e^B = \begin{bmatrix} L_m \\ 0 \\ 0 \end{bmatrix} \tag{4.5}$$

The dynamics of the end-effector in the inertial frame are, in the Newton-Euler formulation, given by [20]

$$\begin{bmatrix} \sum \mathbf{f} \\ \sum \boldsymbol{\tau} \end{bmatrix} = \begin{bmatrix} m\mathbf{1} & mS(\boldsymbol{\xi}_e^B) \\ -mS(\boldsymbol{\xi}_e^B) & I - mS(\boldsymbol{\xi}_e^B)S(\boldsymbol{\xi}_e^B) \end{bmatrix} \begin{bmatrix} \ddot{\boldsymbol{\xi}}_e^I \\ \dot{\boldsymbol{\nu}} \end{bmatrix} + \begin{bmatrix} -mS(\boldsymbol{\nu})S(\boldsymbol{\nu})\boldsymbol{\xi}_e^B \\ S(\boldsymbol{\nu})\left(I - mS(\boldsymbol{\xi}_e^B)S(\boldsymbol{\xi}_e^B)\right)\boldsymbol{\nu} \end{bmatrix} \tag{4.6}$$

When the UAV is in static equilibrium, these dynamics are greatly simplified,

$$\begin{bmatrix} \sum \mathbf{f} \\ \sum \boldsymbol{\tau} \end{bmatrix} = [\mathbf{0}] \tag{4.7}$$

If the UAV is in static contact with the wall, that is, in contact and in static equilibrium, then the net force acting on the UAV is

$$m\ddot{\boldsymbol{\xi}}_e^I = \mathbf{f}_{\mathrm{g}}^I + \mathbf{f}_t^I + \mathbf{f}_c^I = \mathbf{0} \tag{4.8}$$

where $\mathbf{f}_{\mathrm{g}}^I = [0, 0, -mg]^T$ is the force of gravity and $\mathbf{f}_t^I$ is the thrust force. The thrust force (in the body frame) $\mathbf{f}_t^B$, was previously referred to as $\mathbf{f}^B$ in chapter 3 and has the form

$$\mathbf{f}_t^B = \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix} \tag{4.9}$$

In the inertial frame $\mathbf{f}_t^I$ has the form

$$\mathbf{f}_t^I = T \begin{bmatrix} S_\phi S_\psi + C_\phi C_\psi S_\theta \\ C_\phi S_\psi S_\theta - C_\psi S_\phi \\ C_\phi C_\theta \end{bmatrix} \tag{4.10}$$

Rearranging equation 4.8,

$$\mathbf{f}_c^I = -T \begin{bmatrix} S_\phi S_\psi + C_\phi C_\psi S_\theta \\ C_\phi S_\psi S_\theta - C_\psi S_\phi \\ C_\phi C_\theta \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} \tag{4.11}$$

With the contact force calculated above, it is now possible to calculate the torques experienced by the UAV during contact. The contact force $\mathbf{f}_c^I$ is rotated into the body-fixed frame of the UAV and the contact torques are found as follows

$$\boldsymbol{\tau}_c^B = \boldsymbol{\xi}_e \times R_I^B \mathbf{f}_c^I = S(\boldsymbol{\xi}_e) R_I^B \mathbf{f}_c^I = L_m \begin{bmatrix} 0 \\ T - mgC_\phi C_\theta \\ mgC_\theta S_\phi \end{bmatrix} \tag{4.12}$$

Here we see that the magnitude of the torques is governed by the contact forces and dependent on the length of the manipulator.

In the specific case where the UAV is maintaining altitude in free flight, the z component of the net force is zero, so $TC_\phi C_\theta = mg$, that is, $T = \frac{mg}{C_\phi C_\theta}$ and

$$\mathbf{f}_c^I = -\frac{mg}{C_\phi C_\theta} \begin{bmatrix} S_\phi S_\psi + C_\phi C_\psi S_\theta \\ C_\phi S_\psi S_\theta - C_\psi S_\phi \\ 0 \end{bmatrix} \tag{4.13}$$

This equation makes it apparent that the forces being applied to the environment are governed entirely by the mass of the UAV and its orientation. Regardless of the thrust available to the UAV, the primary task of the flight controller is to keep it in the air. In a static contact scenario this constraint limits the potential force to a small range around hovering. The only way for the UAV to apply force is to tilt towards the object and this action is limited by the torque from contact.

Of the interaction forces applied by the UAV, the component perpendicular to the contact surface of the object is of primary interest, in this case the $x-$component. This component will be referred to as *the interaction force* throught this chapter. Ideally, for static force application, the end-effector is motionless on the surface. To ensure that this is the case, the forces perpendicular to the contact surface must remain within the bounds of equation 4.2. Combined with the definition of $\mathbf{f}_c^I$ in equations 4.3 and 4.4 and its derivation in equation 4.11, the following inequality emerges

$$\sqrt{(T(C_\phi S_\psi S_\theta - C_\psi S_\phi))^2 + (TC_\phi C_\theta - mg)^2} \leq \mu |T(S_\phi S_\psi + C_\phi C_\psi S_\theta)| \tag{4.14}$$

For the following analysis of friction and later torque from contact, it will be assumed that the UAV is facing the wall head on, that is, with a yaw $\psi = 0$. The inequality then reduces to

$$\sqrt{(TS_\phi)^2 + (TC_\phi C_\theta - mg)^2} \leq \mu |TC_\phi S_\theta| \tag{4.15}$$

Furthermore, it is also assumed that the UAV's roll $\phi = 0$, further reducing the inequality to

$$|TC_\theta - mg| \leq \mu |TS_\theta| \tag{4.16}$$

Figure 4.1 show the region for which the inequality 4.16 holds. Outside of this region the UAV is unable to remain static.

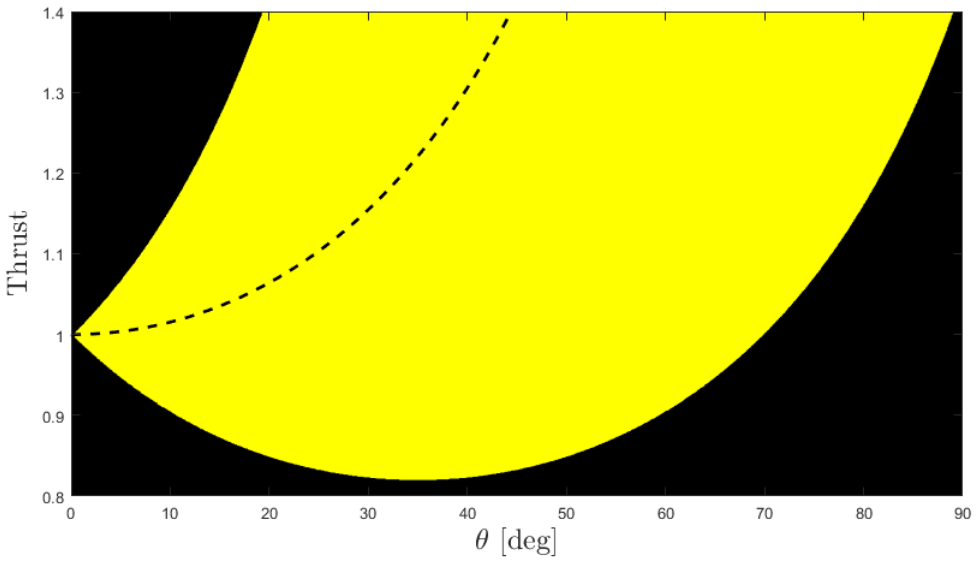


Figure 4.1: *Friction Bound:* The bound on the pitch angle and thrust force defined by the inequality 4.16. The x-axis is the pitch angle in degrees and the y-axis is the thrust, scaled by mg. The yellow area is the region where the inequality is true, that is, the friction force is able to keep the end-effector in place. In the region above the yellow, the end effector will slip up, and in the region below and to the right it slips down. The dashed black line indicates hovering thrust as a function of pitch angle, $T = mg/C_\theta$.

Friction is not the only constraint on the interaction force that can feasibly be applied to an object by the UAV. To remain stable, the contact torque must also be countered. While the UAV can only generate a force in a single direction, it is capable of generating a torque around all three of its axes. These torques are generated by manipulating the balance of forces generated by each rotor. In the previous chapter the motor mixer, equation 3.22, was defined to translate a set of desired thrust and torques into motor speeds. The motor mixer does not, however, take into account the physical limitations of the UAV. Specifically, the operating range of the brushless DC motors. The ability of the motors to counteract the contact torque prove to be a limitation on the static interaction force which can be applied.

In the following, the contact torque will be traced through the motor mixer to the motor speeds. By considering the limits of the rotors, a limit is found to the torque that can be generated and a bound on the pitch angle and thrust force of the UAV. This analysis is only valid within the friction bound in figure 4.1.

Taking the contact torque from equation 4.12 to create the desired wrench vector

$$\mathbf{w}_c^B = \begin{bmatrix} T \\ -\boldsymbol{\tau}_c^B \end{bmatrix} = \begin{bmatrix} T \\ 0 \\ -L_m(T - mgC_\phi C_\theta) \\ -L_m mgC_\theta S_\phi \end{bmatrix} \tag{4.17}$$

and applying the assumptions from above that the roll $\phi = 0$,

$$\mathbf{w}_c^B = \begin{bmatrix} T \\ 0 \\ L_m(mgC_\theta - T) \\ 0 \end{bmatrix} \tag{4.18}$$

Note that the torque $\boldsymbol{\tau}_c^B$ in equation 4.17 assumes that the static friction bound is not exceeded.

Passing the $\mathbf{w}_c^B$ through the motor mixer results in

$$\boldsymbol{\Omega}_c = M\mathbf{w}_c^B = \frac{1}{6k}\begin{bmatrix} T + \sqrt{3}s_{mr}(T - mgC_\theta) \\ T \\ T - \sqrt{3}s_{mr}(T - mgC_\theta) \\ T - \sqrt{3}s_{mr}(T - mgC_\theta) \\ T \\ T + \sqrt{3}s_{mr}(T - mgC_\theta) \end{bmatrix} \tag{4.19}$$

where $\boldsymbol{\Omega}_c$ is the vector of squared motor speeds and $s_{mr} = L_m/L_r$ is the ratio of the manipulator length to the distance of the rotors from the center of mass of the UAV.

For the motor speeds to be valid, the values in $\boldsymbol{\Omega}_c$ must be non-negative, because the rotors are only capable of rotating in the direction of upward force.

The bound on θ is plotted as a function of s_{mr}, with $T = mg/(C_\theta C_\phi)$ in figure 4.2 below. The figure shows that the maximum θ and therefore the maximum applied interaction force that can be achieved is limited by the length of the manipulator. As the end-effector moves further away from the center of the UAV, its power is reduced. In practice, an s_{mr} of 1.5 or more is required for the end-effector to extend far enough past the propellers to prevent them from contacting the wall, even for large pitch angles. For the rest of this section s_{mr} will be fixed at 2.

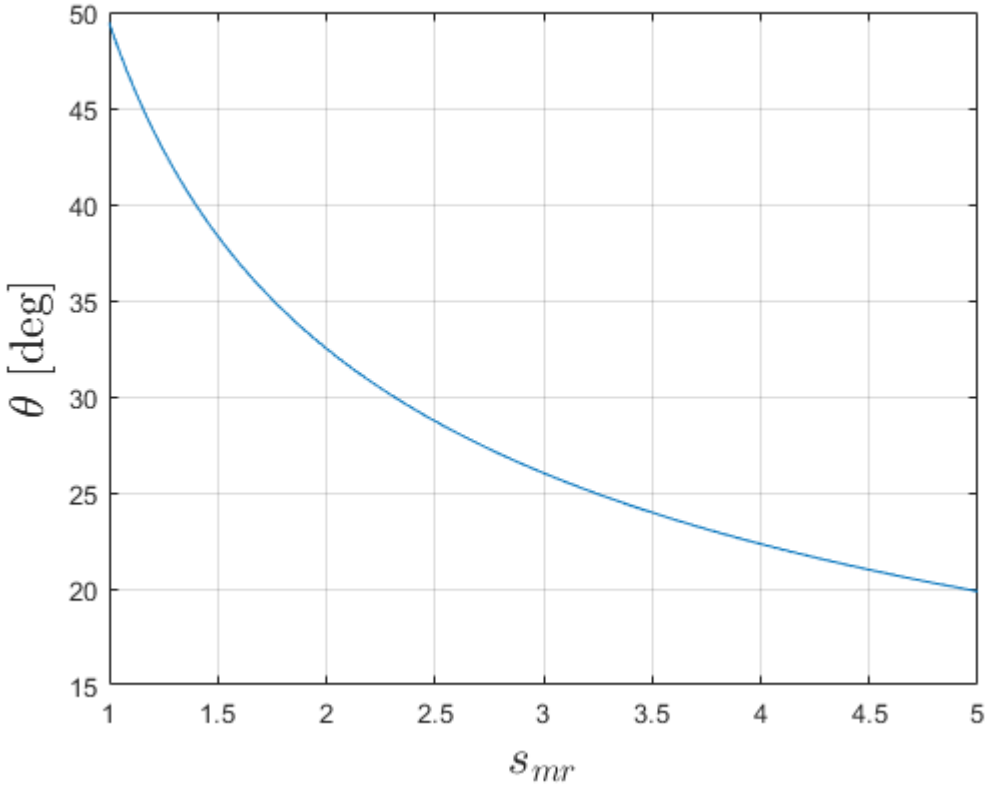


Figure 4.2: Here the maximum pitch angle, θ, is plotted against the ratio of manipulator length to rotor distance, $s_{mr} = L_m/L_r$, given a hover thrust of $T = mg/(C_\phi C_\theta)$.

Plugging these values of θ back into equation 4.13 reveals a maximum interaction force application, assuming ϕ and ψ are 0, of

$$|f_n| \leq \frac{mg}{\sqrt{\sqrt{3}s_{mr} - 1}} \tag{4.20}$$

Figure 4.3, below, shows the maximum force that can be applied as a function of s_{mr}.

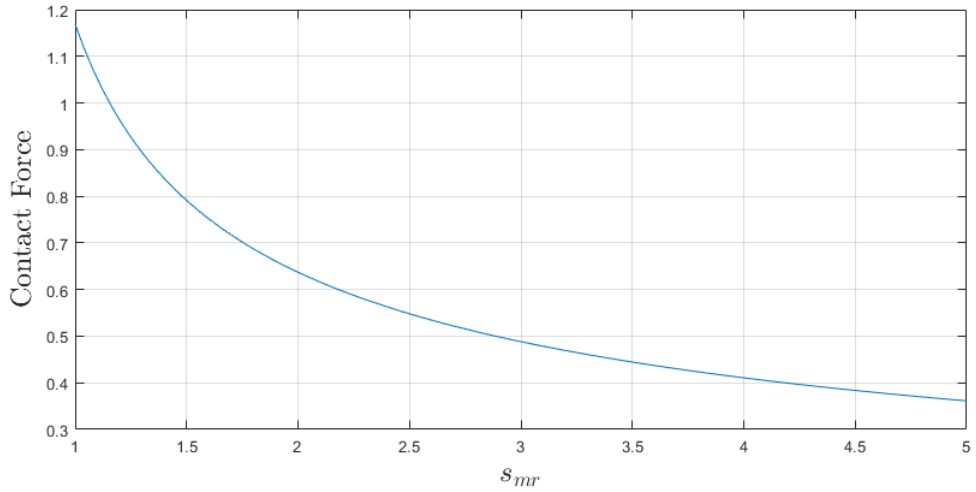


Figure 4.3: Limits on the interaction force that can be applied by the UAV in a static state.

Looking at the relationship more between θ and T more generally, figure 4.4 plots the bounds on θ and T from the inequality **??**. Within the blue region of the plot, the motors are capable of running at the speeds required to generate the torque to resist the contact torque. Outside of this region the motor mixer demands that some of the motors run at negative speeds, which is infeasible for the UAV.

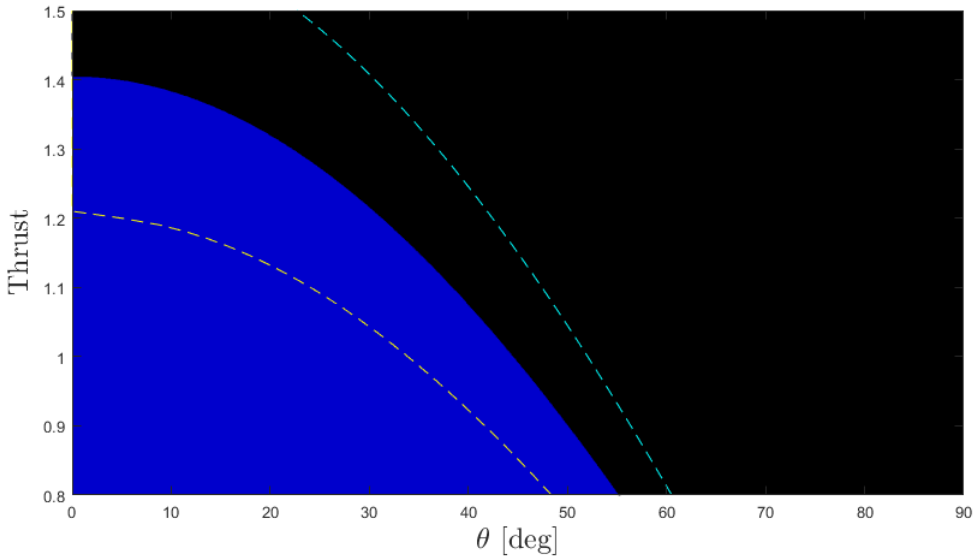


Figure 4.4: *Torque Bound*: Bound on stability from contact induced torque. The x-axis is the pitch degree in angles and the y-axis is the thrust, scaled by mg. The inequality **??** is true in the blue region, where the UAV is able to maintain stable contact. Outside of this region it will begin to pitch uncontrollably. The dashed blue line represents the bound with $s_{mr} = 1.5$ and the dashed yellow represents the bound with $s_{mr} = 3.4$ used in the experiments in section 4.4. Note that this bound is only valid within the friction bound in figure 4.1.

Figure 4.4 shows the bounds for three different values of s_{mr}. The larger the s_{mr}, the smaller the region within the stability bound. Given a UAV with some rotor length L_r, it is advantageous to add as short a manipulator as feasible.

Figure 4.5 visualizes the bounds on θ and T from the torque constraints with the bounds from friction. The green region of overlap represents the region of pitch angle and thrust force in which the UAV can operate.

The torque constraints are given by inequality **??**. Within the region where the inequality is true, the UAV is capable of counteracting the torque arising from contact and remains stable. Outside of this region, necessary counter torque exceeds the capability of the UAV, so the UAV will pitch uncontrollably towards the contact surface.

The friction constraints are given by inequality 4.16. Within the region where the inequality is true, the friction in the z direction falls below the critical value of $\mu||f_n||$ and the end-effector remains motionless on the wall. Should the force exerted on the wall by the UAV's end-effector exceed this bound, the end-effector will begin sliding either up or down depending on which side of the yellow region the UAV falls on.

Only in the green region of the plot can the UAV remain stable with its end-effector fixed in place on the wall.

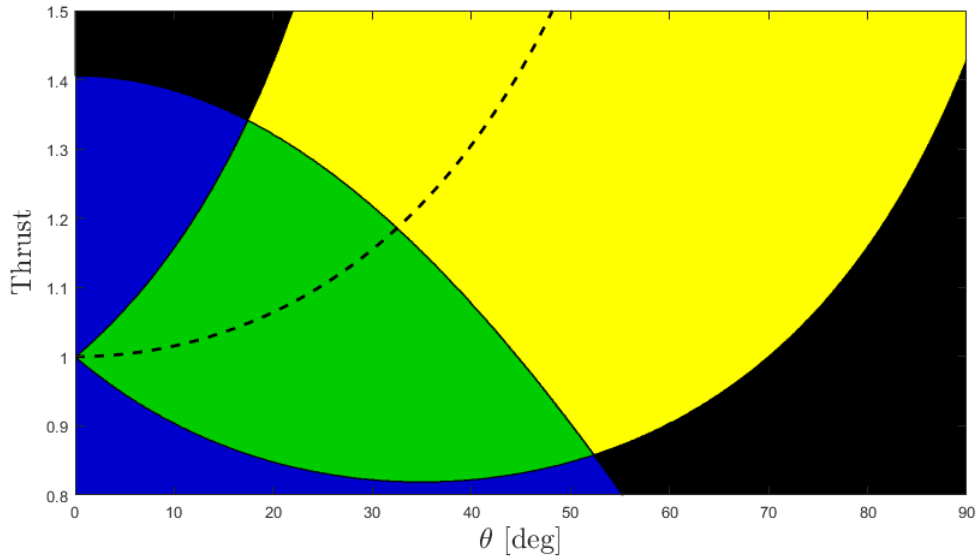


Figure 4.5: *Operating Region:* Limits on UAV pitch angle and thrust force. The x-axis is the pitch angle in degrees and the y-axis is the thrust force, scaled by mg. The domed blue and green area in the lower left of the plot is described by the inequality **??**, where the UAV is stable. The yellow and green banana shaped region is described by the inequality 4.16, where the UAV's end-effector is held in position by friction.

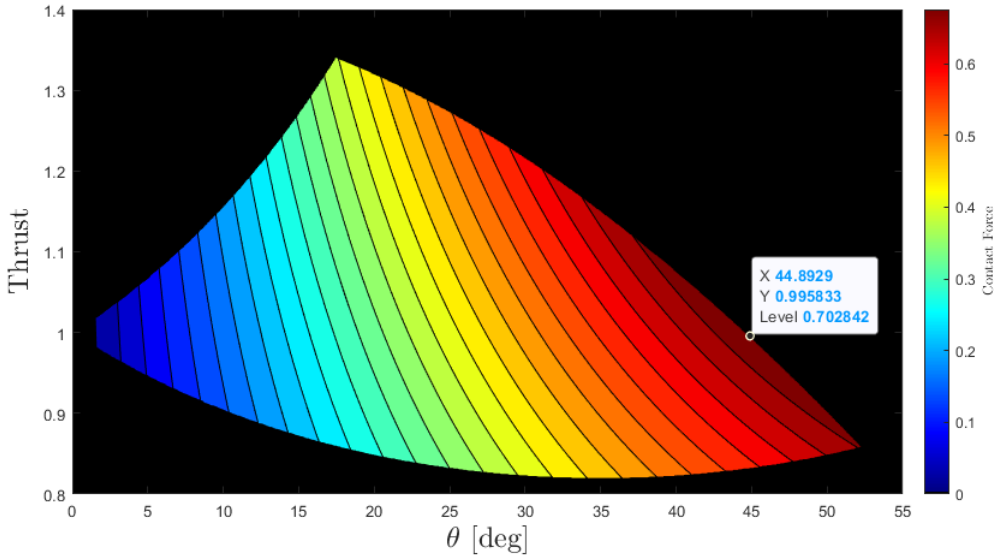


Figure 4.6: Contact force as a function of pitch angle and thrust force. Both the thrust force and contact force are scaled by mg. Note that the maximum contact force $|f_n| = 0.702842mg$ occurs towards the lower right of the operating region.

Given the operating region of the UAV, it is now possible to map the contact forces

that the UAV can handle. Figure 4.6 shows the magnitude of the normal force within the operating range, with the maximum achievable interaction force in this range being $|f_n| = 0.702842mg$ at $\theta = 44.8929°, T = 0.995833mg$.

In this section, the dynamic model from chapter 3 was used to develop a model for static contact using a rigidly attached manipulator, actuated only by the motion of the UAV's body. The physical bounds on static contact were evaluated by considering friction force, figure 4.1, and the achievable torque, figure 4.4. These bounds were combined to reveal the operating region of pitch angle and thrust force of the UAV, figure 4.5, across which the force was mapped, figure 4.6. The ultimate limits on the magnitude of the achievable interaction force were found to be the mass m of the UAV and the ratio s_{mr} of manipulator length to the rotor radius.

4.2 Controller

In this section the controllers from chapter 3 are added to the dynamic model. The contact force is traced back through the controllers to the inputs to the position controller, resulting in the description of a force controller. The constraints of the cascaded PID control scheme are evaluated to identify a realisable subregion of the operating region and establish the maximum realisable contact force. Finally, the effects of PID controller gains are discussed.

4.2.1 Mapping force to controller input

Mapping the desired contact force to the requisite input to the position controller is a three step process. First, map the desired contact force to the pitch angle. Second, map the physical pitch angle to the reference pitch input to the attitude controller. Third, map the reference pitch to the reference x-position input to the position controller.

As the outer loop, the position controller takes inputs directly from the user. In section 4.1, the operating region of the UAV was analysed in terms of thrust force and pitch angle. Where the pitch angle is related to the choice of the reference x-position, the thrust force is related to the reference z-position, the altitude reference. In this section it will be assumed that the end-effector initiates contact with the wall at the height z_{ref} which will be held constant. As the UAV pitches, the body will rotate around the end-effector and the center of mass of the UAV will rise. The location of the center of mass is then given by,

$$z_{\text{com}} = z_{\text{ref}} + L_m S_\theta \tag{4.21}$$

Taking the altitude controller from equations 3.29 and 3.31 in section 3.3.3,

$$T_{\text{ref}} = \frac{m}{C_\theta} u_z = \frac{m}{C_\theta}(k_{p,z} e_{p,z} + k_{d,z} e_{d,z} + k_{i,z} e_{i,z} + g) \tag{4.22}$$

The UAV is stationary and therefore the derivative of the error is zero, $e_{d,z} = 0$. Setting the integral gain to zero, $k_{i,z} = 0$, the relevant parameters become the proportional

components, $k_{p,z}$ and $e_{p,z}$. (This process will be repeated for the other PID controllers below. The integral gain will be discussed in section 4.2.3.) The z-position of the UAV is measured from its center of mass, so the error is $e_{p,z} = z_{\text{ref}} - z_{\text{com}}$ and

$$T_{\text{ref}} = \frac{m}{C_\theta}(k_{p,z}(z_{\text{ref}} - z_{\text{com}}) + g) = \frac{mg}{C_\theta} - \frac{mk_{p,z}L_m S_\theta}{C_\theta} \tag{4.23}$$

where mg/C_θ is the hovering thrust and the other term represents the altitude controller action. This is the thrust force T that will be assumed for the duration of this section.

To map the contact force to the pitch angle, recall equation 4.11 from section 4.1, and the assumptions $\phi = 0$ and $\psi = 0$,

$$\mathbf{f}_c^I = -T \begin{bmatrix} S_\theta \\ 0 \\ C_\theta \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} = \begin{bmatrix} -TS_\theta \\ 0 \\ -TC_\theta + mg \end{bmatrix} \tag{4.24}$$

The normal force f_n is the force of interest, here the x component $|f_n| = |TS_\theta|$. The mapping, then, from pitch to force is

$$|f_n| = \left| \frac{mgS_\theta}{C_\theta} - \frac{mk_{p,z}L_m S_\theta^2}{C_\theta} \right| \tag{4.25}$$

From this equation there is no clear derivation of the reverse mapping. Figure 4.7 plots the reverse mapping computed numerically.

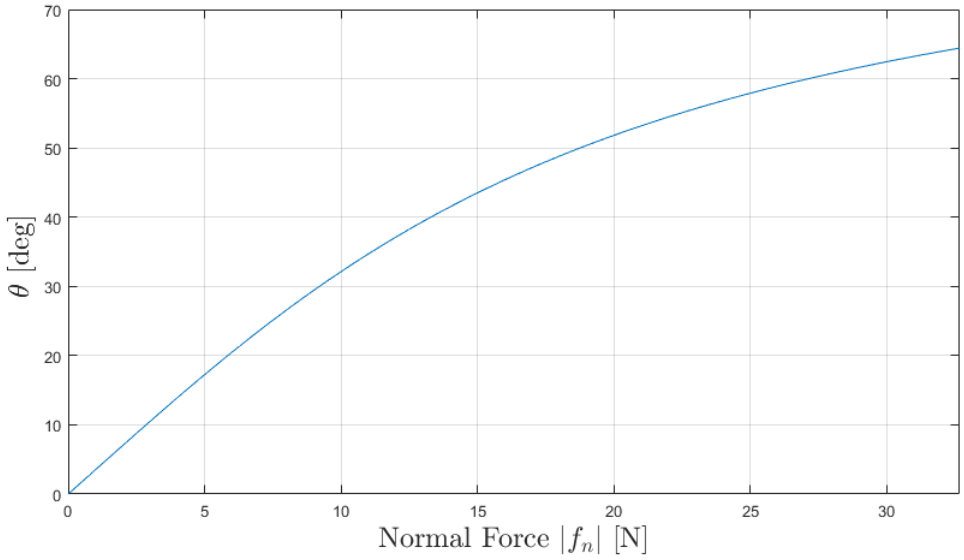


Figure 4.7: Mapping from normal force to pitch angle with $m = 2.2$kg. Note that only pitch angles $\theta \leq 32.5$ are within the operating region.

To map the physical pitch angle to the reference pitch angle, recall the attitude controller defined in equation 3.27 and 3.28, and consider only the pitch component

$$\tau_\theta^B = \frac{u_\theta}{I_{yy}} = \frac{1}{I_{yy}}(k_{p,\theta}e_{p,\theta} + k_{i,\theta}e_{i,\theta} + k_{d,\theta}e_{d,\theta})$$

As before, the relevant parameters are the proportional components, $k_{p,\theta}$ and $e_{p,\theta}$. To remain stationary, the pitch torque output of the attitude controller must counteract the pitch torque from equation 4.18

$$\tau_\theta^B = \frac{k_{p,\theta}e_{p,\theta}}{I_{yy}} = \frac{k_{p,\theta}(\theta_{\text{ref}} - \theta)}{I_{yy}} = L_m(T - mgC_\theta) \tag{4.26}$$

Solving for θ_{ref},

$$\theta_{\text{ref}} = \theta - \frac{I_{yy}L_m(mgC_\theta - T)}{k_{p,\theta}} \tag{4.27}$$

Figure 4.8 shows the required reference pitch to maintain stationary contact as a function of the pitch angle. The plot shows that θ is only responsive to changes in θ_{ref} up to the inflection point at $\theta = 15.3°, \theta_{\text{ref}} = 8.74227°$. In other words, the inflection point is the upper bound on the realisable pitch angles. Through equation 4.25, plotted in figure 4.7, the inflection point at $\theta = 15.3°$ bounds the realisable interaction force at $|f_n| = 5.77619$.

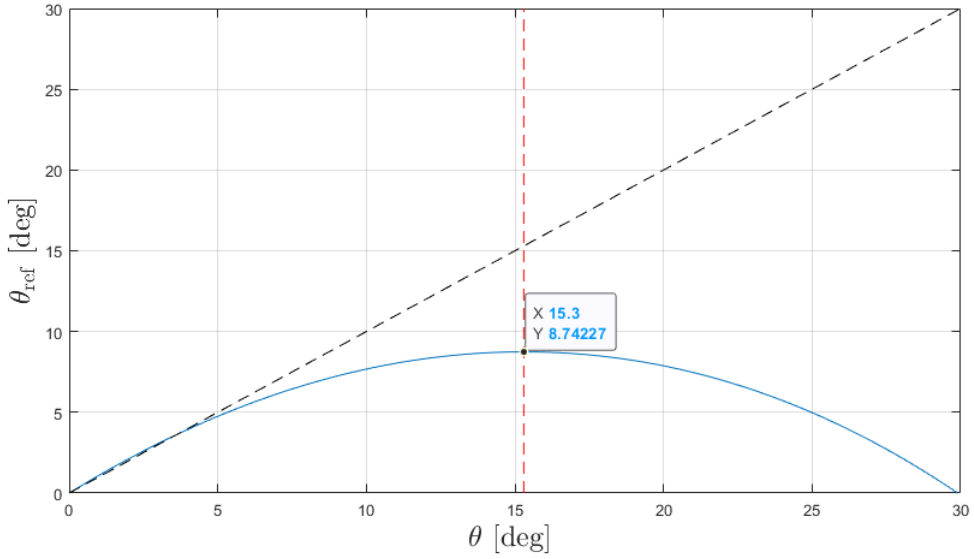


Figure 4.8: Reference pitch (in blue) required for stationary contact as a function of pitch angle θ. The dashed red line marks the inflection point at $\theta = 24.2°, \theta_{\text{ref}} = 13.1458°$. The dashed black line represents the mapping in free flight, namely the identity function. The parameters used can be found in table 3.1. Additionally, $L_m = 0.62$m.

To map the reference pitch to the reference x-position input to the position controller recall the x-position controller from equation 3.35 in section 3.3.4,

$$\theta_{\text{ref}} = u_x = k_{p,x}e_{p,x} + k_{d,x}e_{d,x} + k_{i,x}e_{i,x} \tag{4.28}$$

As before, the relevant parameters are the proportional components,

$$\theta_{\text{ref}} = k_{p,x} e_{p,x} = k_{p,x}(x_{\text{ref}} - x) \tag{4.29}$$

When in static contact, the position x of the UAV is stationary and equal to

$$x_c = x_w - L_m C_\theta \tag{4.30}$$

where x_c is the position of the UAV in contact and x_w is the x-position of the contact surface.

Once in contact with the wall, the position controller becomes a proxy for the attitude controller with a scaling of $k_{p,x}$. That is, to pass some reference pitch θ_{ref} to the attitude controller, the reference x-position error should be

$$e_{p,x} = x_{\text{ref}} - x_c = \frac{\theta_{\text{ref}}}{k_{p,x}} \tag{4.31}$$

The complete mapping of the interaction force to the position error is the combination of equations 4.25, 4.27, and 4.31. Figure 4.9 plots the mapping, computed numerically.

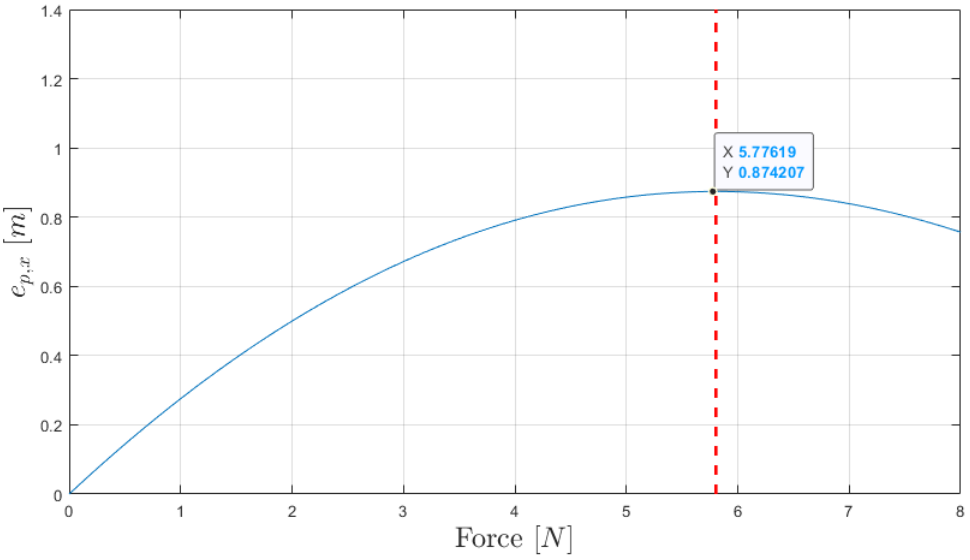


Figure 4.9: Mapping from interaction force to x-position error input, with $m = 2.2$kg. The dashed red line indicates the upper bound on the realisable interaction force.

This is effectively a mapping to x-position reference because, given some position error, it will always be possible to choose some position reference to achieve it. The map can be used to choose an x position reference to achieve a desired interaction force. In effect, this allows for the creation of a force controller as an outer loop. Letting the mapping

from interaction force to x-position error be denoted as $e_{p,x} = E(f_n)$, a force controller can then be written

$$u_f = E(f_n) + x \tag{4.32}$$

where $u_f = x_{\text{ref}}$. The mapping $E(f_n)$ does not lend itself to an analytic solution. A solution might be implemented with a lookup table of pre-computed values, or a polynomial can be used to fit the numerical data.

4.2.2 Realisable region of pitch angle and thrust force

The mapping from interaction force to x position reference was done assuming a fixed z reference. Within the operating region, the mapping traces a line, shown in figure 4.10. The line represents the pitch angles and thrust forces realisable by varying the x component reference input to the position controller in the range of the mapping.

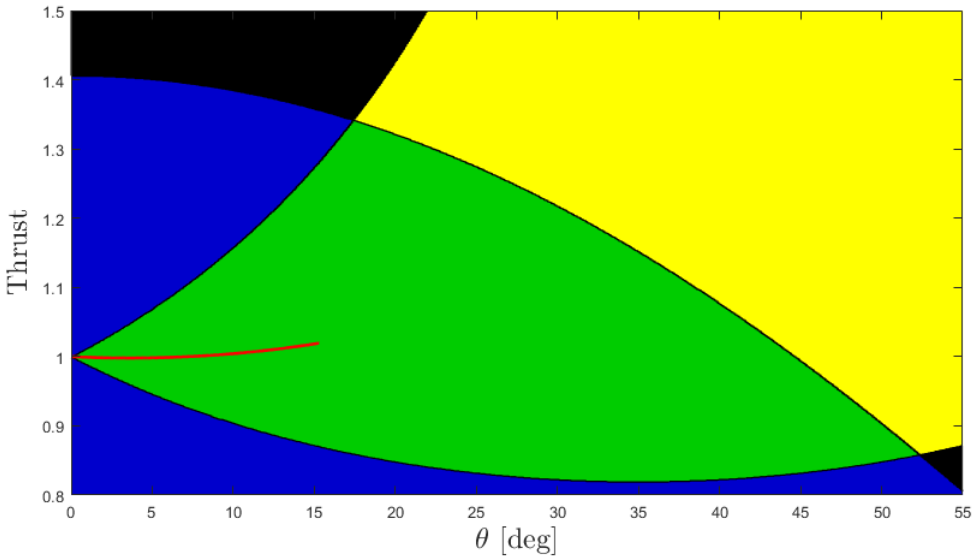


Figure 4.10: The realisable pitch angle and thrust force for a fixed z_{ref} plotted in red. The line lies within a subsection of the operating region realisable when using a cascaded PID control scheme.

More pitch angles and thrust forces can be realized by also varying the z component reference input to the position controller. When varying z_{ref} by some amount δ_z, the thrust from equation 4.23 changes to,

$$T = \frac{m}{C_\theta}(k_{p,z}(z_{\text{ref}} + \delta_z - z_{\text{com}}) + g) = \frac{mg}{C_\theta} + \frac{mk_{p,z}(\delta_z - L_m S_\theta)}{C_\theta} \tag{4.33}$$

The bound on the realisable pitch angles was found by identifying the inflection point in equation 4.27. More formally, the bound was found by solving

$$\frac{d\theta_{\text{ref}}}{d\theta} = 1 - \frac{L_m I_{yy}}{k_{p,\theta}} \left(\frac{\partial}{\partial \theta} T + mgS_\theta \right) = 0 \tag{4.34}$$

To find the entire realisable region of pitch angles and thrust, this process was repeated for thrusts T with varying δ_z. Figure 4.11 plots this realisable region in pink.

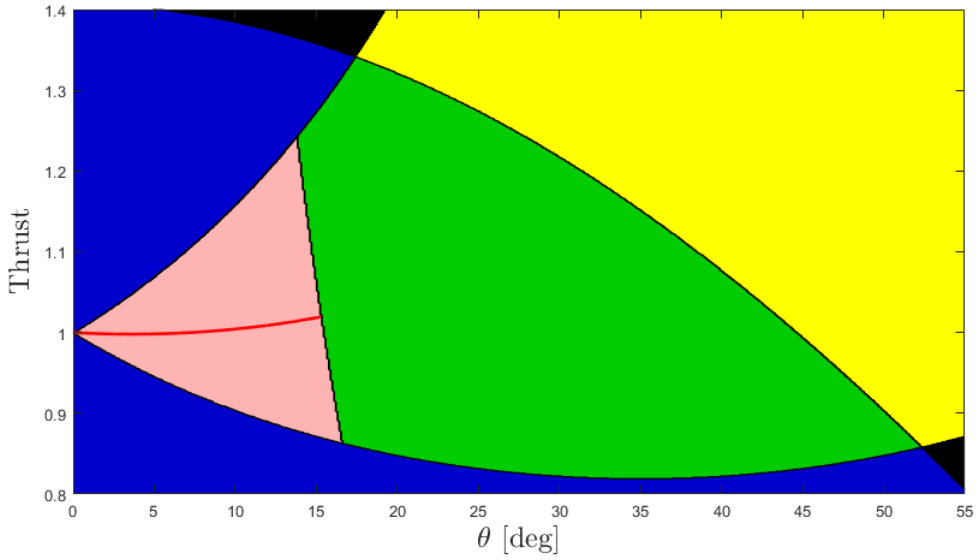


Figure 4.11: *Realisable Region:* The realisable region of pitch angle and thrust force is shown in pink. This region represents the set of achievable pitch angles and thrust forces when using a cascaded PID control scheme.

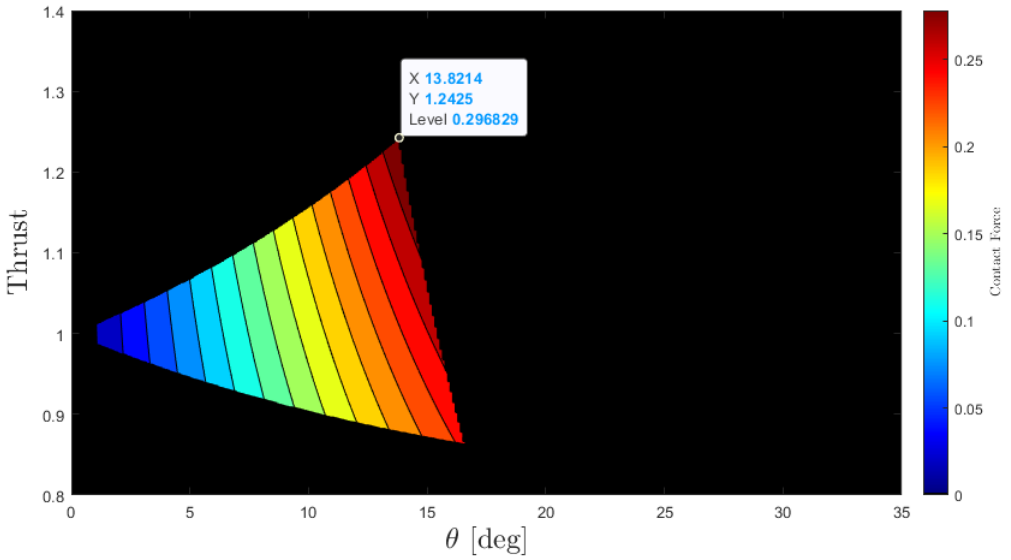


Figure 4.12: Contact force magnitude in the realisable region of pitch angle and thrust force. The maximum possible interaction force $|f_n| = 0.296829mg$ occurs at $\theta = 13.8214°, T = 1.2425mg$.

Looking closely at equation 4.34, it is evident that the location of the of the bound on the realisable region scales with manipulator length L_m and the attitude controller gain $k_{p,\theta}$. To optimize the UAV design for a large realisable region, the manipulator should be made as short as possible and the attitude controller gains should be made large.

4.2.3 Effects of PID controller gains

In this section, the mapping of the realisable region involved three PID controllers, the pitch controller, the x position controller and the z-position controller. The realisable region in figure 4.11 was generated with the assumption that the integral gains k_i of all three were held at 0. What would be the effect of non-zero integral gains?

The most interesting case is that of the pitch controller. The bound on the realisable region was found when mapping from the pitch angle to the pitch reference. Equations 4.35 and 4.36 reintroduce the integral gain to the mapping from pitch angle to pitch reference.

$$\tau_\theta^B = k_{p,\theta}e_{p,\theta} + k_{i,\theta}e_{i,\theta} = k_{p,\theta}(\theta_{\text{ref}} - \theta) + k_{i,\theta}\int_0^t e_{p,\theta} = I_{yy}L_m(T - mgC_\theta) \tag{4.35}$$

$$\theta_{\text{ref}} = \theta - \frac{L_m I_{yy}}{k_{p,\theta}}(T - mgC_\theta) - \frac{k_{i,\theta}}{k_{p,\theta}}\int_0^t e_{p,\theta} \tag{4.36}$$

The pitch error is invariably negative, which can be in figure 4.8, so the final integral term brings the pitch reference closer to the pitch angle. With a non-zero integral gain

$k_{i,\theta}$, the integral action of the controller grows over time. This has the effect of pushing the inflection point away from zero, effectively extending the range of the realisable region. This effect grows stronger with larger values of $k_{i,\theta}$.

The effect of non-zero integral gain is not so helpful for the x position controller. The position error $e_{p,x}$ is what produces the θ_{ref} required to maintain contact. The constant relationship between the proportional error and the pitch reference is what allows for a meaningful mapping between x position reference and interaction force. A non-zero integral gain interferes with this relationship and will always drive the UAV to larger and larger values for θ_{ref}.

For the altitude controller, the effect of a non-zero integral gain is not so pronounced. Where the x position controller and the pitch controller both affect the realisable pitch angle, the altitude controller controls the thrust force. From figure 4.11 it is evident that the realisable region is largely constrained by the pitch angle. Consequently the altitude controller minimally affects the realisable region.

4.3 Simulation

To verify the results of the analysis in sections 4.1 and 4.2, the UAV is simulated with the manipulator. Simulations of contact are done using the PID position controllers from chapter 3. A force controller is implemented and simulated using the map from interaction force to x position reference found in section 4.2.

In the simulation model, the rigid manipulator is modelled as a thin cylinder extending out from the center of mass with a spherical tip. The wall is a rectangular block with the contact surface located at $x = 2\text{m}$. The UAV starts at $x = 0$ and so the simulation begins with the UAV in free flight before establishing contact. The manipulator and contact implementation in Simulink is shown in figure 4.13.

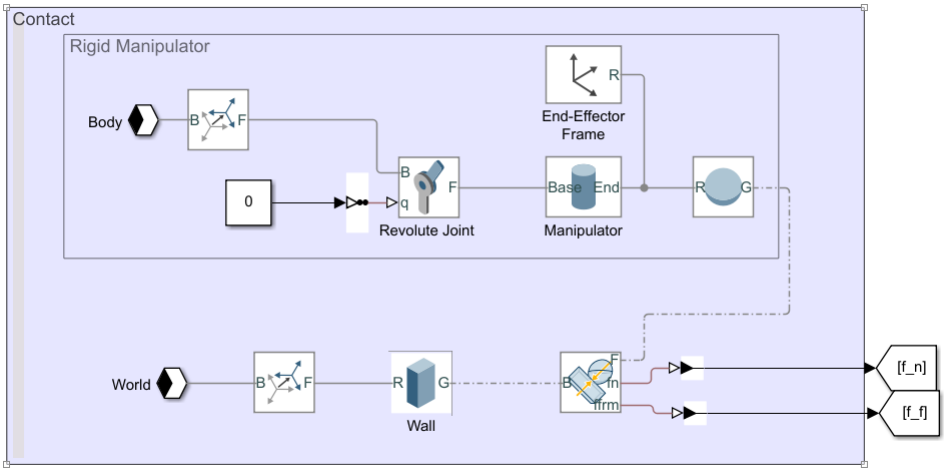


Figure 4.13: Implementation of the rigid manipulator and contact in Simulink using blocks from the Simscape Multibody Toolbox.

Simulations made use of the parameters listed in table 3.1, as well as additional parameters listed in table 4.1.

Coefficient of static friction	μ	0.7
Manipulator length	L_m	0.62 m
Manipulator to rotor length ratio	s_{mr}	2

Table 4.1: Simulation parameters.

To simulate a contact scenario, the simulated UAV was fed position and yaw references. The yaw and y position references were set to 0 and the z position reference was held at 1m. The x position reference was a step input to bring the end-effector into contact with the wall, followed by an increase of 0.1m every 5 seconds. The x and z positions and pitch angle θ of the simulated UAV are shown in figure 4.14. Note how the controlled values diverge from their reference inputs. The divergence of the pitch angle from its reference is of particular interest as it was this divergence in the model, in figure 4.8, which set the upper bound on the realisable pitch angles.

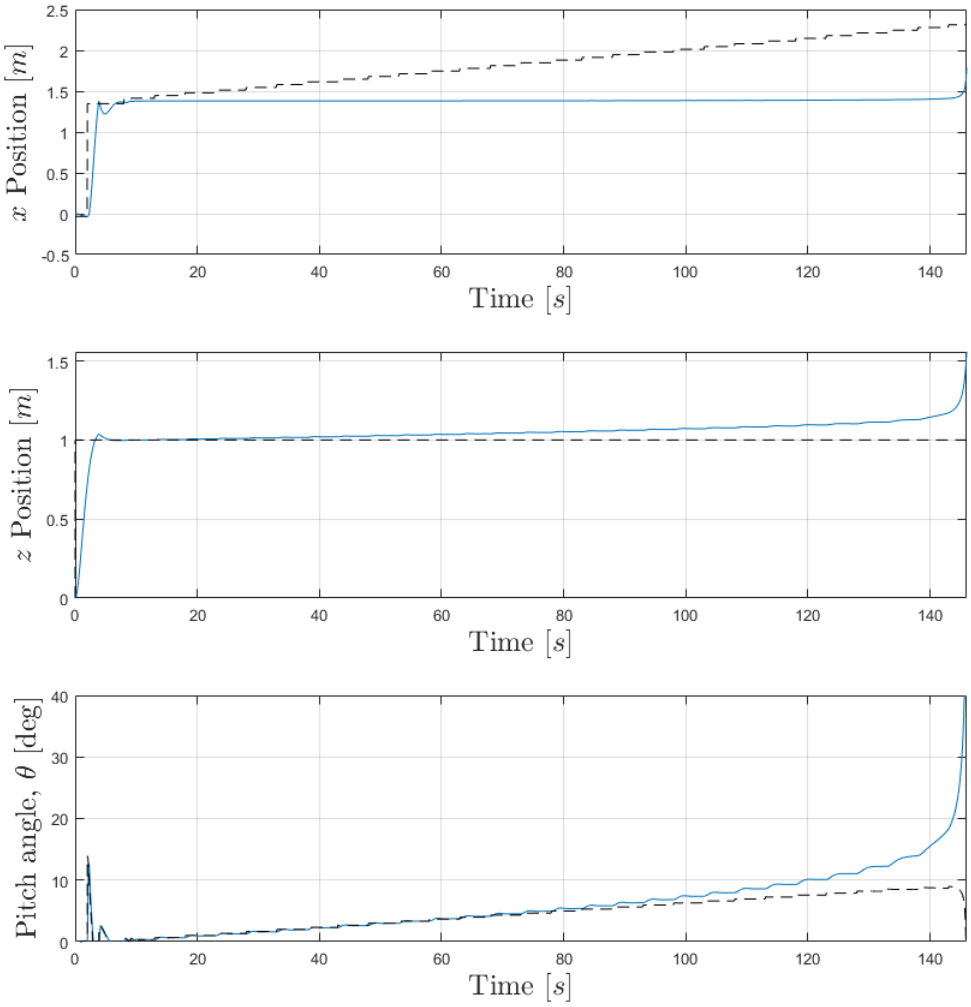


Figure 4.14: From top to bottom: x position, z position, and pitch angle θ of simulated UAV with rigid manipulator. The reference provided to the controllers is shown by the dashed black line, the real positions/angles are shown in blue.

It is evident from the plots in figure 4.14 that the UAV becomes unstable after the final step at $t = 138$s. The UAV moves beyond the pitch bound of 15.3° afterwords at $t = 139.73$s as shown in figure 4.15 by the horizontal and vertical dashed red lines.

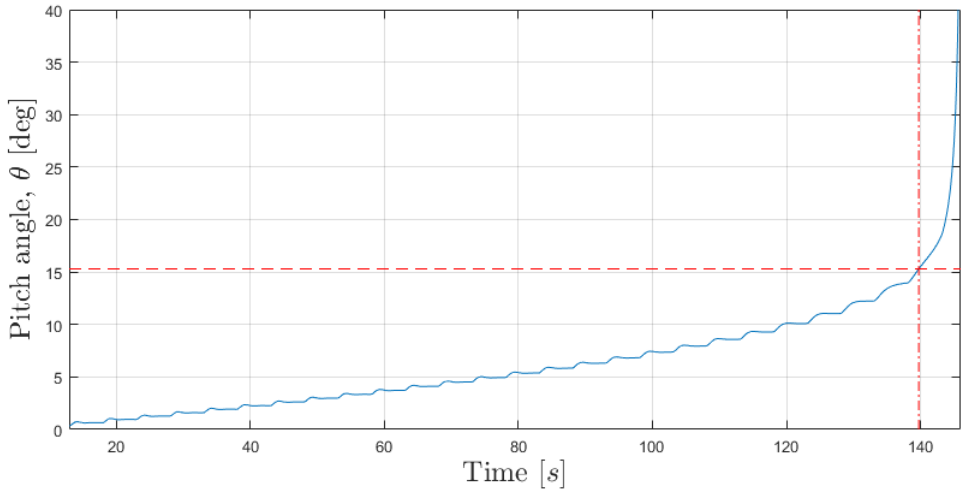


Figure 4.15: Plot of the pitch angle during simulation from figure 4.14, omitting the UAVs initial free flight. The dashed horizontal line is the inflection point from figure 4.8. The dashed vertical line is the time of occurrence.

Figure 4.16 plots the simulated pitch reference against the pitch angle from the simulation, creating a staircase shape in blue. The modelled pitch reference calculated using equation 4.27 and pitch and thrust data from the simulation is shown in orange. The rightmost 'inner' corner of each step represents the steady state for each step in θ_{ref}. For all but the final step, the steady state position lies on the orange line. On the final step, the pitch reference has moved beyond the orange line and so is outside of the realisable region. Here the UAV is not capable of maintaining a steady state.

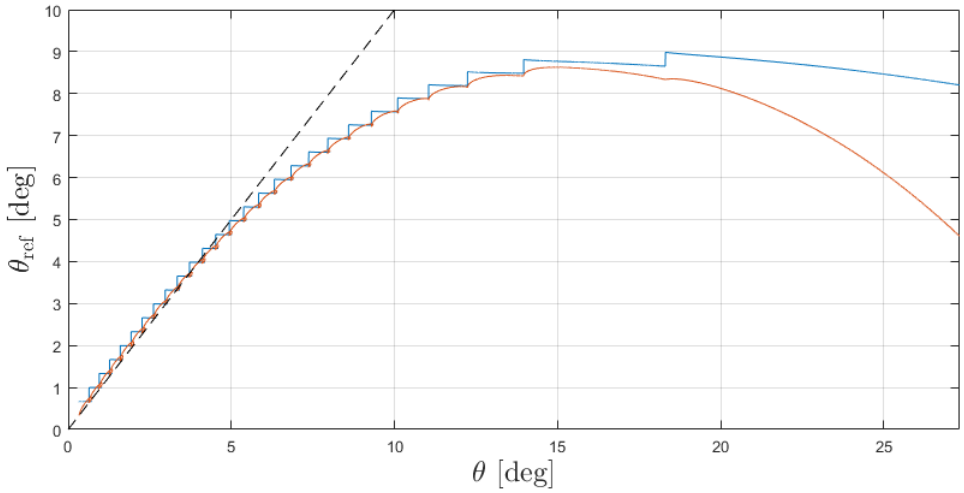


Figure 4.16: Pitch angle reference plotted against pitch angle. The blue line shows the simulation result. The orange line shows the θ_{ref} calculated from the model using the simulation θ and T. At the final step, the pitch reference and pitch angle have moved beyond the realisable region.

Figure 4.17 plots the position error $e_{p,x}$ against the generated interaction force, with the rightmost inner corner of each step once again showing the steady state. The vertical lines of the steps are caused by step changes in the input, the horizontal component comes from the UAV responding to the new input. The orange line plots the simulation data smoothed with a moving average filter and is very close to the modelled mapping in figure 4.7. It is clear from the figure that the simulated UAV is no longer responds to changes in the x-reference at $e_{p,x} = 0.85, |f_n| = 6$.

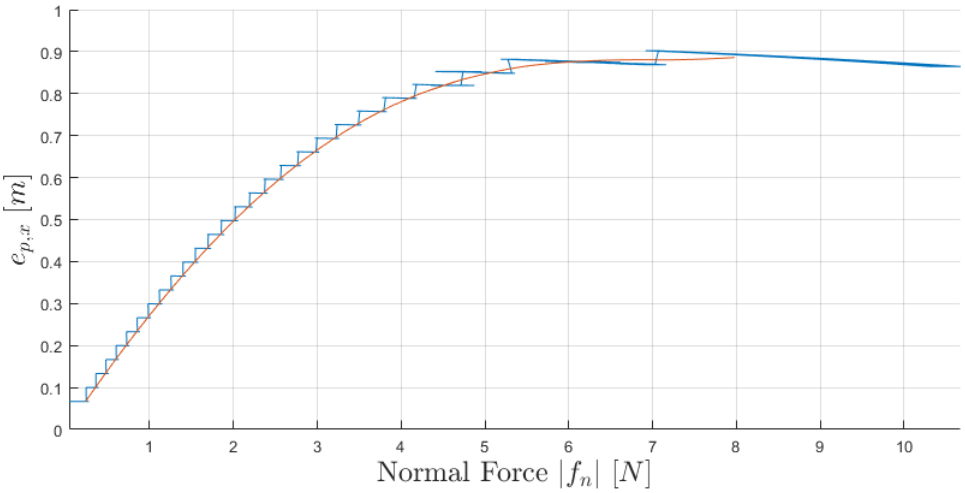


Figure 4.17: The x position error plotted against the applied interaction force.

Figure 4.18 shows the path of the UAV through the operational region. The simulated UAV initiates contact in the realisable region of pitch angle and thrust force. At $t = 139.73$s, the UAV becomes unstable as it crosses the inflection point and pitches uncontrollably towards the wall.

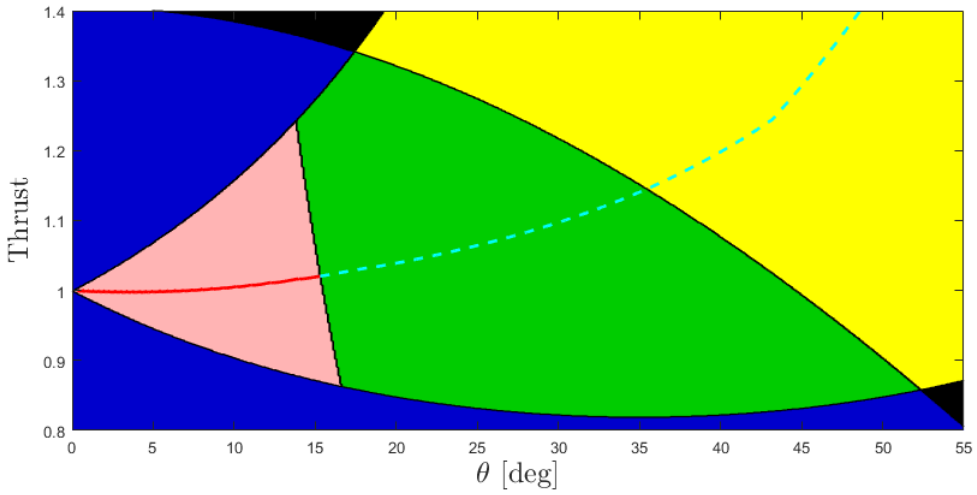


Figure 4.18: The path of the UAV, through the realisable region, in red, and the operating region, dashed cyan line.

The correspondences between the simulation and the predictions of the dynamic model in figures 4.16, 4.17 and 4.18 demonstrate that the simulation verifies the dynamic model.

Video of the simulation can be found at `https://youtu.be/YDhtoYAdNfE`.

To add the force controller from equation 4.32 to the UAV in simulation, the mapping $E(f_n)$ was implemented by fitting a third order polynomial to the simulation data in figure 4.7. The force controller was implemented such that, once the UAV was in contact with the contact surface, the computed error term would be added to the measured position. The implementation in Simulink is shown in figure 4.19.

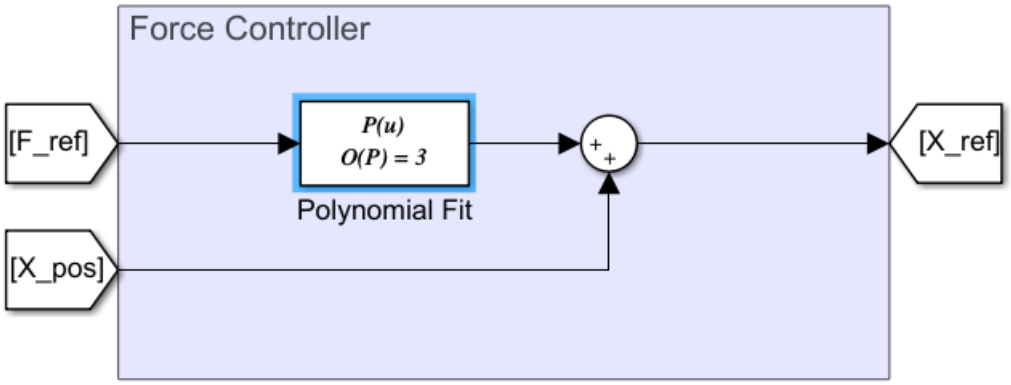


Figure 4.19: Implementation of force controller in Simulink using a fitted polynomial.

Figure 4.20 shows the results of a simulation with the force controller. The output force stops responding to the input reference at 6.3N, setting the maximum interaction force at 38.7% of the UAVs weight.

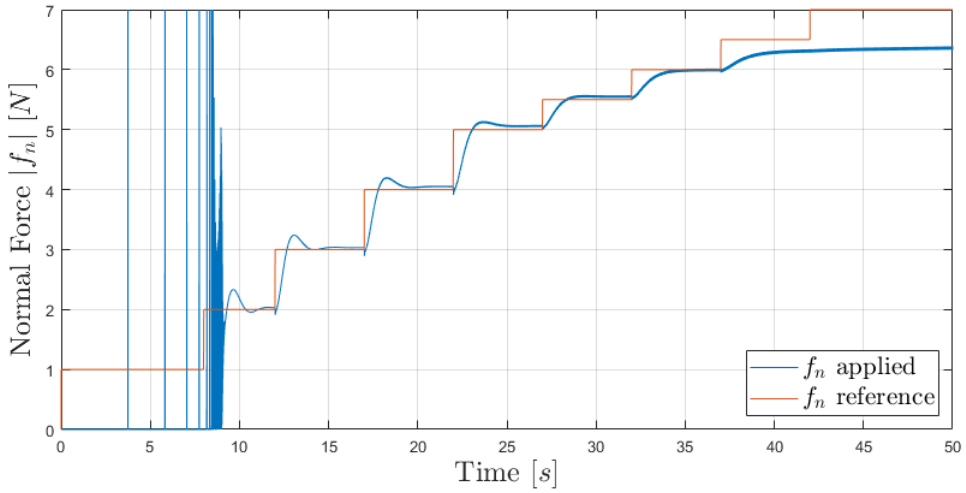


Figure 4.20: Force input, in orange, and simulated output, in blue, of UAV with force control. The output of the controller stop being responsive to the input at 6.3N which is 38.7% of the UAV's weight.

4.4 Experimentation

Experimental flight tests were conducted to verify the modelling and analysis from the previous sections. This section will present the data from one such flight test with three contact events. The experimental setup is described first, followed by presentation of

the flight data. Finally, the correspondence between the experimental data and the predictions of the model and simulations is assessed and the impacts of model assumptions are discussed.

4.4.1 Setup

The physical UAV was affixed with a narrow length of carbon fiber tube attached rigidly to the leftmost rotor arm. The test flight was conducted by setting the z position reference to 1m and the y position reference to 0 for the duration of the flight. The x position reference was stepped up until the UAV made contact with the wall and then onward until either the UAV became unstable or the manipulator slipped. In the cases where the manipulator slipped, the x reference was pulled away from the wall and another attempt was made. When the UAV became unstable, a 'kill' switch was pulled which stopped the motors to limit the damage from a crash.

The parameters of the physical UAV used for the experiments in this section are listed in table 4.2. The value for the coeffecient for static friction μ was estimated from the flight data. The manipulator length L_m was chosen to ensure ample clearance between the propellers and the contact surface.

Coefficient of static friction	μ	0.7
Manipulator length	L_m	1.05 m
Rotor length	L_r	0.31 m
Manipulator to rotor length ratio	s_{mr}	3.4
Mass	m	2.2 kg

Table 4.2: Parameters of the physical UAV used for experiments.

4.4.2 Results

Figures 4.21 and 4.22 show the x-position and pitch angle data collected from a contact flight. Three contact attempts were made, which can clearly be seen in the pitch angle data. The first two contact events ended with the manipulator slipping. The final contact event ended with the UAV pitching and yawing around the point of contact.

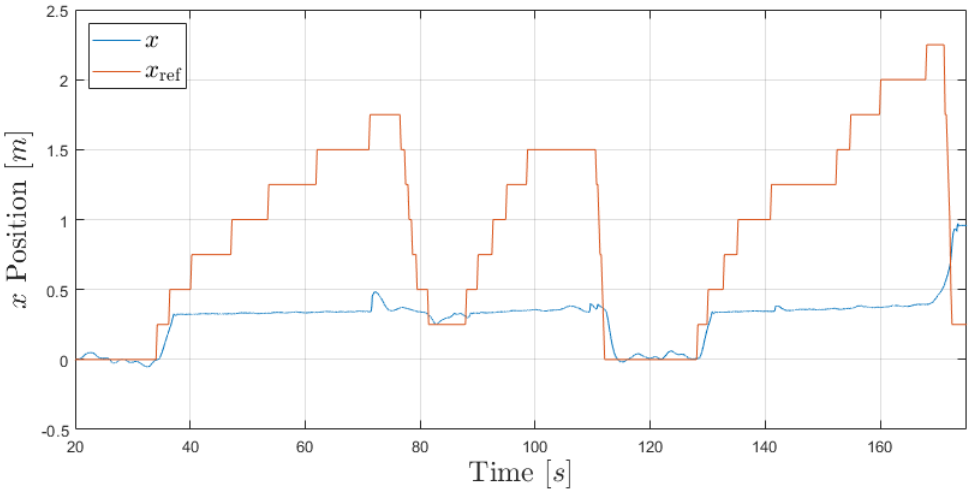


Figure 4.21: Plot of the x position of the UAV, in blue, and the x position reference, in orange.

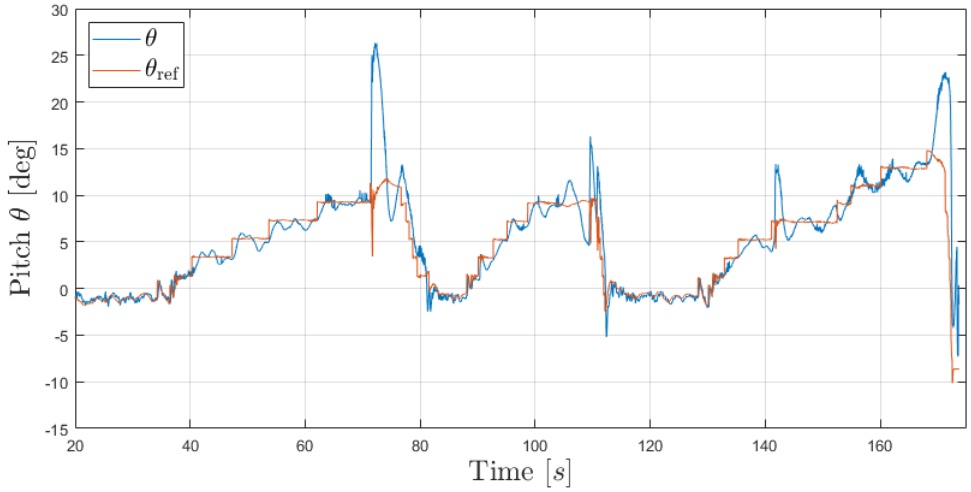
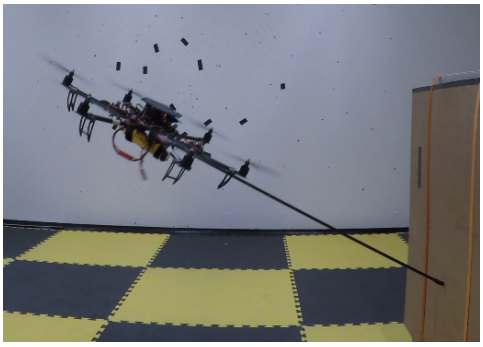
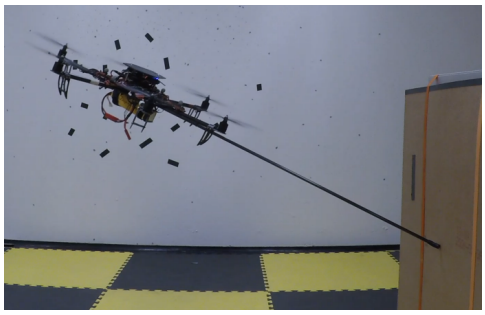


Figure 4.22: Plot of the pitch angle θ, in blue and the pitch angle reference θ_{ref}, in orange.

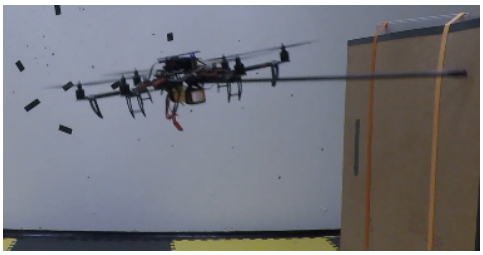
(a) UAV in contact.



(b) UAV after slipping.



(c) UAV in pitching and yawing uncontrollably.



(d) UAV after losing contact.

Figure 4.23: Images of the UAV during different stages of contact. In (a) the UAV is in stable contact shortly before the end-effector begins slipping. In (b) the UAV immediately after the end-effector has slipped. In (c) the UAV in the process of pitching uncontrollably. In (d) the UAV has lost contact with the wall as a result of uncontrollable pitching and yawing.

The parameters of the physical UAV are shown in table 4.2. Figures 4.24, 4.25, and 4.26 show the path of the UAV through the realisable region of pitch and thrust for each of the contact events. They feature a point cloud of the path with the gradient from black to red indicating the chronological order, plotted on top of the realisable region map, redrawn considering the parameters in table 4.2.

The first contact event lasted 30 seconds and concluded with the end-effector slipping down the contact surface and the UAV suddenly pitching. Figure 4.24 shows a scatter plot of the measured thrust against the measured pitch, plotted over the realisable region. The points are colored, from black to red, to represent the chronological order. The path of the UAV through the realisable region of pitch reveals why the end-effector slips. The path leads out of the operating region, falling below the friction bound, causing the end-effector to begin to slip. The force applied by friction falls as it changes from static friction to kinetic friction and the contact torque suddenly drops, leading to an imbalance between the torque generated by the UAV and the torque generated from contact. This imbalance causes the UAV to suddenly pitch, the large distances between the final points indicating speed of the change.

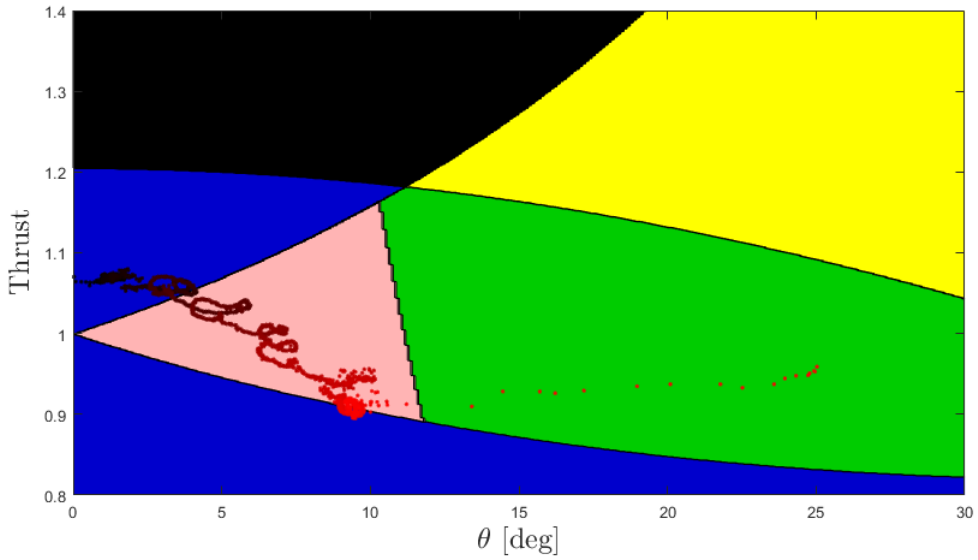


Figure 4.24: First contact event: Path of the UAV through the realisable region of pitch angle and thrust force.

The second contact event lasted for 20 seconds and once again ended with the end-effector slipping. The path of the UAV in figure 4.25 during this contact event tells a similar story. Once again the UAV falls below the operating region, leading to the end-effector slipping and a rapid increase in pitch.

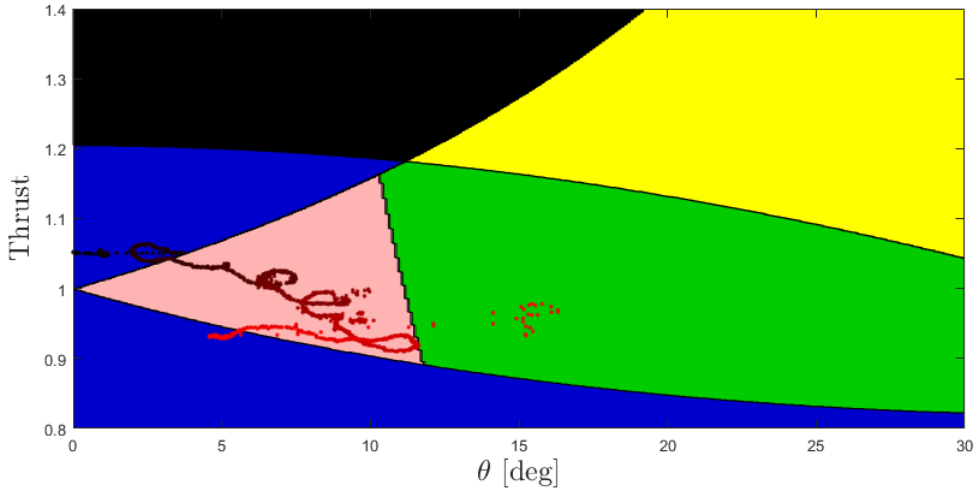


Figure 4.25: Second contact event: Path of the UAV through the realisable region of pitch angle and thrust force.

The third and final contact event of the flight had a duration of 45 seconds and concluded with the UAV pitching uncontrollably. The path of the UAV in figure 4.26 for the final

contact event goes in a different direction. The UAV remains in the operating region for the duration of the event and instead moves beyond the rightmost bound on the realisable region, where it begins to pitch uncontrollably. Along with the pitch, the UAV also begins to yaw to the left, which is not depicted in the figure.

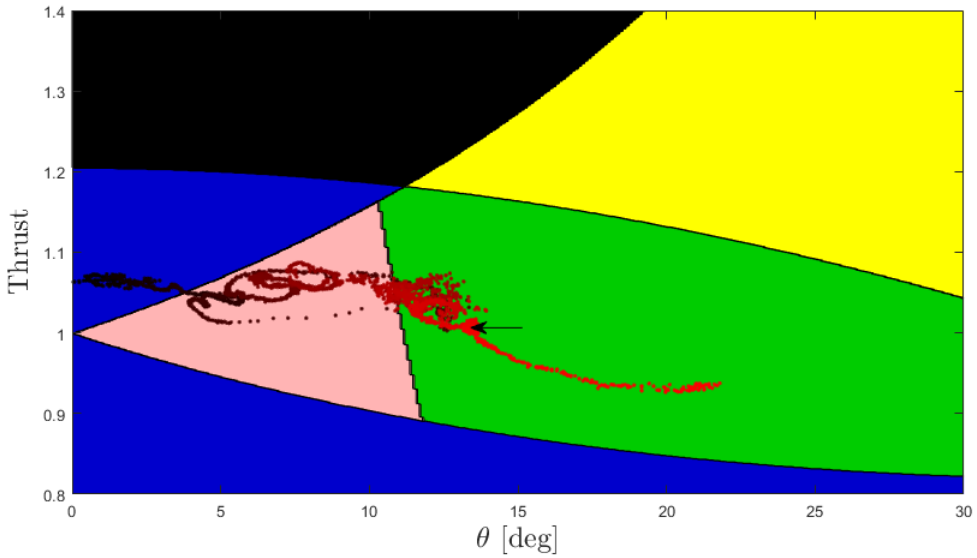


Figure 4.26: Third contact event: Path of the UAV through the realisable region of pitch angle and thrust force. The black arrow points to the apparent actual bound on the realisable region.

Figure 4.27 shows the mapping from pitch angle to pitch reference calculated using the parameters of the physical UAV. The inflection point occurs much earlier than in figure 4.8, where L_m is shorter.

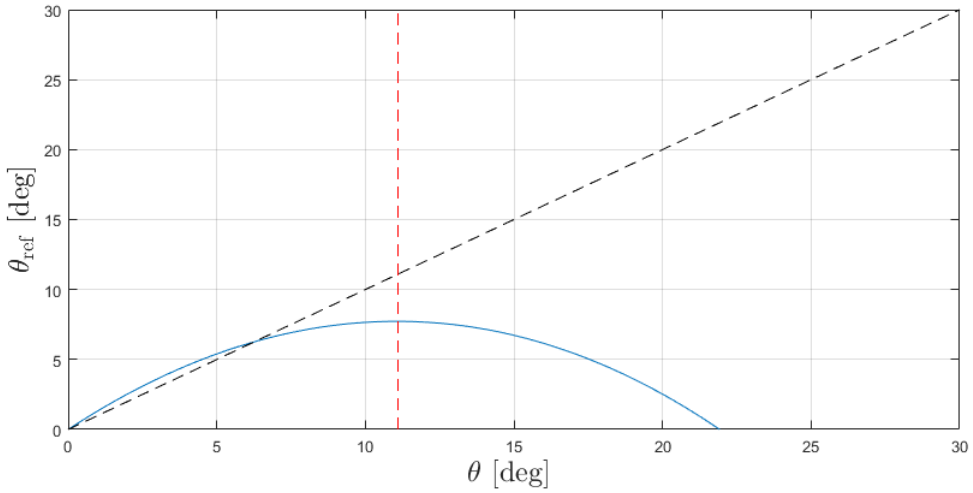


Figure 4.27: Mapping from pitch angle to pitch reference. Inflection point in red.

Figure 4.28 shows the estimated interaction force during the course of the flight test and the three contact events. The estimate is calculated using equation 4.11 with the measured orientation and the estimated thrust force. The largest interaction force achieved during the test flight was 5N, representing $0.2318mg$.

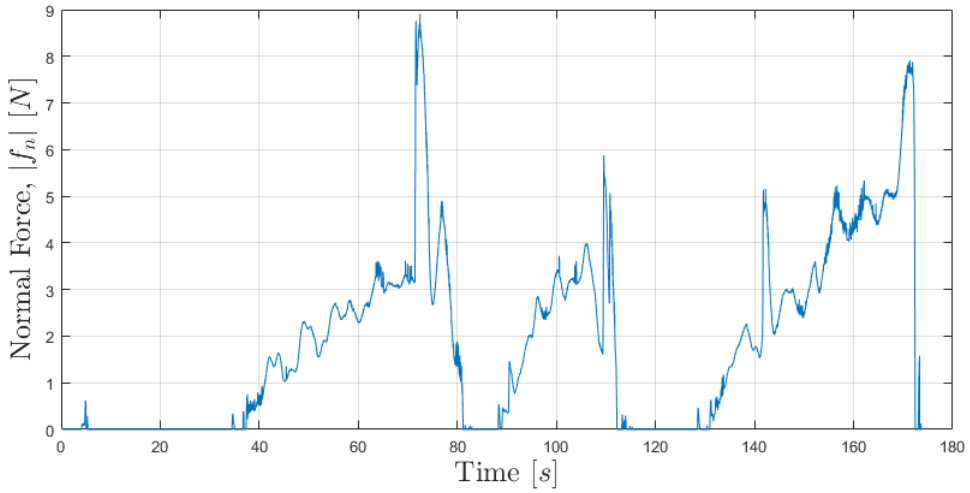


Figure 4.28: Estimated interaction force.

Video of the experiment can be found at `https://youtu.be/HGvCuHRceUw`. The flight data can be found at `https://github.com/paullassen/Flight-Data-Modelling-and-Control-of-an-Aerial-Manipulator/`.

4.4.3 Discussion

The three contact events presented in this section match the predictions of the model and simulations.

The friction bounds found in section 4.1 explain the end-effector slipping in the first two contact events. With regard to the correspondence between the friction bound and the occurrence of the end-effector slippage in the flight data, it should be noted that the coefficient of static friction μ and the resulting friction bound were estimated from the flight data. Ideally, μ should have been measured independently.

The uncontrolled pitching in the third contact event can be explained by the UAV pitching beyond the inflection point in figure 4.27 and out of the realisable region. In figure 4.26, the path of the UAV seems to pass beyond the bound of the realisable region several times without becoming unstable, indicating that the calculated pitch bound on the realisable region, in pink, is short of the apparent actual bound, indicated by the black arrow, by approximately 2°. This difference between the calculated bound and the apparent actual bound is likely due to differences between the onboard attitude controller and the proportional pitch controller considered in section 4.2. Equation 4.36 implies that the bound on the realisable region is fundamentally tied to the attitude control structure. The onboard attitude controller is a nonlinear controller which feeds angular velocity references to a PID angular rate controller [18] and so has a different behaviour than that of the PID controller considered in section 4.2, as discussed in section 3.4.3. Even if it were a PID controller, different gains would lead to different bounds of the realisable region.

Modelling the attitude controller as a PID controller is one of several simplifying assumptions made when developing the dynamic and simulation models. Other assumptions include perfect measurement, windless environment, and zero yaw and roll, which are not accurate in the actual experiments. Despite these challenges, the experiments reflect the predictions from the modelling and simulations.

4.5 1-DOF Manipulator

4.5.1 Modelling

The second manipulator configuration in this chapter is a 1-DOF Manipulator. This manipulator configuration adds a revolute joint between the center of mass and the base of the manipulator. The revolute joint rotates around the $y-$axis of the body fixed frame parameterized by the angle α. The position of the end-effector in this configuration is then given by

$$\boldsymbol{\xi}_e^B = \begin{bmatrix} C_\alpha & 0 & S_\alpha \\ 0 & 1 & 0 \\ -S_\alpha & 0 & C_\alpha \end{bmatrix} \begin{bmatrix} L_m \\ 0 \\ 0 \end{bmatrix} = L_m \begin{bmatrix} C_\alpha \\ 0 \\ -S_\alpha \end{bmatrix} \tag{4.37}$$

This configuration provides a crucial advantage over the rigidly connected manipulator in the previous section by allowing the manipulator to rotate to counter the pitching motion of the UAV. By setting $\alpha = -\theta$, and once again assuming that the roll and yaw angles are negligible, the position of the end-effector in the body and local frame become

$$\boldsymbol{\xi}_e^B = \begin{bmatrix} L_m C_\theta \\ 0 \\ L_m S_\theta \end{bmatrix} \tag{4.38}$$

$$\boldsymbol{\xi}_e^L = R_B^L \boldsymbol{\xi}_e^B = \begin{bmatrix} L_m \\ 0 \\ 0 \end{bmatrix} \tag{4.39}$$

The equation shows that this configuration allows the UAV to keep the end-effector aligned with the x-axis in the local frame. Recalling the discussion at the beginning of section 4.1, the axes of the local frame are parallel to those of the inertial frame, and their x-axis is perpendicular to the contact surface and, thereby, parallel to the normal force. This alignment limits the generation of torques from contact.

The force generated by contact in equation 4.11 is unaffected by the manipulator configuration. With the assumption that the roll and yaw angles are zero, the contact force is

$$\mathbf{f}_c^I = \begin{bmatrix} -TS_\theta \\ 0 \\ mg - TC_\theta \end{bmatrix} \tag{4.40}$$

The z component is the friction force. Once again, the analysis in this section is only valid within the static friction bound from figure 4.1.

The torques on the UAV resulting from contact are calculated by rotating the contact forces into the body frame and applying them at the end-effector, $\boldsymbol{\xi}_e^B$ in equation 4.38,

$$\boldsymbol{\tau}_c^B = S(\boldsymbol{\xi}_e^B) R_I^B \mathbf{f}_c^I = \begin{bmatrix} 0 \\ L_m(TC_\theta - mg) \\ 0 \end{bmatrix} \tag{4.41}$$

By keeping the end-effector perpendicular to the contact surface, the torque experienced by the UAV during contact can be minimized. This can most clearly be seen in the specific case of hovering thrust, $T = mg/C_\theta$, where the torque becomes zero,

$$\boldsymbol{\tau}_c^B = \begin{bmatrix} 0 \\ L_m(\frac{mgC_\theta}{C_\theta} - mg) \\ 0 \end{bmatrix} = \mathbf{0} \tag{4.42}$$

To find the limits of the UAV in contact, the counter torque is once again followed

through the motor mixer. Setting the wrench $\mathbf{w}_c$,

$$\mathbf{w}_c^B = \begin{bmatrix} T \\ -\boldsymbol{\tau}_c^B \end{bmatrix} = \begin{bmatrix} T \\ 0 \\ L_m(mg - TC_\theta) \\ 0 \end{bmatrix} \tag{4.43}$$

and passing it through the motor mixer M, the squared motor speeds $\mathbf{\Omega_c^B}$ required during contact are

$$\mathbf{\Omega}_c = M\mathbf{w}_c^B = \frac{1}{6k}\begin{bmatrix} T + \sqrt{3}s_{mr}(TC_\theta - mg) \\ T \\ T - \sqrt{3}s_{mr}(TC_\theta - mg) \\ T - \sqrt{3}s_{mr}(TC_\theta - mg) \\ T \\ T + \sqrt{3}s_{mr}(TC_\theta - mg) \end{bmatrix} \tag{4.44}$$

The motor speeds are bounded by zero and $\omega_{\max}$. Therefore the front motor pair (1st and 6th) must satisfy

$$0 \leq \frac{1}{6k}\left(T + \sqrt{3}s_{mr}(TC_\theta - mg)\right) \leq \omega_{\max} \tag{4.45}$$

and the rear motor pair (3rd and 4th) must satisfy

$$0 \leq \frac{1}{6k}\left(T - \sqrt{3}s_{mr}(TC_\theta - mg)\right) \leq \omega_{\max} \tag{4.46}$$

Solving for θ,

$$\arccos\left(\frac{mg}{T} - \frac{1}{\sqrt{3}s_{mr}} + \frac{6k\omega_{\max}}{\sqrt{3}s_{mr}T}\right) \leq \theta \leq \arccos\left(\frac{mg}{T} - \frac{1}{\sqrt{3}s_{mr}}\right) \tag{4.47}$$

$$\arccos\left(\frac{mg}{T} + \frac{1}{\sqrt{3}s_{mr}} - \frac{6k\omega_{\max}}{\sqrt{3}s_{mr}T}\right) \geq \theta \geq \arccos\left(\frac{mg}{T} + \frac{1}{\sqrt{3}s_{mr}}\right) \tag{4.48}$$

Figure 4.29 plots these bounds.

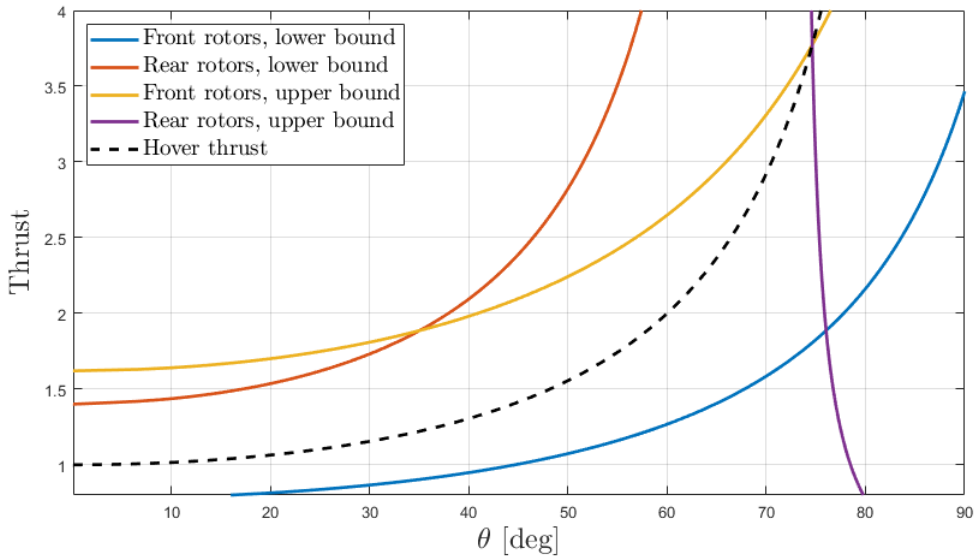


Figure 4.29: Motor bounds on pitch angle and thrust force from equations 4.47 and 4.48.

The dashed black line describes the hover thrust and thus the line of zero contact torque, per equation 4.41. The point where the upper bounds of both the front and rear motor pairs meet the dashed line of the hover thrust represents the maximum achievable thrust $T = 3.7mg$. The hover thrust separates the behaviour of the UAV into two modes, under-thrust and over-thrust. When under-thrusting, the body of the UAV begins to accelerate downward. Friction at the end-effector results in a negative torque and the UAV must generate enough counter torque to avoid tilting backwards. When the thrust is below the front rotors lower bound or the pitch exceeds the rear rotors upper bound, the front rotors can no longer contribute sufficiently to the torque and so the UAV tilts backwards and falls. When the total thrust is at more than half of the maximum, the upper bound on the rear motors takes over as the limiting factor. Conversely, when over-thrusting, the UAV must generate torque to counter the contact torque tilting it into the contact surface. When the thrust exceeds the front rotors upper bound or the rear rotors lower bound, the UAV tilts forward into the contact surface.

Figure 4.30 shows the region bounded by the inequalities depicted in figure 4.29.

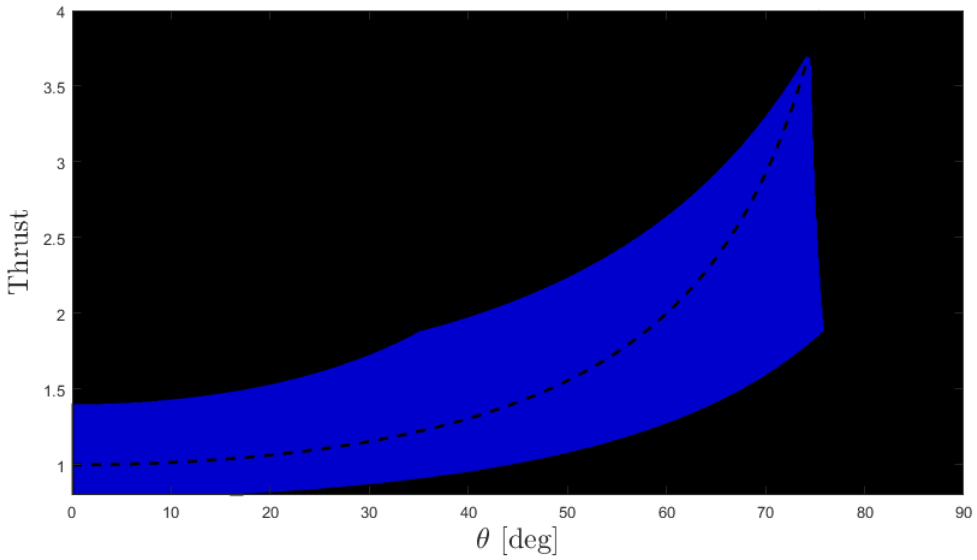


Figure 4.30: Combined motor bounds on pitch angle and thrust force. Note that this bound is only valid within the friction bound in figure 4.1.

Figure 4.31 plots the motor bounds on top of the friction bound from figure 4.1. The motor bound is only valid in the yellow region of the static friction bound. The green region represents the operating region of pitch angle and thrust force.

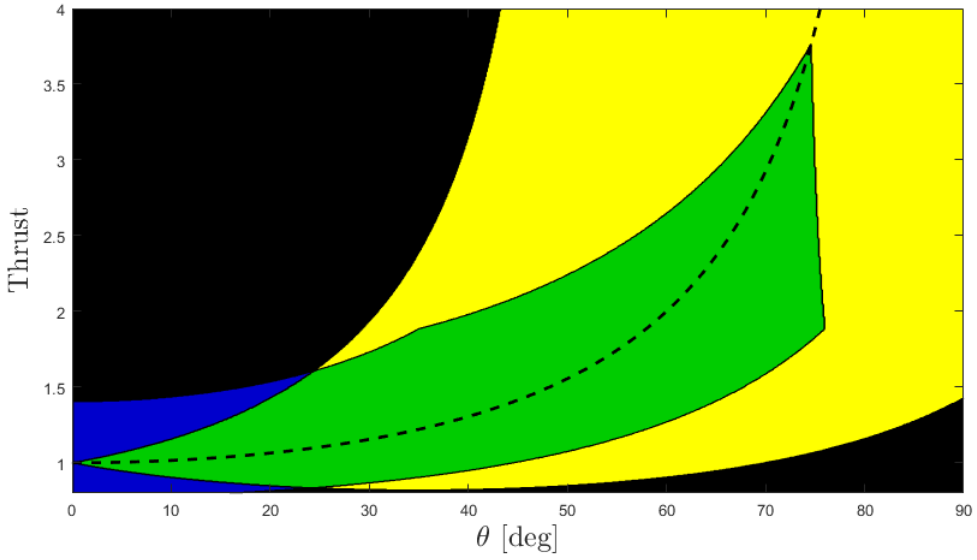


Figure 4.31: Operating region for a 1-DOF manipulator.

Figure 4.32 plots the magnitude of the contact force across the operating region.

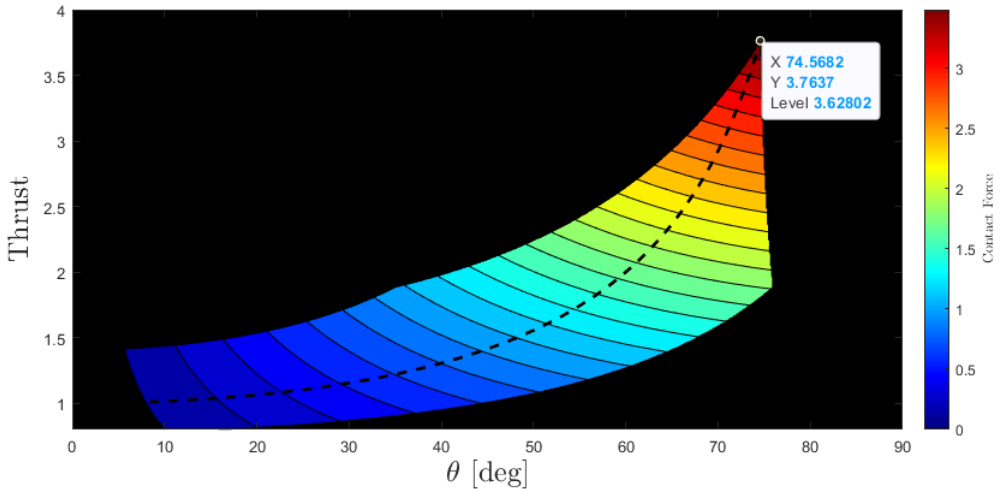


Figure 4.32: Contact force as a function of pitch angle and thrust force. Both the thrust force and contact force are scaled by mg. Note that the maximum contact force $|f_n| = 3.62802mg$ occurs at the upper right of the operating region.

Notice that the maximum contact force occurs at hover thrust with all the rotors at maximum speed. The UAV doesn't need to generate any torque because there is zero contact torque from the 1-DOF manipulator to counter at hover thrust.

The situation with the rigid manipulator in section 4.1 was very different. The contact force was always accompanied with contact torque and the UAV needed to counter with increasing torque at higher pitch angles. Consequently, the torque bound from figure 4.4 placed a pitch bound on the operating region in figure 4.5. Moreover, the analysis in section 4.2 found that the realisable region, achieved using the controllers, was limited by a smaller realisable pitch bound. The realisable pitch angle is bound by the inflection point of the mapping from pitch angle θ to reference pitch θ_{ref} in figure 4.8, the point where the attitude controller stops being responsive to increases in the reference pitch. The divergence of the reference pitch from the pitch angle is entirely driven by the torque from contact, as the mapping is derived from equation 4.27 which relates the pitch error to the contact torque.

For the 1-DOF manipulator, the only meaningful torques appear far from the hover thrust, which is not useful for contact. At hover thrust, $T = mg/C_\theta$, the contact torque is zero and the mapping from pitch angle to pitch reference for the 1-DOF becomes

$$\theta_{\text{ref}} = \theta - \frac{L_m I_{yy}(mg - TC_\theta)}{k_{p,\theta}} = \theta \tag{4.49}$$

and the pitch angle is perfectly responsive to reference pitch. Consequently, the entire operating region is realisable and the force controller is simpler to derive than for the rigid manipulator in section 4.2 and can be done analytically.

As in section 4.2, the force controller is derived from the mapping from interaction force to x position error. Taking the normal force from equation 4.40 and assuming hover thrust,

$$f_n = TS_\theta = \frac{mg}{C_\theta} S_\theta = mgT_\theta \tag{4.50}$$

Rearranging this to solve for θ and considering the position controller, like in equation 4.29,

$$\theta = \arctan\left(\frac{f_n}{mg}\right) = k_{p,x} e_{p,x} \tag{4.51}$$

Finally, solving for $e_{p,x}$

$$e_{p,x} = \frac{1}{k_{p,x}} \arctan\left(\frac{f_n}{mg}\right) \tag{4.52}$$

Figure 4.33 plots the mapping from contact force to x position error. Note that the mapping increases monotonically in contrast to the corresponding mapping for the rigid manipulator in figure 4.7.

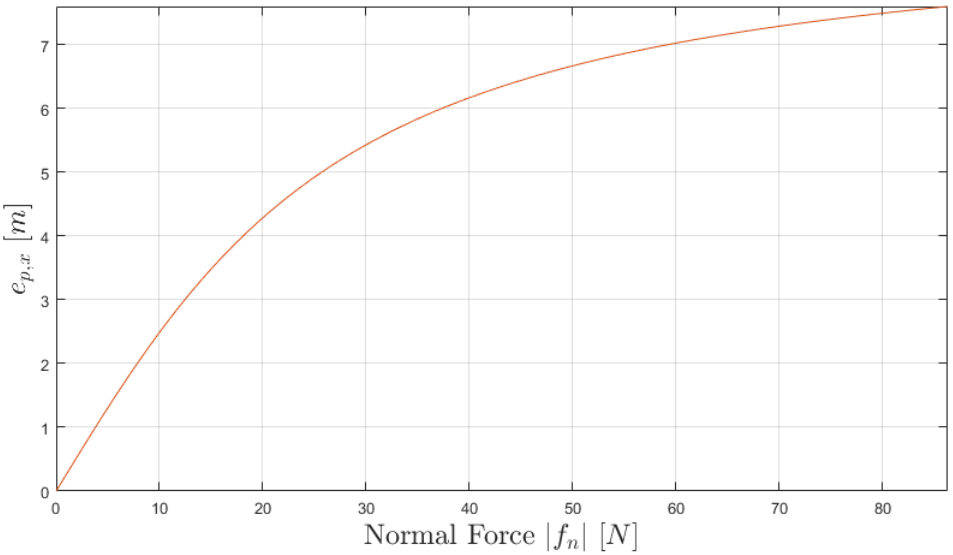


Figure 4.33: Mapping from interaction force to x position error. Note that $e_{p,x}$ increases with the force over the entire range.

The force controller described by this mapping is then

$$u_f = \frac{1}{k_{p,x}} \arctan\left(\frac{f_n}{mg}\right) + x \tag{4.53}$$

where $x_{\text{ref}} = u_f$.

4.5.2 Simulation

To verify the results of the analysis in section 4.5.1, the UAV is simulated with the 1-DOF manipulator.

In the simulation model, the 1-DOF manipulator is modelled as a thin cylinder extending out from the center of mass with a spherical tip connected to the center of mass through a revolute joint. The wall is a rectangular block with the contact surface located at $x = 2$m. The UAV starts at $x = 0$ and so the simulation begins with the UAV in free flight before establishing contact. The manipulator and contact implementation in Simulink is shown in figure 4.34.

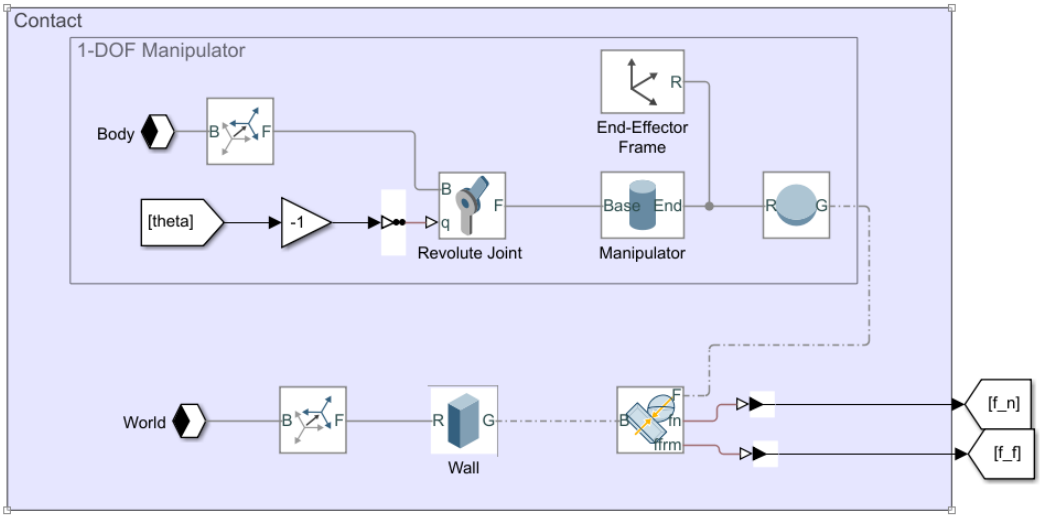


Figure 4.34: Implementation of the 1-DOF manipulator in Simulink using the Simscape Multibody Toolbox.

To simulate a contact scenario, the simulated UAV was fed position and yaw references. The yaw and y position references were set to 0 and the z position reference was held at 1m. The x position reference was a step input to bring the end-effector into contact with the wall, followed by an increase of 0.1m every 5 seconds. The simulation was aborted at $t = 87$s when the UAV crashed. The x and z positions and pitch angle θ of the simulated UAV are shown in figure 4.35.

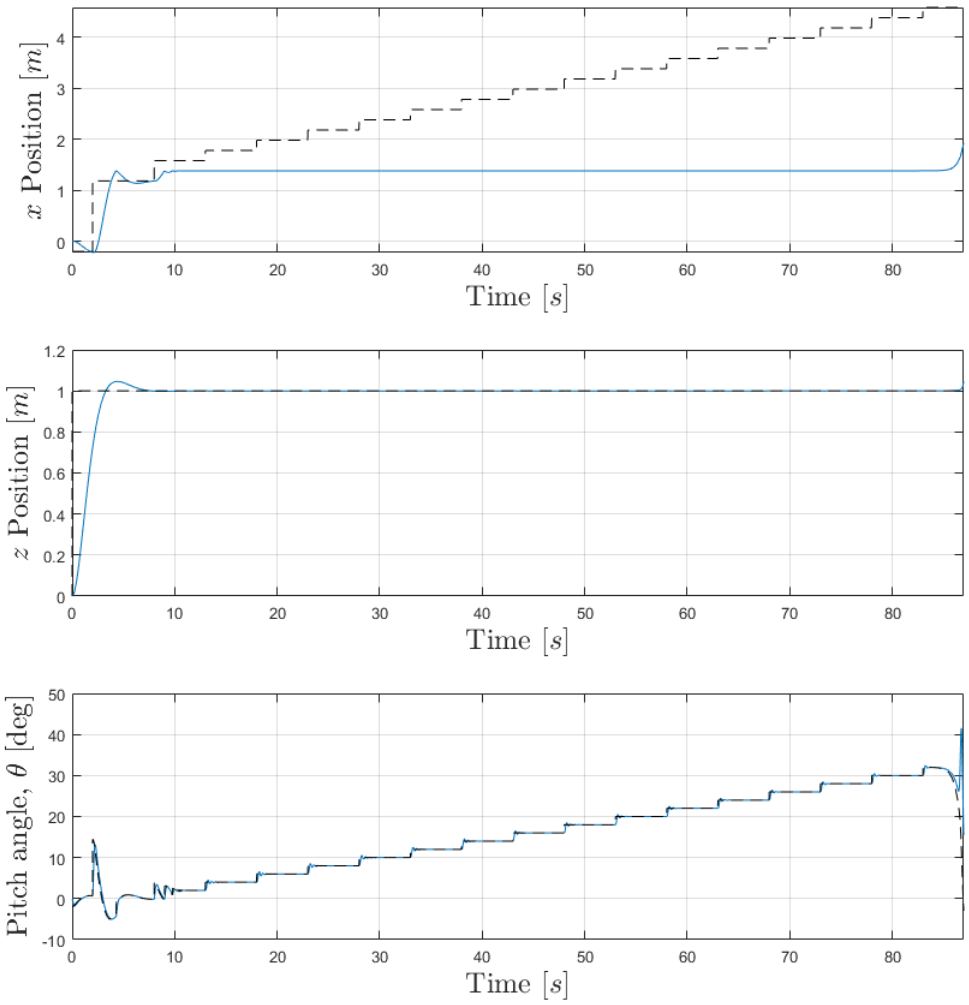


Figure 4.35: From top to bottom: x position, z position and pitch angle θ of simulated UAV with 1-DOF manipulator. The reference provided to the controllers is shown by the black dashed line, the real positions and angles are shown in blue. The pitch angle exactly matches the pitch reference and does not exhibit the divergence seen in the case of the rigid manipulator in figure 4.14.

The simulated UAV is able to achieve a pitch angle of 30° before yawing uncontrollably and ultimately crashing. Figure 4.36 plots the pitch, roll and yaw of the UAV in the last seconds of the simulation. Notice that the roll and yaw angles become non-zero in the last 5 seconds of the simulation.

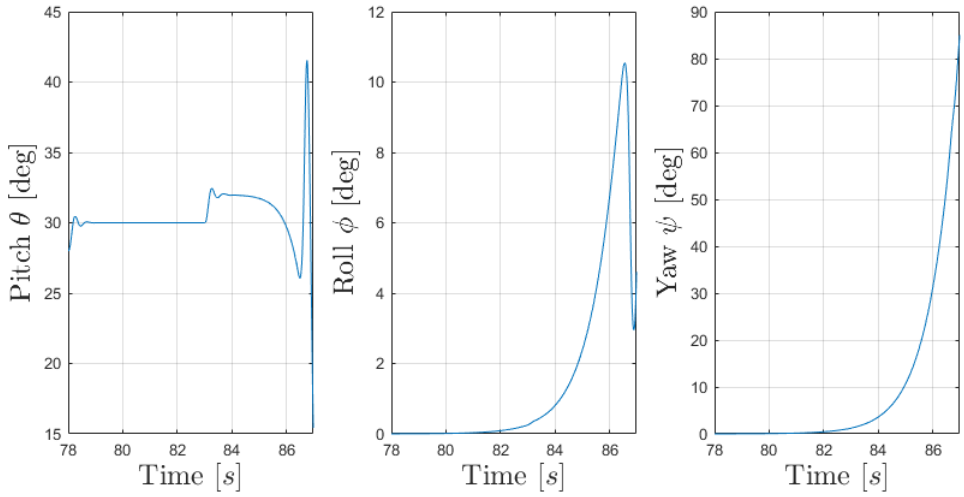


Figure 4.36: From top to botttom: pitch angle θ, roll angle ϕ, and yaw angle ψ during the last 10s of the simulation.

Figure 4.37 plots the relationship between the pitch angle and the pitch reference. This relationship follows the identity closely, as suggested by equation 4.49.

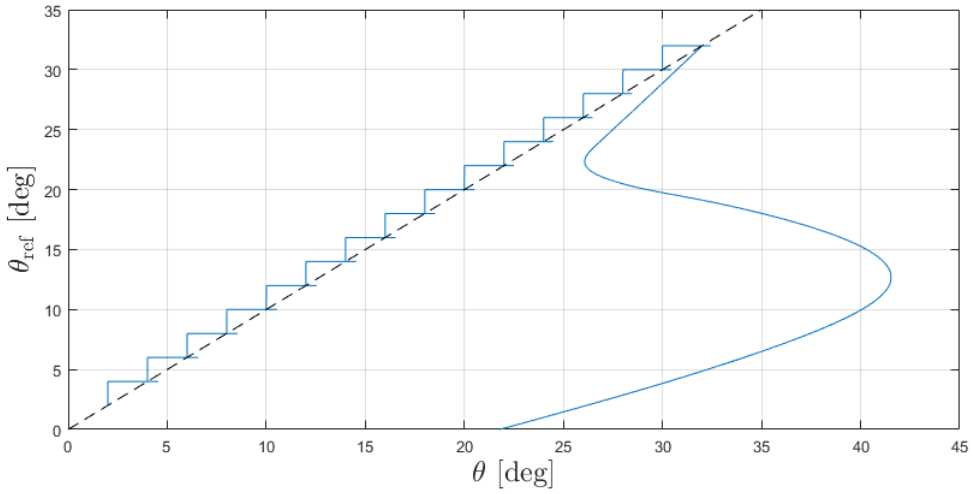


Figure 4.37: Reference pitch plotted against the pitch angle. The blue line shows the simulation result. The dashed black line shows the identity, where $\theta_{\text{ref}} = \theta$. The inner corner of each step represents the steady state, which fall along the identity. The long tail which falls below the dashed black line represents the last 5s of the simulation.

Figure 4.38 shows the relationship between the interaction force and the x position error. Up until the UAV loses control, it follows the mapping in equation 4.33 perfectly.

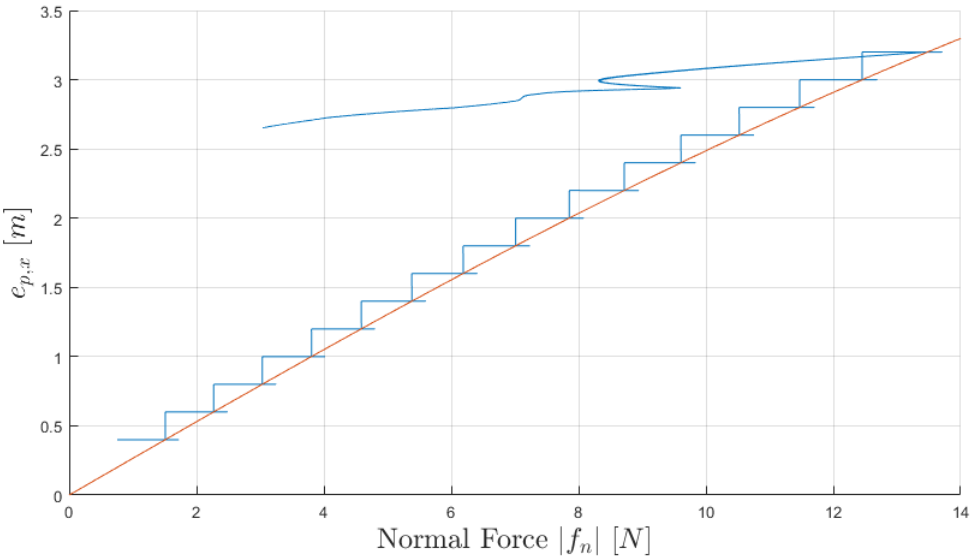


Figure 4.38: The applied interaction force plotted against the x position error, in blue. The orange line shows the mapping from interaction force to x position error as calculated in equation 4.52. The line that travels to the left from the final step represents the last 5s of the simulation.

Up until the UAV loses stability and crashes, it performs as predicted in the model in section 4.5.1. The pitch angle follows the pitch reference throughout the simulation. The failure exhibited in the crash was not captured in the model. The model was developed around the assumption of zero roll and yaw. Figure 4.36 shows that these assumptions were violated in the last 5s of the simulation. While the roll and yaw are zero at $t = 0$, inspection of the simulation data reveals that they become infinitesimally non-zero already after the first time step, $t = 50\mu$s. These slight errors compound over the course of the simulation until they reach a visible order of magnitude at $t = 82$s. While this is technically a small simulation error, a real UAV in flight would also experience errors from a variety of sources.

The crash illustrates a weakness in the model, namely that it does not assess the roll or yaw stability of the UAV in contact.

Video of the simulation can be found at `https://youtu.be/W1jwW-kKRy8`.

4.6 Discussion

In this chapter the dynamic model of a UAV in free flight was extended to describe the dynamics of contact, including the contact forces and torques experienced by a UAV with a mounted manipulator. For a UAV with a rigidly attached manipulator arm, an operating region of pitch angle and thrust force based on the physical parameters of the

UAV was identified. A smaller realisable subregion was identified by analysing the flight controllers. A force controller outer loop was designed and implemented in simulation. Simulations were done to validate the modelling. Experiments were conducted with a physical UAV mounted with a rigid manipulator, further verifying the modelling results. The analyses of the rigid manipulator reveal that the primary UAV design parameter affecting the applicable interaction force is the mass of the UAV, which should be maximized. Another important design parameter is the manipulator length, which should be minimized. The applied interaction force is proportional to the mass, roughly equal to mgS_θ, per the discussion in section 4.1. Moreover, a shorter manipulator arm leads to a larger realisable pitch θ and thus a larger applied interaction force. Attitude controller design also influenced the maximum force application of the rigid manipulators. The bound on the realisable region was found to scale with the proportional gain $k_{p,\theta}$ of the pitch controller.

A similar process was repeated for an actuated manipulator with one degree of freedom. Analysis of the realisable and operating regions of both of the manipulators show that the 1-DOF manipulator configuration is the most suited to contact. Its ability to decouple the contact torque from the UAV pitch angle allows the UAV to reach larger pitch angles and thereby achieve larger interaction forces. This is not, however, the full story. The results of the simulation in section 4.5.2 indicate that the UAV is sensitive to yaw and roll, which were omitted from the dynamic model, by holding them at zero.

The experiments done in this chapter were of limited scope as explained in chapter 1. In particular, the 1-DOF manipulator was never physically implemented and therefore lacks experimental validation. Further experimentation would have included varying the thrust T to map the entire realisable region experimentally. Additionally, flight tests with varying manipulator lengths L_m would verify the model predictions of the effects of this UAV design parameter. The force output of the UAV in the experiments was estimated from the pitch and thrust data, rather than by measurement, which would have provided direct verification.

Future work would be to extend the modelling of the realisable region to other attitude control schemes, in particular to the non-linear attitude controller found on the PX4 flight controller. Moreover, to complete the description of the total applicable wrench, the analysis in section 4.1 could be extended to consider the application of torque at the point of contact. This would extend the task space of the UAV to the real world surface contact applications motivating this thesis, including contact inspection, polishing, and drilling.

CHAPTER 5

Conclusion

This thesis has evaluated the capabilities and limitations of drone control strategies for aerial manipulators, motivated by real world surface contact applications such as contact inspection, drilling, and polishing.

First, related research was surveyed, covering controller design, manipulator design, and customized UAV designs for aerial manipulation. Previous studies omitted analyses of the capabilities of the simplest manipulators and standard flight control schemes. This thesis filled this gap.

The standard flight controller chosen for this thesis was the PixHawk 4, which is a PX4 flight controller. A coplanar hexarotor was built around it. Two simple manipulator configurations were evaluated, a rigid manipulator arm and a 1-DOF manipulator arm.

The capabilities and limitations of the chosen UAV platform and manipulators were evaluated through dynamic modelling, simulations, and experimentation. Evaluation of a UAV in free flight was done first, to lay the groundwork, in chapter 3. This was then extended, in chapter 4, to include contact analysis.

The dynamic model was developed from the Newton-Euler formulation of the laws of motion, first in free flight and then with a manipulator in contact with a surface. A motor mixer to control the rotor speeds to produce a desired wrench was derived.

The simulation model was developed in MATLAB and Simulink, with the Simscape Multibody Toolbox, modeling the evolution of the UAV's state under the influence of the forces and torques acting on it. Moreover, a cascaded control scheme, with a PID attitude controller inner loop and a PID position controller outer loop, was implemented in simulation.

The physical UAV was constructed and the experimental setup, with the MOCAP system and the ground station, was built. A command line interface was developed to control the UAV over a ROS network. Using ROS to receive measurements from the MOCAP system and targets from the CLI, and MAVSDK to send commands to the Pixhawk 4, the PID position controller was implemented on the UAV flight computer. A method to compensate for the motor voltage dropping as the battery discharged was developed and included in the position controller implementation.

A test rig was designed, printed, and assembled to collect the data required to tune the PID attitude controller gains in simulation.

The capabilities of the hexarotor platform with a simple manipulator were evaluated

in terms of pitch angles and thrust forces. The goal of static contact is to apply an interaction force to a surface while stationary, confining the pitch angle and thrust force within the limits of static friction. An operating region was identified based on the physical limitations of the platform, specifically the limitations of the motors, which translate to a limit on the achievable reaction torque. This is particularly pronounced for a UAV with a rigid manipulator, as the nature of contact results in torque. For a UAV with a 1-DOF manipulator, much of this torque can be eliminated by actuation and so the operating region extends to larger pitch angles and thrust forces where the interaction force is larger.

A force controller was developed by tracing the contact force back through the cascaded controller to create a mapping from the interaction force to position controller input. This analysis revealed that, for the rigid manipulator, not all of the operating region could be reached, but only a subregion was realisable using the controller. The attitude controller proved to be the limiting factor, specifically the proportional gain of the pitch controller.

The force controller was implemented and validated in simulation. Simulations of the UAV with each of the two manipulators were conducted. For the rigid manipulator, the simulated UAV became unstable outside the realisable region, validating the analysis. For the 1-DOF manipulator, the UAV failed unexpectedly, revealing its sensitivity to yaw and roll, which were held at zero in the dynamic model.

Experiments were conducted with a physical UAV mounted with a rigid manipulator, further verifying the modelling results.

Two UAV design parameters were identified as key to maximizing the magnitude of the interaction force. The primary design parameter is the mass of the UAV. Increasing the mass of the UAV increases the interaction force which can be applied. The other key design parameter is the manipulator length, particularly in the case of the rigid manipulator. The shorter the manipulator, the smaller the contact torques experienced by the UAV, leading to larger realisable pitch angles.

Attitude controller design also influenced the maximum force application of the rigid manipulators. The bound on the realisable region was found to scale with the proportional gain $k_{p,\theta}$ of the pitch controller.

This thesis has met the objectives stated in chapter 1. The capabilities of a standard UAV configuration were evaluated by modelling, simulation, and experimentation. The dynamic models and experiences from the simulations and flight tests suggested recommendations for designing simple manipulators for coplanar multirotors.

APPENDIX A

Source Code

Two programs are presented in this appendix, the C++ program implementing the position controller on the Raspberry Pi 3+ and a Python script for communicating with the flight computer from the control computer using a command line interface, running Ubuntu 18.04. Both the C++ program and the Python script were included in a single ROS package.

Two launch files are included in this package. The first, startup.launch, is started on the control computer and used to initialize the environment. It begins broadcasting data from the MOCAP system and starts drone_ui.py containing the CLI.

The second launch file, fly.launch, is started on the Raspberry Pi flight computer and launches the flight code. The flight code consists of uav_monitor.h, uav_monitor.cpp, mavsdk_helper.h, mavsdk_helper.cpp and ros_interface.cpp.

- uav_monitor.h provides a description of the UavMonitor class, which implements the position controller and ROS subscriber callbacks, and the description and implementation of the Triplet class, which provides convenient arithmetic for the many triplets found in the program.
- uav_monitor.cpp provides the implementation of the UavMonitor class.
- mavsdk_helper.h and mavsdk_helper.cpp describe and implement many useful helper functions for interfacing with the Pixhawk 4 over MAVlink.
- ros_interface.cpp is the main file. It initializes the MAVlink connection with the Pixhawk 4, the ROS connection and the UavMonitor instance. It also handles the broadcasting of data from the UAV.

It should be noted the mavsdk_helper.h and mavsdk_helper.cpp are adapted from one of the MAVSDK example programs. ros_interface also makes use of this example code. The example code can be found at: `https://github.com/mavlink/MAVSDK/blob/main/examples/offboard_velocity/offboard_velocity.cpp`

The rest of this appendix is dedicated to the raw source code.

The structure of the ROS package is:

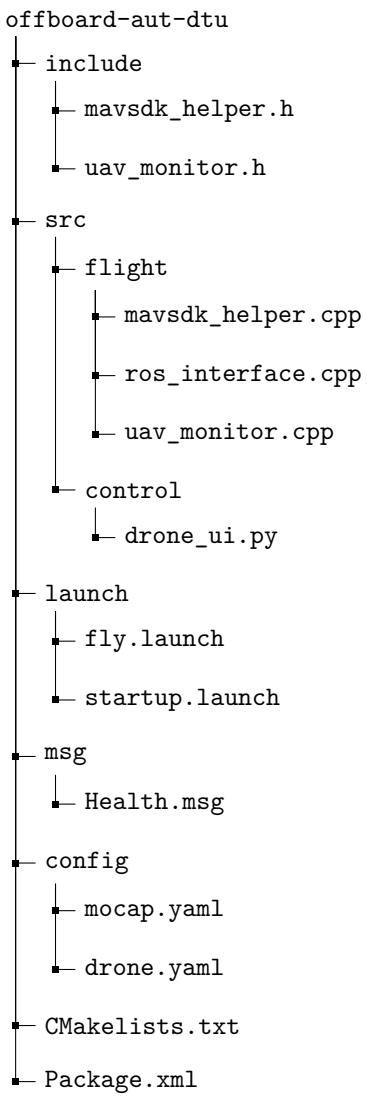


Figure A.1: ROS package structure.

List of source codes

```cpp
#ifndef UAV_MONITOR_H
#define UAV_MONITOR_H

#include <geometry_msgs/Point.h>
#include <geometry_msgs/PoseStamped.h>
#include <offboard/Health.h>
#include <ros/ros.h>
#include <std_msgs/Bool.h>
#include <std_msgs/Float32.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <tf2_ros/buffer.h>
#include <tf2_ros/transform_listener.h>

#include <semaphore.h>
#include <chrono>
#include <cmath>
#include <future>
#include <iostream>
#include <thread>

#include <mavsdk/mavsdk.h>
#include <mavsdk/plugins/action/action.h>
#include <mavsdk/plugins/info/info.h>
#include <mavsdk/plugins/offboard/offboard.h>
#include <mavsdk/plugins/telemetry/telemetry.h>
#include "mavsdk_helper.h"

#define LIST_SIZE 10
#define MANIPULATOR_MAXIMUM 605
#define MANIPULATOR_MINIMUM 405

struct param_struct {
        float mass;
        float length;
        int mode;
};

template <class T>
class Triplet {
 private:
  T x;
  T y;
  T z;

 public:
  Triplet() : x(0), y(0), z(0) {}
  Triplet(T w_) : x(w_), y(w_), z(w_) {}
  Triplet(T x_, T y_, T z_) : x(x_), y(y_), z(z_) {}
  Triplet(const Triplet<float>& obj)
      : x(obj.get_x()), y(obj.get_y()), z(obj.get_z()) {}
  Triplet(const Triplet<double>& obj)
      : x(obj.get_x()), y(obj.get_y()), z(obj.get_z()) {}
  Triplet(const Triplet<int>& obj)
      : x(obj.get_x()), y(obj.get_y()), z(obj.get_z()) {}

  void set_x(T x_) { x = x_; }
  void set_y(T y_) { y = y_; }
  void set_z(T z_) { z = z_; }

  void set(T x_, T y_, T z_) {
    set_x(x_);
    set_y(y_);
    set_z(z_);
  }

  void set(const Triplet<T>& obj) {
    set_x(obj.get_x());
    set_y(obj.get_y());
    set_z(obj.get_z());
```

```
70    }
71
72    void set(const geometry_msgs::Point& obj) {
73      set_x(obj.x);
74      set_y(obj.y);
75      set_z(obj.z);
76    }
77
78    T get_x(void) const { return x; }
79    T get_y(void) const { return y; }
80    T get_z(void) const { return z; }
81    void get(float* x, float* y, float* z) {
82      *x = get_x();
83      *y = get_y();
84      *z = get_z();
85    }
86
87    geometry_msgs::Point to_point() {
88      geometry_msgs::Point result;
89      result.x = get_x();
90      result.y = get_y();
91      result.z = get_z();
92      return result;
93    }
94
95    void saturate(T minmax) { saturate(-minmax, minmax); }
96    void saturate(T min, T max) {
97      if (x > max) {
98        set_x(max);
99      } else if (x < min) {
100       set_x(min);
101     }
102     if (y > max) {
103       set_y(max);
104     } else if (y < min) {
105       set_y(min);
106     }
107     if (z > max) {
108       set_z(max);
109     } else if (z < min) {
110       set_z(min);
111     }
112   }
113   void saturate(T mx, T my, T mz) {
114     if (x > mx) {
115       set_x(mx);
116     } else if (x < -mx) {
117       set_x(-mx);
118     }
119     if (y > my) {
120       set_y(my);
121     } else if (y < -my) {
122       set_y(-my);
123     }
124     if (z > mz) {
125       set_z(mz);
126     } else if (z < -mz) {
127       set_z(-mz);
128     }
129   }
130
131   void print() {
132     std::cout << "\n-------------" << std::endl;
133     std::cout << "x: " << x << std::endl;
134     std::cout << "y: " << y << std::endl;
135     std::cout << "z: " << z << std::endl;
136     std::cout << "-------------" << std::endl;
137   }
```

```
138
139   Triplet& operator+=(const Triplet& obj) {
140     x += (T)obj.x;
141     y += (T)obj.y;
142     z += (T)obj.z;
143     return *this;
144   }
145
146   Triplet operator+(const Triplet& obj) {
147     Triplet<T> result(*this);
148     result += obj;
149     return result;
150   }
151
152   Triplet& operator-=(const Triplet& obj) {
153     x -= (T)obj.x;
154     y -= (T)obj.y;
155     z -= (T)obj.z;
156     return *this;
157   }
158
159   Triplet operator-(const Triplet& obj) {
160     Triplet<T> result(*this);
161     result -= obj;
162     return result;
163   }
164
165   Triplet& operator*=(const Triplet& obj) {
166     x *= (T)obj.x;
167     y *= (T)obj.y;
168     z *= (T)obj.z;
169     return *this;
170   }
171
172   Triplet operator*(const Triplet& obj) {
173     Triplet<T> result(*this);
174     result *= obj;
175     return result;
176   }
177
178   Triplet operator/=(const Triplet& obj) {
179     x /= (T)obj.x;
180     y /= (T)obj.y;
181     z /= (T)obj.z;
182     return *this;
183   }
184
185   Triplet operator/(double obj) {
186     Triplet<T> div(obj);
187     Triplet<T> result(*this);
188     result /= div;
189     return result;
190   }
191 };
192
193 using namespace mavsdk;
194 class UavMonitor {
195  public:
196   UavMonitor() { sem_init(&begin, 0, 0); }
197   UavMonitor(struct param_struct ps) {
198     sem_init(&begin, 0, 0);
199     drone_params = ps;
200   }
201   virtual ~UavMonitor() {}
202   uint64_t dur = 0;
203   ros::Time last_time = ros::Time::now();
204
205   int list_counter = 0;
206   float x_list[LIST_SIZE] = {};
```

```
207     float y_list[LIST_SIZE] = {};
208     float z_list[LIST_SIZE] = {};
209     Triplet<float> pos_list[LIST_SIZE] = {};
210     ros::Time t_list[LIST_SIZE] = {};
211
212     Triplet<float> position;
213     Triplet<float> velocity;
214     Triplet<float> target;
215
216     Triplet<float> erp;  // Position Error
217     Triplet<float> erd;  // Derivative Error
218     Triplet<float> eri;  // Integrated Error
219
220     Triplet<float> kp;  // Position Gain
221     Triplet<float> kd;  // Derivative Gain
222     Triplet<float> ki;  // Integrated Gain
223
224     Triplet<double> mocap_attitude;
225     Triplet<float> rpy;
226     Triplet<float> uav_rpy;
227
228     Triplet<float> trim;
229     bool last_trim_msg;
230     bool trimmed;
231
232     float baseline = 0.1;
233     float uav_thrust = 0;
234
235     float target_yaw = 0.0;
236     float offset_yaw = 0.0;
237
238     geometry_msgs::TransformStamped transform;
239     geometry_msgs::TransformStamped yaw_transform;
240     ros::ServiceClient manip_client;
241
242     // Health
243     offboard::Health health;
244     // Battery
245     float battery = 0.0;
246     // Params
247     struct param_struct drone_params;
248
249     // Set Functions
250     void set_health(Telemetry::Health);
251     void set_battery(Telemetry::Battery);
252     void set_angle(Telemetry::EulerAngle);
253
254     // This function can be overrided in a derived class to insert a new
255     // controller
256     virtual void set_attitude_targets(Offboard::Attitude* attitude);
257     void calculate_error();
258
259     void set_trim();
260
261     float saturate(double in, double minmax);
262     float saturate_minmax(double in, double min, double max);
263
264     // Get Functions
265     bool get_health(void);
266     float get_battery(void);
267
268     // Other Functions
269     void print();
270
271     int ch = ' ';
272     volatile bool done = false;
273     volatile bool kill = false;
274     sem_t begin;
275     // Callback Functions
276     void kpCb(const geometry_msgs::Point::ConstPtr& msg);
277     void kdCb(const geometry_msgs::Point::ConstPtr& msg);
```

```
278   void kiCb(const geometry_msgs::Point::ConstPtr& msg);
279   void killCb(const std_msgs::Bool::ConstPtr& msg);
280   void startCb(const std_msgs::Bool::ConstPtr& msg);
281   void trimCb(const std_msgs::Bool::ConstPtr& msg);
282   void baselineCb(const std_msgs::Float32::ConstPtr& msg);
283   void targetCb(const geometry_msgs::Point::ConstPtr& msg);
284   void yawCb(const std_msgs::Float32::ConstPtr& msg);
285   void mocapCb(const geometry_msgs::PoseStamped::ConstPtr& msg);
286
287   // Manipulator Functions
288   bool deploy_manipulator(void);
289   bool retract_manipulator(void);
290   bool command_manipulator(int);
291
292   // Threads
293   static void* offboard_control(void* arg);
294   static void* ros_run(void* args);
295 };
296
297 /* Struct for passing arguments between main function and threads */
298 struct thread_data {
299   UavMonitor* uav;
300   ros::NodeHandle* nh;
301   std::shared_ptr<Offboard> offboard;
302   std::shared_ptr<Action> action;
303 };
304 #endif
```

Listing 1: uav_monitor.h

```cpp
// ROS libraries
#include <dynamixel_workbench_msgs/DynamixelCommand.h>
#include <geometry_msgs/Point.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/TransformStamped.h>
#include <ros/ros.h>
#include <std_msgs/Bool.h>
#include <std_msgs/Float32.h>
#include <tf/tf.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <tf2_ros/buffer.h>
#include <tf2_ros/transform_listener.h>
// Standard C++ libraries
#include <chrono>
#include <cmath>
#include <future>
#include <iostream>
#include <queue>
#include <thread>
// Standard C libraries
#include <pthread.h>
#include <semaphore.h>
#include <string.h>
#include <unistd.h>
// MAVSDK libraries
#include <mavsdk/mavsdk.h>
#include <mavsdk/plugins/action/action.h>
#include <mavsdk/plugins/info/info.h>
#include <mavsdk/plugins/offboard/offboard.h>
#include <mavsdk/plugins/telemetry/telemetry.h>
// Application libraries
#include "mavsdk_helper.h"
#include "uav_monitor.h"

using namespace mavsdk;

void UavMonitor::kpCb(const geometry_msgs::Point::ConstPtr &msg) {
  kp.set(msg->x, msg->y, msg->z);
}

void UavMonitor::kdCb(const geometry_msgs::Point::ConstPtr &msg) {
  kd.set(msg->x, msg->y, msg->z);
}

void UavMonitor::kiCb(const geometry_msgs::Point::ConstPtr &msg) {
  ki.set(msg->x, msg->y, msg->z);
}

void UavMonitor::targetCb(const geometry_msgs::Point::ConstPtr &msg) {
  yaw_transform.transform.rotation =
      tf::createQuaternionMsgFromYaw(-target_yaw * M_PI / 180);
  target.set(msg->x, msg->y, msg->z);
}

void UavMonitor::yawCb(const std_msgs::Float32::ConstPtr &msg) {
  target_yaw = msg->data;
}

void UavMonitor::mocapCb(const geometry_msgs::PoseStamped::ConstPtr &msg) {
  // create quaternion
  geometry_msgs::PoseStamped mocap;
  try {
    tf2::doTransform(*msg, mocap, transform);
  } catch (tf2::TransformException &ex) {
    ROS_WARN("%s", ex.what());
  }

```

```
68    tf::Quaternion q(mocap.pose.orientation.x, mocap.pose.orientation.y,
69                     mocap.pose.orientation.z, mocap.pose.orientation.w);
70
71    // get rotation matrix
72    tf::Matrix3x3 m(q);
73    // get r,p,y
74    double r, p, y;
75    m.getRPY(r, p, y);
76    r += r > 0 ? -M_PI : M_PI;
77    if ((ros::Time::now() - last_time) > ros::Duration(0.5)) {
78      // get offset
79      offset_yaw = (float)y * 180 / M_PI - rpy.get_z();
80      last_time = ros::Time::now();
81    }
82    mocap_attitude.set(r, p, y);
83    // Fill the list if it is not yet initialized
84    if (pos_list[0].get_x() == 0.0 && list_counter == 0) {
85      for (int i = 0; i < LIST_SIZE; i++) {
86        t_list[i] = ros::Time::now();
87      }
88    }
89    list_counter++;
90    list_counter %= LIST_SIZE;
91    //  std::cout << list_counter << std::endl;
92
93    int prev = (list_counter + 1) % LIST_SIZE;
94    pos_list[list_counter].set(mocap.pose.position.x, mocap.pose.position.y,
95                               -mocap.pose.position.z);
96    t_list[list_counter] = msg->header.stamp;
97
98    ros::Duration dt = t_list[list_counter] - t_list[prev];
99
100   velocity.set((pos_list[list_counter] - pos_list[prev]) / dt.toSec());
101
102   calculate_error();
103 }
104
105 void UavMonitor::baselineCb(const std_msgs::Float32::ConstPtr &msg) {
106   // baseline = msg->data;
107 }
108
109 void UavMonitor::killCb(const std_msgs::Bool::ConstPtr &msg) {
110   kill = msg->data;
111 }
112
113 void UavMonitor::startCb(const std_msgs::Bool::ConstPtr &msg) {
114   if (msg->data) {
115     sem_post(&begin);
116   }
117 }
118
119 void UavMonitor::trimCb(const std_msgs::Bool::ConstPtr &msg) {
120   if (!last_trim_msg && msg->data) {
121     set_trim();
122     trimmed = true;
123   }
124   last_trim_msg = msg->data;
125 }
126
127 // Health Functions
128 void UavMonitor::set_health(Telemetry::Health h) {
129   health.gyro = h.is_gyrometer_calibration_ok;
130   health.accel = h.is_accelerometer_calibration_ok;
131   health.mag = h.is_magnetometer_calibration_ok;
132   health.level = h.is_level_calibration_ok;
133
134   health.local = h.is_local_position_ok;
135   health.globe = h.is_global_position_ok;
```

```
136   health.home = h.is_home_position_ok;
137 }
138
139 bool UavMonitor::get_health() {
140   return health.gyro && health.accel && health.mag && health.level;
141 }
142
143 // Battery Functions
144 void UavMonitor::set_battery(Telemetry::Battery bat) {
145   health.battery = bat.voltage_v;
146   baseline = (9.80665 * 2.2 / 0.3368) * 1 / (health.battery * health.battery);
147 }
148
149 float UavMonitor::get_battery() { return battery; }
150
151 // Attitude Functions
152 void UavMonitor::set_angle(Telemetry::EulerAngle angle) {
153   rpy.set(angle.roll_deg, angle.pitch_deg, angle.yaw_deg);
154 }
155
156 bool UavMonitor::command_manipulator(int value) {
157   dynamixel_workbench_msgs::DynamixelCommand srv;
158
159   srv.request.command = "";
160   srv.request.id = 4;
161   srv.request.addr_name = "Goal_position";
162   srv.request.value = value;
163
164   return manip_client.call(srv);
165 }
166
167 bool UavMonitor::deploy_manipulator() {
168   dynamixel_workbench_msgs::DynamixelCommand srv;
169   return command_manipulator(MANIPULATOR_MAXIMUM);
170 }
171
172 bool UavMonitor::retract_manipulator() {
173   dynamixel_workbench_msgs::DynamixelCommand srv;
174   return command_manipulator(MANIPULATOR_MINIMUM);
175 }
176
177 // Other Functions
178 void *UavMonitor::offboard_control(void *arg) {
179   std::cout << "Starting control thread ..." << std::endl;
180   const std::string offb_mode = "ATTITUDE";
181
182   struct thread_data *args = (struct thread_data *)arg;
183
184   UavMonitor *m = args->uav;
185   std::shared_ptr<mavsdk::Offboard> offboard = args->offboard;
186   std::shared_ptr<mavsdk::Action> action = args->action;
187
188   ros::Rate rate(100);
189
190   Offboard::Attitude attitude;
191   attitude.roll_deg = 0.0f;
192   attitude.pitch_deg = -0.0f;
193   attitude.yaw_deg = 0.0f;
194   attitude.thrust_value = 0.1f;
195
196   if (sem_wait(&(m->begin)) == -1) {
197     std::cout << "Thread Sync Error. Aborting ... " << std::endl;
198     m->done = true;
199     pthread_exit(NULL);
200   }
201
202   Action::Result arm_result = action->arm();
203   action_error_exit(arm_result, "Arming failed");
204   std::cout << "Armed" << std::endl;
205
```

```
206   offboard->set_attitude(attitude);
207   Offboard::Result offboard_result = offboard->start();
208   offboard_error_exit(offboard_result, "Offboard start failed");
209   offboard_log(offb_mode, "Offboard started");
210   std::cout << "Offboard" << std::endl;
211
212   ros::Time start = ros::Time::now();
213   while (!m->kill) {
214     // Control Loop Timer
215     ros::Time end = ros::Time::now();
216     ros::Duration t = end - start;
217     start = end;
218     // Control Loop
219     m->set_attitude_targets(&attitude);
220     offboard->set_attitude(attitude);
221     rate.sleep();
222   }
223
224
225   std::cout << "Zeroing control inputs ..." << std::endl;
226   attitude.thrust_value = 0.0f;
227   attitude.roll_deg = 0.0f;
228   attitude.pitch_deg = 0.0f;
229   attitude.yaw_deg = 0.0f;
230   offboard->set_attitude(attitude);
231
232   sleep(1);
233   std::cout << "Sending kill command ..." << std::endl;
234
235   const Action::Result kill_result = action->kill();
236
237   m->done = true;
238   pthread_exit(NULL);
239 }
240
241 void UavMonitor::set_attitude_targets(Offboard::Attitude *attitude) {
242   Triplet<float> attitude_target(kp * erp + kd * erd + ki * eri);
243   attitude_target *= Triplet<float>(-1, 1, 1);
244   attitude_target.saturate(8, 8, 0.3);
245   attitude_target.get(&(attitude->pitch_deg), &(attitude->roll_deg),
246                       &(attitude->thrust_value));
247   attitude->thrust_value += baseline;
248   attitude->pitch_deg -= trim.get_x();
249   attitude->roll_deg += trim.get_y();
250   attitude->yaw_deg = target_yaw - offset_yaw;
251
252   uav_thrust = attitude->thrust_value;
253   uav_rpy.set_x(attitude->roll_deg);
254   uav_rpy.set_y(attitude->pitch_deg);
255   uav_rpy.set_z(attitude->yaw_deg);
256 }
257
258 float UavMonitor::saturate(double in, double minmax) {
259   return saturate_minmax(in, -minmax, minmax);
260 }
261
262 float UavMonitor::saturate_minmax(double in, double min, double max) {
263   if (in > max) {
264     return (float)max;
265   } else if (in < min) {
266     return (float)min;
267   }
268
269   return (float)in;
270 }
271
272 void UavMonitor::calculate_error() {
273   Triplet<float> offset;
274   if (drone_params.mode == 1) {
```

```
275       offset.set_z(sin(uav_rpy.get_x() * M_PI / 180) * drone_params.length);
276     }
277     Triplet<float> tmp(target + offset - pos_list[list_counter]);
278     geometry_msgs::Point error(tmp.to_point());
279
280     geometry_msgs::Point error_transformed;
281     tf2::doTransform(error, error_transformed, yaw_transform);
282
283     tmp.set(Triplet<float>(-1) * velocity);
284     geometry_msgs::Point derror(tmp.to_point());
285
286     geometry_msgs::Point derror_transformed;
287     tf2::doTransform(derror, derror_transformed, yaw_transform);
288
289     erp.set(error_transformed);
290     erd.set(derror_transformed);
291
292     int sval;
293     sem_getvalue(&begin, &sval);
294     if (sval > 0) {
295       eri += (erp / 100);
296       if (trimmed) {
297         eri.saturate(1, 1, 1);
298       } else {
299         eri.saturate(6, 6, 6);
300       }
301     } else {
302       eri.set(erp);
303       eri /= Triplet<float>(100, 100, 100);
304     }
305   }
306
307   void UavMonitor::set_trim() {
308     trim.set(eri);
309     trim.set_z(0);
310     eri.set_x(0);
311     eri.set_y(0);
312   }
313
314   void *UavMonitor::ros_run(void *arg) {
315     std::cout << "Starting Callbacks ..." << std::endl;
316     struct thread_data *args = (struct thread_data *)arg;
317
318     UavMonitor *uav = args->uav;
319     ros::NodeHandle *nh = args->nh;
320
321     ros::Subscriber kill_switch =
322         nh->subscribe<std_msgs::Bool>("kill", 10, &UavMonitor::killCb, uav);
323     ros::Subscriber start_sub =
324         nh->subscribe<std_msgs::Bool>("start", 10, &UavMonitor::startCb, uav);
325     ros::Subscriber trim_sub =
326         nh->subscribe<std_msgs::Bool>("trim", 10, &UavMonitor::trimCb, uav);
327     ros::Subscriber kp_sub =
328         nh->subscribe<geometry_msgs::Point>("kp", 10, &UavMonitor::kpCb, uav);
329     ros::Subscriber kd_sub =
330         nh->subscribe<geometry_msgs::Point>("kd", 10, &UavMonitor::kdCb, uav);
331     ros::Subscriber ki_sub =
332         nh->subscribe<geometry_msgs::Point>("ki", 10, &UavMonitor::kiCb, uav);
333
334     ros::Subscriber baseline_sub = nh->subscribe<std_msgs::Float32>(
335         "baseline", 10, &UavMonitor::baselineCb, uav);
336     ros::Subscriber mocap_sub = nh->subscribe<geometry_msgs::PoseStamped>(
337         "mocap", 10, &UavMonitor::mocapCb, uav);
338     ros::Subscriber target_sub = nh->subscribe<geometry_msgs::Point>(
339         "target", 10, &UavMonitor::targetCb, uav);
340     ros::Subscriber yw_sub = nh->subscribe<std_msgs::Float32>(
341         "yaw_target", 10, &UavMonitor::yawCb, uav);
342
343     uav->manip_client =
```

```
344         nh->serviceClient<dynamixel_workbench_msgs::DynamixelCommand>(
345             "/dynamixel_workbench/dynamixel_command");
346   ros::spin();
347
348   pthread_exit(NULL);
349 }
```

Listing 2: uav_monitor.cpp

```
#ifndef MAVSDK_HELPER_H
#define MAVSDK_HELPER_H

#include <chrono>
#include <cmath>
#include <future>
#include <iostream>
#include <thread>

#include <mavsdk/mavsdk.h>
#include <mavsdk/plugins/action/action.h>
#include <mavsdk/plugins/info/info.h>
#include <mavsdk/plugins/offboard/offboard.h>
#include <mavsdk/plugins/telemetry/telemetry.h>

using namespace mavsdk;

#define ERROR_CONSOLE_TEXT "\033[31m"      // Turn text on console red
#define TELEMETRY_CONSOLE_TEXT "\033[34m"  // Turn text on console blue
#define NORMAL_CONSOLE_TEXT "\033[0m"      // Restore normal console colour
void action_error_exit(Action::Result result, const std::string& message);
void offboard_error_exit(Offboard::Result result, const std::string& message);
void offboard_log(const std::string& offb_mode, const std::string msg);
void connection_error_exit(ConnectionResult result, const std::string& message);
void usage(std::string bin_name);

void wait_until_discover(Mavsdk& dc);

#endif
```

Listing 3: mavsdk_helper.h

```cpp
#include <chrono>
#include <cmath>
#include <future>
#include <iostream>
#include <thread>

#include <mavsdk/mavsdk.h>
#include <mavsdk/plugins/action/action.h>
#include <mavsdk/plugins/offboard/offboard.h>
#include <mavsdk/plugins/telemetry/telemetry.h>

using namespace mavsdk;
using std::chrono::milliseconds;
using std::chrono::seconds;
using std::this_thread::sleep_for;

#define ERROR_CONSOLE_TEXT "\033[31m"      // Turn text on console red
#define TELEMETRY_CONSOLE_TEXT "\033[34m"  // Turn text on console blue
#define NORMAL_CONSOLE_TEXT "\033[0m"      // Restore normal console colour

// Handles Action's result
inline void action_error_exit(Action::Result result,
                              const std::string& message) {
  if (result != Action::Result::Success) {
    std::cerr << ERROR_CONSOLE_TEXT << message << result << NORMAL_CONSOLE_TEXT
              << std::endl;
    exit(EXIT_FAILURE);
  }
}

// Handles Offboard's result
inline void offboard_error_exit(Offboard::Result result,
                                const std::string& message) {
  if (result != Offboard::Result::Success) {
    std::cerr << ERROR_CONSOLE_TEXT << message << result << NORMAL_CONSOLE_TEXT
              << std::endl;
    exit(EXIT_FAILURE);
  }
}

// Handles connection result
inline void connection_error_exit(ConnectionResult result,
                                  const std::string& message) {
  if (result != ConnectionResult::Success) {
    std::cerr << ERROR_CONSOLE_TEXT << message << result << NORMAL_CONSOLE_TEXT
              << std::endl;
    exit(EXIT_FAILURE);
  }
}

// Logs during Offboard control
inline void offboard_log(const std::string& offb_mode, const std::string msg) {
  std::cout << "[" << offb_mode << "] " << msg << std::endl;
}

/**
 * Does Offboard control using NED co-ordinates.
 *
 * returns true if everything went well in Offboard control, exits with a log
 * otherwise.
 */
bool offb_ctrl_ned(std::shared_ptr<mavsdk::Offboard> offboard) {
  const std::string offb_mode = "NED";
  // Send it once before starting offboard, otherwise it will be rejected.
  const Offboard::VelocityNedYaw stay{};
  offboard->set_velocity_ned(stay);

  Offboard::Result offboard_result = offboard->start();
  offboard_error_exit(offboard_result, "Offboard start failed");
  offboard_log(offb_mode, "Offboard started");
```

```
71
72    offboard_log(offb_mode, "Turn to face East");
73
74    Offboard::VelocityNedYaw turn_east{};
75    turn_east.yaw_deg = 90.0f;
76    offboard->set_velocity_ned(turn_east);
77    sleep_for(seconds(1));  // Let yaw settle.
78
79    {
80      const float step_size = 0.01f;
81      const float one_cycle = 2.0f * (float)M_PI;
82      const unsigned steps = 2 * unsigned(one_cycle / step_size);
83
84      offboard_log(offb_mode, "Go North and back South");
85      for (unsigned i = 0; i < steps; ++i) {
86        float vx = 5.0f * sinf(i * step_size);
87        Offboard::VelocityNedYaw north_and_back_south{};
88        north_and_back_south.north_m_s = vx;
89        north_and_back_south.yaw_deg = 90.0f;
90        offboard->set_velocity_ned(north_and_back_south);
91        sleep_for(milliseconds(10));
92      }
93    }
94
95    offboard_log(offb_mode, "Turn to face West");
96    Offboard::VelocityNedYaw turn_west{};
97    turn_west.yaw_deg = 270.0f;
98    offboard->set_velocity_ned(turn_west);
99    sleep_for(seconds(2));
100
101   offboard_log(offb_mode, "Go up 2 m/s, turn to face South");
102   Offboard::VelocityNedYaw up_and_south{};
103   up_and_south.down_m_s = -2.0f;
104   up_and_south.yaw_deg = 180.0f;
105   offboard->set_velocity_ned(up_and_south);
106   sleep_for(seconds(4));
107
108   offboard_log(offb_mode, "Go down 1 m/s, turn to face North");
109   Offboard::VelocityNedYaw down_and_north{};
110   up_and_south.down_m_s = 1.0f;
111   offboard->set_velocity_ned(down_and_north);
112   sleep_for(seconds(4));
113
114   // Now, stop offboard mode.
115   offboard_result = offboard->stop();
116   offboard_error_exit(offboard_result, "Offboard stop failed: ");
117   offboard_log(offb_mode, "Offboard stopped");
118
119   return true;
120 }
121
122 /**
123  * Does Offboard control using body co-ordinates.
124  *
125  * returns true if everything went well in Offboard control, exits with a log
126  * otherwise.
127  */
128 bool offb_ctrl_body(std::shared_ptr<mavsdk::Offboard> offboard) {
129   const std::string offb_mode = "BODY";
130
131   // Send it once before starting offboard, otherwise it will be rejected.
132   Offboard::VelocityBodyYawspeed stay{};
133   offboard->set_velocity_body(stay);
134
135   Offboard::Result offboard_result = offboard->start();
136   offboard_error_exit(offboard_result, "Offboard start failed: ");
137   offboard_log(offb_mode, "Offboard started");
138
```

```
139   offboard_log(offb_mode, "Turn clock-wise and climb");
140   Offboard::VelocityBodyYawspeed cc_and_climb{};
141   cc_and_climb.down_m_s = -1.0f;
142   cc_and_climb.yawspeed_deg_s = 60.0f;
143   offboard->set_velocity_body(cc_and_climb);
144   sleep_for(seconds(5));
145
146   offboard_log(offb_mode, "Turn back anti-clockwise");
147   Offboard::VelocityBodyYawspeed ccw{};
148   ccw.down_m_s = -1.0f;
149   ccw.yawspeed_deg_s = -60.0f;
150   offboard->set_velocity_body(ccw);
151   sleep_for(seconds(5));
152
153   offboard_log(offb_mode, "Wait for a bit");
154   offboard->set_velocity_body(stay);
155   sleep_for(seconds(2));
156
157   offboard_log(offb_mode, "Fly a circle");
158   Offboard::VelocityBodyYawspeed circle{};
159   circle.forward_m_s = 5.0f;
160   circle.yawspeed_deg_s = 30.0f;
161   offboard->set_velocity_body(circle);
162   sleep_for(seconds(15));
163
164   offboard_log(offb_mode, "Wait for a bit");
165   offboard->set_velocity_body(stay);
166   sleep_for(seconds(5));
167
168   offboard_log(offb_mode, "Fly a circle sideways");
169   circle.right_m_s = -5.0f;
170   circle.yawspeed_deg_s = 30.0f;
171   offboard->set_velocity_body(circle);
172   sleep_for(seconds(15));
173
174   offboard_log(offb_mode, "Wait for a bit");
175   offboard->set_velocity_body(stay);
176   sleep_for(seconds(8));
177
178   offboard_result = offboard->stop();
179   offboard_error_exit(offboard_result, "Offboard stop failed: ");
180   offboard_log(offb_mode, "Offboard stopped");
181
182   return true;
183 }
184
185 /**
186  * Does Offboard control using attitude commands.
187  *
188  * returns true if everything went well in Offboard control, exits with a log
189  * otherwise.
190  */
191 bool offb_ctrl_attitude(std::shared_ptr<mavsdk::Offboard> offboard) {
192   const std::string offb_mode = "ATTITUDE";
193
194   // Send it once before starting offboard, otherwise it will be rejected.
195   Offboard::Attitude roll{};
196   roll.roll_deg = 30.0f;
197   roll.thrust_value = 0.6f;
198   offboard->set_attitude(roll);
199
200   Offboard::Result offboard_result = offboard->start();
201   offboard_error_exit(offboard_result, "Offboard start failed");
202   offboard_log(offb_mode, "Offboard started");
203
204   offboard_log(offb_mode, "ROLL 30");
205   offboard->set_attitude(roll);
206   sleep_for(seconds(2));  // rolling
207
```

```
208     offboard_log(offb_mode, "ROLL -30");
209     roll.roll_deg = -30.0f;
210     offboard->set_attitude(roll);
211     sleep_for(seconds(2));  // Let yaw settle.
212
213     offboard_log(offb_mode, "ROLL 0");
214     roll.roll_deg = 0.0f;
215     offboard->set_attitude(roll);
216     sleep_for(seconds(2));  // Let yaw settle.
217
218     // Now, stop offboard mode.
219     offboard_result = offboard->stop();
220     offboard_error_exit(offboard_result, "Offboard stop failed: ");
221     offboard_log(offb_mode, "Offboard stopped");
222
223     return true;
224   }
225
226   void wait_until_discover(Mavsdk& dc) {
227     std::cout << "Waiting to discover system..." << std::endl;
228     std::promise<void> discover_promise;
229     auto discover_future = discover_promise.get_future();
230
231     dc.register_on_discover([&discover_promise](uint64_t uuid) {
232       std::cout << "Discovered system with UUID: " << uuid << std::endl;
233       discover_promise.set_value();
234     });
235
236     discover_future.wait();
237   }
238
239   void usage(std::string bin_name) {
240     std::cout << NORMAL_CONSOLE_TEXT << "Usage : " << bin_name
241               << " <connection_url>" << std::endl
242               << "Connection URL format should be :" << std::endl
243               << " For TCP : tcp://[server_host][:server_port]" << std::endl
244               << " For UDP : udp://[bind_host][:bind_port]" << std::endl
245               << " For Serial : serial:///path/to/serial/dev[:baudrate]"
246               << std::endl
247               << "For example, to connect to the simulator use URL: udp://:14540"
248               << std::endl;
249   }
250
251   Telemetry::LandedStateCallback landed_state_callback(
252       std::shared_ptr<Telemetry>& telemetry, std::promise<void>& landed_promise) {
253     return [&landed_promise, &telemetry](Telemetry::LandedState landed) {
254       switch (landed) {
255         case Telemetry::LandedState::OnGround:
256           std::cout << "On ground" << std::endl;
257           break;
258         case Telemetry::LandedState::TakingOff:
259           std::cout << "Taking off..." << std::endl;
260           break;
261         case Telemetry::LandedState::Landing:
262           std::cout << "Landing..." << std::endl;
263           break;
264         case Telemetry::LandedState::InAir:
265           std::cout << "Taking off has finished." << std::endl;
266           telemetry->subscribe_landed_state(nullptr);
267           landed_promise.set_value();
268           break;
269         case Telemetry::LandedState::Unknown:
270           std::cout << "Unknown landed state." << std::endl;
271           break;
272       }
273     };
274   }
275
276   int mavmain(int argc, char** argv) {
```

```
277   Mavsdk dc;
278   std::string connection_url;
279   ConnectionResult connection_result;
280
281   if (argc == 2) {
282     connection_url = argv[1];
283     connection_result = dc.add_any_connection(connection_url);
284   } else {
285     usage(argv[0]);
286     return 1;
287   }
288
289   if (connection_result != ConnectionResult::Success) {
290     std::cout << ERROR_CONSOLE_TEXT
291               << "Connection failed: " << connection_result
292               << NORMAL_CONSOLE_TEXT << std::endl;
293     return 1;
294   }
295
296   // Wait for the system to connect via heartbeat
297   wait_until_discover(dc);
298
299   // System got discovered.
300   System& system = dc.system();
301   auto action = std::make_shared<Action>(system);
302   auto offboard = std::make_shared<Offboard>(system);
303   auto telemetry = std::make_shared<Telemetry>(system);
304
305   while (!telemetry->health_all_ok()) {
306     std::cout << "Waiting for system to be ready" << std::endl;
307     sleep_for(seconds(1));
308   }
309   std::cout << "System is ready" << std::endl;
310
311   std::promise<void> in_air_promise;
312   auto in_air_future = in_air_promise.get_future();
313
314   Action::Result arm_result = action->arm();
315   action_error_exit(arm_result, "Arming failed");
316   std::cout << "Armed" << std::endl;
317
318   Action::Result takeoff_result = action->takeoff();
319   action_error_exit(takeoff_result, "Takeoff failed");
320
321   telemetry->subscribe_landed_state(
322       landed_state_callback(telemetry, in_air_promise));
323   in_air_future.wait();
324
325   //  using attitude control
326   bool ret = offb_ctrl_attitude(offboard);
327   if (ret == false) {
328     return EXIT_FAILURE;
329   }
330
331   //  using local NED co-ordinates
332   ret = offb_ctrl_ned(offboard);
333   if (ret == false) {
334     return EXIT_FAILURE;
335   }
336
337   //  using body co-ordinates
338   ret = offb_ctrl_body(offboard);
339   if (ret == false) {
340     return EXIT_FAILURE;
341   }
342
343   const Action::Result land_result = action->land();
344   action_error_exit(land_result, "Landing failed");
345
```

```
346     // Check if vehicle is still in air
347     while (telemetry->in_air()) {
348       std::cout << "Vehicle is landing..." << std::endl;
349       sleep_for(seconds(1));
350     }
351     std::cout << "Landed!" << std::endl;
352
353     // We are relying on auto-disarming but let's keep watching the telemetry for
354     // a bit longer.
355     sleep_for(seconds(3));
356     std::cout << "Finished..." << std::endl;
357
358     return EXIT_SUCCESS;
359 }
```

Listing 4: mavsdk_helper.cpp

```cpp
#include <geometry_msgs/Point.h>
#include <geometry_msgs/PointStamped.h>
#include <offboard/ActuatorArray.h>
#include <offboard/Health.h>
#include <ros/ros.h>
#include <std_msgs/Bool.h>
#include <std_msgs/Float32.h>
#include <std_msgs/Header.h>
#include <tf/tf.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <tf2_ros/buffer.h>
#include <tf2_ros/transform_listener.h>

#include <chrono>
#include <cmath>
#include <future>
#include <iostream>
#include <thread>

#include <pthread.h>
#include <string.h>
#include <unistd.h>

#include <mavsdk/mavsdk.h>
#include <mavsdk/plugins/action/action.h>
#include <mavsdk/plugins/info/info.h>
#include <mavsdk/plugins/offboard/offboard.h>
#include <mavsdk/plugins/telemetry/telemetry.h>
#include "mavsdk_helper.h"
#include "uav_monitor.h"

using namespace mavsdk;
using std::chrono::milliseconds;
using std::chrono::seconds;
using std::this_thread::sleep_for;

int main(int argc, char **argv) {
  /* Start ROS */
  ros::init(argc, argv, "interface");
  ros::NodeHandle private_node_handle("~");
  ros::NodeHandle nh;

  Mavsdk dc;
  std::string connection_url;
  ConnectionResult connection_result;
  /* Get connection url */
  if (private_node_handle.getParam("url", connection_url)) {
    std::cout << "Found param" << std::endl;
    connection_result = dc.add_any_connection(connection_url);
  } else {
    usage(argv[0]);
    return 1;
  }

  struct param_struct drone_params;
  if (private_node_handle.getParam("mode", drone_params.mode)) {
    std::cout << "Mode Found " << std::endl;
  } else {
    drone_params.mode = 0;
  }

  if (private_node_handle.getParam("mass", drone_params.mass)) {
    std::cout << "Mass Found" << std::endl;
  } else {
    drone_params.mass = 0;
  }

  if (private_node_handle.getParam("length", drone_params.length)) {
    std::cout << "Length Found" << std::endl;
```

```
70    } else {
71      drone_params.length = 0;
72    }
73
74    /* Attempt connection */
75    if (connection_result != ConnectionResult::Success) {
76      std::cout << "Connection Failed: " << connection_result << std::endl;
77      return 1;
78    }
79
80    wait_until_discover(dc);
81    System &system = dc.system();
82
83    /* Once connected initialise plugins */
84    auto action = std::make_shared<Action>(system);
85    auto info = std::make_shared<Info>(system);
86    auto offboard = std::make_shared<Offboard>(system);
87    auto telemetry = std::make_shared<Telemetry>(system);
88
89    /* Create ROS publishers */
90    ros::Publisher health_pub = nh.advertise<offboard::Health>("health", 10);
91    ros::Publisher att_pub =
92        nh.advertise<geometry_msgs::PointStamped>("attitude", 10);
93    ros::Publisher matt_pub =
94        nh.advertise<geometry_msgs::PointStamped>("mocap_att", 10);
95    ros::Publisher err_pub = nh.advertise<geometry_msgs::PointStamped>("err", 10);
96    ros::Publisher erd_pub = nh.advertise<geometry_msgs::PointStamped>("erd", 10);
97    ros::Publisher eri_pub = nh.advertise<geometry_msgs::PointStamped>("eri", 10);
98    ros::Publisher in_pub =
99        nh.advertise<geometry_msgs::PointStamped>("att_in", 10);
100   ros::Publisher thrust_pub = nh.advertise<std_msgs::Float32>("thrust", 10);
101   ros::Rate rate(100.0);
102
103   /* Create UavMonitor to handle controllers */
104   UavMonitor uav(drone_params);
105   std::cout << "UAV created with:\n\tMass:\t" << uav.drone_params.mass
106             << std::endl;
107   std::cout << "\tLength:\t" << uav.drone_params.length << std::endl;
108   /* Start Battery Subscriber */
109   Telemetry::Result set_rate_result = telemetry->set_rate_battery(10.0);
110   if (set_rate_result != Telemetry::Result::Success) {
111     std::cout << "Setting batt rate failed:" << set_rate_result << std::endl;
112   }
113
114   /* Start Attitude Subscriber */
115   set_rate_result = telemetry->set_rate_attitude(100.0);
116   if (set_rate_result != Telemetry::Result::Success) {
117     std::cout << "Setting att rate failed:" << set_rate_result << std::endl;
118   }
119
120   telemetry->subscribe_health(
121       [&uav](Telemetry::Health health) { uav.set_health(health); });
122
123   telemetry->subscribe_battery(
124       [&uav](Telemetry::Battery battery) { uav.set_battery(battery); });
125
126   telemetry->subscribe_attitude_euler(
127       [&uav](Telemetry::EulerAngle angle) { uav.set_angle(angle); });
128
129   /* struct to pass objects to threads */
130   struct thread_data thread_args;
131   thread_args.uav = &uav;
132   thread_args.offboard = offboard;
133   thread_args.action = action;
134   thread_args.nh = &nh;
135
136   /* structure for measuring loop time */
137
```

```
138   /* Find transformation*/
139   tf2_ros::Buffer tBuffer;
140   tf2_ros::TransformListener tfListener(tBuffer);
141   uav.transform = tBuffer.lookupTransform("HexyBoi", "base_link", ros::Time(0),
142                                           ros::Duration(1.0));
143
144   /* start listening for msgs */
145   pthread_t offboard_thread, callback_thread;
146   pthread_create(&callback_thread, NULL, &UavMonitor::ros_run,
147                  (void *)&thread_args);
148
149   /* go into offboard control */
150   pthread_create(&offboard_thread, NULL, &UavMonitor::offboard_control,
151                  (void *)&thread_args);
152
153   std::cout << "Starting publishers ..." << std::endl;
154
155   /*Create ROS msgs*/
156   offboard::Health health;
157   geometry_msgs::PointStamped attitude;
158   geometry_msgs::PointStamped mocap_att;
159   geometry_msgs::PointStamped err;
160   geometry_msgs::PointStamped erd;
161   geometry_msgs::PointStamped eri;
162   geometry_msgs::PointStamped in;
163
164   std_msgs::Float32 thrust;
165
166   std_msgs::Header header;
167   header.stamp = ros::Time::now();
168
169   while (!uav.done) {
170     ros::Time start = ros::Time::now();
171
172     health = uav.health;
173
174     header.stamp = ros::Time::now();
175     attitude.point = uav.rpy.to_point();
176     attitude.header = header;
177     mocap_att.point = uav.mocap_attitude.to_point();
178     mocap_att.header = header;
179     err.point = uav.erp.to_point();
180     err.header = header;
181     erd.point = uav.erd.to_point();
182     erd.header = header;
183     eri.point = uav.eri.to_point();
184     eri.header = header;
185     in.point = uav.uav_rpy.to_point();
186     in.header = header;
187
188     thrust.data = uav.uav_thrust;
189
190     att_pub.publish(attitude);
191     matt_pub.publish(mocap_att);
192     health_pub.publish(health);
193     err_pub.publish(err);
194     erd_pub.publish(erd);
195     eri_pub.publish(eri);
196     in_pub.publish(in);
197     thrust_pub.publish(thrust);
198     ros::Time end = ros::Time::now();
199     rate.sleep();
200   }
201   pthread_join(offboard_thread, NULL);
202
203   std::cout << "Shutting down ROS ..." << std::endl;
204
205   ros::shutdown();
206
207   pthread_join(callback_thread, NULL);
```

```
208
209     std::cout << "Ending the program ..." << std::endl;
210
211     return 0;
212 }
```

Listing 5: ros_interface.cpp

```
#!/usr/bin/env python
import rospy
import std_msgs.msg as smsg
import geometry_msgs.msg as geo
import mavros_msgs.msg as mav
import offboard.msg as test

import os
import sys
import termios
import fcntl
import atexit
import time

old_settings=None
done = False

def init_anykey():
    global old_settings
    old_settings = termios.tcgetattr(sys.stdin)
    new_settings = termios.tcgetattr(sys.stdin)
    new_settings[3] = new_settings[3] & ~(termios.ECHO | termios.ICANON) # lflags
    new_settings[6][termios.VMIN] = 0  # cc
    new_settings[6][termios.VTIME] = 0 # cc
    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, new_settings)

@atexit.register
def term_anykey():
    global old_settings
    if old_settings:
        termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_settings)

def anykey():
    ch_set = []
    ch = os.read(sys.stdin.fileno(), 1)
    while ch != None and len(ch) > 0:
        ch_set.append( ord(ch[0]) )
        ch = os.read(sys.stdin.fileno(), 1)
    return ch_set;

def intro():
    print("================================================================")
    print("Command Line Control Interface")
    print("================================================================")
    print("Press Q to kill")
    print("Press A to abort this program")
    print("Press E/D to change target yaw by +/- 10")
    print("Press T/G to change target height by +/- 0.25")
    print("Press R/F to change baseline thrust by +/- 0.01")
    print("Press Y/H to change Kpxy by +/- 0.1")
    print("Press U/J to change Kdxy by +/- 0.1")
    print("Press I/K to change Kpz  by +/- 0.01")
    print("Press O/L to change Kdz  by +/- 0.01")

def printBool(boolean):
    if boolean:
        return "OK"
    else:
        return "FAIL"


class Status:
    def __init__(self):
        self.health = test.Health()
        self.kp = geo.Point()
        self.kd = geo.Point()
        self.ki = geo.Point()
        self.zz = geo.Point()

        self.bl = smsg.Float32()
```

```
71          self.kl = smsg.Bool()
72          self.st = smsg.Bool()
73          self.st.data = False
74          self.pos = geo.Point()
75          self.vel = geo.Point()
76          self.err = geo.Point()
77          self.erd = geo.Point()
78          self.eri = geo.Point()
79          self.yaw = smsg.Float32()
80          self.attitude = geo.Point()
81          self.mocap_att = geo.Point()
82          self.thrust = 0.0
83
84          self.kp.x = 8.0
85          self.kp.y = 8.0
86          self.kp.z = 0.05
87
88          self.kd.x = 10.0
89          self.kd.y = 10.0
90          self.kd.z = 0.1
91
92          self.ki.x = 1.0
93          self.ki.y = 1.0
94          self.ki.z = 0.02
95          self.bl.data = 0.14
96          self.yaw.data = 0.0
97
98          self.kp_pub = rospy.Publisher('kp', geo.Point, queue_size=1)
99          self.kd_pub = rospy.Publisher('kd', geo.Point, queue_size=1)
100         self.ki_pub = rospy.Publisher('ki', geo.Point, queue_size=1)
101         self.bl_pub = rospy.Publisher('baseline', smsg.Float32, queue_size=1)
102         self.zz_pub = rospy.Publisher('target', geo.Point, queue_size=1)
103         self.kl_pub = rospy.Publisher('kill', smsg.Bool, queue_size=1)
104         self.st_pub = rospy.Publisher('start', smsg.Bool, queue_size=1)
105         self.yw_pub = rospy.Publisher('yaw_target', smsg.Float32, queue_size=1)
106
107     def healthCb(self, msg):
108         self.health = msg
109
110     def attCb(self, msg):
111         self.attitude = msg.point
112
113     def mattCb(self, msg):
114         self.mocap_att = msg.point
115
116     def posCb(self, msg):
117         self.pos= msg.point
118
119     def velCb(self, msg):
120         self.vel = msg.point
121
122     def errCb(self, msg):
123         self.err = msg.point
124
125     def erdCb(self, msg):
126         self.erd = msg.point
127
128     def eriCb(self, msg):
129         self.eri = msg.point
130
131     def thrustCb(self, msg):
132         self.thrust = msg.data
133
134     def baselineCb(self, msg):
135         self.bl = msg
136
137     def targetCb(self, msg):
138         self.zz = msg
139
140     def killCb(self, msg):
```

```
141         self.kl = msg
142 
143     def startCb(self, msg):
144         self.st = msg
145 
146     def yawCb(self, msg):
147         self.yaw = msg
148 
149     def publish(self):
150         self.kp_pub.publish(self.kp)
151         self.kd_pub.publish(self.kd)
152         self.ki_pub.publish(self.ki)
153         self.bl_pub.publish(self.bl)
154         self.zz_pub.publish(self.zz)
155         self.kl_pub.publish(self.kl)
156         self.st_pub.publish(self.st)
157         self.yw_pub.publish(self.yaw)
158 
159 
160     def print_status(self, first=False):
161         if not first:
162             print("\r\033[26A")
163         else:
164             pass
165         zzx = round(self.zz.x,5)
166         zzy = round(self.zz.y,5)
167         zzz = round(self.zz.z,5)
168         kpx = round(self.kp.x,5)
169         kpy = round(self.kp.y,5)
170         kpz = round(self.kp.z,5)
171         kdx = round(self.kd.x,5)
172         kdy = round(self.kd.y,5)
173         kdz = round(self.kd.z,5)
174         kix = round(self.ki.x,5)
175         kiy = round(self.ki.y,5)
176         kiz = round(self.ki.z,5)
177         bl  = round(self.bl.data, 5)
178 
179 
180         px = round(self.pos.x, 5)
181         py = round(self.pos.y, 5)
182         pz = round(self.pos.z, 5)
183         vx = round(self.vel.x, 5)
184         vy = round(self.vel.y, 5)
185         vz = round(self.vel.z, 5)
186         ex = round(self.err.x, 5)
187         ey = round(self.err.y, 5)
188         ez = round(self.err.z, 5)
189         exd = round(self.erd.x, 5)
190         eyd = round(self.erd.y, 5)
191         ezd = round(self.erd.z, 5)
192         exi = round(self.eri.x, 5)
193         eyi = round(self.eri.y, 5)
194         ezi = round(self.eri.z, 5)
195 
196         t = round(self.thrust, 5)
197 
198         pr = round(self.attitude.x, 5)
199         pp = round(self.attitude.y, 5)
200         pw = round(self.attitude.z, 5)
201 
202         mr = round(self.mocap_att.x, 5)
203         mp = round(self.mocap_att.y, 5)
204         my = round(self.mocap_att.z, 5)
205 
206         health = self.health;
207         print("------------------------------------------------------------")
208         print("Health".ljust(55))
```

```
209             print("-------------------------------------------------------------")
210             print((" Gyro  : "+printBool(health.gyro)).ljust(15) + (" Local    :
                ↪  "+printBool(health.local)).ljust(20))
211             print((" Accel : "+printBool(health.accel)).ljust(15) + (" Global   :
                ↪  "+printBool(health.globe)).ljust(20))
212             print((" Mag   : "+printBool(health.mag)).ljust(15) + (" Home     :
                ↪  "+printBool(health.home)).ljust(20))
213             print((" Level : "+printBool(health.level)).ljust(15) + (" Battery  :
                ↪  "+str(round(health.battery,5)).ljust(20)))
214             print("-------------------------------------------------------------")
215
                ↪  print("Uav_Ang".ljust(15)+"Mocap_Ang".ljust(15)+"Errors".ljust(15)+"Gains".ljust(10)+
                ↪  ')
216             print("-------------------------------------------------------------")
217             print((' r: '+str(pr)).ljust(15) + (' r: '+str(mr)).ljust(15) +
218                   (' ex: '+str(ex)).ljust(15) + (' kpx: '+str(kpx)).ljust(10)+'   ')
219
220             print((' p: '+str(pp)).ljust(15) + (' p: '+str(mp)).ljust(15) +
221                   (' ey: '+str(ey)).ljust(15) + (' kpy: '+str(kpy)).ljust(10)+'   ')
222
223             print((' y: '+str(pw)).ljust(15) + (' y: '+str(my)).ljust(15) +
224                   (' ez: '+str(ez)).ljust(15) + (' kpz: '+str(kpz)).ljust(10)+'   ')
225
226             print("-------------------------------------------------------------")
227
                ↪  print("Current_Pos".ljust(15)+"Target_Pos".ljust(15)+"Errors".ljust(15)+"Gains".ljust
                ↪  ')
228             print("-------------------------------------------------------------")
229             print((' x: '+str(px)).ljust(15) + (' x: '+str(zzx)).ljust(15) +
230                   (' exd: '+str(exd)).ljust(15) + (' kdx: '+str(kdx)).ljust(10)+'   ')
231
232             print((' y: '+str(py)).ljust(15) + (' y: '+str(zzy)).ljust(15) +
233                   (' eyd: '+str(eyd)).ljust(15) + (' kdy: '+str(kdy)).ljust(10)+'   ')
234
235             print((' z: '+str(pz)).ljust(15) + (' z: '+str(zzz)).ljust(15) +
236                   (' ezd: '+str(ezd)).ljust(15) + (' kdz: '+str(kdz)).ljust(10)+'   ')
237
238             print("-------------------------------------------------------------")
239             print("Current_Vel".ljust(15)+("Yaw:
                ↪  "+str(self.yaw.data)).ljust(30)+"Kill:"+str(self.kl.data).rjust(6))
240             print("-------------------------------------------------------------")
241             print((' x: '+str(vx)).ljust(15) + (' T: '+str(t)).ljust(15) +
242                   ('exi: '+str(exi)).ljust(15) + (' kix: '+str(kix)).ljust(10)+'   ')
243
244             print((' y: '+str(vy)).ljust(15) + ('Baseline').ljust(15) +
245                   ('eyi: '+str(eyi)).ljust(15) + (' kiy: '+str(kiy)).ljust(10)+'   ')
246
247             print((' z: '+str(vz)).ljust(15) + (' T: '+str(bl)).ljust(15) +
248                   ('ezi: '+str(ezi)).ljust(15) + (' kiz: '+str(kiz)).ljust(10)+'   ')
249
250     def kill(self):
251         self.kl.data = True
252         self.bl.data = 0.0
253     def incKpz(self):
254         self.kp.z += 0.01
255
256     def decKpz(self):
257         self.kp.z -= 0.01
258
259     def incKpxy(self):
260         self.kp.x += 0.1
261         self.kp.y += 0.1
262
263     def decKpxy(self):
264         self.kp.x -= 0.1
265         self.kp.y -= 0.1
266
267     def incKdz(self):
```

```
268             self.kd.z += 0.01
269
270         def decKdz(self):
271             self.kd.z -= 0.01
272
273         def incKdxy(self):
274             self.kd.x += 0.1
275             self.kd.y += 0.1
276
277         def decKdxy(self):
278             self.kd.x -= 0.1
279             self.kd.y -= 0.1
280
281         def incKiz(self):
282             self.ki.z += 0.01
283
284         def decKiz(self):
285             self.ki.z -= 0.01
286
287         def incKixy(self):
288             self.ki.x += 0.01
289             self.ki.y += 0.01
290
291         def decKixy(self):
292             self.ki.x -= 0.01
293             self.ki.y -= 0.01
294
295         def incBl(self):
296             self.bl.data += 0.01
297
298         def decBl(self):
299             self.bl.data -= 0.01
300
301         def incZz(self):
302             self.zz.z += 0.12
303
304         def decZz(self):
305             self.zz.z -= 0.12
306
307         def incXx(self):
308             self.zz.x += 0.25
309
310         def decXx(self):
311             self.zz.x -= 0.25
312
313         def incYy(self):
314             self.zz.y += 0.25
315
316         def decYy(self):
317             self.zz.y -= 0.25
318
319         def start(self):
320             self.st.data = True
321
322         def incYaw(self):
323             self.yaw.data += 2.5
324
325         def decYaw(self):
326             self.yaw.data -= 2.5
327
328         def manExt(self):
329             os.system("rosrun offboard dynamixel_ctrl position:=1")
330
331         def manRet(self):
332             os.system("rosrun offboard dynamixel_ctrl position:=0")
333
334     def main():
335         global done
336         init_anykey()
337
338         rospy.init_node('test_ui', anonymous=True)
```

```
339
340     rate = rospy.Rate(5.0)
341
342     stat = Status()
343
344     rospy.Subscriber('health', test.Health, stat.healthCb)
345     rospy.Subscriber('attitude', geo.PointStamped, stat.attCb)
346     rospy.Subscriber('mocap_att', geo.PointStamped, stat.mattCb)
347     rospy.Subscriber('pose', geo.PointStamped, stat.posCb)
348     rospy.Subscriber('vel', geo.PointStamped, stat.velCb)
349     rospy.Subscriber('err', geo.PointStamped, stat.errCb)
350     rospy.Subscriber('erd', geo.PointStamped, stat.erdCb)
351     rospy.Subscriber('eri', geo.PointStamped, stat.eriCb)
352     rospy.Subscriber('thrust', smsg.Float32, stat.thrustCb)
353
354     rospy.Subscriber('baseline_', smsg.Float32, stat.baselineCb)
355     rospy.Subscriber('target_', geo.Point, stat.targetCb)
356     rospy.Subscriber('kill_', smsg.Bool, stat.killCb)
357     rospy.Subscriber('start_', smsg.Bool, stat.startCb)
358     rospy.Subscriber('yaw_target_', smsg.Float32, stat.yawCb)
359     intro()
360     stat.print_status(True)
361     while not done:
362         stat.print_status()
363         key = anykey()
364         if key != []:
365             for k in key:
366                 if chr(k) == 'a':
367                     stat.kill()
368                     stat.publish()
369                     done=True
370                     #print("\n"*13)
371                 elif chr(k) == 'q':
372                     stat.kill()
373                 elif chr(k) == 'r':
374                     stat.manExt()
375                 elif chr(k) == 'f':
376                     stat.manRet()
377                 elif chr(k) == 't':
378                     stat.incZz()
379                 elif chr(k) == 'g':
380                     stat.decZz()
381                 elif chr(k) == 'z':
382                     stat.incXx()
383                 elif chr(k) == 'x':
384                     stat.decXx()
385                 elif chr(k) == 'y':
386                     stat.incKpxy()
387                 elif chr(k) == 'h':
388                     stat.decKpxy()
389                 elif chr(k) == 'u':
390                     stat.incKdxy()
391                 elif chr(k) == 'j':
392                     stat.decKdxy()
393                 elif chr(k) == 'i':
394                     stat.incKpz()
395                 elif chr(k) == 'k':
396                     stat.decKpz()
397                 elif chr(k) == 'o':
398                     stat.incKdz()
399                 elif chr(k) == 'l':
400                     stat.decKdz()
401                 elif chr(k) == 'n':
402                     stat.incKiz()
403                 elif chr(k) == 'm':
404                     stat.decKiz()
405                 elif chr(k) == 'v':
```

```
406                     stat.incKixy()
407                 elif chr(k) == 'b':
408                     stat.decKixy()
409                 elif chr(k) == 'c':
410                     stat.start()
411                 elif chr(k) == 'e':
412                     stat.incYaw()
413                 elif chr(k) == 'd':
414                     stat.decYaw()
415
416
417
418         stat.publish()
419         rate.sleep()
420
421 if __name__ == '__main__':
422     try:
423         main()
424     except rospy.ROSInterruptException:
425         pass
```

Listing 6: drone_ui.py

```
<launch>
        <!--group ns="HexyBoi"-->

                <!--include file="$(find offboard)/launch/dynamixel.launch"/-->
                <arg name="drone_param_file"
                         default="$(find offboard)/config/drone.yaml" />

                <node pkg="offboard"
                          type="ros_interface"
                          name="fly"
                          output="screen">

                                <param name="~url" type="string" value="serial:///dev/ttyACM0" />
                                <param name="~mode" type="int" value="0"/>
                                <!--param name="~mass" type="double" value="2.2"/-->
                                <!--param name="~length" type="double" value="1"/-->
                                <rosparam file="$(arg drone_param_file)" command="load" />
                </node>

        <!-- /group-->
</launch>
```

Listing 7: fly.launch

```
<launch>
  <!-- Pass in mocap_config_file:=/path/to/config.yaml to change options. -->
  <arg name="mocap_config_file"
       default="$(find offboard)/config/mocap.yaml" />

  <arg name="drone_name"
                 default="HexyBoi" />

  <!-- Setup machine to launch nodes directly on drone -->
  <machine name="rpi"
                   user="pi"
                   address="hexyboi"
                   password="raspberry"
                   env-loader="/home/pi/catkin_ws/env.sh"/>

  <group ns="$(arg drone_name)">

  <!-- Mocap Data -->
                <node pkg="mocap_optitrack"
                        type="mocap_node"
                        name="mocap_node"
                        respawn="false"
                        launch-prefix=""
                        required="true">
                    <rosparam file="$(arg mocap_config_file)" command="load" />
                  </node>

  <!-- Publish fixed coordinate systems transform -->
                  <node pkg="tf"
                          type="static_transform_publisher"
                              name="static_transform"
                              args="0.0 0.0 0.0 0.0 0.0 3.14159 base_link $(arg drone_name) 10">
                  </node>

  <!-- Start UI -->
                <node pkg="offboard"
                              type="drone_ui.py"
                              name="clui"
                              output="screen"
                        required="true">
                  </node>


  <!-- Start Flight Node on Drone
                <node machine="rpi"
                          pkg="offboard"
                          type="ros_interface"
                          name="flight">
                          <param name="~url" type="string" value="serial:///dev/ttyACM0" />
                </node>
  -->
  </group>

</launch>
```

Listing 8: startup.launch

```yaml
#
# Definition of all trackable objects
# Identifier corresponds to Trackable ID set in Tracking Tools
#
rigid_bodies:
    '1':
        pose: mocap
        pose2d: mocap2d
        child_frame_id: base_link
        parent_frame_id: world
        use_new_coordinates: false
optitrack_config:
        multicast_address: 239.255.42.99
```

Listing 9: mocap.yaml

```yaml
#
#Drone Parameters
#
mass: 2.2
length: 1.07
```

Listing 10: drone.yaml

```
bool gyro
bool accel
bool mag
bool level
bool local
bool globe
bool home
float32 battery
```

Listing 11: Health.msg

Bibliography

[1] Mike Allenspach et al. "Design and optimal control of a tiltrotor micro-aerial vehicle for efficient omnidirectional flight." eng. In: *International Journal of Robotics Research* 39.10-11 (2020), pages 1305–1325. ISSN: 17413176, 02783649. DOI: 10.1177/0278364920943654.

[2] Auterion. *The story of PX4 and Pixhawk*. https://auterion.com/company/the-history-of-pixhawk/. [Online; accessed 11-April-2021]. 2020.

[3] Karen Bodie et al. "An Omnidirectional Aerial Manipulation Platform for Contact-Based Inspection." eng. In: *Robotics: Science and Systems Xv* (2019).

[4] Dario Brescianini, Markus Hehn, and Raffaello D'Andrea. *Nonlinear Quadrocopter Attitude Control. Technical Report*. en. Technical report. Zürich, 2013. DOI: 10.3929/ethz-a-009970340.

[5] Xilun Ding et al. "A review of aerial manipulation of small-scale rotorcraft unmanned robotic systems." eng. In: *Chinese Journal of Aeronautics* 32.1 (2019), pages 200–214. ISSN: 10009361, 25889230. DOI: 10.1016/j.cja.2018.05.012.

[6] Simulink Documentation. *Simulation and Model-Based Design*. [Online; accessed 9-April-2021]. 2020. URL: https://www.mathworks.com/products/simulink.html.

[7] Engineering ToolBox. *Friction and Friction Coefficients*. https://www.engineeringtoolbox.com/friction-coefficients-d_778.html. [Online; accessed 9-April-2021]. 2004.

[8] M. Fumagalli et al. "Modeling and Control of a Flying Robot for Contact Inspection." eng. In: *2012 Ieee/rsj International Conference on Intelligent Robots and Systems (iros)* (2012), pages 3532–3537. ISSN: 21530866, 21530858.

[9] A. E. Jimenez-Cano et al. "Aerial manipulator for structure inspection by contact from the underside." In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pages 1879–1884. DOI: 10.1109/IROS.2015.7353623.

[10] Mina Kamel et al. "The Voliro Omniorientational Hexacopter: An Agile and Maneuverable Tiltable-Rotor Aerial Vehicle." eng. In: *Ieee Robotics and Automation Magazine* 25.4 (2018), page 8485627. ISSN: 10709932, 1558223x. DOI: 10.1109/MRA.2018.2866758.

[11] Suseong Kim, Hoseong Seo, and H. Jin Kim. "Operating an unknown drawer using an aerial manipulator." eng. In: *2015 Ieee International Conference on Robotics and Automation (icra)* (2015), pages 5503–5508. DOI: 10.1109/ICRA.2015.7139968.

[12] A. Koubâa et al. "Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey." In: *IEEE Access* 7 (2019), pages 87658–87680. DOI: 10.1109/ACCESS.2019.2924410.

[13] Vijay Kumar and Nathan Michael. "Opportunities and challenges with autonomous micro aerial vehicles." eng. In: *Springer Tracts in Advanced Robotics* 100 (2017), pages 41–58. ISSN: 1610742x, 16107438. DOI: 10.1007/978-3-319-29363-9_3.

[14] Teppo Luukkonen. *Modelling and control of quadcopter*. Technical report Mat-2.4108. Espoo: Aalto University, School of Science, August 2011.

[15] Reza Olfati-Saber. "Nonlinear control of underactuated mechanical systems with application to robotics and aerospace vehicles." eng. In: (2001).

[16] M. Orsag et al. "Dexterous Aerial Robots—Mobile Manipulation Using Unmanned Aerial Systems." In: *IEEE Transactions on Robotics* 33.6 (2017), pages 1453–1466. DOI: 10.1109/TRO.2017.2750693.

[17] Sangyul Park et al. "ODAR: Aerial Manipulation Platform Enabling Omnidirectional Wrench Generation." eng. In: *Ieee/asme Transactions on Mechatronics* 23.4 (2018), pages 1–1. ISSN: 1941014x, 10834435. DOI: 10.1109/TMECH.2018.2848255.

[18] PX4 contributors. *PX4 User Guide*. https://docs.px4.io/master/en/. [Online; accessed 7-April-2021]. 2021.

[19] Morgan Quigley et al. "ROS: an open-source Robot Operating System." In: *ICRA workshop on open source software*. Volume 3. 3.2. Kobe, Japan. 2009, page 5.

[20] ETH Zurich Robotic Systems Lab. *Robot Dynamics Lecture Notes*. https://ethz.ch/content/dam/ethz/special-interest/mavt/robotics-n-intelligent-systems/rsl-dam/documents/RobotDynamics2017/RD_HS2017script.pdf. [Online; accessed 11-April-2021]. 2017.

[21] Fabio Ruggiero, Vincenzo Lippiello, and Anibal Ollero. "Aerial manipulation: A literature review." In: *IEEE Robotics and Automation Letters* 3.3 (2018), pages 1957–1964.

[22] Markus Ryll et al. "6D interaction control with aerial robots: The flying end-effector paradigm." und. In: *International Journal of Robotics Research* 38.9 (2019), pages 1045–1062. ISSN: 17413176, 02783649. DOI: 10.1177/0278364919856694.

[23] Inc. The MathWorks. *Simscape Multibody*. [Online; accessed 9-April-2021]. Natick, Massachusetts, United State, 2021. URL: https://www.mathworks.com/help/physmod/sm.

[24] Marco Tognon et al. "A Truly-Redundant Aerial Manipulator System With Application to Push-and-Slide Inspection in Industrial Plants." eng. In: *Ieee Robotics and Automation Letters* 4.2 (2019), pages 1846–1851. ISSN: 23773774, 23773766. DOI: 10.1109/LRA.2019.2895880.

[25] Miguel Ángel Trujillo et al. “Novel Aerial Manipulator for Accurate and Robust Industrial NDT Contact Inspection: A New Tool for the Oil and Gas Inspection Industry.” und. In: *Sensors (basel, Switzerland)* 19.6 (2019), page 1305. ISSN: 14248220, 14243210. DOI: 10.3390/s19061305.

[26] Hideyuki Tsukagoshi et al. “Aerial manipulator with perching and door-opening capability.” eng. In: *Proceedings - Ieee International Conference on Robotics and Automation* 2015-.June (2015), page 7139845. ISSN: 10504729. DOI: 10.1109/ICRA.2015.7139845.

[27] Akash Varshney, Deeksha Gupta, and Bharti Dwivedi. “Speed response of brushless DC motor using fuzzy PID controller under varying load condition.” In: *Journal of Electrical Systems and Information Technology* 4.2 (2017), pages 310–321. ISSN: 2314-7172. DOI: https://doi.org/10.1016/j.jesit.2016.12.014. URL: https://www.sciencedirect.com/science/article/pii/S2314717217300077.

[28] H. W. Wopereis et al. “Application of substantial and sustained force to vertical surfaces using a quadrotor.” eng. In: *2017 Ieee International Conference on Robotics and Automation (icra)* (2017), pages 2704–2709. DOI: 10.1109/ICRA.2017.7989314.

[29] Han W. Wopereis et al. “Multimodal Aerial Locomotion: An Approach to Active Tool Handling.” eng. In: *Ieee Robotics and Automation Magazine* 25.4 (2018), pages 57–65. ISSN: 10709932, 1558223x. DOI: 10.1109/MRA.2018.2869527.

DTU Electrical Engineering
Department of Electrical Engineering
Technical University of Denmark
Ørsteds Plads
Building 348
DK-2800 Kgs. Lyngby
Denmark
Tel: (+45) 45 25 38 00
www.elektro.dtu.dk